Grundlagen Großer Sprachmodelle

Tong Xiao and Jingbo Zhu

18. September 2025

NLP Lab, Northeastern University & NiuTrans Research

Der Inhalt dieses Buches ist aus dem NLPBook ausgewählt und übersetzt. Weitere Sprachversionen:
https://github.com/NiuTrans/NLPBookTranslations

Übersetzung durch LaTexTrans, das automatische Übersetzungsprogramm für arXiv-Paper:
https://github.com/NiuTrans/LaTeXTrans

Natural Language Processing Lab, Northeastern University
&
NiuTrans Research

18. September 2025

Vorwort

Große Sprachmodelle stammen ursprünglich aus der Verarbeitung natürlicher Sprache, sind aber in den letzten Jahren zweifellos zu einer der revolutionärsten technologischen Entwicklungen im Bereich der künstlichen Intelligenz geworden. Eine wichtige Erkenntnis, die große Sprachmodelle gebracht haben, ist, dass Wissen über die Welt und Sprachen durch umfangreiche Sprachmodellierungsaufgaben erworben werden kann und dass wir auf diese Weise ein universelles Modell schaffen können, das vielfältige Probleme bewältigt. Diese Entdeckung hat die Forschungsmethoden in der Verarbeitung natürlicher Sprache und vielen verwandten Disziplinen tiefgreifend beeinflusst. Wir haben uns von dem Training spezialisierter Systeme von Grund auf, das eine große Menge an annotierten Daten verwendet, zu einem neuen Paradigma verlagert, das umfangreiches Vortraining nutzt, um Basismodelle zu erhalten, die dann feingetunt, ausgerichtet und durch Prompts gesteuert werden.

Dieses Buch zielt darauf ab, die grundlegenden Konzepte großer Sprachmodelle darzulegen und die damit verbundenen Techniken vorzustellen. Wie der Titel schon andeutet, konzentriert sich das Buch mehr auf die grundlegenden Aspekte großer Sprachmodelle, anstatt eine umfassende Abdeckung aller hochmodernen Methoden zu bieten. Das Buch besteht aus fünf Kapiteln:

- Kapitel 1 führt in die Grundlagen des Vortrainings ein. Dies ist die Grundlage großer Sprachmodelle, und gängige Vortrainingsmethoden sowie Modellarchitekturen werden hier besprochen.
- Kapitel 2 führt generative Modelle ein, welche die großen Sprachmodelle sind, auf die wir uns heute üblicherweise beziehen. Nach der Darstellung des grundlegenden Prozesses zur Erstellung dieser Modelle werden wir auch untersuchen, wie man das Modelltraining skaliert und lange Texte verarbeitet.
- Kapitel 3 führt Prompting-Methoden für große Sprachmodelle ein. Wir werden verschiedene Prompting-Strategien sowie fortgeschrittenere Methoden wie das Chain-of-Thought-Reasoning und das automatische Prompt-Design diskutieren.
- Kapitel 4 führt Alignment-Methoden für große Sprachmodelle ein. Wir werden uns auf das anweisungsbasierte Fine-Tuning und das Alignment basierend auf menschlichem Feedback konzentrieren.
- Kapitel 5 führt Inferenzmethoden für große Sprachmodelle ein. Wir werden verschiedene Dekodierungsalgorithmen, Beschleunigungsmethoden und das Problem der Skalierung zur Inferenzzeit diskutieren.

Wenn Leser über Vorkenntnisse im maschinellen Lernen und in der Verarbeitung natürlicher Sprache sowie über ein gewisses Verständnis von neuronalen Netzwerken wie Transformern verfügen, wird die Lektüre dieses Buches recht einfach sein. Aber auch ohne dieses Vorwissen ist es völlig in Ordnung, da wir den Inhalt jedes Kapitels so in sich geschlossen wie möglich gestaltet haben, um sicherzustellen, dass die Leser nicht mit zu großen Leseschwierigkeiten belastet werden.

Der hier präsentierte Inhalt ist Teil einer umfassenden Einführungsressource zu neuronalen Netzwerken und großen Sprachmodellen in der Verarbeitung natürlicher Sprache. Leser, die mehr über Hintergrundthemen wie Sequenzmodellierung und Aufmerksamkeitsmechanismen erfahren möchten, können für weitere Informationen https://github.com/NiuTrans/NLPBook oder https://niutrans.github.io/NLPBook besuchen.

Wir möchten den Studierenden in unserem Labor und all unseren Freunden danken, die ihre Ansichten zu großen Sprachmodellen mit uns geteilt und bei der Korrektur von Schreibfehlern geholfen haben. Insbesondere möchten wir Weiqiao Shan, Yongyu Mu, Chenglong Wang, Kaiyan Chang, Yuchun Fan, Hang Zhou, Chuanhao Lv, Xinyu Liu, Tao Zhou, Huiwen Bao, Tong Zheng, Junhao Ruan, Yingfeng Luo, Yuzhang Wu, Yifu Huo, Xiaoqian Liu, Ziming Zhu, Shunjie Xing, Bingsen Zhou, Shan Xie, Yonghao Shi, Dan Chen, Xihan Yang, Junxin Wang, Miaohe Niu und Qi Meng danken.

Notation

a	Variable
$\mathbf{a}$	Zeilenvektor oder Matrix
$f(a)$	Funktion von a
$\max f(a)$	Maximalwert von $f(a)$
$\arg\max_a f(a)$	Wert von a, der $f(a)$ maximiert
$\mathbf{x}$	Eingabe-Tokensequenz eines Modells
x_j	Eingabe-Token an Position j
$\mathbf{y}$	Ausgabe-Tokensequenz, die von einem Modell erzeugt wird
y_i	Ausgabe-Token an Position i
θ	Modellparameter
$\Pr(a)$	Wahrscheinlichkeit von a
$\Pr(a\|b)$	Bedingte Wahrscheinlichkeit von a gegeben b
$\Pr(\cdot\|b)$	Wahrscheinlichkeitsverteilung einer Variablen gegeben b
$\Pr_\theta(a)$	Wahrscheinlichkeit von a, parametrisiert durch θ
$\mathbf{h}_t$	Verborgener Zustand zum Zeitschritt t in sequenziellen Modellen
$\mathbf{H}$	Matrix aller verborgenen Zustände über die Zeit in einer Sequenz
$\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$	Query-, Key- und Value-Matrizen in Aufmerksamkeitsmechanismen
$\text{Softmax}(\mathbf{A})$	Softmax-Funktion, die den Eingabevektor oder die Eingabematrix $\mathbf{A}$ normalisiert
$\mathcal{L}$	Verlustfunktion
$\mathcal{D}$	Datensatz, der für das Training oder Fine-Tuning eines Modells verwendet wird
$\frac{\partial \mathcal{L}}{\partial \theta}$	Gradient der Verlustfunktion $\mathcal{L}$ in Bezug auf die Parameter θ
$\text{KL}(p \,\|\|\, q)$	KL-Divergenz zwischen den Verteilungen p und q

Inhaltsverzeichnis

1 Vortraining 1

1.1 Vortrainieren von NLP-Modellen . . . 2

1.1.1 Unüberwachtes, überwachtes und selbstüberwachtes Vortraining . . . 3

1.1.2 Anpassung vortrainierter Modelle . . . 4

1.2 Selbstüberwachte Pre-Training-Aufgaben . . . 8

1.2.1 Reines Dekodierer-Vortraining . . . 8

1.2.2 Reines Encoder-Vortraining . . . 10

1.2.3 Encoder-Decoder-Vortraining . . . 17

1.2.4 Vergleich von Vortrainingsaufgaben . . . 23

1.3 Beispiel: BERT . . . 24

1.3.1 Das Standardmodell . . . 25

1.3.2 Mehr Training und größere Modelle . . . 30

1.3.3 Effizientere Modelle . . . 31

1.3.4 Mehrsprachige Modelle . . . 32

1.4 Anwendung von BERT-Modellen . . . 35

1.5 Zusammenfassung . . . 40

2 Generative Modelle 42

2.1 Eine kurze Einführung in LLMs . . . 43

2.1.1 Reine Decoder-Transformer . . . 44

2.1.2 Training von LLMs . . . 47

2.1.3 Feinabstimmung von LLMs . . . 49

2.1.4 Angleichung von LLMs an die Welt . . . 54

2.1.5 Prompting von LLMs . . . 60

2.2 Training in großem Maßstab . . . 65

2.2.1 Datenaufbereitung . . . 65

2.2.2 Modellmodifikationen . . . 67

2.2.3 Verteiltes Training . . . 70

2.2.4 Skalierungsgesetze . . . 74

2.3 Modellierung langer Sequenzen . . . 77

2.3.1 Optimierung aus der Perspektive des Hochleistungsrechnens . . . 78

2.3.2 Effiziente Architekturen . . . 80

2.3.3 Cache und Speicher . . . 82

2.3.4 Teilung über Heads und Schichten hinweg . . . 92
2.3.5 Positionsextrapolation und -interpolation . . . 95
2.3.6 Bemerkungen . . . 106
2.4 Zusammenfassung . . . 109

3 Prompting 111
3.1 Allgemeines Prompt-Design . . . 112
3.1.1 Grundlagen . . . 112
3.1.2 In-Kontext-Lernen . . . 115
3.1.3 Strategien für das Prompt-Engineering . . . 117
3.1.4 Weitere Beispiele . . . 121
3.2 Fortgeschrittene Prompting-Methoden . . . 132
3.2.1 Gedankenkette . . . 132
3.2.2 Problemdekomposition . . . 136
3.2.3 Selbstverfeinerung . . . 144
3.2.4 Ensembling . . . 150
3.2.5 RAG und Werkzeugnutzung . . . 155
3.3 Prompt-Lernen . . . 160
3.3.1 Prompt-Optimierung . . . 161
3.3.2 Weiche Prompts . . . 165
3.3.3 Reduzierung der Prompt-Länge . . . 175
3.4 Zusammenfassung . . . 177

4 Alignment 179
4.1 Ein Überblick über die LLM-Abstimmung . . . 180
4.2 Anpassung an Anweisungen . . . 181
4.2.1 Überwachtes Fine-Tuning . . . 182
4.2.2 Erfassung von Feinabstimmungsdaten . . . 187
4.2.3 Feinabstimmung mit weniger Daten . . . 192
4.2.4 Instruktionsgeneralisierung . . . 194
4.2.5 Verwendung schwacher Modelle zur Verbesserung starker Modelle . . . 196
4.3 Anpassung an menschliche Präferenzen: RLHF . . . 200
4.3.1 Grundlagen des Verstärkungslernens . . . 201
4.3.2 Training von Belohnungsmodellen . . . 208
4.3.3 Training von LLMs . . . 211
4.4 Verbesserte Anpassung an menschliche Präferenzen . . . 216

4.4.1 Bessere Belohnungsmodellierung . . . 216
4.4.2 Direkte Präferenzoptimierung . . . 224
4.4.3 Automatische Generierung von Präferenzdaten . . . 228
4.4.4 Schrittweise Anpassung . . . 229
4.4.5 Anpassung zur Inferenzzeit . . . 232
4.5 Zusammenfassung . . . 234

5 Inferenz 236
5.1 Prefilling und Dekodierung . . . 237
5.1.1 Grundlagen . . . 237
5.1.2 Ein Zwei-Phasen-Framework . . . 242
5.1.3 Dekodierungsalgorithmen . . . 243
5.1.4 Evaluierungsmetriken für die LLM-Inferenz . . . 256
5.2 Effiziente Inferenztechniken . . . 258
5.2.1 Weiteres Caching . . . 258
5.2.2 Batching . . . 259
5.2.3 Parallelisierung . . . 269
5.2.4 Anmerkungen . . . 270
5.3 Skalierung zur Inferenzzeit . . . 272
5.3.1 Kontextskalierung . . . 273
5.3.2 Suchskalierung . . . 274
5.3.3 Output-Ensembling . . . 275
5.3.4 Erzeugen und Verifizieren von Denkpfaden . . . 276
5.4 Zusammenfassung . . . 285

Literaturverzeichnis 287

Kapitel 1

Vortraining

Die Entwicklung von neuronalen Sequenzmodellen, wie z. B. Transformern, hat zusammen mit den Verbesserungen im Bereich des groß angelegten selbst-überwachten Lernens die Tür zu universellem Sprachverständnis und universeller Sprachgenerierung geöffnet. Diese Errungenschaft ist maßgeblich durch das Vortraining motiviert: Wir trennen gemeinsame Komponenten aus vielen auf neuronalen Netzwerken basierenden Systemen und trainieren sie dann auf riesigen Mengen unmarkierter Daten mittels Selbst-Überwachung. Diese vortrainierten Modelle dienen als Grundlagenmodelle, die durch Feinabstimmung oder Prompting leicht an verschiedene Aufgaben angepasst werden können. Infolgedessen hat sich das Paradigma des NLP enorm verändert. In vielen Fällen ist ein groß angelegtes überwachtes Lernen für spezifische Aufgaben nicht mehr erforderlich, stattdessen müssen wir nur vortrainierte Grundlagenmodelle anpassen.

Obwohl das Vortraining in der jüngsten NLP-Forschung an Popularität gewonnen hat, reicht dieses Konzept Jahrzehnte bis in die Anfänge des Deep Learning zurück. Frühe Versuche, Deep-Learning-Systeme vorzutrainieren, umfassen beispielsweise unüberwachtes Lernen für RNNs, tiefe vorwärtsgekoppelte Netzwerke, Autoencoder und andere [Schmidhuber, 2015]. In der modernen Ära des Deep Learning erlebten wir eine Wiederbelebung des Vortrainings, teilweise verursacht durch das groß angelegte unüberwachte Lernen verschiedener Worteinbettungsmodelle [Mikolov et al., 2013b; Pennington et al., 2014]. Im gleichen Zeitraum erregte das Vortraining auch in der Computer Vision großes Interesse, wo die Backbone-Modelle auf relativ großen, markierten Datensätzen wie ImageNet trainiert und dann auf verschiedene nachgelagerte Aufgaben angewendet wurden [He et al., 2019; Zoph et al., 2020]. Die groß angelegte Forschung zum Vortraining im NLP begann mit der Entwicklung von Sprachmodellen unter Verwendung von selbst-überwachtem Lernen. Diese Modellfamilie umfasst mehrere bekannte Beispiele wie BERT [Devlin et al., 2019] und GPT [Brown et al., 2020], die alle auf der ähnlichen Idee beruhen, dass allgemeines Sprachverständnis und allgemeine Sprachgenerierung durch das Trainieren der Modelle zur Vorhersage maskierter Wörter in einer riesigen Textmenge erreicht werden können. Trotz der einfachen Natur dieses Ansatzes zeigen die resultierenden Modelle eine bemerkenswerte Fähigkeit zur Modellierung linguistischer Strukturen, obwohl sie nicht explizit darauf trainiert wurden, dies zu erreichen. Die Allgemeinheit der Vortrainingsaufgaben führt zu Systemen, die bei einer Vielzahl von NLP-Problemen eine starke Leistung aufweisen und sogar zuvor gut entwickelte überwachte Systeme übertreffen. In jüngerer Zeit haben vortrainierte große Sprachmodelle noch größere Erfolge erzielt und zeigen die aufregenden Aussichten für eine allgemeinere künstliche Intelligenz [Bubeck et al., 2023].

Dieses Kapitel behandelt das Konzept des Vortrainings im Kontext von NLP. Es beginnt mit einer allgemeinen Einführung in Vortrainingsmethoden und deren Anwendungen. Anschließend wird BERT als Beispiel verwendet, um zu veranschaulichen, wie ein Sequenzmodell über eine selbst-überwachte Aufgabe, die als maskierte Sprachmodellierung bezeichnet wird, trainiert wird. Darauf folgt eine Diskussion von Methoden zur Anpassung vortrainierter Sequenzmodelle für verschiedene NLP-Aufgaben. Beachten Sie, dass

wir uns in diesem Kapitel hauptsächlich auf das Paradigma des Vortrainings im NLP konzentrieren und daher nicht beabsichtigen, Details zu generativen großen Sprachmodellen zu behandeln. Eine detaillierte Diskussion dieser Modelle wird den nachfolgenden Kapiteln überlassen.

1.1 Vortrainieren von NLP-Modellen

Die Entwicklung von neuronalen Sequenzmodellen, wie zum Beispiel Transformers [Vaswani et al., 2017], hat zusammen mit den Fortschritten im Bereich des selbstüberwachten Lernens im großen Maßstab die Tür zu universellem Sprachverständnis und universeller Sprachgenerierung geöffnet. Diese Errungenschaft ist größtenteils durch das Pre-Training motiviert: Wir trennen gemeinsame Komponenten aus vielen auf neuronalen Netzen basierenden Systemen und trainieren sie dann mithilfe von Selbstüberwachung auf riesigen Mengen unmarkierter Daten. Diese vortrainierten Modelle dienen als Basismodelle (Foundation Models), die durch Feinabstimmung (Fine-Tuning) oder Prompting leicht an verschiedene Aufgaben angepasst werden können. Infolgedessen hat sich das Paradigma des NLP (Natural Language Processing) enorm verändert. In vielen Fällen ist ein umfangreiches überwachtes Lernen für spezifische Aufgaben nicht mehr erforderlich, stattdessen müssen wir nur vortrainierte Basismodelle anpassen.

$$\begin{aligned} \mathbf{o} &= g(x_0, x_1, ..., x_m; \theta) \\ &= g_\theta(x_0, x_1, ..., x_m) \end{aligned} \tag{1.1}$$

wobei $\{x_0, x_1, ..., x_m\}$ eine Sequenz von Eingabe-Token[1] bezeichnet, x_0 ein spezielles Symbol ($\langle s \rangle$ oder [CLS]) ist, das an den Anfang einer Sequenz angehängt wird, $g(\cdot; \theta)$ (auch als $g_\theta(\cdot)$ geschrieben) ein neuronales Netzwerk mit Parametern θ ist und $\mathbf{o}$ die Ausgabe des neuronalen Netzwerks darstellt. Unterschiedliche Probleme können je nach Form der Ausgabe $\mathbf{o}$ variieren. Zum Beispiel ist bei Token-Vorhersageproblemen (wie bei der Sprachmodellierung) $\mathbf{o}$ eine Verteilung über ein Vokabular; bei Sequenzkodierungsproblemen ist $\mathbf{o}$ eine Repräsentation der Eingabesequenz, oft ausgedrückt als eine reellwertige Vektorsequenz.

Hier gibt es zwei grundlegende Probleme.

- Optimierung von θ bei einer Vortrainingsaufgabe. Im Gegensatz zu herkömmlichen Lernproblemen in der NLP wird beim Vortraining nicht von spezifischen nachgelagerten Aufgaben ausgegangen, auf die das Modell angewendet werden soll. Stattdessen besteht das Ziel darin, ein Modell zu trainieren, das auf eine Vielzahl von Aufgaben generalisieren kann.

- Anwendung des vortrainierten Modells $g_{\hat{\theta}}(\cdot)$ auf nachgelagerte Aufgaben. Um das Modell an diese Aufgaben anzupassen, müssen die Parameter $\hat{\theta}$ unter Verwendung von gelabelten Daten geringfügig angepasst oder das Modell mit Aufgabenbeschreibungen instruiert werden.

[1] Hier gehen wir davon aus, dass Token grundlegende Texteinheiten sind, die durch Tokenisierung getrennt werden. Manchmal werden wir die Begriffe Token und Wort austauschbar verwenden, obwohl sie im NLP eng verwandte, aber leicht unterschiedliche Bedeutungen haben.

In diesem Abschnitt erörtern wir die grundlegenden Ansätze zur Behandlung dieser Probleme.

1.1.1 Unüberwachtes, überwachtes und selbstüberwachtes Vortraining

Im tiefen Lernen bezieht sich das Vortraining auf den Prozess der Optimierung eines neuronalen Netzwerks, bevor es weiter trainiert/feinabgestimmt und auf die interessierenden Aufgaben angewendet wird. Dieser Ansatz basiert auf der Annahme, dass ein auf einer Aufgabe vortrainiertes Modell angepasst werden kann, um eine andere Aufgabe auszuführen. Daher müssen wir kein tiefes, komplexes neuronales Netzwerk von Grund auf für Aufgaben mit begrenzten gelabelten Daten trainieren. Stattdessen können wir Aufgaben nutzen, bei denen Überwachungssignale leichter zu erhalten sind. Dies verringert die Abhängigkeit von aufgabenspezifischen gelabelten Daten und ermöglicht die Entwicklung allgemeinerer Modelle, die nicht auf bestimmte Probleme beschränkt sind.

Während des Wiederauflebens neuronaler Netzwerke durch das tiefe Lernen konzentrierten sich viele frühe Versuche, ein Vortraining zu erreichen, auf das unüberwachtes Lernen. Bei diesen Methoden werden die Parameter eines neuronalen Netzwerks unter Verwendung eines Kriteriums optimiert, das nicht direkt mit bestimmten Aufgaben zusammenhängt. Zum Beispiel können wir die Rekonstruktions-Kreuzentropie des Eingabevektors für jede Schicht minimieren [Bengio et al., 2006]. Unüberwachtes Vortraining wird üblicherweise als vorbereitender Schritt vor dem überwachten Lernen eingesetzt und bietet mehrere Vorteile, wie z. B. die Unterstützung bei der Entdeckung besserer lokaler Minima und das Hinzufügen eines Regularisierungseffekts zum Trainingsprozess [Erhan et al., 2010]. Diese Vorteile machen die nachfolgende Phase des überwachten Lernens einfacher und stabiler.

Ein zweiter Ansatz zum Vortraining besteht darin, ein neuronales Netzwerk auf Aufgaben des überwachtes Lernen vorzutrainieren. Betrachten wir zum Beispiel ein Sequenzmodell, das dazu dient, Eingabesequenzen in bestimmte Repräsentationen zu kodieren. Im Vortraining wird dieses Modell mit einer Klassifikationsschicht kombiniert, um ein Klassifikationssystem zu bilden. Dieses System wird dann auf einer Vortrainingsaufgabe trainiert, wie z. B. der Klassifizierung von Sätzen nach ihrer Stimmung (z. B. der Bestimmung, ob ein Satz eine positive oder negative Stimmung vermittelt). Anschließend passen wir das Sequenzmodell an eine nachgelagerte Aufgabe an. Wir erstellen ein neues Klassifikationssystem, das auf diesem vortrainierten Sequenzmodell und einer neuen Klassifikationsschicht basiert (z. B. zur Bestimmung, ob eine Sequenz subjektiv oder objektiv ist). Typischerweise müssen wir die Parameter des neuen Modells mithilfe aufgabenspezifischer gelabelter Daten feinabstimmen, um sicherzustellen, dass das Modell optimal angepasst ist, um bei dieser neuen Art von Daten gute Leistungen zu erbringen. Das feinabgestimmte Modell wird dann zur Klassifizierung neuer Sequenzen für diese Aufgabe eingesetzt. Ein Vorteil des überwachten Vortrainings besteht darin, dass der Trainingsprozess, sei es in der Vortrainings- oder Feinabstimmungsphase, unkompliziert ist, da er dem gut untersuchten allgemeinen Paradigma des überwachten Lernens im maschinellen Lernen folgt. Mit zunehmender Komplexität des neuronalen Netzwerks wächst jedoch auch der Bedarf an mehr gelabelten Daten. Dies wiederum erschwert die Vortrainingsaufgabe, insbesondere wenn keine großen Mengen an gelabelten Daten verfügbar sind.

Ein dritter Ansatz zum Vortraining ist das selbstüberwachtes Lernen. Bei diesem Ansatz wird ein neuronales Netzwerk mithilfe von Überwachungssignalen trainiert, die es selbst erzeugt, anstatt von Menschen bereitgestellter. Dies geschieht im Allgemeinen, indem eigene Trainingsaufgaben direkt aus ungelabelten Daten konstruiert werden, beispielsweise indem das System Pseudo-Label erstellt. Obwohl das selbstüberwachte Lernen in letzter Zeit zu einer sehr beliebten Methode in der NLP geworden ist, ist es kein neues Konzept. Im maschinellen Lernen ist ein verwandtes Konzept das Selbsttraining, bei dem ein Modell iterativ verbessert wird, indem es von den einem Datensatz zugewiesenen Pseudo-Labeln lernt. Dazu benötigen wir einige Startdaten, um ein initiales Modell zu erstellen. Dieses Modell erzeugt dann Pseudo-Label für ungelabelte Daten, und diese Pseudo-Label werden anschließend verwendet, um das Modell selbst iterativ zu verfeinern und zu bootstrappen. Eine solche Methode wurde in mehreren NLP-Bereichen erfolgreich eingesetzt, wie z. B. bei der Wortsinndisambiguierung [Yarowsky, 1995] und der Dokumentenklassifikation [Blum and Mitchell, 1998]. Im Gegensatz zur Standardmethode des Selbsttrainings ist das selbstüberwachte Vortraining in der NLP nicht auf ein initiales Modell zur Annotation der Daten angewiesen. Stattdessen werden alle Überwachungssignale aus dem Text erstellt, und das gesamte Modell wird von Grund auf trainiert. Ein bekanntes Beispiel hierfür ist das Trainieren von Sequenzmodellen, indem sukzessive ein maskiertes Wort anhand seiner vorangehenden oder umgebenden Wörter in einem Text vorhergesagt wird. Dies ermöglicht ein groß angelegtes selbstüberwachtes Lernen für tiefe neuronale Netzwerke, was zum Erfolg des Vortrainings bei vielen Aufgaben des Verstehens, Schreibens und Schlussfolgerns führt.

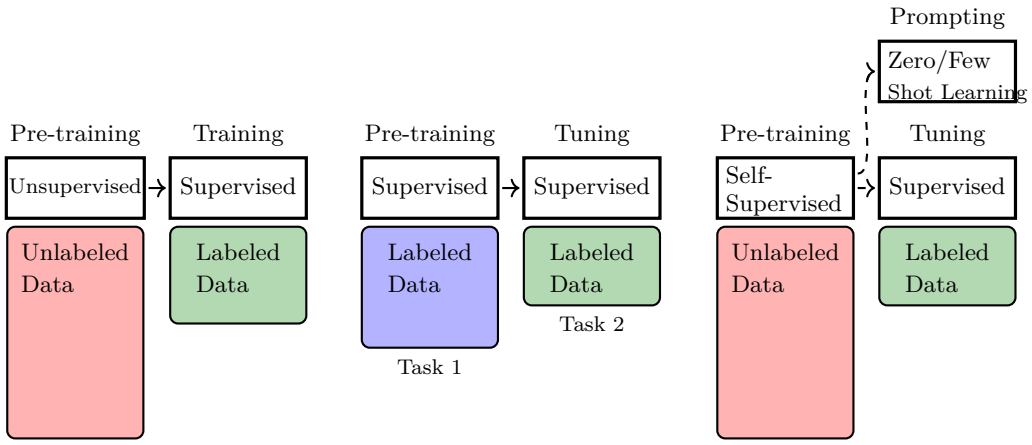
Abbildung 1.1 zeigt einen Vergleich der oben genannten drei Vortrainingsansätze. Das selbstüberwachte Vortraining ist so erfolgreich, dass die meisten aktuellen NLP-Modelle auf dem Stand der Technik auf diesem Paradigma basieren. Daher werden wir uns in diesem Kapitel und im gesamten Buch auf das selbstüberwachte Vortraining konzentrieren. Wir werden zeigen, wie Sequenzmodelle durch Selbstüberwachung vortrainiert und wie die vortrainierten Modelle angewendet werden.

1.1.2 Anpassung vortrainierter Modelle

Wie bereits erwähnt, werden zwei Haupttypen von Modellen für das NLP-Vortraining häufig verwendet.

- Sequence Encoding Models. Gegeben eine Sequenz von Wörtern oder Tokens, stellt ein Sequenzkodierungsmodell diese Sequenz entweder als reellwertigen Vektor oder als eine Sequenz von Vektoren dar und erhält so eine Repräsentation der Sequenz. Diese Repräsentation wird typischerweise als Eingabe für ein anderes Modell verwendet, wie zum Beispiel ein System zur Satzklassifizierung.

- Sequence Generation Models. In der NLP bezieht sich die Sequenzgenerierung im Allgemeinen auf das Problem, eine Sequenz von Tokens basierend auf einem gegebenen Kontext zu erzeugen. Der Begriff context hat je nach Anwendung unterschiedliche Bedeutungen. Zum Beispiel bezieht er sich in der Sprachmodellierung auf die vorhergehenden Tokens und in der maschinellen Übersetzung auf die quellsprachliche



(a) Unsupervised Pre-training (b) Supervised Pre-training (c) Self-supervised Pre-training

Abbildung 1.1: Darstellung von unüberwachtem, überwachtem und selbstüberwachtem Vortraining. Beim unüberwachten Vortraining wird das Training auf großen Mengen unmarkierter Daten durchgeführt. Es kann als ein vorbereitender Schritt angesehen werden, um einen guten Ausgangspunkt für den nachfolgenden Optimierungsprozess zu schaffen, obwohl nach dem Vortraining noch erheblicher Aufwand erforderlich ist, um das Modell mit markierten Daten weiter zu trainieren. Beim überwachten Vortraining ist die zugrunde liegende Annahme, dass verschiedene (überwachte) Lernaufgaben miteinander in Beziehung stehen. So kann das Modell zunächst auf einer Aufgabe trainiert und das resultierende Modell mit einigem Trainings- oder Abstimmungsaufwand auf eine andere Aufgabe übertragen werden. Beim selbstüberwachten Vortraining wird ein Modell auf großen Mengen unmarkierter Daten durch Selbstüberwachung vortrainiert. Das Modell kann auf diese Weise gut trainiert werden, und wir können es durch Feinabstimmung oder Prompting effizient an neue Aufgaben anpassen.

Sequenz[2].

Wir benötigen unterschiedliche Techniken, um diese Modelle nach dem Vortraining auf nachgelagerte Aufgaben anzuwenden. Hier interessieren wir uns für die folgenden zwei Methoden.

1.1.2.1 Feinabstimmung vortrainierter Modelle

Für das Vortraining der Sequenzkodierung ist die Feinabstimmung eine gängige Methode zur Anpassung vortrainierter Modelle. Sei $\text{Encode}_\theta(\cdot)$ ein Encoder mit den Parametern θ, zum Beispiel kann $\text{Encode}_\theta(\cdot)$ ein standardmäßiger Transformer-Encoder sein. Vorausgesetzt, wir haben dieses Modell auf irgendeine Weise vortrainiert und die optimalen Parameter $\hat{\theta}$ erhalten, können wir es verwenden, um jede Sequenz zu modellieren und die entsprechende Repräsentation zu erzeugen, wie hier gezeigt:

$$\mathbf{H} = \text{Encode}_{\hat{\theta}}(\mathbf{x}) \tag{1.2}$$

wobei $\mathbf{x}$ die Eingabesequenz $\{x_0, x_1, ..., x_m\}$ und $\mathbf{H}$ die Ausgaberepräsentation ist, die eine Sequenz von reellwertigen Vektoren $\{\mathbf{h}_0, \mathbf{h}_1, ..., \mathbf{h}_m\}$ ist. Da der Encoder nicht als eigenständiges NLP-System funktioniert, wird er oft als Komponente in ein größeres System

[2]Genauer gesagt wird bei der autoregressiven Dekodierung der maschinellen Übersetzung jedes zielsprachliche Token sowohl auf der Grundlage seiner vorhergehenden Tokens als auch der quellsprachlichen Sequenz generiert.

integriert. Betrachten wir zum Beispiel ein Textklassifikationsproblem, bei dem wir die Polarität (d. h. positiv, negativ und neutral) eines gegebenen Textes identifizieren. Wir können ein Textklassifikationssystem erstellen, indem wir einen Klassifikator auf den Encoder aufsetzen. Sei $\text{Classify}_{\omega}(\cdot)$ ein neuronales Netz mit den Parametern ω. Dann kann das Textklassifikationsmodell in der folgenden Form ausgedrückt werden:

$$\begin{aligned} \text{Pr}_{\omega,\hat{\theta}}(\cdot|\mathbf{x}) &= \text{Classify}_{\omega}(\mathbf{H}) \\ &= \text{Classify}_{\omega}(\text{Encode}_{\hat{\theta}}(\mathbf{x})) \end{aligned} \tag{1.3}$$

Hier ist $\text{Pr}_{\omega,\hat{\theta}}(\cdot|\mathbf{x})$ eine Wahrscheinlichkeitsverteilung über die Label-Menge $\{\text{positiv}, \text{negativ}, \text{neutral}\}$, und das Label mit der höchsten Wahrscheinlichkeit in dieser Verteilung wird als Ausgabe ausgewählt. Um die Notation übersichtlich zu halten, verwenden wir $F_{\omega,\hat{\theta}}(\cdot)$, um $\text{Classify}_{\omega}(\text{Encode}_{\hat{\theta}}(\cdot))$ zu bezeichnen.

Da die Modellparameter ω und $\hat{\theta}$ nicht für die Klassifikationsaufgabe optimiert sind, können wir dieses Modell nicht direkt verwenden. Stattdessen müssen wir eine modifizierte Version des Modells verwenden, die an die Aufgabe angepasst ist. Ein typischer Weg ist die Feinabstimmung des Modells durch explizite Kennzeichnung in nachgelagerten Aufgaben. Wir können $F_{\omega,\hat{\theta}}(\cdot)$ auf einem gelabelten Datensatz trainieren und es als eine übliche Aufgabe des überwachten Lernens behandeln. Das Ergebnis der Feinabstimmung sind die Parameter $\tilde{\omega}$ und $\tilde{\theta}$, die weiter optimiert werden. Alternativ können wir die Encoder-Parameter $\hat{\theta}$ einfrieren, um ihren vortrainierten Zustand beizubehalten, und uns ausschließlich auf die Optimierung von ω konzentrieren. Dies ermöglicht eine effiziente Anpassung des Klassifikators, um mit dem vortrainierten Encoder zusammenzuarbeiten.

Sobald wir ein feinabgestimmtes Modell erhalten haben, können wir es zur Klassifizierung eines neuen Textes verwenden. Nehmen wir zum Beispiel an, wir haben einen Kommentar, der auf einer Reise-Website veröffentlicht wurde:

I love the food here. It's amazing!

Zuerst tokenisieren wir diesen Text in Tokens[3], und speisen dann die Token-Sequenz $\mathbf{x}_{\text{new}}$ in das feinabgestimmte Modell $F_{\tilde{\omega},\tilde{\theta}}(\cdot)$ ein. Das Modell erzeugt eine Verteilung über die Klassen durch

$$F_{\tilde{\omega},\tilde{\theta}}(\mathbf{x}_{\text{neu}}) = \begin{bmatrix} \text{Pr}(\text{positiv}|\mathbf{x}_{\text{neu}}) & \text{Pr}(\text{negativ}|\mathbf{x}_{\text{neu}}) & \text{Pr}(\text{neutral}|\mathbf{x}_{\text{neu}}) \end{bmatrix} \tag{1.4}$$

Und wir wählen das Label des Eintrags mit dem Maximalwert als Ausgabe. In diesem Beispiel ist es positiv.

Im Allgemeinen ist die Menge der für die Feinabstimmung verwendeten gelabelten Daten im Vergleich zu den Vortrainingsdaten gering, sodass die Feinabstimmung weniger rechenintensiv ist. Dies macht die Anpassung vortrainierter Modelle in der Praxis sehr effizient: Gegeben ein vortrainiertes Modell und eine nachgelagerte Aufgabe, müssen wir nur einige gelabelte Daten sammeln und die Modellparameter auf diesen Daten leicht anpassen. Eine detailliertere Diskussion der Feinabstimmung findet sich in Abschnitt 1.4.

[3] Der Text kann auf viele verschiedene Arten tokenisiert werden. Eine der einfachsten ist die Segmentierung des Textes in englische Wörter und Satzzeichen {I, love, the, food, here, ., It, 's, amazing, !}

1.1.2.2 Prompting von vortrainierten Modellen

Im Gegensatz zu Sequenzkodierungsmodellen werden Sequenzgenerierungsmodelle oft unabhängig eingesetzt, um Sprachgenerierungsaufgaben wie Fragebeantwortung und maschinelle Übersetzung zu lösen, ohne dass zusätzliche Module erforderlich sind. Es ist daher unkompliziert, diese Modelle als vollständige Systeme für nachgelagerte Aufgaben feinabzustimmen. Zum Beispiel können wir ein vortrainiertes mehrsprachiges Encoder-Decoder-Modell mit einigen zweisprachigen Daten feinabstimmen, um seine Leistung bei einer spezifischen Übersetzungsaufgabe zu verbessern.

Unter den verschiedenen Sequenzgenerierungsmodellen sind die auf sehr großen Datenmengen trainierten großen Sprachmodelle ein bemerkenswertes Beispiel. Diese Sprachmodelle werden trainiert, um einfach das nächste Token basierend auf den vorhergehenden Tokens vorherzusagen. Obwohl die Vorhersage von Tokens eine so einfache Aufgabe ist, dass sie lange Zeit nur auf die „Sprachmodellierung“beschränkt war, hat sich gezeigt, dass durch die sehr häufige Wiederholung dieser Aufgabe das Erlernen allgemeinen Wissens über Sprachen ermöglicht wird. Das Ergebnis ist, dass die vortrainierten großen Sprachmodelle bemerkenswert gute Fähigkeiten bei der Vorhersage von Tokens aufweisen, was es ermöglicht, zahlreiche NLP-Probleme durch Prompting der großen Sprachmodelle in einfache Textgenerierungsaufgaben umzuwandeln. Zum Beispiel können wir das oben genannte Textklassifikationsproblem als Textgenerierungsaufgabe formulieren

I love the food here. It’s amazing! I’m _____

Hier zeigt __ das Wort oder die Phrase an, die wir vorhersagen möchten (nennen wir es die completion). Wenn das vorhergesagte Wort glücklich, oder froh, oder zufrieden oder ein verwandtes positives Wort ist, können wir den Text als positive klassifizieren. Dieses Beispiel zeigt eine einfache Prompting-Methode, bei der wir den Eingabetext mit Ich bin verketten, um einen Prompt zu bilden. Die Vervollständigung hilft dann zu entscheiden, welches Label dem ursprünglichen Text zugewiesen wird.

Angesichts der starken Leistung beim Sprachverständnis und der Sprachgenerierung von großen Sprachmodellen kann ein Prompt die Modelle anweisen, komplexere Aufgaben auszuführen. Hier ist ein Prompt, mit dem wir das LLM anweisen, eine Polaritätsklassifikation mit einer Anweisung durchzuführen.

> Assume that the polarity of a text is a label chosen from {positive, negative, neutral}. Identify the polarity of the input.
>
> Input: I love the food here. It’s amazing!
>
> Polarity: _____

Die ersten beiden Sätze sind eine Beschreibung der Aufgabe. Input und Polarity sind Indikatoren für die Eingabe bzw. Ausgabe. Wir erwarten, dass das Modell den Text vervollständigt und gleichzeitig das korrekte Polaritätslabel angibt. Durch die Verwendung von anweisungsbasierten Prompts können wir große Sprachmodelle anpassen, um NLP-Probleme ohne zusätzliches Training zu lösen.

Dieses Beispiel demonstriert auch die Fähigkeit des Zero-Shot-Lernens von großen Sprachmodellen, die Aufgaben ausführen können, die während der Trainingsphase nicht beobachtet wurden. Eine weitere Methode, um neue Fähigkeiten in einem neuronalen Netzwerk zu ermöglichen, ist das Few-Shot-Lernen. Dies wird typischerweise durch in-context learning (ICT) erreicht. Genauer gesagt, fügen wir einige Beispiele hinzu, die zeigen, wie eine Eingabe einer Ausgabe entspricht. Diese Beispiele, bekannt als demonstrations, werden verwendet, um großen Sprachmodellen beizubringen, wie sie die Aufgabe ausführen sollen. Unten folgt ein Beispiel mit Demonstrationen

> Assume that the polarity of a text is a label chosen from {positive, negative, neutral}. Identify the polarity of the input.
>
> Input: The traffic is terrible during rush hours, making it difficult to reach the airport on time.
>
> Polarity: Negative
>
> Input: The weather here is wonderful.
>
> Polarity: Positive
>
> Input: I love the food here. It's amazing!
>
> Polarity: _____

Prompting und In-Context-Lernen spielen eine wichtige Rolle beim jüngsten Aufstieg großer Sprachmodelle. Wir werden diese Themen in Kapitel 3ausführlicher behandeln. Es ist jedoch erwähnenswert, dass, obwohl Prompting eine leistungsstarke Methode zur Anpassung großer Sprachmodelle ist, dennoch ein gewisser Abstimmungsaufwand erforderlich ist, um sicherzustellen, dass die Modelle Anweisungen genau befolgen können. Zusätzlich ist der Feinabstimmungsprozess entscheidend, um die Werte dieser Modelle mit menschlichen Werten in Einklang zu bringen. Ausführlichere Diskussionen zur Feinabstimmung finden sich in Kapitel 4.

1.2 Selbstüberwachte Pre-Training-Aufgaben

In diesem Abschnitt betrachten wir selbstüberwachte Pre-Training-Ansätze für verschiedene neuronale Architekturen, einschließlich reiner Decoder-, reiner Encoder- und Encoder-Decoder-Architekturen. Wir beschränken unsere Diskussion auf Transformer , da sie die Grundlage für die meisten vortrainierten Modelle im NLP bilden. Jedoch ist Pre-Training ein breites Konzept, und daher geben wir nur eine kurze Einführung in die grundlegenden Ansätze, um diesen Abschnitt kurz zu halten.

1.2.1 Reines Dekodierer-Vortraining

Die reine Dekodierer-Architektur wurde weithin bei der Entwicklung von Sprachmodellen verwendet [Radford et al., 2018]. Zum Beispiel können wir einen Transformer-Dekodierer

als Sprachmodell verwenden, indem wir einfach die Kreuz-Aufmerksamkeits-Sublayer daraus entfernen. Ein solches Modell sagt die Verteilung der Token an einer Position voraus, gegeben seine vorangehenden Token, und die Ausgabe ist das Token mit der maximalen Wahrscheinlichkeit. Die Standardmethode zum Trainieren dieses Modells, wie beim Sprachmodellierungsproblem, besteht darin, eine Verlustfunktion über eine Sammlung von Token-Sequenzen zu minimieren. Sei $\text{Decoder}_\theta(\cdot)$ ein Dekodierer mit den Parametern θ. An jeder Position i erzeugt der Dekodierer eine Verteilung der nächsten Token basierend auf seinen vorangehenden Token $\{x_0, ..., x_i\}$, bezeichnet als $\Pr_\theta(\cdot|x_0, ..., x_i)$ (oder kurz $\mathbf{p}^\theta_{i+1}$). Angenommen, wir haben die Goldstandard-Verteilung an derselben Position, bezeichnet als $\mathbf{p}^{\text{gold}}_{i+1}$. Für die Sprachmodellierung können wir uns $\mathbf{p}^{\text{gold}}_{i+1}$ als eine One-Hot-Repräsentation des korrekten vorhergesagten Wortes vorstellen. Wir definieren dann eine Verlustfunktion $\mathcal{L}(\mathbf{p}^\theta_{i+1}, \mathbf{p}^{\text{gold}}_{i+1})$, um den Unterschied zwischen der Modellvorhersage und der wahren Vorhersage zu messen. Im NLP wird typischerweise der logarithmische Kreuzentropie-Verlust verwendet.

Gegeben eine Sequenz von m Token $\{x_0, ..., x_m\}$, ist der Verlust für diese Sequenz die Summe der Verluste über die Positionen $\{0, ..., m-1\}$, gegeben durch

$$\begin{aligned} \text{Loss}_\theta(x_0, ..., x_m) &= \sum_{i=0}^{m-1} \mathcal{L}(\mathbf{p}^\theta_{i+1}, \mathbf{p}^{\text{gold}}_{i+1}) \\ &= \sum_{i=0}^{m-1} \text{LogCrossEntropy}(\mathbf{p}^\theta_{i+1}, \mathbf{p}^{\text{gold}}_{i+1}) \end{aligned} \tag{1.5}$$

wobei $\text{LogCrossEntropy}(\cdot)$ die logarithmische Kreuzentropie ist und $\mathbf{p}^{\text{gold}}_{i+1}$ die One-Hot-Repräsentation von x_{i+1} ist.

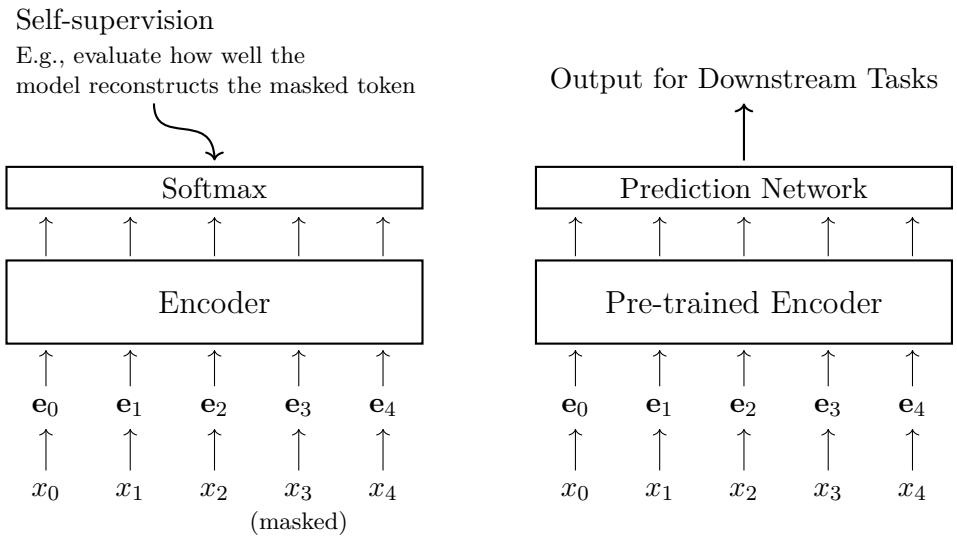
Diese Verlustfunktion kann auf eine Menge von Sequenzen $\mathcal{D}$ erweitert werden. In diesem Fall besteht das Ziel des Vortrainings darin, die besten Parameter zu finden, die den Verlust auf $\mathcal{D}$ minimieren

$$\hat{\theta} = \arg\min_\theta \sum_{\mathbf{x} \in \mathcal{D}} \text{Loss}_\theta(\mathbf{x}) \tag{1.6}$$

Beachten Sie, dass dieses Ziel mathematisch äquivalent zur Maximum-Likelihood-Schätzung ist und wie folgt umformuliert werden kann

$$\begin{aligned} \hat{\theta} &= \arg\max_\theta \sum_{\mathbf{x} \in \mathcal{D}} \log \Pr_\theta(\mathbf{x}) \\ &= \arg\max_\theta \sum_{\mathbf{x} \in \mathcal{D}} \sum_{i=0}^{i-1} \log \Pr_\theta(x_{i+1}|x_0, ..., x_i) \end{aligned} \tag{1.7}$$

Mit diesen optimierten Parametern $\hat{\theta}$ können wir das vortrainierte Sprachmodell $\text{Decoder}_{\hat{\theta}}(\cdot)$ verwenden, um die Wahrscheinlichkeit $\Pr_{\hat{\theta}}(x_{i+1}|x_0, ..., x_i)$ an jeder Position einer gegebenen Sequenz zu berechnen.



(a) Pre-training (b) Applying the Pre-trained Encoder

Abbildung 1.2: Vortraining eines Transformer-Encoders (links) und anschließende Anwendung des vortrainierten Encoders (rechts). In der Vortrainingsphase wird der Encoder zusammen mit einer Softmax-Schicht mittels Selbstüberwachung trainiert. In der Anwendungsphase wird die Softmax-Schicht entfernt und der vortrainierte Encoder mit einem Prädiktionsnetzwerk kombiniert, um spezifische Probleme zu lösen. In der Regel wird das System zur besseren Anpassung an diese Aufgaben mithilfe von gelabelten Daten feinabgestimmt.

1.2.2 Reines Encoder-Vortraining

Wie in Abschnitt 1.1.2.1 definiert, ist ein Encoder $\text{Encoder}_\theta(\cdot)$ eine Funktion, die eine Sequenz von Token $\mathbf{x} = x_0 ... x_m$ liest und eine Sequenz von Vektoren $\mathbf{H} = \mathbf{h}_0 ... \mathbf{h}_m$ erzeugt[4]. Das Training dieses Modells ist nicht trivial, da wir keine Gold-Standard-Daten haben, um zu messen, wie gut die Ausgabe der reellwertigen Funktion ist. Ein typischer Ansatz für das Encoder-Vortraining besteht darin, den Encoder mit einigen Ausgabeschichten zu kombinieren, um Supervisionssignale zu erhalten, die leichter zu beschaffen sind. Abbildung 1.2 zeigt eine gängige Architektur für das Vortraining von Transformer -Encodern, bei der wir eine Softmax-Schicht über den Transformer -Encoder legen. Offensichtlich ist diese Architektur dieselbe wie die eines dekodiererbasierten Sprachmodells, und die Ausgabe ist eine Sequenz von Wahrscheinlichkeitsverteilungen

$$\begin{bmatrix} \mathbf{p}_1^{\mathbf{W},\theta} \\ \vdots \\ \mathbf{p}_m^{\mathbf{W},\theta} \end{bmatrix} = \text{Softmax}_{\mathbf{W}}(\text{Encoder}_\theta(\mathbf{x})) \tag{1.9}$$

Hier ist $\mathbf{p}_i^{\mathbf{W},\theta}$ die Ausgabeverteilung $\Pr(\cdot|\mathbf{x})$ an Position i. Wir verwenden $\text{Softmax}_{\mathbf{W}}(\cdot)$,

[4] Wenn wir $\mathbf{h}_i$ als Zeilenvektor betrachten, kann $\mathbf{H}$ als

$$\mathbf{H} = \begin{bmatrix} \mathbf{h}_0 \\ \vdots \\ \mathbf{h}_m \end{bmatrix} \tag{1.8}$$

geschrieben werden

um zu kennzeichnen, dass die Softmax-Schicht durch $\mathbf{W}$ parametrisiert ist, d.h. $\text{Softmax}_{\mathbf{W}}(\mathbf{H}) = \text{Softmax}(\mathbf{H} \cdot \mathbf{W})$. Zur Vereinfachung der Notation werden wir gelegentlich die hochgestellten Indizes $\mathbf{W}$ und θ bei jeder Wahrscheinlichkeitsverteilung weglassen.

Der Unterschied zwischen diesem Modell und Standard-Sprachmodellen besteht darin, dass die Ausgabe $\mathbf{p}_i$ beim Encoder-Vortraining und bei der Sprachmodellierung unterschiedliche Bedeutungen hat. Bei der Sprachmodellierung ist $\mathbf{p}_i$ die Wahrscheinlichkeitsverteilung für die Vorhersage des nächsten Wortes. Dies folgt einem autoregressiven Dekodierungsprozess: Ein Sprachmodell beobachtet nur die Wörter bis zur Position i und sagt das nächste voraus. Im Gegensatz dazu kann beim Encoder-Vortraining die gesamte Sequenz auf einmal beobachtet werden, und daher ist es nicht sinnvoll, eines der Token in dieser Sequenz vorherzusagen.

1.2.2.1 Maskierte Sprachmodellierung

Eine der populärsten Methoden des Encoder-Vortrainings ist die maskierte Sprachmodellierung, die die Grundlage des bekannten BERT-Modells bildet [Devlin et al., 2019]. Die Idee der maskierten Sprachmodellierung besteht darin, Vorhersageaufgaben zu erstellen, indem einige der Tokens in der Eingabesequenz maskiert und ein Modell trainiert wird, um die maskierten Tokens vorherzusagen. In diesem Sinne ist das konventionelle Sprachmodellierungsproblem, das manchmal als kausale Sprachmodellierung bezeichnet wird, ein Spezialfall der maskierten Sprachmodellierung: An jeder Position maskieren wir die Tokens im rechten Kontext und sagen das Token an dieser Position unter Verwendung seines linken Kontexts voraus. Bei der kausalen Sprachmodellierung verwenden wir jedoch nur den linken Kontext für die Wortvorhersage, während die Vorhersage von Tokens im rechten Kontext abhängen kann. Im Gegensatz dazu werden bei der maskierten Sprachmodellierung alle nicht-maskierten Tokens für die Wortvorhersage verwendet, was zu einem bidirektionalen Modell führt, das Vorhersagen auf der Grundlage sowohl des linken als auch des rechten Kontexts trifft.

Formaler ausgedrückt, für eine Eingabesequenz $\mathbf{x} = x_0 ... x_m$, nehmen wir an, dass wir die Tokens an den Positionen $\mathcal{A}(\mathbf{x}) = \{i_1, ..., i_u\}$ maskieren. Dadurch erhalten wir eine maskierte Tokensequenz $\bar{\mathbf{x}}$, bei der das Token an jeder Position in $\mathcal{A}(\mathbf{x})$ durch ein spezielles Symbol [MASK] ersetzt wird. Zum Beispiel für die folgende Sequenz

The early bird catches the worm

könnten wir eine maskierte Tokensequenz wie diese haben

The [MASK] bird catches the [MASK]

wobei wir die Tokens early und worm maskieren (d.h. $i_1 = 2$ und $i_2 = 6$).

Nun haben wir zwei Sequenzen $\mathbf{x}$ und $\bar{\mathbf{x}}$. Das Modell wird dann so optimiert, dass wir $\mathbf{x}$ auf der Grundlage von $\bar{\mathbf{x}}$ korrekt vorhersagen können. Dies kann als ein autoencoderähnlicher Prozess betrachtet werden, und das Trainingsziel ist die Maximierung der Rekonstruktionswahrscheinlichkeit $\Pr(\mathbf{x}|\bar{\mathbf{x}})$. Beachten Sie, dass es eine einfache positionsweise

Ausrichtung zwischen $\mathbf{x}$ und $\bar{\mathbf{x}}$ gibt. Da ein nicht-maskiertes Token in $\bar{\mathbf{x}}$ mit dem Token in $\mathbf{x}$ an der gleichen Position identisch ist, muss die Vorhersage für dieses nicht-maskierte Token nicht berücksichtigt werden. Dies führt zu einem vereinfachten Trainingsziel, das nur die Wahrscheinlichkeiten für maskierte Tokens maximiert. Wir können dieses Ziel im Stil einer Maximum-Likelihood-Schätzung ausdrücken

$$(\widehat{\mathbf{W}}, \hat{\theta}) \quad = \quad \arg\max_{\mathbf{W},\theta} \sum_{\mathbf{x}\in\mathcal{D}} \sum_{i\in\mathcal{A}(\mathbf{x})} \log \Pr_i^{\mathbf{W},\theta}(x_i|\bar{\mathbf{x}}) \tag{1.10}$$

oder es alternativ mit dem Kreuzentropie-Verlust ausdrücken

$$(\widehat{\mathbf{W}}, \hat{\theta}) \quad = \quad \arg\min_{\mathbf{W},\theta} \sum_{\mathbf{x}\in\mathcal{D}} \sum_{i\in\mathcal{A}(\mathbf{x})} \text{LogCrossEntropy}(\mathbf{p}_i^{\mathbf{W},\theta}, \mathbf{p}_i^{\text{gold}}) \tag{1.11}$$

wobei $\Pr_k^{\mathbf{W},\theta}(x_k|\bar{\mathbf{x}})$ die Wahrscheinlichkeit des wahren Tokens x_k an Position k bei gegebener beschädigter Eingabe $\bar{\mathbf{x}}$ ist und $\mathbf{p}_k^{\mathbf{W},\theta}$ die Wahrscheinlichkeitsverteilung an Position k bei gegebener beschädigter Eingabe $\bar{\mathbf{x}}$ ist. Zur Veranschaulichung betrachten wir das obige Beispiel, in dem zwei Tokens der Sequenz „the early bird catches the worm“maskiert sind. Für dieses Beispiel ist das Ziel, die Summe der logarithmierten Wahrscheinlichkeiten zu maximieren

$$\begin{aligned} \text{Loss} \quad = \quad & \log \Pr(x_2 = \text{early}|\bar{\text{x}} = \text{[CLS] The } \underbrace{\text{[MASK]}}_{\bar{x}_2} \text{ bird catches the } \underbrace{\text{[MASK]}}_{\bar{x}_6}) + \\ & \log \Pr(x_6 = \text{worm}|\bar{\text{x}} = \text{[CLS] The } \underbrace{\text{[MASK]}}_{\bar{x}_2} \text{ bird catches the } \underbrace{\text{[MASK]}}_{\bar{x}_6}) \end{aligned} \tag{1.12}$$

Sobald wir die optimierten Parameter $\widehat{\mathbf{W}}$ und $\hat{\theta}$ erhalten, können wir $\widehat{\mathbf{W}}$ verwerfen. Dann können wir den vortrainierten Encoder $\text{Encoder}_{\hat{\theta}}(\cdot)$ weiter feinabstimmen oder ihn direkt auf nachgelagerte Aufgaben anwenden.

1.2.2.2 Permutierte Sprachmodellierung

Obwohl die maskierte Sprachmodellierung einfach und weit verbreitet ist, führt sie neue Probleme ein. Ein Nachteil ist die Verwendung eines speziellen Tokens, [MASK], das nur während des Trainings, aber nicht zur Testzeit verwendet wird. Dies führt zu einer Diskrepanz zwischen Training und Inferenz. Darüber hinaus übersieht der Auto-Encoding-Prozess die Abhängigkeiten zwischen den maskierten Tokens. Im obigen Beispiel wird beispielsweise die Vorhersage von x_2 (d. h. dem ersten maskierten Token) unabhängig von x_6 (d. h. dem zweiten maskierten Token) gemacht, obwohl x_6 im Kontext von x_2 berücksichtigt werden sollte.

Diese Probleme können durch den Ansatz der permutierten Sprachmodellierung für das Pre-Training gelöst werden [Yang et al., 2019]. Ähnlich wie bei der kausalen Sprachmodellierung beinhaltet die permutierte Sprachmodellierung die sequentielle Vorhersage von Tokens. Im Gegensatz zur kausalen Modellierung, bei der die Vorhersagen der natürlichen Reihenfolge des Textes folgen (wie von links nach rechts oder von rechts nach links), ermöglicht die permutierte Sprachmodellierung Vorhersagen in beliebiger Reihenfolge. Der

Ansatz ist unkompliziert: Wir bestimmen eine Reihenfolge für die Token-Vorhersagen und trainieren das Modell dann auf die übliche Weise der Sprachmodellierung, wie in Abschnitt 1.2.1 beschrieben. Beachten Sie, dass bei diesem Ansatz die tatsächliche Reihenfolge der Tokens im Text unverändert bleibt und sich nur die Reihenfolge, in der wir diese Tokens vorhersagen, von der Standard-Sprachmodellierung unterscheidet. Betrachten wir zum Beispiel eine Sequenz von 5 Tokens $x_0x_1x_2x_3x_4$. Sei $\mathbf{e}_i$ die Einbettung von x_i (d. h. eine Kombination aus der Token-Einbettung und der positionalen Einbettung). Bei der Standard-Sprachmodellierung würden wir diese Sequenz in der Reihenfolge $x_0 \rightarrow x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow x_4$ generieren. Die Wahrscheinlichkeit der Sequenz kann durch einen Generierungsprozess modelliert werden.

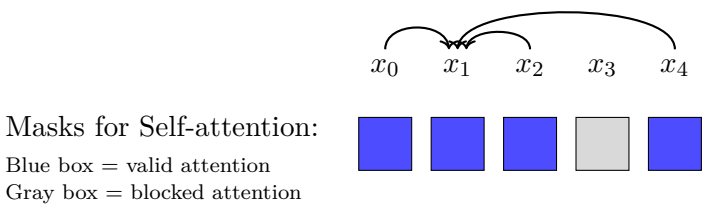
$$\begin{aligned} \Pr(\mathbf{x}) &= \Pr(x_0) \cdot \Pr(x_1|x_0) \cdot \Pr(x_2|x_0, x_1) \cdot \Pr(x_3|x_0, x_1, x_2) \cdot \\ &\quad \Pr(x_4|x_0, x_1, x_2, x_3) \\ &= \Pr(x_0) \cdot \Pr(x_1|\mathbf{e}_0) \cdot \Pr(x_2|\mathbf{e}_0, \mathbf{e}_1) \cdot \Pr(x_3|\mathbf{e}_0, \mathbf{e}_1, \mathbf{e}_2) \cdot \\ &\quad \Pr(x_4|\mathbf{e}_0, \mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3) \end{aligned} \tag{1.13}$$

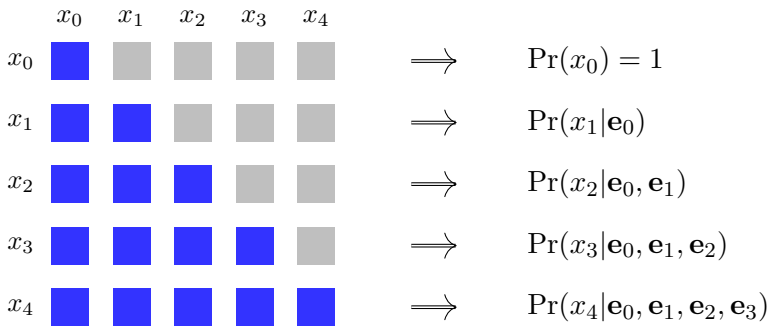
Betrachten wir nun eine andere Reihenfolge für die Token-Vorhersage: $x_0 \rightarrow x_4 \rightarrow x_2 \rightarrow x_1 \rightarrow x_3$. Der Sequenzgenerierungsprozess kann dann wie folgt ausgedrückt werden:

$$\begin{aligned} \Pr(\mathbf{x}) &= \Pr(x_0) \cdot \Pr(x_4|\mathbf{e}_0) \cdot \Pr(x_2|\mathbf{e}_0, \mathbf{e}_4) \cdot \Pr(x_1|\mathbf{e}_0, \mathbf{e}_4, \mathbf{e}_2) \cdot \\ &\quad \Pr(x_3|\mathbf{e}_0, \mathbf{e}_4, \mathbf{e}_2, \mathbf{e}_1) \end{aligned} \tag{1.14}$$

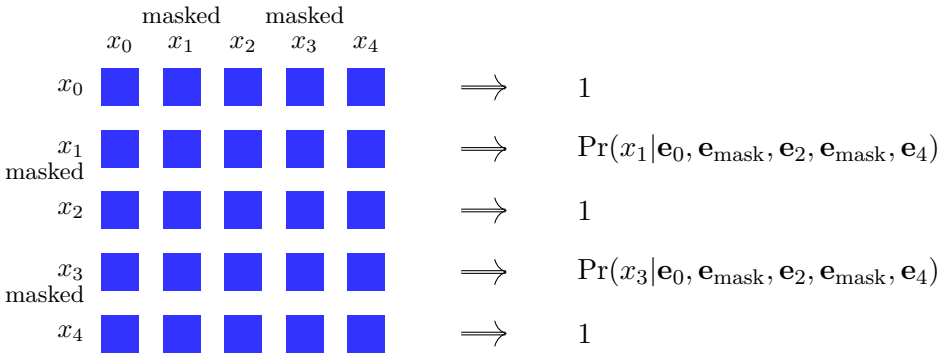
Diese neue Vorhersagereihenfolge ermöglicht es, dass die Generierung einiger Tokens von einem breiteren Kontext abhängig gemacht wird, anstatt nur auf die vorhergehenden Tokens beschränkt zu sein, wie es bei Standard-Sprachmodellen der Fall ist. Zum Beispiel berücksichtigt das Modell bei der Generierung von x_3 sowohl den linken Kontext (d. h. $\mathbf{e}_0, \mathbf{e}_1, \mathbf{e}_2$) als auch den rechten Kontext (d. h. $\mathbf{e}_4$). Die Einbettungen $\mathbf{e}_0, \mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_4$ enthalten die Positionsinformationen von x_0, x_1, x_2, x_4 und bewahren so die ursprüngliche Reihenfolge der Tokens. Folglich ähnelt dieser Ansatz gewissermaßen der maskierten Sprachmodellierung: Wir maskieren x_3 und verwenden die umgebenden Tokens x_0, x_1, x_2, x_4, um dieses Token vorherzusagen.

Die Implementierung von permutierten Sprachmodellen ist für Transformer relativ einfach. Da das Self-Attention-Modell unempfindlich gegenüber der Reihenfolge der Eingaben ist, müssen wir die Sequenz nicht explizit neu anordnen, um eine Faktorisierung wie in Gl. (1.14) zu erhalten. Stattdessen kann die Permutation durch das Setzen geeigneter Masken für die Self-Attention erfolgen. Betrachten wir zum Beispiel den Fall der Berechnung von $\Pr(x_1|\mathbf{e}_0, \mathbf{e}_4, \mathbf{e}_2)$. Wir können x_0, x_1, x_2, x_3, x_4 in der richtigen Reihenfolge anordnen und die Attention von x_3 auf x_1 in der Self-Attention blockieren, wie unten dargestellt

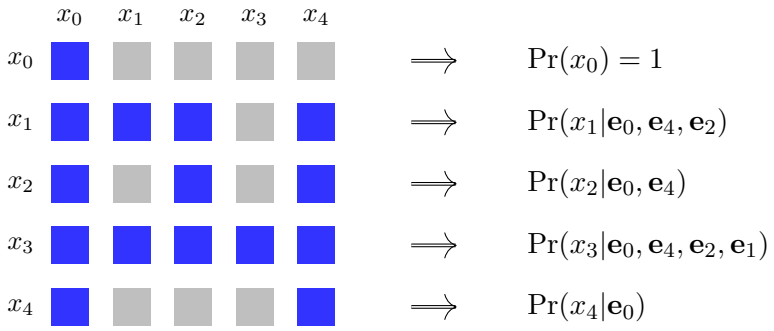




(a) Causal Language Modeling (order: $x_0 \to x_1 \to x_2 \to x_3 \to x_4$)



(b) Masked Language Modeling (order: $x_0, [\text{MASK}], x_2, [\text{MASK}], x_4 \to x_1, x_3$)



(c) Permuted Language Modeling (order: $x_0 \to x_4 \to x_2 \to x_1 \to x_3$)

Abbildung 1.3: Vergleich der Ergebnisse der Selbst-Aufmerksamkeits-Maskierung bei kausaler Sprachmodellierung, maskierter Sprachmodellierung und permutierter Sprachmodellierung. Die graue Zelle bedeutet, dass das Token an Position j das Token an Position i nicht beachtet. Die blaue Zelle (i, j) bedeutet, dass das Token an Position j das Token an Position i beachtet. $\mathbf{e}_{\text{mask}}$ repräsentiert die Einbettung des Symbols [MASK], die eine Kombination aus der Token-Einbettung und der Positionseinbettung ist.

Für ein anschaulicheres Beispiel vergleichen wir in Abbildung 1.3 die Ergebnisse der Self-Attention-Maskierung für die kausale Sprachmodellierung, die maskierte Sprachmodellierung und die permutierte Sprachmodellierung.

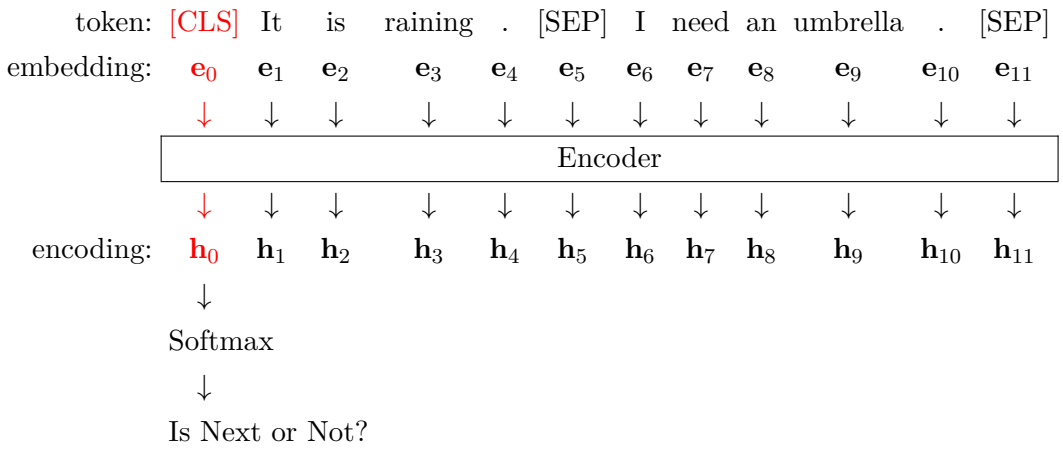
1.2.2.3 Vortrainieren von Encodern als Klassifikatoren

Eine weitere häufig verwendete Idee zum Trainieren eines Encoders ist die Betrachtung von Klassifizierungsaufgaben. Beim selbstüberwachten Lernen geschieht dies typischerweise durch die Erstellung neuer Klassifizierungsherausforderungen aus dem unmarkierten Text. Es gibt viele verschiedene Möglichkeiten, die Klassifizierungsaufgaben zu gestalten. Hier stellen wir zwei populäre Aufgaben vor.

Eine einfache Methode, genannt Nächster-Satz-Vorhersage (NSP), wird in der Originalarbeit von BERT vorgestellt [Devlin et al., 2019]. Die Annahme von NSP ist, dass ein guter Text-Encoder die Beziehung zwischen zwei Sätzen erfassen sollte. Um eine solche Beziehung zu modellieren, können wir bei NSP die Ausgabe der Kodierung von zwei aufeinanderfolgenden Sätzen Sent_A und Sent_B verwenden, um zu bestimmen, ob Sent_B der nächste Satz nach Sent_A ist. Nehmen wir zum Beispiel an, Sent_A = 'Es regnet .' und Sent_B = 'Ich brauche einen Schirm .'. Die Eingabesequenz des Encoders könnte sein

[CLS] It is raining . [SEP] I need an umbrella . [SEP]

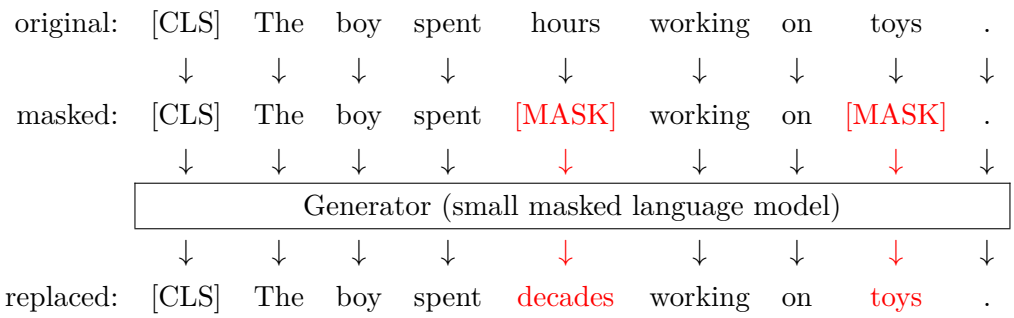
wobei [CLS] das Startsymbol ist (d. h. x_0), das üblicherweise beim Vortraining von Encodern verwendet wird, und [SEP] ein Trennzeichen ist, das die beiden Sätze trennt. Die Verarbeitung dieser Sequenz folgt einem Standardverfahren der Transformer-Kodierung: Wir repräsentieren zuerst jedes token x_i als sein entsprechendes Embedding $\mathbf{e}_i$ und speisen dann die Embedding-Sequenz $\{\mathbf{e}_0, ..., \mathbf{e}_m\}$ in den Encoder ein, um die Ausgabesequenz $\{\mathbf{h}_0, ..., \mathbf{h}_m\}$ zu erhalten. Da $\mathbf{h}_0$ im Allgemeinen als die Repräsentation der gesamten Sequenz betrachtet wird, fügen wir eine Softmax-Schicht darüber hinzu, um ein binäres Klassifizierungssystem zu erstellen. Dieser Prozess wird wie folgt veranschaulicht



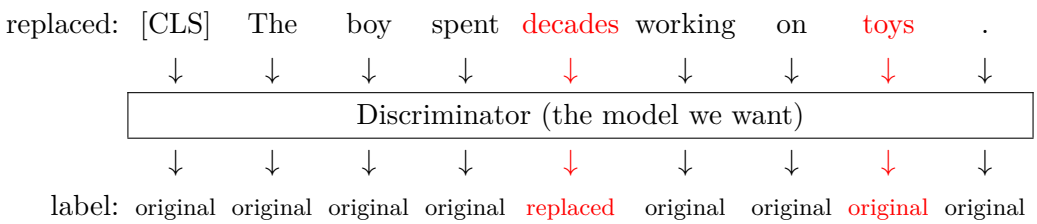
Um Trainingsbeispiele zu generieren, benötigen wir jedes Mal zwei Sätze, einen für Sent_A und den anderen für Sent_B. Eine einfache Möglichkeit, dies zu tun, besteht darin, die natürliche Abfolge von zwei aufeinanderfolgenden Sätzen im Text zu nutzen. Zum Beispiel erhalten wir ein positives Beispiel, indem wir tatsächlich aufeinanderfolgende Sätze verwenden, und ein negatives Beispiel, indem wir zufällig ausgewählte Sätze verwenden. Folglich ist das Trainieren dieses Modells dasselbe wie das Trainieren eines Klassifikators.

Typischerweise wird NSP als zusätzliche Trainings-Verlustfunktion für das Vortraining auf Basis von maskierter Sprachmodellierung verwendet.

Ein zweites Beispiel für das Trainieren von Transformer-Encodern als Klassifikatoren ist die Anwendung von klassifizierungsbasierten Überwachungssignalen auf jede Ausgabe eines Encoders. Zum Beispiel schlagen Clark et al. [2019] in ihrem ELECTRA-Modell vor, einen Transformer-Encoder zu trainieren, um zu identifizieren, ob jedes Eingabe-Token mit der ursprünglichen Eingabe identisch ist oder auf irgendeine Weise verändert wurde. Der erste Schritt dieser Methode besteht darin, aus einer gegebenen Sequenz von Tokens eine neue Sequenz zu generieren, in der einige der Tokens verändert sind. Dazu wird ein kleines maskiertes Sprachmodell (nennen wir es den Generator) angewendet: Wir maskieren zufällig einige der Tokens und trainieren dieses Modell, um die maskierten Tokens vorherzusagen. Für jedes Trainingsbeispiel gibt dieses maskierte Sprachmodell an jeder maskierten Position ein Token aus, das sich vom ursprünglichen Token unterscheiden kann. Gleichzeitig trainieren wir einen weiteren Transformer-Encoder (nennen wir ihn den Diskriminator), um zu bestimmen, ob jedes vorhergesagte Token mit dem ursprünglichen Token identisch oder verändert ist. Genauer gesagt verwenden wir den Generator, um eine Sequenz zu erzeugen, in der einige der Tokens ersetzt werden. Unten ist eine Illustration.



Dann verwenden wir den Diskriminator, um jedes dieser Tokens als original oder replaced zu kennzeichnen, wie folgt



Für das Training wird der Generator als maskiertes Sprachmodell mit Maximum-Likelihood-Schätzung optimiert, und der Diskriminator wird als Klassifikator unter Verwendung eines klassifizierungsbasierten Verlusts optimiert. In ELECTRA werden der Maximum-Likelihood-basierte Verlust und der klassifizierungsbasierte Verlust kombiniert, um sowohl den Generator als auch den Diskriminator gemeinsam zu trainieren. Ein alternativer Ansatz ist die Verwendung von generativen adversariellen Netzwerken (GANs), das heißt, der Generator wird trainiert, um den Diskriminator zu täuschen, und der Diskriminator wird trainiert, um die Ausgabe des Generators von der wahren Verteilung zu

unterscheiden. Jedoch verkompliziert das Training im GAN-Stil die Trainingsaufgabe und ist schwieriger zu skalieren. Dennoch wird der Generator nach Abschluss des Trainings verworfen, und der Kodierungsteil des Diskriminators wird als vortrainiertes Modell für nachgelagerte Aufgaben verwendet.

1.2.3 Encoder-Decoder-Vortraining

Im NLP werden Encoder-Decoder-Architekturen oft verwendet, um Sequenz-zu-Sequenz-Probleme wie maschinelle Übersetzung und die Beantwortung von Fragen zu modellieren. Zusätzlich zu diesen typischen Sequenz-zu-Sequenz-Problemen im NLP können Encoder-Decoder-Modelle erweitert werden, um viele andere Probleme zu behandeln. Eine einfache Idee ist es, Text sowohl als Eingabe als auch als Ausgabe eines Problems zu betrachten, sodass wir Encoder-Decoder-Modelle direkt anwenden können. Zum Beispiel können wir bei einem gegebenen Text ein Modell bitten, einen Text auszugeben, der die Stimmung des Eingabetextes beschreibt, wie positiv, negativ und neutral.

Eine solche Idee ermöglicht es uns, ein einziges Text-zu-Text-System zu entwickeln, um jedes NLP-Problem zu lösen. Wir können verschiedene Probleme in dasselbe Text-zu-Text-Format formulieren. Wir trainieren zuerst ein Encoder-Decoder-Modell, um durch selbstüberwachtes Lernen allgemeines Sprachwissen zu erlangen. Dieses Modell wird dann für spezifische nachgelagerte Aufgaben unter Verwendung gezielter Text-zu-Text-Daten feinabgestimmt.

1.2.3.1 Maskiertes Encoder-Decoder-Vortraining

Im T5-Modell von Raffel et al. [2020] werden viele verschiedene Aufgaben als dieselbe Text-zu-Text-Aufgabe formuliert. Jedes Beispiel in T5 folgt dem Format

$$\text{Quelltext} \quad \rightarrow \quad \text{Zieltext}$$

Hier trennt $\rightarrow$ den Quelltext, der aus einer Aufgabenbeschreibung oder Anweisung und der dem System gegebenen Eingabe besteht, vom Zieltext, der die Antwort auf die Eingabeaufgabe ist. Betrachten wir als Beispiel eine Aufgabe der Übersetzung aus dem Chinesischen ins Englische. Ein Trainingsbeispiel kann ausgedrückt werden als

$$\text{[CLS] Übersetze von Chinesisch nach Englisch: 你好 !} \quad \rightarrow \quad \langle s \rangle \text{ Hallo!}$$

wobei [CLS] und $\langle s \rangle$ die Startsymbole auf der Quell- bzw. Zielseite sind[5].

[5] Wir könnten dasselbe Startsymbol für verschiedene Sequenzen verwenden. Hier verwenden wir unterschiedliche Symbole, um die Sequenzen auf der Encoder- und Decoder-Seite zu unterscheiden.

Ebenso können wir andere Aufgaben auf die gleiche Weise ausdrücken. Zum Beispiel

[CLS] Answer: when was Albert Einstein born?
→ ⟨s⟩ He was born on March 14, 1879.

[CLS] Simplify: the professor, who has published numerous papers in his field, will be giving a lecture on the topic next week.
→ ⟨s⟩ The experienced professor will give a lecture next week.

[CLS] Score the translation from English to Chinese. English: when in Rome, do as the Romans do. Chinese: 人 在 罗马 就 像 罗马 人 一样 做事 。
→ ⟨s⟩ 0.81

wobei Anweisungen grau hinterlegt sind. Ein interessanter Fall ist, dass wir im letzten Beispiel das Bewertungsproblem als ein Texterzeugungsproblem umformulieren. Unser Ziel ist es, einen Text zu generieren, der die Zahl 0.81 darstellt, anstatt sie als numerischen Wert auszugeben.

Der oben beschriebene Ansatz bietet ein neues Rahmenwerk für universelles Sprachverständnis und Sprachgenerierung. Sowohl die Aufgabenanweisungen als auch die Problemeingaben werden dem System in Textform zur Verfügung gestellt. Das System folgt dann den Anweisungen, um die Aufgabe zu erledigen. Diese Methode fasst verschiedene Probleme zusammen, mit dem Vorteil, ein einziges Modell zu trainieren, das viele Aufgaben gleichzeitig ausführen kann.

Im Allgemeinen ist eine Feinabstimmung notwendig, um das vortrainierte Modell an eine spezifische nachgelagerte Aufgabe anzupassen. In diesem Prozess kann man das Modell auf unterschiedliche Weise für die Aufgabe instruieren, beispielsweise durch die Verwendung eines kurzen Namens der Aufgabe als Präfix für die eigentliche Eingabesequenz oder durch die Bereitstellung einer detaillierten Beschreibung der Aufgabe. Da die Aufgabenanweisungen in Textform ausgedrückt und als Teil der Eingabe einbezogen werden, kann das allgemeine Wissen über Anweisungen durch das Erlernen der Sprachverständnismodelle in der Vortrainingsphase gewonnen werden. Dies kann helfen, Zero-Shot-Lernen zu ermöglichen. Zum Beispiel können vortrainierte Modelle generalisieren, um neue Probleme zu lösen, bei denen die Aufgabenanweisungen noch nie zuvor angetroffen wurden.

Es gab bereits mehrere leistungsstarke Methoden des selbstüberwachten Lernens für entweder Transformer-Encoder oder -Decoder. Die Anwendung dieser Methoden zum Vortrainieren von Encoder-Decoder-Modellen ist relativ einfach. Eine übliche Wahl ist es, Encoder-Decoder-Modelle als Sprachmodelle zu trainieren. Zum Beispiel empfängt der Encoder ein Sequenzpräfix, während der Decoder die restliche Sequenz generiert. Dies unterscheidet sich jedoch von der standardmäßigen kausalen Sprachmodellierung, bei der die gesamte Sequenz autoregressiv vom ersten Token an generiert wird. In unserem Fall verarbeitet der Encoder das Präfix auf einmal, und dann sagt der Decoder nachfolgende Tokens in der Art der kausalen Sprachmodellierung voraus. Genauer gesagt handelt es sich um ein Problem der Präfix-Sprachmodellierung: Ein Sprachmodell sagt die nachfolgende Sequenz voraus, gegeben ein Präfix, das als Kontext für die Vorhersage dient.

Betrachten Sie das folgende Beispiel

$$\underbrace{\text{[CLS] The puppies are frolicking}}_{\text{Prefix}} \rightarrow \underbrace{\langle s \rangle \text{ outside the house .}}_{\text{Subsequent Sequence}}$$

Wir können ein Encoder-Decoder-Modell direkt mit solchen Beispielen trainieren. Dann lernt der Encoder, das Präfix zu verstehen, und der Decoder lernt, auf der Grundlage dieses Verständnisses weiterzuschreiben. Für das groß angelegte Vortraining ist es einfach, eine große Anzahl von Trainingsbeispielen aus unmarkiertem Text zu erstellen.

Es ist erwähnenswert, dass vortrainierte Encoder-Decoder-Modelle, um bei mehrsprachigen und sprachübergreifenden Aufgaben wie der maschinellen Übersetzung effektiv zu sein, mit mehrsprachigen Daten trainiert werden sollten. Dies erfordert typischerweise, dass das Vokabular Tokens aus allen Sprachen enthält. Dadurch können die Modelle gemeinsame Repräsentationen über verschiedene Sprachen hinweg lernen und so Fähigkeiten sowohl im Sprachverständnis als auch in der Sprachgenerierung in einem mehrsprachigen und sprachübergreifenden Kontext ermöglichen.

Ein zweiter Ansatz zum Vortrainieren von Encoder-Decoder-Modellen ist die maskierte Sprachmodellierung. Bei diesem Ansatz werden, wie in Abschnitt 1.2.2 erläutert, Tokens in einer Sequenz zufällig durch ein Maskensymbol ersetzt, und das Modell wird dann trainiert, diese maskierten Tokens basierend auf der gesamten maskierten Sequenz vorherzusagen.

Zur Veranschaulichung betrachten wir die Aufgabe, den Satz zu maskieren und zu rekonstruieren

The puppies are frolicking outside the house .

Indem wir zwei Tokens (sagen wir, frolicking und the) maskieren, erhalten wir die BERT-artige Eingabe und Ausgabe des Modells wie folgt

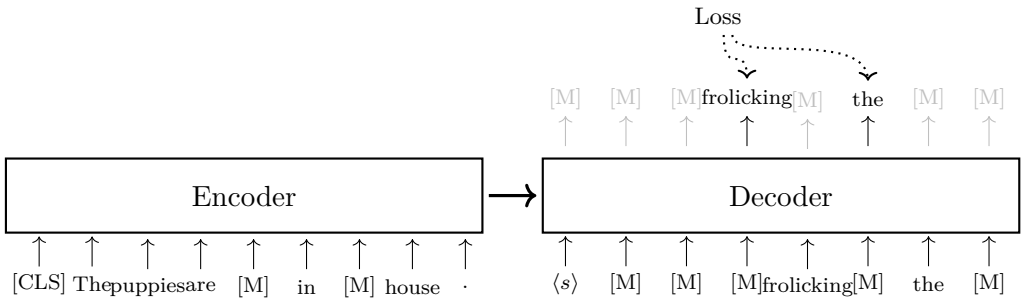
$$\begin{aligned} & \text{[CLS] The puppies are [MASK] outside [MASK] house .} \\ \rightarrow\ & \langle s \rangle \ \_\_ \ \_\_ \ \_\_ \text{ frolicking } \_\_ \text{ the } \_\_ \ \_\_ \end{aligned}$$

Hier bezeichnet __ die maskierte Position, an der wir keine Token-Vorhersagen machen. Indem der Prozentsatz der maskierten Tokens im Text variiert wird, kann dieser Ansatz entweder in Richtung eines BERT-artigen Trainings oder eines Trainings im Stil der Sprachmodellierung verallgemeinert werden [Song et al., 2019]. Wenn wir zum Beispiel alle Tokens ausmaskieren, wird das Modell trainiert, die gesamte Sequenz zu generieren

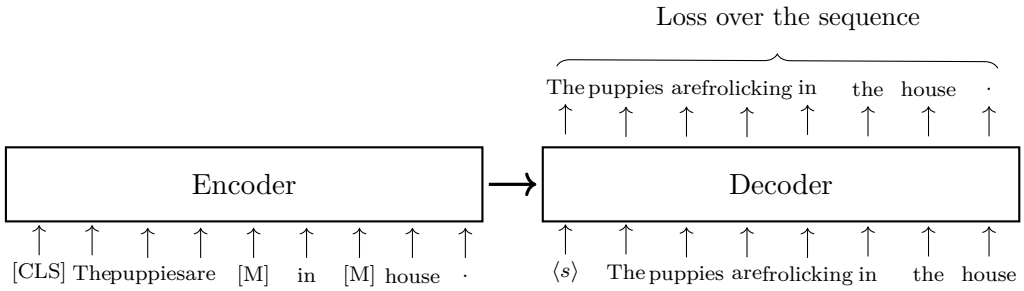
$$\begin{aligned} & \text{[CLS] [MASK] [MASK] [MASK] [MASK] [MASK] [MASK] [MASK] [MASK]} \\ \rightarrow\ & \langle s \rangle \text{ The puppies are frolicking outside the house .} \end{aligned}$$

In diesem Fall trainieren wir den Decoder als Sprachmodell.

Beachten Sie, dass wir im Kontext der Encoder-Decoder-Architektur den Encoder verwenden können, um die maskierte Sequenz zu lesen, und den Decoder, um die ursprüngliche Sequenz vorherzusagen. Mit diesem Ziel haben wir im Wesentlichen einen



(a) Training an encoder-decoder model with BERT-style masked language modeling



(b) Training an encoder-decoder model with denoising autoencoding

Abbildung 1.4: Training eines Encoder-Decoder-Modells unter Verwendung von BERT-artigen und Denoising-Autoencoding-Methoden. Bei beiden Methoden ist die Eingabe für den Encoder eine korrumpierte Token-Sequenz, bei der einige Tokens maskiert und durch [MASK] (oder kurz [M]) ersetzt werden. Der Decoder sagt diese maskierten Tokens voraus, jedoch auf unterschiedliche Weise. Beim BERT-artigen Training muss der Decoder nur den Verlust für die maskierten Tokens berechnen, während die übrigen Tokens in der Sequenz einfach als [MASK]-Tokens behandelt werden können. Beim Denoising Autoencoding sagt der Decoder die Sequenz aller Tokens auf autoregressive Weise voraus. Folglich wird der Verlust durch die Akkumulation der Verluste all dieser Tokens ermittelt, wie bei der Standard-Sprachmodellierung.

Denoising Autoencoder: Der Encoder transformiert eine beschädigte Eingabe in eine verborgene Repräsentation, und der Decoder rekonstruiert die unbeschädigte Eingabe aus dieser verborgenen Repräsentation. Hier ist ein Beispiel für Eingabe und Ausgabe für das Denoising-Training.

[CLS] The puppies are [MASK] outside [MASK] house .
→ ⟨s⟩ The puppies are frolicking outside the house .

Indem das Modell lernt, von dieser beschädigten Sequenz auf ihr unbeschädigtes Gegenstück abzubilden, erlangt es die Fähigkeit, auf der Encoder-Seite zu verstehen und auf der Decoder-Seite zu generieren. Siehe Abbildung 1.4 für eine Veranschaulichung, wie ein Encoder-Decoder-Modell mit BERT-artigen und Denoising-Autoencoding-Zielen trainiert wird.

Da wir Tokens zufällig zum Maskieren auswählen, können wir sicherlich auch aufeinanderfolgende Tokens maskieren [Joshi et al., 2020]. Hier ist ein Beispiel.

[CLS] The puppies are [MASK] outside [MASK] [MASK] .
→ ⟨s⟩ The puppies are frolicking outside the house .

Eine andere Möglichkeit, aufeinanderfolgende maskierte Tokens zu betrachten, besteht darin, sie als Spans darzustellen. Hier folgen wir der Arbeit von Raffel et al. [2020] und verwenden [X], [Y] und [Z], um Sentinel-Tokens zu bezeichnen, die einen oder mehrere aufeinanderfolgende maskierte Tokens abdecken. Mit dieser Notation können wir das obige Trainingsbeispiel umformulieren als

[CLS] The puppies are [X] outside [Y] .
→ ⟨s⟩ [X] frolicking [Y] the house [Z]

Die Idee ist, dass wir die beschädigte Sequenz als eine Sequenz darstellen, die Platzhalter-Slots enthält. Die Trainingsaufgabe besteht darin, diese Slots mit den korrekten Tokens unter Verwendung des umgebenden Kontexts zu füllen. Ein Vorteil dieses Ansatzes ist, dass die im Training verwendeten Sequenzen kürzer wären, was das Training effizienter macht. Beachten Sie, dass die maskierte Sprachmodellierung ein sehr allgemeines Rahmenwerk für das Training von Encoder-Decoder-Modellen bietet. Verschiedene Einstellungen können angepasst werden, um unterschiedliche Trainingsversionen zu erhalten, wie z.B. die Änderung des Prozentsatzes der maskierten Tokens und der maximalen Länge der maskierten Spans.

1.2.3.2 Training mit Denoising

Wenn wir das Problem des Trainings von Encoder-Decoder-Modellen als ein Problem des Trainings von entrauschenden Autoencodern betrachten, gibt es typischerweise viele verschiedene Methoden, um Eingabekorruption einzuführen und die Eingabe zu rekonstruieren. Zum Beispiel können wir über das zufällige Maskieren von Tokens hinaus auch einige von ihnen ändern oder ihre Reihenfolge neu anordnen.

Angenommen, wir haben ein Encoder-Decoder-Modell, das eine Eingabesequenz $\mathbf{x}$ auf eine Ausgabesequenz $\mathbf{y}$ abbilden kann

$$\begin{aligned} \mathbf{y} &= \text{Decode}_{\omega}(\text{Encode}_{\theta}(\mathbf{x})) \\ &= \text{Model}_{\theta,\omega}(\mathbf{x}) \end{aligned} \tag{1.15}$$

wobei θ und ω die Parameter des Encoders bzw. des Decoders sind. Bei Problemen des entrauschenden Autoencodings fügen wir $\mathbf{x}$ etwas Rauschen hinzu, um eine verrauschte, korrumpierte Eingabe $\mathbf{x}_{\text{noise}}$ zu erhalten. Indem wir $\mathbf{x}_{\text{noise}}$ in den Encoder einspeisen, soll der Decoder die ursprüngliche Eingabe ausgeben. Das Trainingsziel kann wie folgt definiert werden:

$$(\hat{\theta}, \hat{\omega}) = \arg\min_{\theta,\omega} \text{Loss}(\text{Model}_{\theta,\omega}(\mathbf{x}_{\text{noise}}), \mathbf{x}) \tag{1.16}$$

Hier bewertet die Verlustfunktion $\text{Loss}(\text{Model}_{\theta,\omega}(\mathbf{x}_{\text{noise}}), \mathbf{x})$, wie gut das Modell $\text{Model}_{\theta,\omega}(\mathbf{x}_{\text{noise}})$ die ursprüngliche Eingabe $\mathbf{x}$ rekonstruiert. Wir können wie üblich den Kreuzentropie-Verlust wählen.

Nachdem die Modellarchitektur und der Trainingsansatz entwickelt wurden, ist das verbleibende Problem die Korruption der Eingabe. Lewis et al. [2020] schlagen in ihrem BART-Modell vor, die Eingabesequenz auf verschiedene Weisen zu korrumpieren.

- Token-Maskierung. Dies ist die gleiche Maskierungsmethode, die wir beim maskierten Sprachmodellieren verwendet haben. Die Token in der Eingabesequenz werden zufällig ausgewählt und maskiert.

- Token-Löschung. Diese Methode ähnelt der Token-Maskierung. Anstatt jedoch die ausgewählten Token durch ein spezielles Symbol [MASK] zu ersetzen, werden diese Token aus der Sequenz entfernt. Siehe das folgende Beispiel für einen Vergleich der Methoden der Token-Maskierung und der Token-Löschung.

Original ($\mathbf{x}$):	The puppies are frolicking outside the house .
Token-Maskierung ($\mathbf{x}_{\text{noise}}$):	The puppies are [MASK] outside [MASK] house .
Token-Löschung ($\mathbf{x}_{\text{noise}}$):	The puppies are ~~frolicking~~ outside ~~the~~ house .

  wobei die unterstrichenen Token in der ursprünglichen Sequenz maskiert oder gelöscht werden.

- Span-Maskierung. Über die Sequenz werden nicht überlappende Spannen zufällig ausgewählt. Jede Spanne wird durch [MASK] maskiert. Wir betrachten auch Spannen der Länge 0, und in solchen Fällen wird [MASK] einfach an einer Position in der Sequenz eingefügt. Zum Beispiel können wir die obige Sequenz mittels Span-Maskierung wie folgt verändern:

Original ($\mathbf{x}$):	The 0 puppies are frolicking outside the house .
Span-Maskierung ($\mathbf{x}_{\text{noise}}$):	The [MASK] puppies are [MASK] house .

  Hier wird die Spanne frolicking outside the durch ein einziges [MASK] ersetzt. 0 bezeichnet eine Spanne der Länge 0, und so fügen wir ein [MASK] zwischen The und puppies ein. Die Span-Maskierung führt neue Vorhersageherausforderungen ein, bei denen das Modell wissen muss, wie viele Token aus einer Spanne generiert werden. Dieses Problem ist der Fertilitätsmodellierung in der maschinellen Übersetzung sehr ähnlich [Brown et al., 1993].

Wenn wir eine Sequenz betrachten, die aus mehreren Sätzen besteht, können zusätzliche Korruptionsmethoden angewendet werden. Im BART-Modell gibt es zwei solcher Methoden.

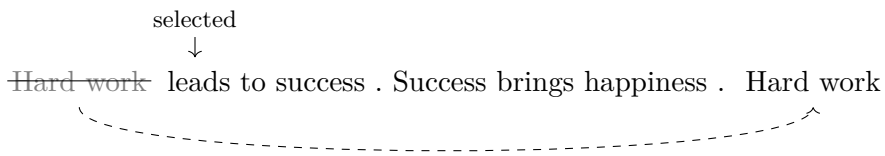
- Satz-Neuordnung. Diese Methode permutiert die Sätze zufällig, sodass das Modell lernen kann, Sätze in einem Dokument neu zu ordnen. Betrachten wir zum Beispiel zwei aufeinanderfolgende Sätze

Hard work leads to success . Success brings happiness .

Wir können die beiden Sätze neu anordnen, um eine korrumpierte Eingabesequenz zu erhalten

Success brings happiness . Hard work leads to success .

- Dokument-Rotation. Das Ziel dieser Aufgabe ist es, den Start-Token der Sequenz zu identifizieren. Zuerst wird ein Token zufällig aus der Sequenz ausgewählt. Dann wird die Sequenz so rotiert, dass der ausgewählte Token der erste Token ist. Nehmen wir zum Beispiel an, wir wählen den Token leads aus der obigen Sequenz aus. Die rotierte Sequenz ist



  wobei die Untersequenz Hard work vor leads an das Ende der Sequenz angehängt wird.

Für das Vortraining können wir mehrere Korruptionsmethoden anwenden, um robuste Modelle zu lernen, zum Beispiel, indem wir für jede Trainingsstichprobe zufällig eine davon auswählen. In der Praxis hängt das Ergebnis des Vortrainings von Encoder-Decoder-Modellen stark von den verwendeten Eingabekorruptionsmethoden ab, und daher müssen wir typischerweise durch sorgfältige Experimente geeignete Trainingsziele auswählen.

1.2.4 Vergleich von Vortrainingsaufgaben

Bisher haben wir eine Reihe von Vortrainingsaufgaben besprochen. Da dasselbe Trainingsziel auf verschiedene Architekturen angewendet werden kann (z. B. die Verwendung von maskierter Sprachmodellierung sowohl für das nur-Encoder- als auch für das Encoder-Decoder-Vortraining), erscheint eine Kategorisierung von Vortrainingsaufgaben ausschließlich nach der Modellarchitektur nicht ideal. Stattdessen fassen wir diese Aufgaben anhand der Trainingsziele zusammen.

- Sprachmodellierung. Typischerweise bezieht sich dieser Ansatz auf ein autoregressives Generierungsverfahren von Sequenzen. Dabei wird zu einem Zeitpunkt das nächste Token basierend auf seinem vorherigen Kontext vorhergesagt.

- Maskierte Sprachmodellierung. Die maskierte Sprachmodellierung gehört zu einem allgemeinen Mask-Predict-Framework. Es maskiert zufällig Tokens in einer Sequenz und sagt diese Tokens unter Verwendung der gesamten maskierten Sequenz voraus.

- Permutierte Sprachmodellierung. Die permutierte Sprachmodellierung folgt einer ähnlichen Idee wie die maskierte Sprachmodellierung, berücksichtigt jedoch die Reihenfolge der (maskierten) Token-Vorhersage. Sie ordnet die Eingabesequenz neu an und sagt die Tokens sequenziell voraus. Jede Vorhersage basiert auf einigen zufällig ausgewählten Kontext-Tokens.

- Diskriminatives Training. Beim diskriminativen Training werden Supervisionssignale aus Klassifikationsaufgaben erstellt. Modelle für das Vortraining werden in Klassifikatoren integriert und zusammen mit den übrigen Teilen der Klassifikatoren trainiert, um deren Klassifikationsleistung zu verbessern.

- Denoising Autoencoding. Dieser Ansatz wird für das Vortraining von Encoder-Decoder-Modellen angewendet. Die Eingabe ist eine beschädigte Sequenz, und die Encoder-Decoder-Modelle werden trainiert, um die ursprüngliche Sequenz zu rekonstruieren.

Tabelle 1.1 veranschaulicht diese Methoden und ihre Varianten anhand von Beispielen. Die Verwendung dieser Beispiele unterscheidet nicht zwischen Modellen, aber wir kennzeichnen die Modellarchitekturen, bei denen die Vortrainingsaufgaben angewendet werden können. In jedem Beispiel besteht die Eingabe aus einer Token-Sequenz, und die Ausgabe ist entweder eine Token-Sequenz oder einige Wahrscheinlichkeiten. Bei Generierungsaufgaben, wie der Sprachmodellierung, werden hochgestellte Zeichen verwendet, um die Generierungsreihenfolge auf der Zielseite anzugeben. Wenn die hochgestellten Zeichen weggelassen werden, bedeutet dies, dass die Ausgabesequenz entweder autoregressiv oder simultan erzeugt werden kann. Auf der Quellseite gehen wir davon aus, dass die Sequenz einen standardmäßigen Transformer-Codierungsprozess durchläuft, was bedeutet, dass jedes Token in der Selbst-Aufmerksamkeit die gesamte Sequenz sehen kann. Die einzige Ausnahme ist die permutierte Sprachmodellierung, bei der ein autoregressiver Generierungsprozess durch das Setzen von Aufmerksamkeitsmasken auf der Encoder-Seite implementiert wird. Um die Diskussion zu vereinfachen, entfernen wir das Token $\langle s \rangle$ von der Zielseite jedes Beispiels.

Obwohl diese Vortrainingsaufgaben unterschiedlich sind, ist es möglich, sie im selben Framework und experimentellen Aufbau zu vergleichen [Dong et al., 2019; Raffel et al., 2020; Lewis et al., 2020]. Beachten Sie, dass wir hier nicht alle Vortrainingsaufgaben auflisten können, da es sehr viele davon gibt. Für weitere Diskussionen über Vortrainingsaufgaben wird der interessierte Leser auf einige Übersichtsartikel zu diesem Thema verwiesen [Qiu et al., 2020; Han et al., 2021].

1.3 Beispiel: BERT

In diesem Abschnitt stellen wir BERT-Modelle vor, die zu den beliebtesten und am weitesten verbreiteten vortrainierten Sequenzkodierungsmodellen im Bereich NLP gehören.

Method	Enc	Dec	E-D	Input	Output
Causal LM		•	•		The^1 kitten^2 is^3 chasing^4 the^5 ball^6 .^{7}
Prefix LM		•	•	[C] The kitten is	chasing^1 the^2 ball^3 .^{4}
Masked LM	•		•	[C] The kitten [M] chasing the [M] .	_ _ is _ _ ball _
MASS-style	•		•	[C] The kitten [M] [M] [M] ball .	_ _ is chasing the _ _
BERT-style	•		•	[C] The kitten [M] playing the [M] .	_ kitten is chasing _ ball _
Permuted LM	•			[C] The kitten is chasing the ball .	The^5 kitten^7 is^6 chasing^1 the^4 ball^2 .^{3}
Next Sentence Prediction	•			[C] The kitten is chasing the ball . Birds eat worms .	Pr(IsNext | representation-of-[C])
Sentence Comparison	•			Encode a sentence as h_a and another sentence as h_b	Score(h_a, h_b)
Token Classification	•			[C] The kitten is chasing the ball .	Pr(·|The) Pr(·|kitten) ... Pr(·|.)
Token Reordering			•	[C] . kitten the chasing The is ball	The^1 kitten^2 is^3 chasing^4 the^5 ball^6 .^{7}
Token Deletion			•	[C] The kitten is ~~chasing~~ the ball .	The^1 kitten^2 is^3 chasing^4 the^5 ball^6 .^{7}
Span Masking			•	[C] The kitten [M] is [M] .	The^1 kitten^2 is^3 chasing^4 the^5 ball^6 .^{7}
Sentinel Masking			•	[C] The kitten [X] the [Y]	[X]^1 is^2 chasing^3 [Y]^4 ball^5 .^{6}
Sentence Reordering			•	[C] The ball rolls away swiftly . The kitten is chasing the ball .	The^1 kitten^2 is^3 chasing^4 the^5 ball^6 .^{7} The^8 ball^9 rolls^{10} away^{11} swiftly^{12} .^{13}
Document Rotation			•	[C] chasing the ball . The ball rolls away swiftly . The kitten is	The^1 kitten^2 is^3 chasing^4 the^5 ball^6 .^{7} The^8 ball^9 rolls^{10} away^{11} swiftly^{12} .^{13}

Tabelle 1.1: Vergleich von Pre-Training-Aufgaben, einschließlich language modeling, masked language modeling, permuted language modeling, discriminative training und denoising autoencoding. [C] = [CLS], [M] = [MASK], [X], [Y] = Sentinel-Tokens. Enc, Dec und E-D geben an, ob der Ansatz auf reine Encoder-, reine Decoder- bzw. Encoder-Decoder-Modelle angewendet werden kann. Für Generierungsaufgaben werden Hochstellungen verwendet, um die Reihenfolge der Tokens darzustellen.

1.3.1 Das Standardmodell

Das Standard-BERT-Modell, das in der Arbeit von Devlin et al. [2019] vorgeschlagen wurde, ist ein Transformer Encoder, der sowohl mit den Aufgaben der maskierten Sprachmodellierung als auch der Vorhersage des nächsten Satzes trainiert wird. Der beim Training dieses Modells verwendete Verlust ist die Summe der Verluste der beiden Aufgaben.

$$\text{Loss}_{\text{BERT}} = \text{Loss}_{\text{MLM}} + \text{Loss}_{\text{NSP}} \tag{1.17}$$

Wie beim Training tiefer neuronaler Netze üblich, optimieren wir die Modellparameter, indem wir diesen Verlust minimieren. Dazu wird eine Anzahl von Trainingsbeispielen gesammelt. Während des Trainings wird jeweils ein Batch von Trainingsbeispielen zufällig aus dieser Sammlung ausgewählt, und $\text{Loss}_{\text{BERT}}$ wird über diese Trainingsbeispiele akkumuliert. Anschließend werden die Modellparameter mittels Gradientenabstieg oder dessen Varianten aktualisiert. Dieser Prozess wird viele Male wiederholt, bis ein Abbruchkriterium erfüllt ist, beispielsweise wenn der Trainingsverlust konvergiert.

1.3.1.1 Verlustfunktionen

Im Allgemeinen werden BERT-Modelle verwendet, um einen einzelnen Satz oder ein Satzpaar darzustellen, und können daher verschiedene nachgeschaltete Sprachverständnisprobleme bewältigen. In diesem Abschnitt nehmen wir an, dass die Eingaberepräsentation eine Sequenz ist, die zwei Sätze Sent_A und Sent_B enthält, ausgedrückt als

$$\text{[CLS]} \quad \text{Sent}_A \quad \text{[SEP]} \quad \text{Sent}_B \quad \text{[SEP]}$$

Hier folgen wir der Notation im BERT-Paper und verwenden [SEP], um das Trennzeichen zu bezeichnen.

Gegeben diese Sequenz, können wir Loss_{MLM} und Loss_{NSP} separat erhalten. Für die maskierte Sprachmodellierung sagen wir eine Teilmenge der Tokens in der Sequenz voraus. Typischerweise wird ein bestimmter Prozentsatz der Tokens zufällig ausgewählt, zum Beispiel werden im Standard-BERT-Modell 15% der Tokens in jeder Sequenz ausgewählt. Dann wird die Sequenz auf drei Arten modifiziert

- Token-Maskierung. 80% der ausgewählten Token werden maskiert und durch das Symbol [MASK] ersetzt. Zum Beispiel

Original:	[CLS] It is raining . [SEP] I need an umbrella . [SEP]
Maskiert:	[CLS] It is [MASK] . [SEP] I need [MASK] umbrella . [SEP]

  wobei die ausgewählten Token unterstrichen sind. Die Vorhersage maskierter Token veranlasst das Modell, zu lernen, Token aus ihrem umgebenden Kontext darzustellen.

- Zufälliger Ersatz. 10% der ausgewählten Token werden durch ein zufälliges Token ersetzt. Zum Beispiel

Original:	[CLS] It is raining . [SEP] I need an umbrella . [SEP]
Zufälliges Token:	[CLS] It is raining . [SEP] I need an hat . [SEP]

  Dies hilft dem Modell, zu lernen, ein Token aus einer verrauschten Eingabe wiederherzustellen.

- Unverändert. 10% der ausgewählten Token bleiben unverändert. Zum Beispiel,

Original:	[CLS] It is raining . [SEP] I need an umbrella . [SEP]
Unverändertes Token:	[CLS] It is raining . [SEP] I need an umbrella . [SEP]

  Dies ist keine schwierige Vorhersageaufgabe, kann das Modell aber anleiten, einfachere Anhaltspunkte für die Vorhersage zu verwenden.

Sei $\mathcal{A}(\mathbf{x})$ die Menge der ausgewählten Positionen einer gegebenen Token-Sequenz $\mathbf{x}$,

und sei $\bar{\mathbf{x}}$ die modifizierte Sequenz von $\mathbf{x}$. Die Verlustfunktion der maskierten Sprachmodellierung kann definiert werden als

$$\text{Loss}_{\text{MLM}} \quad = \quad -\sum_{i \in \mathcal{A}(\mathbf{x})} \log \Pr_i(x_i|\bar{\mathbf{x}}) \tag{1.18}$$

wobei $\Pr_i(x_i|\bar{\mathbf{x}})$ die Wahrscheinlichkeit ist, x_i an der Position i gegeben $\bar{\mathbf{x}}$ vorherzusagen. Abbildung 1.5 zeigt ein laufendes Beispiel für die Berechnung von Loss_{MLM}.

Für die Nächster-Satz-Vorhersage folgen wir der in Abschnitt 1.2.2.3 beschriebenen Methode. Jedes Trainingsbeispiel wird in einen Label-Satz $\{\text{IsNext}, \text{NotNext}\}$ klassifiziert, zum Beispiel,

Sequence: [CLS] It is raining . [SEP] I need an umbrella . [SEP]
Label: IsNext

Sequence: [CLS] The cat sleeps on the windowsill . [SEP] Apples grow on trees . [SEP]
Label: NotNext

Der Ausgabevektor des Encoders für das erste Token [CLS] wird als die Sequenzrepräsentation betrachtet, bezeichnet mit $\mathbf{h}_{\text{cls}}$ (oder $\mathbf{h}_0$). Ein Klassifikator wird auf $\mathbf{h}_{\text{cls}}$ aufgebaut. Dann können wir die Wahrscheinlichkeit eines Labels c gegeben $\mathbf{h}_{\text{cls}}$ berechnen, d.h. $\Pr(c|\mathbf{h}_{\text{cls}})$. Es gibt viele Verlustfunktionen, die man für Klassifikationsprobleme wählen kann. Zum Beispiel können wir im Maximum-Likelihood-Training Loss_{NSP} definieren als

$$\text{Loss}_{\text{NSP}} \quad = \quad -\log \Pr(c_{\text{gold}}|\mathbf{h}_{\text{cls}}) \tag{1.19}$$

wobei c_{gold} das korrekte Label für dieses Beispiel ist.

1.3.1.2 Modellaufbau

Wie in Abbildung 1.6 gezeigt, basieren BERT-Modelle auf der Standard-Transformer-Encoder-Architektur. Die Eingabe ist eine Sequenz von Embeddings, wobei jedes die Summe aus dem Token-Embedding, dem Positions-Embedding und dem Segment-Embedding ist.

$$\mathbf{e} \quad = \quad \mathbf{x} + \mathbf{e}_{\text{pos}} + \mathbf{e}_{\text{seg}} \tag{1.20}$$

Sowohl das Token-Embedding ($\mathbf{x}$) als auch das Positions-Embedding ($\mathbf{e}_{\text{pos}}$) sind regulär, wie bei Transformer-Modellen. Das Segment-Embedding ($\mathbf{e}_{\text{seg}}$) ist eine neue Art von Embedding, das anzeigt, ob ein Token zu Sent_A oder Sent_B gehört. Dies kann durch das folgende Beispiel veranschaulicht werden.

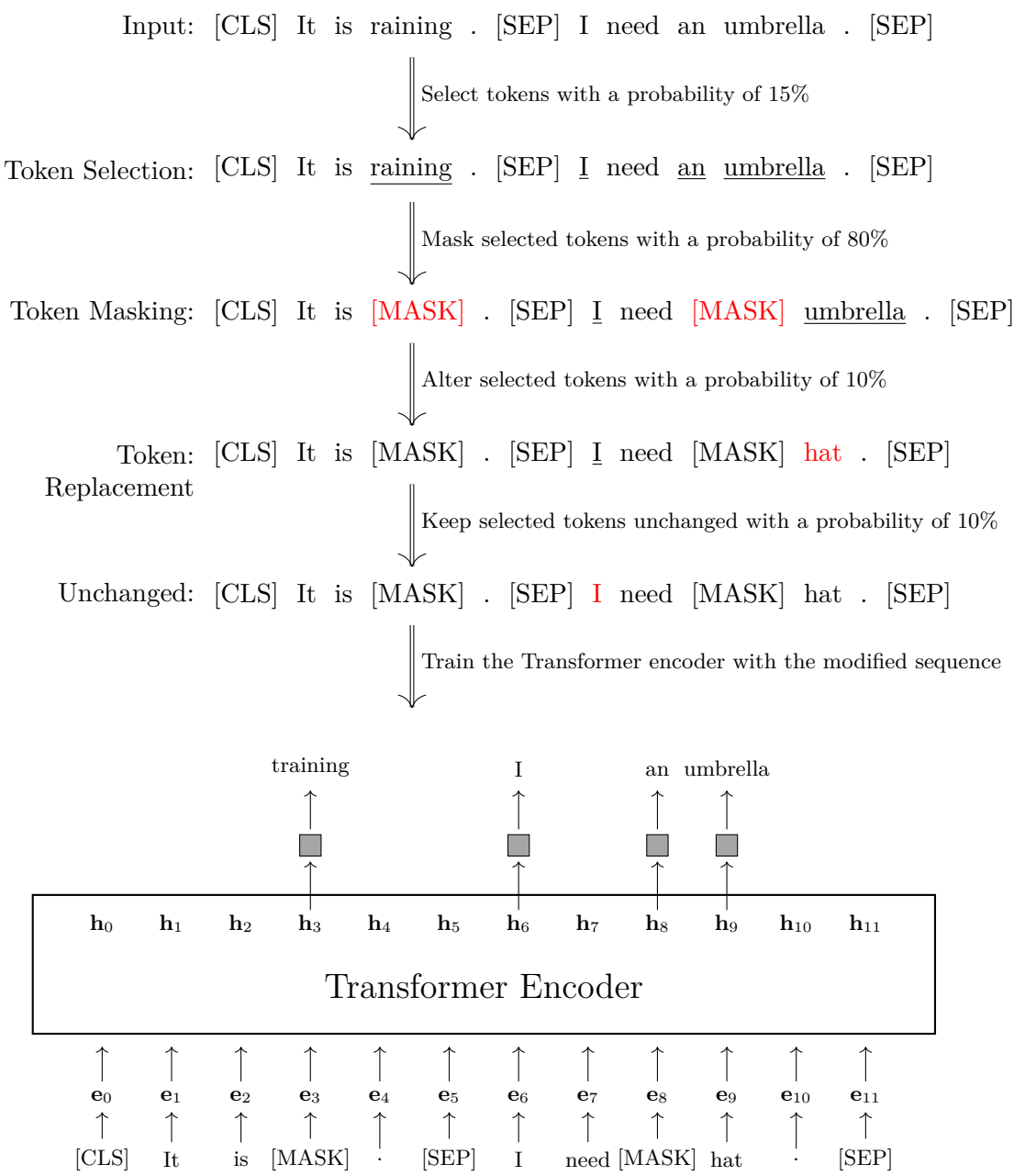


Abbildung 1.5: Ein laufendes Beispiel für die maskierte Sprachmodellierung im BERT-Stil. Zuerst werden 15% der Tokens zufällig ausgewählt. Diese ausgewählten Tokens werden dann auf eine von drei Arten verarbeitet: durch ein [MASK]-Token ersetzt (in 80% der Fälle), durch ein zufälliges Token ersetzt (in 10% der Fälle) oder unverändert beibehalten (in 10% der Fälle). Das Modell wird trainiert, diese ausgewählten Tokens basierend auf der modifizierten Sequenz vorherzusagen. $\mathbf{e}_i$ repräsentiert die Einbettung des Tokens an der Position i. Graue Kästen stellen die Softmax-Schichten dar.

Token	[CLS]	It	is	raining	.	[SEP]	I	need	an	umbrella	.	[SEP]
$\mathbf{x}$	$\mathbf{x}_0$	$\mathbf{x}_1$	$\mathbf{x}_2$	$\mathbf{x}_3$	$\mathbf{x}_4$	$\mathbf{x}_5$	$\mathbf{x}_6$	$\mathbf{x}_7$	$\mathbf{x}_8$	$\mathbf{x}_9$	$\mathbf{x}_{10}$	$\mathbf{x}_{11}$
$\mathbf{e}_{\text{pos}}$	PE(0)	PE(1)	PE(2)	PE(3)	PE(4)	PE(5)	PE(6)	PE(7)	PE(8)	PE(9)	PE(10)	PE(11)
$\mathbf{e}_{\text{seg}}$	$\mathbf{e}_A$	$\mathbf{e}_A$	$\mathbf{e}_A$	$\mathbf{e}_A$	$\mathbf{e}_A$	$\mathbf{e}_A$	$\mathbf{e}_B$	$\mathbf{e}_B$	$\mathbf{e}_B$	$\mathbf{e}_B$	$\mathbf{e}_B$	$\mathbf{e}_B$

Der Hauptteil von BERT-Modellen ist ein mehrschichtiges Transformer-Netzwerk.

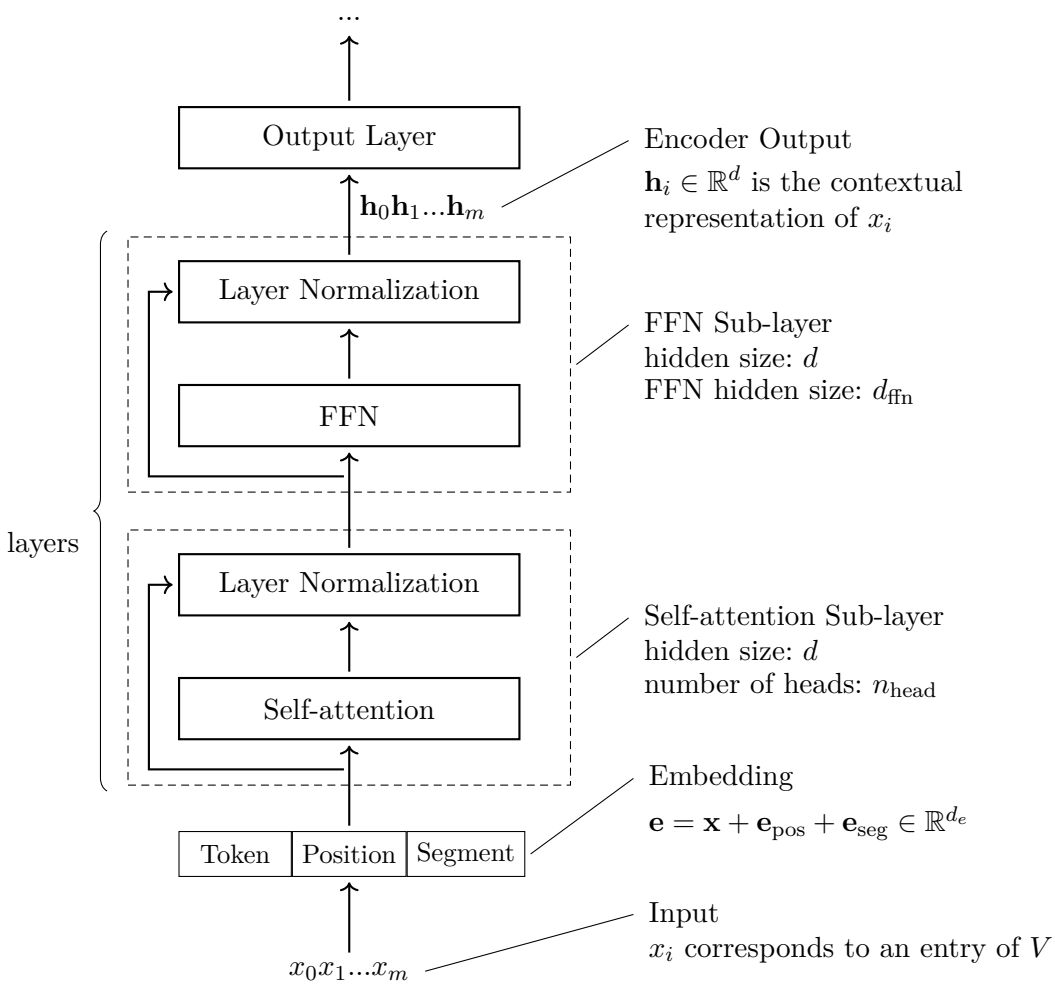


Abbildung 1.6: Die Modellarchitektur von BERT (Transformer -Encoder). Die Eingabe-Tokens werden zunächst als Embeddings dargestellt, von denen jedes die Summe aus dem entsprechenden Token-Embedding, Positions-Embedding und Segment-Embedding ist. Anschließend wird die Embedding-Sequenz von einem Stapel von Transformer -Schichten verarbeitet. Jede Schicht in diesem Stapel enthält eine Self-Attention-Unterschicht und eine FFN-Unterschicht. Die Ausgabe des BERT-Modells ist eine Sequenz von Vektoren, die von der letzten Transformer -Schicht erzeugt wird.

Eine Transformer-Schicht besteht aus einer Self-Attention-Unterschicht und einer FFN-Unterschicht. Beide folgen der Post-Norm-Architektur: output = LNorm(F(input)+input), wobei $F(\cdot)$ die Kernfunktion der Unterschicht ist (entweder ein Self-Attention-Modell oder ein FFN) und LNorm$(\cdot)$ die Layer-Normalisierungseinheit ist. Typischerweise werden mehrere Transformer-Schichten gestapelt, um ein tiefes Netzwerk zu bilden. An jeder Position der Sequenz ist die Ausgabedarstellung ein reellwertiger Vektor, der von der letzten Schicht des Netzwerks erzeugt wird.

Bei der Entwicklung von BERT-Modellen sind mehrere Aspekte zu berücksichtigen.

- Vokabulargröße ($|V|$). In Transformern wird jedes Eingabe-Token als ein Eintrag in einem Vokabular V dargestellt. Große Vokabulare können mehr Oberflächenformvarianten von Wörtern abdecken, führen aber möglicherweise zu einem erhöhten Speicherbedarf.

- Einbettungsgröße (d_e). Jedes Token wird als ein d_e-dimensionaler reellwertiger Vektor dargestellt. Wie oben dargestellt, ist dieser Vektor die Summe der Token-Einbettung, der Positionseinbettung und der Segmenteinbettung, die alle ebenfalls d_e-dimensionale reellwertige Vektoren sind.

- Verborgene Dimension (d). Die Ein- und Ausgabe einer Unterschicht haben die Dimension d. Zudem sind die meisten verborgenen Zustände einer Unterschicht d-dimensionale Vektoren. Im Allgemeinen kann d grob als die Breite des Netzwerks angesehen werden.

- Anzahl der Köpfe (n_{head}). In Selbst-Aufmerksamkeits-Unterschichten muss die Anzahl der in der Multi-Head-Selbst-Aufmerksamkeit verwendeten Köpfe spezifiziert werden. Je größer diese Zahl, desto mehr Unterräume, in denen die Aufmerksamkeit ausgeführt wird. In praktischen Systemen setzen wir oft $n_{\text{head}} \geq 4$.

- Verborgene FFN-Dimension (d_{ffn}). Die Dimension der verborgenen Schicht der in Transformern verwendeten FFNs ist typischerweise größer als d. Eine typische Einstellung ist beispielsweise $d_{\text{ffn}} = 4d$. Bei größeren Transformern, wie z.B. neueren großen Modellen, kann d_{ffn} auf einen sehr großen Wert gesetzt werden.

- Modelltiefe (L). Die Verwendung tiefer Netzwerke ist eine effektive Methode, um die Ausdruckskraft von Transformern zu verbessern. Bei BERT-Modellen wird L typischerweise auf 12 oder 24 gesetzt. Jedoch sind auch Netzwerke mit noch größerer Tiefe realisierbar und können für weitere Verbesserungen eingesetzt werden.

Unterschiedliche Einstellungen dieser Hyperparameter führen zu unterschiedlichen Modellgrößen. Es gibt zwei weit verbreitete BERT-Modelle.

- $\text{BERT}_{\text{base}}$: $d = 768$, $L = 12$, $n_{\text{head}} = 12$, Gesamtzahl der Parameter = 110M.

- $\text{BERT}_{\text{large}}$: $d = 1,024$, $L = 24$, $n_{\text{head}} = 16$, Gesamtzahl der Parameter = 340M.

Das Trainieren von BERT-Modellen folgt dem Standard-Trainingsprozess von Transformern. Das Trainieren größerer Modelle wie $\text{BERT}_{\text{large}}$ erfordert mehr Trainingsaufwand und Zeit. Dies ist ein häufiges Problem beim Vortraining, insbesondere wenn ein Modell mit einer sehr großen Datenmenge trainiert wird. In der Praxis gibt es oft Überlegungen zur Trainingseffizienz. Eine gängige Praxis ist es beispielsweise, ein BERT-Modell zunächst auf relativ kurzen Sequenzen für eine große Anzahl von Trainingsschritten zu trainieren und es dann für die verbleibenden Trainingsschritte auf Sequenzen voller Länge weiter zu trainieren.

1.3.2 Mehr Training und größere Modelle

BERT ist ein Meilensteinmodell im NLP, das viele nachfolgende Bemühungen zu seiner Verbesserung anregte. Eine Richtung ist die Skalierung des Modells selbst, was die Vergrößerung der Trainingsdaten und die Entwicklung größerer Modelle einschließt.

RoBERTa, eine Erweiterung des Standard-BERT-Modells, ist ein Beispiel für solche Bemühungen [Liu et al., 2019]. Es führt zwei wesentliche Verbesserungen ein. Erstens kann die einfache Verwendung von mehr Trainingsdaten und mehr Rechenleistung BERT-Modelle verbessern, ohne dass die Modellarchitekturen geändert werden müssen. Zweitens verringert das Entfernen des NSP-Verlusts die Leistung bei nachgelagerten Aufgaben nicht, wenn das Training skaliert wird. Diese Ergebnisse legen nahe, eine allgemeine Richtung des Vortrainings zu erkunden: Wir können das Vortraining weiter verbessern, indem wir es bei einfachen Vortrainingsaufgaben skalieren.

Ein zweiter Ansatz zur Verbesserung von BERT-Modellen besteht darin, die Anzahl der Modellparameter zu erhöhen. Zum Beispiel wird in der Arbeit von He et al. [2021] ein BERT-ähnliches Modell mit 1,5 Milliarden Parametern erstellt, indem sowohl die Modelltiefe als auch die Größe der verborgenen Schicht erhöht werden. Jedoch bringt die Skalierung von BERT und verschiedenen anderen vortrainierten Modellen neue Herausforderungen im Training mit sich. Zum Beispiel wird das Training sehr großer Modelle oft instabil und konvergiert schwer. Dies macht das Problem komplizierter und erfordert eine sorgfältige Berücksichtigung verschiedener Aspekte, einschließlich Modellarchitektur, paralleler Berechnung, Parameterinitialisierung und so weiter. In einem anderen Beispiel trainierten Shoeybi et al. [2019] erfolgreich ein BERT-ähnliches Modell mit 3,9 Milliarden Parametern, wobei Hunderte von GPUs verwendet wurden, um den gestiegenen Rechenaufwand zu bewältigen.

1.3.3 Effizientere Modelle

Im Vergleich zu seinen Vorgängern ist BERT ein relativ großes Modell für die Zeit, zu der es vorgeschlagen wurde. Diese Zunahme der Modellgröße führt zu einem höheren Speicherbedarf und einer daraus folgenden Verlangsamung der Systemleistung. Die Entwicklung kleinerer und schnellerer BERT-Modelle ist Teil der umfassenderen Herausforderung, effiziente Transformer zu erstellen, was in den Arbeiten von Tay et al. [2020] und Xiao and Zhu [2023] ausführlich diskutiert wurde. Eine tiefergehende Diskussion dieses allgemeinen Themas würde jedoch den Rahmen unserer aktuellen Erörterung sprengen. Hier betrachten wir stattdessen einige effiziente Varianten von BERT.

Für NLP-Forscher sind bei der Entwicklung effizienter BERT-Modelle mehrere Forschungsstränge von Interesse. Erstens zeigen Arbeiten zur Wissensdestillation, wie das Trainieren von Studentenmodellen mit der Ausgabe von gut trainierten Lehrermodellen, dass kleinere BERT-Modelle durch die Übertragung von Wissen von größeren BERT-Modellen erhalten werden können. Da BERT-Modelle mehrschichtige Netzwerke mit verschiedenen Arten von Schichten sind, kann die Wissensdestillation auf verschiedenen Repräsentationsebenen angewendet werden. Zum Beispiel ist es über die Destillation von Wissen aus den Ausgabeschichten hinaus auch möglich, einen Trainingsverlust einzubeziehen, der den Unterschied in der Ausgabe der verborgenen Schichten zwischen Lehrer- und Studentenmodellen misst [Sun et al., 2020; Jiao et al., 2020]. Tatsächlich ist die Wissensdestillation eine der am weitesten verbreiteten Techniken zum Lernen kleiner vortrainierter Modelle.

Zweitens können herkömmliche Modellkompressionsmethoden direkt zur Komprimierung von BERT-Modellen angewendet werden. Ein gängiger Ansatz ist die Verwendung

allgemeiner Pruning-Methoden, um die Transformer-Kodierernetzwerke zu beschneiden [Gale et al., 2019]. Dies beinhaltet im Allgemeinen das Entfernen ganzer Schichten [Fan et al., 2019] oder eines bestimmten Prozentsatzes von Parametern in den Netzwerken [Sanh et al., 2020; Chen et al., 2020]. Pruning ist auch auf Multi-Head-Attention-Modelle anwendbar. Zum Beispiel zeigen Michel et al. [2019], dass das Entfernen einiger der Köpfe die Leistung von BERT-Modellen nicht signifikant verringert, aber die Inferenz dieser Modelle beschleunigt. Ein weiterer Ansatz zur Komprimierung von BERT-Modellen ist die Quantisierung [Shen et al., 2020]. Durch die Darstellung von Modellparametern als Zahlen mit geringer Präzision können die Modelle stark komprimiert werden. Obwohl diese Methode nicht spezifisch für BERT-Modelle ist, erweist sie sich als wirksam für große Transformer-basierte Architekturen.

Drittens, da BERT-Modelle relativ tiefe und große Netzwerke sind, verwendet ein weiterer Forschungsstrang dynamische Netzwerke, um diese Modelle für eine effiziente Inferenz anzupassen. Eine Idee in diesem Paradigma ist es, die Schichten zur Verarbeitung eines Tokens dynamisch auszuwählen. Zum Beispiel verlassen wir in tiefenadaptiven Modellen bei einer optimalen Tiefe die Verarbeitung und überspringen somit die restlichen Schichten im Schichtenstapel [Xin et al., 2020; Zhou et al., 2020]. Ähnlich können wir längenadaptive Modelle entwickeln, bei denen die Länge der Eingabesequenz dynamisch angepasst wird. Zum Beispiel können wir einige der Tokens in der Eingabesequenz überspringen, sodass das Modell die Rechenlast bei weniger wichtigen Tokens reduzieren und die Gesamteffizienz steigern kann.

Viertens ist es auch möglich, Parameter über Schichten hinweg zu teilen, um die Größe von BERT-Modellen zu reduzieren. Eine einfache Möglichkeit hierfür ist, die Parameter einer ganzen Transformer-Schicht über den gesamten Schichtenstapel hinweg zu teilen [Dehghani et al., 2018; Lan et al., 2020]. Zusätzlich zur reduzierten Anzahl von Parametern ermöglicht dies die Wiederverwendung derselben Schicht in einem mehrschichtigen Transformer-Netzwerk, was zur Testzeit zu Einsparungen beim Speicherbedarf führt.

1.3.4 Mehrsprachige Modelle

Das ursprüngliche BERT-Modell war hauptsächlich auf Englisch ausgerichtet. Kurz nachdem dieses Modell vorgeschlagen wurde, wurde es auf viele Sprachen erweitert. Eine einfache Möglichkeit hierfür ist die Entwicklung eines separaten Modells für jede Sprache. Ein anderer Ansatz, der in jüngsten Arbeiten zu großen Sprachmodellen populärer geworden ist, besteht darin, mehrsprachige Modelle direkt auf Daten aus allen Sprachen zu trainieren. Als Reaktion darauf wurden mehrsprachige BERT (mBERT) Modelle entwickelt, indem sie auf Texten aus 104 Sprachen trainiert wurden [6]. Der Hauptunterschied zu einsprachigen BERT-Modellen besteht darin, dass mBERT-Modelle größere Vokabulare verwenden, um Tokens aus mehreren Sprachen abzudecken. Dadurch werden die Repräsentationen von Tokens aus verschiedenen Sprachen in denselben Raum abgebildet, was den Wissensaustausch zwischen den Sprachen über dieses universelle Repräsentationsmodell ermöglicht.

Eine wichtige Anwendung von mehrsprachig vortrainierten Modellen ist das kreuzlinguale Lernen. Im kreuzlingualen Setting lernen wir ein Modell für Aufgaben in einer

[6] https://github.com/google-research/bert/

Sprache und wenden es auf dieselben Aufgaben in einer anderen Sprache an. Bei der kreuzlingualen Textklassifikation zum Beispiel stimmen wir ein mehrsprachig vortrainiertes Modell auf englisch annotierten Dokumenten fein. Anschließend verwenden wir das feinabgestimmte Modell zur Klassifizierung chinesischer Dokumente.

Eine Verbesserung für mehrsprachig vortrainierte Modelle wie mBERT ist die Einführung bilingualer Daten in das Vortraining. Anstatt ausschließlich auf einsprachigen Daten aus mehreren Sprachen zu trainieren, modelliert das bilinguale Training explizit die Beziehung zwischen Tokens in zwei Sprachen. Das resultierende Modell verfügt über angeborene kreuzlinguale Transferfähigkeiten und kann somit leicht an verschiedene Sprachen angepasst werden. Lample and Conneau [2019] schlagen einen Ansatz zum Vortraining von kreuzlingualen Sprachmodellen (XLMs) vor. In ihrer Arbeit kann ein kreuzlinguales Sprachmodell entweder nach der Methode der kausalen Sprachmodellierung oder der maskierten Sprachmodellierung trainiert werden. Für das Vortraining mit maskierter Sprachmodellierung wird das Modell als Encoder behandelt. Das Trainingsziel ist dasselbe wie bei BERT: Wir maximieren die Wahrscheinlichkeiten einiger zufällig ausgewählter Tokens, die in der Eingabe entweder maskiert, durch zufällige Tokens ersetzt oder unverändert gelassen werden. Wenn wir bilinguale Daten im Vortraining berücksichtigen, sampeln wir jedes Mal ein Paar ausgerichteter Sätze. Anschließend werden die beiden Sätze zu einer einzigen Sequenz zusammengefügt, die für das Training verwendet wird. Betrachten wir zum Beispiel ein englisch-chinesisches Satzpaar

鲸鱼 是 哺乳 动物 。 ↔ Whales are mammals .

Wir können sie zu einer Sequenz zusammenpacken, wie hier gezeigt

[CLS] 鲸鱼 是 哺乳 动物 。 [SEP] Whales are mammals . [SEP]

Wir wählen dann einen bestimmten Prozentsatz der Tokens aus und ersetzen sie durch [MASK].

[CLS] [MASK] 是 [MASK] 动物 。 [SEP] Whales [MASK] [MASK] . [SEP]

Das Ziel des Vortrainings ist es, das Produkt der Wahrscheinlichkeiten der maskierten Tokens gegeben die obige Sequenz zu maximieren. Durch diese Art des Trainings kann das Modell lernen, sowohl die englischen als auch die chinesischen Sequenzen zu repräsentieren und die Korrespondenzen zwischen den Tokens in den beiden Sprachen zu erfassen. Zum Beispiel könnte die Vorhersage des chinesischen Tokens 鲸鱼 die Information des englischen Tokens Whales erfordern. Die Angleichung der Repräsentationen der beiden Sprachen verwandelt das Modell im Wesentlichen in ein „Übersetzungs“-Modell. Daher wird dieses Trainingsziel auch als Übersetzungs-Sprachmodellierung bezeichnet. Abbildung 1.7 zeigt eine Veranschaulichung dieses Ansatzes.

Ein Vorteil von mehrsprachig vortrainierten Modellen ist ihre inhärente Fähigkeit, mit Code-Switching umzugehen. In der NLP und der Linguistik bezieht sich Code-Switching

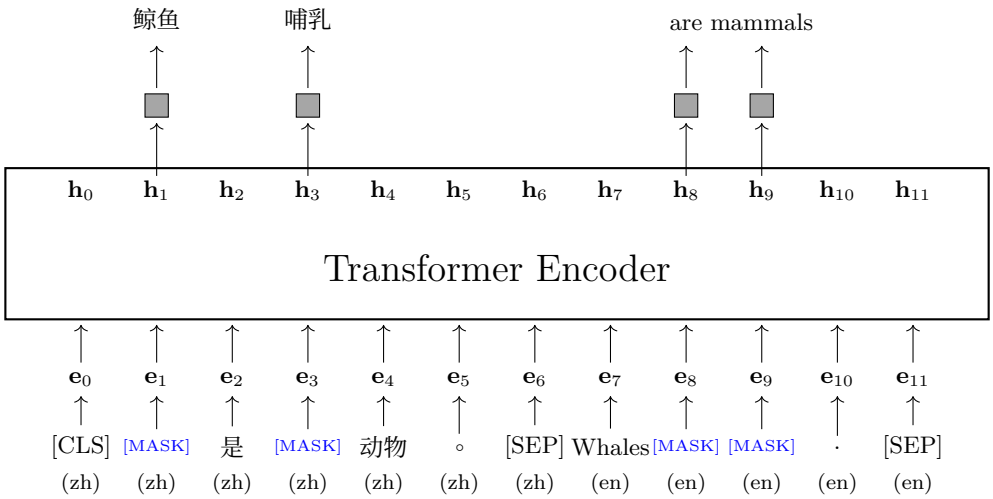


Abbildung 1.7: Eine Veranschaulichung des Translation Language Modelings. Zum besseren Verständnis stellen wir ein einfaches Beispiel dar, in dem alle ausgewählten Tokens maskiert sind. Das Modell wird trainiert, diese maskierten Tokens vorherzusagen. Da die Sequenz Tokens in zwei Sprachen enthält, ermöglicht die Vorhersage eines Tokens in einer Sprache den Zugriff auf Tokens in der anderen Sprache, wodurch eine sprachübergreifende Modellierung ermöglicht wird. In der Arbeit von Lample and Conneau [2019] ist eine Eingabeeinbettung (d. h. $\mathbf{e}_i$) die Summe aus der Token-Einbettung, der Positionseinbettung und der Spracheinbettung. Dies erfordert, dass jedem Token ein Sprach-Label zugewiesen wird. So können wir Tokens in verschiedenen Sprachen unterscheiden. Beim mehrsprachigen Pre-Training, insbesondere bei Arbeiten, die gemeinsame Vokabulare verwenden, ist die Angabe der Sprache, zu der ein Token gehört, nicht notwendig. Die Verwendung von Spracheinbettungen erschwert wiederum den Umgang mit Code-Switching. Daher gehen wir hier davon aus, dass alle Token-Repräsentationen sprachunabhängig sind.

auf den Wechsel zwischen Sprachen innerhalb eines Textes. Zum Beispiel ist der folgende Text ein gemischtsprachiger Text, der sowohl Chinesisch als auch Englisch enthält:

周末 我们 打算 去 做 hiking ， 你 想 一起 来 吗 ？

(We plan to go hiking this weekend, would you like to join us?)

Bei mehrsprachig vortrainierten Modellen müssen wir nicht identifizieren, ob ein Token chinesisch oder englisch ist. Stattdessen ist jedes Token nur ein Eintrag im gemeinsamen Vokabular. Man kann sich das so vorstellen, als würde man eine „neue“Sprache schaffen, die alle Sprachen umfasst, die wir verarbeiten möchten.

Das Ergebnis des mehrsprachigen Vortrainings wird von mehreren Faktoren beeinflusst. Bei gegebener Modellarchitektur muss man die Größe des gemeinsamen Vokabulars, die Anzahl (oder den Prozentsatz) der Samples in jeder Sprache, die Größe des Modells und so weiter festlegen. Conneau et al. [2020] weisen auf mehrere interessante Probleme im Zusammenhang mit dem groß angelegten mehrsprachigen Vortraining für XLM-ähnliche Modelle hin. Erstens wird mit zunehmender Anzahl unterstützter Sprachen ein größeres Modell benötigt, um diese Sprachen zu bewältigen. Zweitens ist ein größeres gemeinsames Vokabular hilfreich, um die zunehmende Vielfalt der Sprachen zu modellieren. Drittens profitieren ressourcenarme Sprachen leichter vom kreuzlingualen Transfer aus ressourcenreichen Sprachen, insbesondere wenn ähnliche ressourcenreiche Sprachen am Vortraining beteiligt sind. Jedoch kann Interferenz auftreten, wenn das Modell über einen längeren

Zeitraum trainiert wird, was bedeutet, dass die Gesamtleistung des vortrainierten Modells ab einem bestimmten Punkt während des Vortrainings zu sinken beginnt. Daher muss man in praktischen Systemen das Vortraining möglicherweise frühzeitig beenden, um Interferenzen zu vermeiden.

1.4 Anwendung von BERT-Modellen

Sobald ein BERT-Modell vortrainiert ist, kann es zur Lösung von NLP-Problemen verwendet werden. BERT-Modelle sind jedoch nicht sofort bereit, spezifische nachgelagerte Aufgaben auszuführen. Im Allgemeinen ist zusätzliche Feinabstimmungsarbeit erforderlich, um sie anzupassen. Als erster Schritt benötigen wir einen Prädiktor, um die Ausgabe des Modells an das jeweilige Problem anzupassen. Sei $\text{BERT}_{\hat{\theta}}(\cdot)$ ein BERT-Modell mit vortrainierten Parametern $\hat{\theta}$ und $\text{Predict}_{\omega}(\cdot)$ ein Vorhersagenetzwerk mit Parametern ω. Durch die Integration des Vorhersagenetzwerks mit der Ausgabe des BERT-Modells entwickeln wir ein Modell, um die nachgelagerten Aufgaben zu bewältigen. Dieses Modell kann ausgedrückt werden als

$$\mathbf{y} \quad = \quad \text{Predict}_{\omega}(\text{BERT}_{\hat{\theta}}(\mathbf{x})) \tag{1.21}$$

wobei $\mathbf{x}$ die Eingabe und $\mathbf{y}$ die Ausgabe ist, die zum Problem passt. Bei Klassifikationsproblemen gibt das Modell beispielsweise eine Wahrscheinlichkeitsverteilung über die Labels aus.

Dann sammeln wir einen Satz von gelabelten Stichproben $\mathcal{D}$ und stimmen das Modell fein ab durch

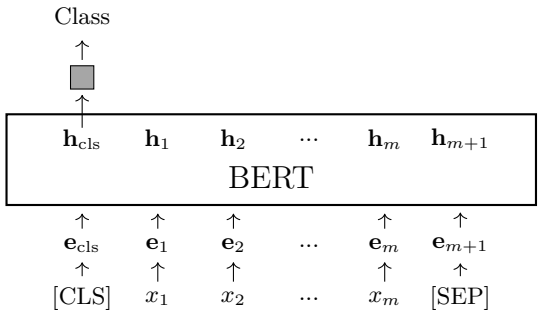
$$(\tilde{\omega}, \tilde{\theta}) \quad = \quad \underset{\omega, \hat{\theta}^{+}}{\arg\min} \sum_{(\mathbf{x}, \mathbf{y}_{\text{gold}}) \in \mathcal{D}} \text{Loss}(\mathbf{y}_{\omega, \hat{\theta}^{+}}, \mathbf{y}_{\text{gold}}) \tag{1.22}$$

wobei $(\mathbf{x}, \mathbf{y}_{\text{gold}})$ ein Tupel aus einer Eingabe und ihrer entsprechenden Ausgabe darstellt. Die Notation dieser Gleichung erscheint etwas kompliziert, aber der Trainings-/Abstimmungsprozess ist Standard. Wir optimieren das Modell, indem wir den Verlust über die Abstimmungsstichproben minimieren. Das Ergebnis sind die optimierten Parameter $\tilde{\omega}$ und $\tilde{\theta}$. Die Optimierung beginnt mit den vortrainierten Parametern $\hat{\theta}$. Hier verwenden wir $\hat{\theta}^{+}$, um anzuzeigen, dass die Parameter mit $\hat{\theta}$ initialisiert werden, und verwenden $\mathbf{y}_{\omega, \hat{\theta}^{+}}$, um die Modellausgabe zu bezeichnen, die mit den Parametern ω und $\hat{\theta}^{+}$ berechnet wird.

Mit den feingestimmten Parametern $\tilde{\omega}$ und $\tilde{\theta}$ können wir das Modell $\text{Predict}_{\tilde{\omega}}(\text{BERT}_{\tilde{\theta}}(\cdot))$ auf neue Daten derselben Aufgaben anwenden, für die das Modell feingestimmt wurde. Die Form der nachgelagerten Aufgaben bestimmt die Ein- und Ausgabeformate des Modells sowie die Architektur des Vorhersagenetzwerks. Im Folgenden listen wir einige Aufgaben auf, für die BERT-Modelle im Allgemeinen geeignet sind.

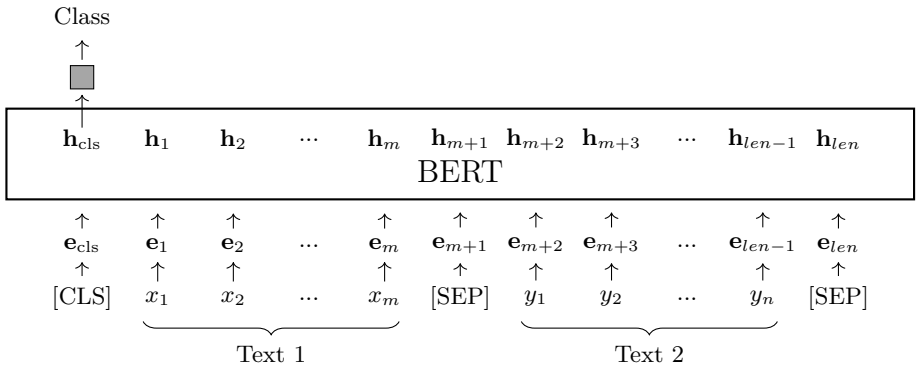
- Klassifikation (Einzeltext). Eine der am weitesten verbreiteten Anwendungen von BERT-Modellen ist die Textklassifikation. In dieser Aufgabe erhält ein BERT-Modell eine Sequenz von Tokens und kodiert sie als eine Sequenz von Vektoren. Der erste

Ausgabevektor $\mathbf{h}_{\text{cls}}$ (oder $\mathbf{h}_0$) wird typischerweise als Repräsentation des gesamten Textes verwendet. Das Vorhersagenetzwerk nimmt $\mathbf{h}_{\text{cls}}$ als Eingabe, um eine Verteilung von Labels zu erzeugen. Sei [CLS]$x_1 x_2 ... x_m$ ein Eingabetext. Unten folgt eine Illustration der auf BERT basierenden Textklassifikation.



Hier bezeichnet die graue Box das Vorhersagenetzwerk. Viele NLP-Probleme können als Textklassifikationsaufgaben kategorisiert werden, und es gibt mehrere Benchmarks zur Textklassifikation zur Bewertung vortrainierter Modelle. Zum Beispiel können wir Texte nach ihrer grammatikalischen Korrektheit (Grammatikalität) oder ihrem emotionalen Ton (Sentiment) klassifizieren [Socher et al., 2013; Warstadt et al., 2019]. Beachte, dass das Vorhersagenetzwerk jedes beliebige Klassifikationsmodell sein kann, wie ein tiefes neuronales Netz oder ein traditionelleres Klassifikationsmodell. Das gesamte Modell kann dann in der Art eines Standardklassifikationsmodells trainiert oder feinabgestimmt werden. Zum Beispiel kann das Vorhersagenetzwerk einfach eine Softmax-Schicht sein, und die Modellparameter können durch Maximierung der Wahrscheinlichkeiten der korrekten Labels optimiert werden.

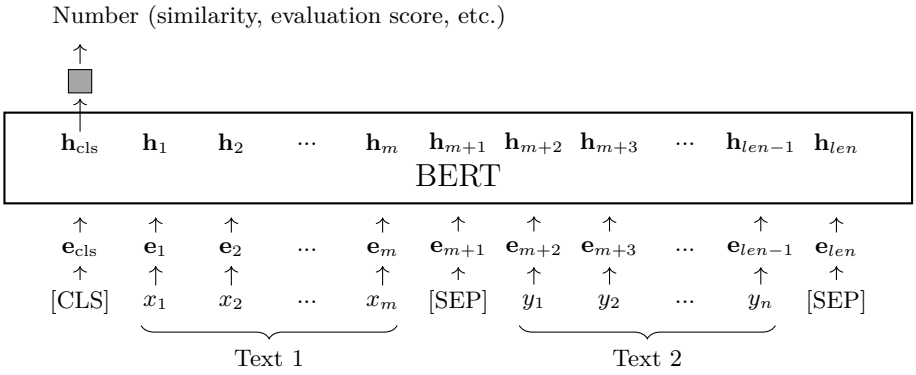
- Klassifikation (Textpaare). Die Klassifikation kann auch auf einem Paar von Texten durchgeführt werden. Angenommen, wir haben zwei Texte, $x_1 ... x_m$ und $y_1 ... y_n$. Wir können diese Texte zu einer einzigen Sequenz mit der Länge len zusammenfügen. Anschließend sagen wir ein Label für diese kombinierte Textsequenz basierend auf dem Vektor $\mathbf{h}_{\text{cls}}$ voraus, wie folgt:



wobei $len = n+m+2$ ist. Die Klassifikation von Textpaaren deckt mehrere Probleme ab, einschließlich der Beurteilung semantischer Äquivalenz (Bestimmung, ob zwei Texte semantisch gleichwertig sind) [Dolan and Brockett, 2005], der Beurteilung

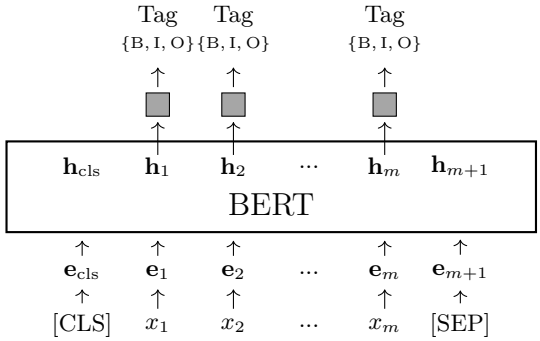
textlicher Folgerungen (Bestimmung, ob eine Hypothese logisch aus einer Prämisse abgeleitet oder gefolgert werden kann) [Bentivogli and Giampiccolo, 2011; Williams et al., 2018], der fundierten Commonsense-Inferenz (Bestimmung, ob ein Ereignis gegeben seinem Kontext wahrscheinlich eintreten wird) [Zellers et al., 2018] und der Frage-Antwort-Inferenz (Bestimmung, ob eine Antwort zu einer gegebenen Frage passt).

- Regression. Anstatt eine Label-Verteilung zu erzeugen, kann das Vorhersagenetzwerk eine reelle Punktzahl ausgeben. Zum Beispiel kann das System durch Hinzufügen einer Sigmoid-Schicht zum Vorhersagenetzwerk verwendet werden, um die Ähnlichkeit zwischen zwei gegebenen Sätzen zu berechnen. Die Architektur ist dieselbe wie bei BERT-basierten Klassifikationssystemen, nur die Ausgabeschicht wird geändert.



Durch training Oder feinschliff können wir den verlust des umkehrfunktion wie üblich minimieren.

- Sequenzkennzeichnung. Sequenzkennzeichnung ist ein maschinelles Lernverfahren, das auf eine breite Palette von NLP-Problemen anwendbar ist. Bei diesem Ansatz wird jedem Token in einer Eingabesequenz ein Label zugewiesen, und einige linguistische Annotationen können anschließend aus dieser Sequenz von Labels abgeleitet werden. Ein Beispiel für Sequenzkennzeichnung in NLP ist die Part-of-Speech-(POS)-Markierung. Wir versehen jedes Wort in einem Satz mit seinem entsprechenden POS-Tag. Ein weiteres Beispiel ist die Named-Entity-Recognition (NER), bei der jedes Wort mit einem NER-Tag versehen wird und benannte Entitäten mithilfe dieser Tags identifiziert werden. Unten folgt eine Illustration der Modellarchitektur für NER.

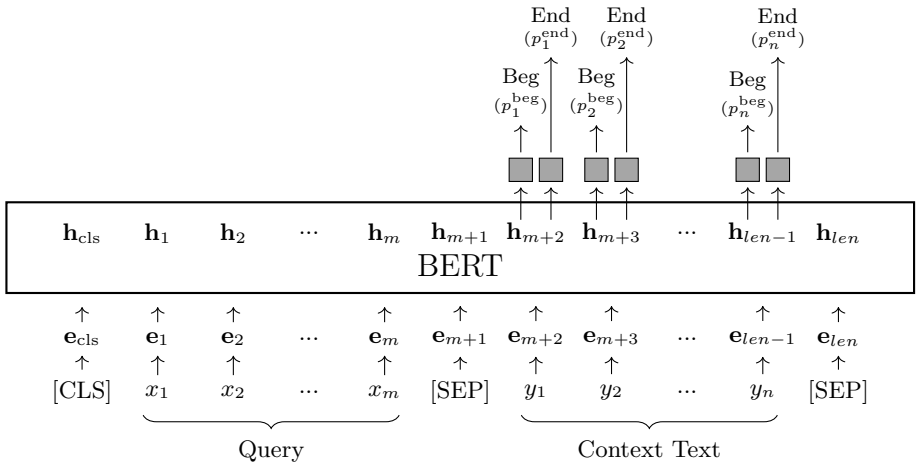


Hier ist $\{\mathrm{B}, \mathrm{I}, \mathrm{O}\}$ die Tag-Menge der NER. Zum Beispiel bedeutet B-ORG den Beginn einer Organisation, I-ORG bedeutet, dass das Wort innerhalb einer Organisation liegt, und O bedeutet, dass das Wort zu keiner benannten Entität gehört. Dieses NER-Modell kann an jeder Position eine Verteilung über die Tag-Menge ausgeben, bezeichnet als $\mathbf{p}_i$. Das Training oder Fine-Tuning des Modells kann über diese Verteilungen $\{\mathbf{p}_1, ..., \mathbf{p}_m\}$ durchgeführt werden. Zum Beispiel sei $p_i(\mathrm{tag}_i)$ die Wahrscheinlichkeit des korrekten Tags an Position i. Der Trainingsverlust kann als negative Likelihood definiert werden.

$$\text{Loss} \quad = \quad -\frac{1}{m}\sum_{i=1}^{m}\log p_i(\mathrm{tag}_i) \tag{1.23}$$

Die Bestimmung der besten Label-Sequenz für ein trainiertes NER-Modell ist ein gut untersuchtes Problem im NLP. Dies wird oft mittels dynamischer Programmierung erreicht, die im Kontext der Pfadsuche über ein Gitter eine lineare Komplexität aufweist [Huang, 2009].

- Span-Vorhersage. Einige NLP-Aufgaben erfordern die Vorhersage eines Spans in einem Text. Ein gängiges Beispiel ist das Leseverständnis. In dieser Aufgabe erhalten wir eine Abfrage $x_1...x_m$ und einen Kontexttext $y_1...y_n$. Das Ziel ist es, einen zusammenhängenden Span in $y_1...y_n$ zu identifizieren, der die Abfrage am besten beantwortet. Dieses Problem kann als eine Sequenzkennzeichnungs-ähnliche Aufgabe formuliert werden, bei der wir für jedes y_j ein Label vorhersagen, um den Beginn oder das Ende des Spans anzugeben. Nach Seo et al. [2017] fügen wir zwei Netzwerke auf dem BERT-Ausgang für y_j hinzu: eines zur Erzeugung der Wahrscheinlichkeit, dass y_j den Beginn des Spans darstellt (bezeichnet als p_j^{beg}), und eines zur Erzeugung der Wahrscheinlichkeit, dass y_j das Ende des Spans darstellt (bezeichnet als p_j^{end}). Die resultierende Modellarchitektur wird wie folgt dargestellt:



Wir packen die Abfrage und den Kontexttext zusammen, um die Eingabesequenz zu erhalten. Die Vorhersagenetzwerke werden nur auf die Ausgaben des Kontexttexts angewendet und erzeugen die Wahrscheinlichkeiten p_j^{beg} und p_j^{end} an jeder Position. Der Verlust kann berechnet werden, indem die negativen Log-Likelihoods der beiden

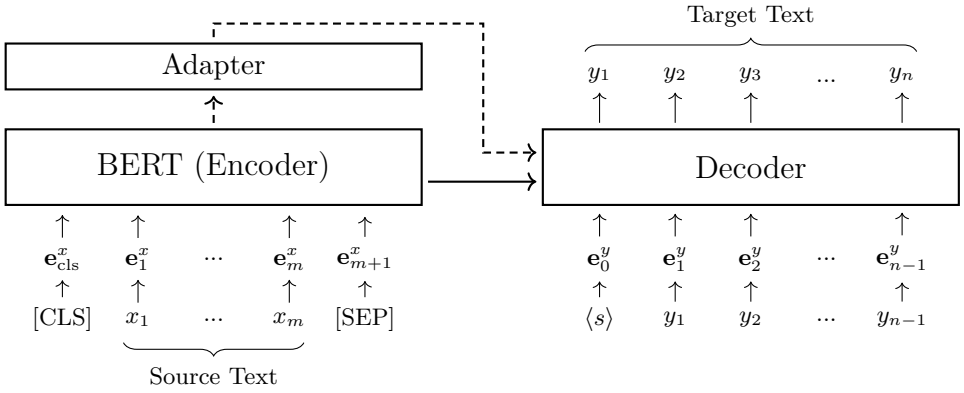
Modelle über den gesamten Kontexttext aufsummiert werden.

$$\text{Loss} \quad = \quad -\frac{1}{n}\sum_{j=1}^{n}\left(\log p_j^{\text{beg}} + \log p_j^{\text{end}}\right) \tag{1.24}$$

Wenn wir den test bestanden haben, sahen wir den perfekten bund

$$(\hat{j}_1, \hat{j}_2) \quad = \quad \underset{1\le j_1\le j_2\le n}{\arg\max} \left(\log p_{j_1}^{\text{beg}} + \log p_{j_2}^{\text{end}}\right) \tag{1.25}$$

- Kodierung für Encoder-Decoder-Modelle. Obwohl sich unser Fokus in diesem Abschnitt hauptsächlich auf Sprachverständnisprobleme gerichtet hat, ist es erwähnenswert, dass BERT-Modelle auf eine breitere Palette von NLP-Aufgaben angewendet werden können. Tatsächlich können BERT-Modelle in allen Szenarien verwendet werden, in denen ein Textstück kodiert werden muss. Eine Anwendung, die wir bisher nicht erwähnt haben, ist die Textgenerierung, die eine Reihe von Aufgaben umfasst, wie maschinelle Übersetzung, Zusammenfassung, Fragebeantwortung und Dialoggenerierung. Diese Aufgaben können als Sequenz-zu-Sequenz-Probleme formuliert werden: Wir verwenden einen Encoder, um den Quelltext zu repräsentieren, und einen Decoder, um den entsprechenden Zieltext zu erzeugen. Eine einfache Methode, BERT-Modelle anzuwenden, besteht darin, sie als Encoder zu betrachten. Vor dem Fine-Tuning können wir die Parameter des Encoders mit denen eines vortrainierten BERT-Modells initialisieren. Anschließend kann das Encoder-Decoder-Modell wie gewohnt auf Textpaaren feinabgestimmt werden. Im Folgenden wird die Architektur eines neuronalen maschinellen Übersetzungssystems gezeigt, bei dem ein BERT-Modell auf der Quellseite angewendet wird.



Hier bezeichnet $x_1...x_m$ die Quellsequenz, $y_1...y_n$ die Zielsequenz, $\mathbf{e}^x_1...\mathbf{e}^x_m$ die Embedding-Sequenz von $x_1...x_m$ und $\mathbf{e}^y_1...\mathbf{e}^y_n$ die Embedding-Sequenz von $y_1...y_n$. Der Adapter, der optional ist, wandelt die Ausgabe des BERT-Modells in eine Form um, die besser zum Decoder passt.

Die Feinabstimmung von BERT-Modellen ist ein kompliziertes technisches Problem, das von vielen Faktoren beeinflusst wird, wie z. B. der Menge der Feinabstimmungsdaten,

der Modellgröße und dem bei der Feinabstimmung verwendeten Optimierer. Im Allgemeinen möchten wir diese Modelle ausreichend feinabstimmen, damit sie bei den nachgelagerten Aufgaben gut abschneiden. Die Feinabstimmung von BERT-Modellen für spezifische Aufgaben kann jedoch zu Überanpassung führen, was wiederum ihre Fähigkeit zur Generalisierung auf andere Aufgaben verringert. Angenommen, wir haben ein BERT-Modell, das bei einer bestimmten Aufgabe gut abschneidet. Wenn wir es dann für neue Aufgaben feinabstimmen, kann dies seine Leistung bei der ursprünglichen Aufgabe verringern. Dieses Problem steht im Zusammenhang mit dem catastrophic forgetting-Problem beim kontinuierlichen Training, bei dem ein neuronales Netzwerk zuvor gelernte Informationen vergisst, wenn es mit neuen Stichproben aktualisiert wird. In praktischen Anwendungen ist eine gängige Methode zur Linderung des katastrophalen Vergessens, einige alte Daten in die Feinabstimmung einzubeziehen und das Modell mit vielfältigeren Daten zu trainieren. Außerdem kann man auf katastrophales Vergessen spezialisierte Methoden verwenden, wie z. B. Experience Replay [Rolnick et al., 2019] und Elastic Weight Consolidation [Kirkpatrick et al., 2017]. Der interessierte Leser sei für detailliertere Diskussionen zu diesem Thema im kontinuierlichen Lernen auf einige Übersichtsartikel verwiesen [Parisi et al., 2019; Wang et al., 2023a;e].

1.5 Zusammenfassung

In diesem Kapitel haben wir die allgemeine Idee des Pre-Training in der NLP erörtert. Insbesondere haben wir das selbstüberwachte Pre-Training und seine Anwendung auf reine Encoder-, reine Decoder- sowie Encoder-Decoder-Architekturen besprochen. Darüber hinaus haben wir eine Vielzahl von Pre-Training-Aufgaben für diese Architekturen vorgestellt und verglichen. Als Beispiel wird BERT verwendet, um zu veranschaulichen, wie Sequenzmodelle durch maskierte Sprachmodellierung vortrainiert und auf verschiedene nachgelagerte Aufgaben angewendet werden.

Die letzten Jahre haben bemerkenswerte Fortschritte in der NLP gezeigt, angeführt durch den großflächigen Einsatz von selbstüberwachtem Pre-Training. Und weitreichende Fortschritte werden bei vielen Aufgaben erzielt, nicht nur in der NLP, sondern auch in der Computer Vision und anderen Bereichen der KI. Eine Idee hinter diesen Fortschritten ist, dass eine beträchtliche Menge an Wissen über die Welt gelernt werden kann, indem man diese KI-Systeme einfach auf riesigen Mengen ungelabelter Daten trainiert. Zum Beispiel kann ein Sprachmodell allgemeines Wissen über eine Sprache lernen, indem es wiederholt maskierte Wörter in umfangreichen Texten vorhersagt. Infolgedessen kann dieses vortrainierte Sprachmodell als Basismodell dienen, das leicht angepasst werden kann, um spezifische nachgelagerte NLP-Aufgaben zu bewältigen. Dieser Paradigmenwechsel in der NLP hat die Entwicklung unglaublich leistungsfähiger Systeme für Sprachverständnis, -erzeugung und -schlussfolgerung ermöglicht [Manning, 2022]. Es ist jedoch wichtig anzuerkennen, dass wir uns noch in den Anfängen der Schaffung wirklich intelligenter Systeme befinden und es noch ein langer Weg ist. Dennoch hat das großangelegte Pre-Training eine Tür zu intelligenten Systemen geöffnet, deren Entwicklung Forscher lange angestrebt haben, obwohl mehrere wichtige Forschungsbereiche noch zur Erkundung offen stehen, wie das effiziente Erlernen von Intelligenz unter Verwendung von Daten angemessen kleiner Größe und der Erwerb komplexer Schlussfolgerungs- und Planungsfähigkeiten.

Beachten Sie, dass dieses Kapitel hauptsächlich einführend ist und nicht alle Aspekte des Pre-Training abdecken kann. Zum Beispiel gibt es viele Methoden, um ein vortrainiertes Modell feinabzustimmen, die verschiedene Möglichkeiten bieten, das Modell besser an unterschiedliche Situationen anzupassen. Darüber hinaus werden große Sprachmodelle, die als eine der bedeutendsten Errungenschaften der KI in den letzten Jahren gelten, in diesem Abschnitt übersprungen. Die Diskussion dieser Themen überlassen wir den folgenden Kapiteln.

Kapitel 2

Generative Modelle

Einer der bedeutendsten Fortschritte in der NLP in den letzten Jahren ist möglicherweise die Entwicklung großer Sprachmodelle (LLMs). Dies hat dazu beigetragen, Systeme zu schaffen, die natürliche Sprachen wie Menschen verstehen und generieren können. Es wurde sogar festgestellt, dass diese Systeme in der Lage sind, zu schlussfolgern, was als ein sehr anspruchsvolles KI-Problem gilt. Mit diesen Errungenschaften hat die NLP große Fortschritte gemacht und ist in eine neue Ära der Forschung eingetreten, in der schwierige Probleme gelöst werden, wie zum Beispiel der Aufbau von Konversationssystemen, die reibungslos mit Menschen kommunizieren können.

Das Konzept der Sprachmodellierung oder probabilistischen Sprachmodellierung geht auf frühe Experimente von Shannon [1951] zurück. In seiner Arbeit wurde ein Sprachmodell entwickelt, um die Vorhersagbarkeit des Englischen zu schätzen — wie gut kann der nächste Buchstabe eines Textes vorhergesagt werden, wenn die vorhergehenden N Buchstaben bekannt sind. Obwohl Shannons Experimente vorläufig waren, sind die grundlegenden Ziele und Methoden der Sprachmodellierung seitdem über die Jahrzehnte weitgehend unverändert geblieben. Über einen recht langen Zeitraum, insbesondere vor 2010, war der dominierende Ansatz zur Sprachmodellierung der n-Gramm-Ansatz [Jurafsky and Martin, 2008]. Bei der n-Gramm-Sprachmodellierung schätzen wir die Wahrscheinlichkeit eines Wortes gegeben seine vorhergehenden $n-1$ Wörter, und somit kann die Wahrscheinlichkeit einer Sequenz durch das Produkt einer Reihe von n-Gramm-Wahrscheinlichkeiten angenähert werden. Diese Wahrscheinlichkeiten werden typischerweise durch das Sammeln geglätteter relativer Häufigkeiten von n-Grammen im Text geschätzt. Obwohl ein solcher Ansatz unkompliziert und einfach ist, wurde er in der NLP ausgiebig verwendet. Zum Beispiel hing der Erfolg moderner statistischer Spracherkennungs- und maschineller Übersetzungssysteme maßgeblich von der Nutzung von n-Gramm-Sprachmodellen ab [Jelinek, 1998; Koehn, 2010].

Die Anwendung neuronaler Netze auf die Sprachmodellierung war lange Zeit attraktiv, aber ein echter Durchbruch gelang erst mit dem Fortschritt der Deep-Learning-Techniken. Eine vielzitierte Studie ist die Arbeit von Bengio et al. [2003], in der n-Gramm-Wahrscheinlichkeiten über ein Feed-Forward-Netzwerk modelliert und durch das Training des Netzwerks auf End-to-End-Weise gelernt werden. Ein Nebenprodukt dieses neuronalen Sprachmodells sind die verteilten Repräsentationen von Wörtern, bekannt als Worteinbettungen. Anstatt Wörter als diskrete Variablen darzustellen, bilden Worteinbettungen Wörter in niedrigdimensionale reellwertige Vektoren ab, was es ermöglicht, die Bedeutungen von Wörtern und Wort-n-Grammen in einem kontinuierlichen Repräsentationsraum zu berechnen. Infolgedessen sind Sprachmodelle nicht mehr mit dem Fluch der Dimensionalität belastet, sondern können exponentiell viele n-Gramme über ein kompaktes und dichtes neuronales Modell repräsentieren.

Die Idee, Wortrepräsentationen durch neuronale Sprachmodelle zu lernen, inspirierte die nachfolgende Forschung zum Repräsentationslernen in der NLP. Dieser Ansatz stieß jedoch in den ersten Jahren nach seinem Vorschlag auf kein signifikantes Interesse bei der Entwicklung von NLP-Systemen. Ab etwa 2012 wurden jedoch Fortschritte beim Lernen

von Worteinbettungen aus umfangreichen Texten durch einfache Wortvorhersageaufgaben erzielt. Mehrere Methoden, wie Word2Vec, wurden vorgeschlagen, um solche Einbettungen effektiv zu lernen, die dann erfolgreich in einer Vielzahl von NLP-Systemen angewendet wurden [Mikolov et al., 2013a;b]. Als Ergebnis dieser Fortschritte begannen Forscher, über das Lernen von Repräsentationen von Sequenzen unter Verwendung leistungsfähigerer Sprachmodelle nachzudenken, wie z.B. LSTM-basierte Modelle [Sutskever et al., 2014; Peters et al., 2018]. Und weitere Fortschritte und das Interesse an der Sequenzrepräsentation explodierten, nachdem der Transformer vorgeschlagen wurde. Neben dem Aufstieg des Transformer wurde das Konzept der Sprachmodellierung verallgemeinert, um Modelle zu umfassen, die lernen, Wörter auf verschiedene Weisen vorherzusagen. Viele leistungsstarke Transformer-basierte Modelle wurden mit diesen Wortvorhersageaufgaben vortrainiert und erfolgreich auf eine Vielzahl von nachgelagerten Aufgaben angewendet [Devlin et al., 2019].

Tatsächlich hat das Training von Sprachmodellen auf großen Datenmengen die NLP-Forschung in aufregende Zeiten geführt. Obwohl die Sprachmodellierung lange Zeit als eine grundlegende Technik ohne direkten Bezug zu den Zielen der künstlichen Intelligenz angesehen wurde, auf die Forscher gehofft hatten, hilft sie uns, die Entstehung intelligenter Systeme zu erkennen, die ein gewisses Maß an Allgemeinwissen durch wiederholtes Vorhersagen von Wörtern im Text lernen können. Jüngste Forschungen zeigen, dass ein einziges, gut trainiertes LLM eine große Anzahl von Aufgaben bewältigen und generalisieren kann, um neue Aufgaben mit geringem Anpassungsaufwand zu erfüllen [Bubeck et al., 2023]. Dies deutet auf einen Schritt in Richtung fortgeschrittenerer Formen der künstlichen Intelligenz hin und inspiriert zu weiterer Forschung bei der Entwicklung leistungsfähigerer Sprachmodelle als Basismodelle.

In diesem Kapitel betrachten wir die grundlegenden Konzepte von generativen LLMs. Der Einfachheit halber verwenden wir die Begriffe große Sprachmodelle oder LLMs, um uns auf generative Modelle wie GPT zu beziehen, obwohl dieser Begriff auch andere Modelltypen wie BERT im weiteren Sinne abdecken kann. Wir beginnen mit einer allgemeinen Einführung in LLMs, einschließlich der wichtigsten Schritte zum Aufbau solcher Modelle. Anschließend erörtern wir zwei Skalierungsprobleme von LLMs: wie LLMs in großem Maßstab trainiert werden und wie LLMs verbessert werden können, um sehr lange Texte zu verarbeiten. Abschließend geben wir eine Zusammenfassung dieser Diskussionen.

2.1 Eine kurze Einführung in LLMs

In diesem Abschnitt geben wir eine Einführung in die grundlegenden Konzepte von LLMs, wie sie für den Rest dieses Kapitels und die folgenden Kapitel erforderlich sind. Wir werden die Begriffe Wort und Token austauschbar verwenden. Beide beziehen sich auf die grundlegenden Einheiten, die in der Sprachmodellierung verwendet werden, obwohl ihre ursprünglichen Bedeutungen unterschiedlich sind.

Bevor wir auf Details eingehen, betrachten wir zunächst, wie Sprachmodelle funktionieren. Das Ziel der Sprachmodellierung ist es, die Wahrscheinlichkeit des Auftretens einer Sequenz von Tokens vorherzusagen. Sei $\{x_0, x_1, ..., x_m\}$ eine Sequenz von Tokens, wobei

x_0 das Startsymbol $\langle s \rangle$ (oder $\langle \text{SOS} \rangle$) ist[1]. Die Wahrscheinlichkeit dieser Sequenz kann mit der Kettenregel definiert werden

$$\begin{aligned} \Pr(x_0, ..., x_m) &= \Pr(x_0) \cdot \Pr(x_1|x_0) \cdot \Pr(x_2|x_0, x_1) \cdots \Pr(x_m|x_0, ..., x_{m-1}) \\ &= \prod_{i=0}^{m} \Pr(x_i|x_0, ..., x_{i-1}) \end{aligned} \tag{2.1}$$

oder alternativ in logarithmischer Form

$$\log \Pr(x_0, ..., x_m) = \sum_{i=0}^{m} \log \Pr(x_i|x_0, ..., x_{i-1}) \tag{2.2}$$

Hier ist $\Pr(x_i|x_0, ..., x_{i-1})$ die Wahrscheinlichkeit des Tokens x_i gegeben alle vorhergehenden Tokens $\{x_0, ..., x_{i-1}\}$ [2]. In der Ära des Deep Learning ist ein typischer Ansatz der Sprachmodellierung, diese Wahrscheinlichkeit mit einem tiefen neuronalen Netzwerk zu schätzen. Neuronale Netzwerke, die trainiert werden, um diese Aufgabe zu erfüllen, erhalten eine Sequenz von Tokens $x_0, ..., x_{i-1}$ und erzeugen eine Verteilung über das Vokabular $\mathcal{V}$ (bezeichnet als $\Pr(\cdot|x_0, ..., x_{i-1})$). Die Wahrscheinlichkeit $\Pr(x_i|x_0, ..., x_{i-1})$ ist der Wert des i-ten Eintrags von $\Pr(\cdot|x_0, ..., x_{i-1})$.

Bei der Anwendung eines trainierten Sprachmodells besteht eine häufige Aufgabe darin, das wahrscheinlichste Token bei gegebenem vorherigen Kontext-Tokens zu finden. Diese Token-Vorhersageaufgabe kann beschrieben werden als

$$\hat{x}_i = \arg\max_{x_i \in \mathcal{V}} \Pr(x_i|x_0, ..., x_{i-1}) \tag{2.3}$$

Wir können die Wortvorhersage mehrmals durchführen, um einen kontinuierlichen Text zu generieren: Jedes Mal sagen wir das beste Token $\hat{x}_i$ voraus und fügen dieses vorhergesagte Token dann dem Kontext hinzu, um das nächste Token $\hat{x}_{i+1}$ vorherzusagen. Dies führt zu einem von links nach rechts verlaufenden Generierungsprozess, der die Gleichungen (2.1) und (2.2) implementiert. Zur Veranschaulichung betrachten wir die Generierung der folgenden drei Wörter bei gegebenem Präfix '$\langle s \rangle$ a', wie in Tabelle 2.1 gezeigt. Nun diskutieren wir, wie LLMs konstruiert, trainiert und angewendet werden.

2.1.1 Reine Decoder-Transformer

Standardmäßig ist die Eingabe eines Sprachmodells eine Sequenz von Tokens (bezeichnet mit $\{x_0, ..., x_{m-1}\}$). In jedem Schritt wird ein Ausgabe-Token erzeugt, wodurch die Sequenz für die nächste Vorhersage um eine Position nach vorne verschoben wird. Dazu gibt das Sprachmodell an jeder Position i eine Verteilung $\Pr(\cdot|x_0, ..., x_{i-1})$ aus, und das Token x_i wird gemäß dieser Verteilung ausgewählt. Dieses Modell wird durch Maximierung der Log-Likelihood $\sum_{i=1}^{m} \log \Pr(x_i|x_0, ..., x_{i-1})$ trainiert[3].

[1] Das Startsymbol kann in Anlehnung an BERT-Modelle auch [CLS] sein.

[2] Wir nehmen an, dass für $i = 0$, $\Pr(x_i|x_0, ..., x_{i-1}) = \Pr(x_0) = 1$ ist. Daher gilt $\Pr(x_0, ..., x_m) = \Pr(x_0) \Pr(x_1, ..., x_m|x_0) = \Pr(x_1, ..., x_m|x_0)$.

[3] Beachten Sie, dass $\sum_{i=1}^{m} \log \Pr(x_i|x_0, ..., x_{i-1}) = \sum_{i=0}^{m} \log \Pr(x_i|x_0, ..., x_{i-1})$, da $\log \Pr(x_0) = 0$.

Context	Predict	Decision Rule	Sequence Probability
$\langle s\rangle$ a	b	$\arg\max_{x_2\in V} \Pr(x_2\|\langle s\rangle\ a)$	$\Pr(\langle s\rangle)\cdot \Pr(a\|\langle s\rangle)\cdot$ $\Pr(b\|\langle s\rangle\ a)$
$\langle s\rangle$ a b	c	$\arg\max_{x_3\in V} \Pr(x_3\|\langle s\rangle\ a\ b)$	$\Pr(\langle s\rangle)\cdot \Pr(a\|\langle s\rangle)\cdot \Pr(b\|\langle s\rangle\ a)\cdot$ $\Pr(c\|\langle s\rangle\ a\ b)$
$\langle s\rangle$ a b c	d	$\arg\max_{x_4\in V} \Pr(x_4\|\langle s\rangle\ a\ b\ c)$	$\Pr(\langle s\rangle)\cdot \Pr(a\|\langle s\rangle)\cdot \Pr(b\|\langle s\rangle\ a)\cdot$ $\Pr(c\|\langle s\rangle\ a\ b)\cdot$ $\Pr(d\|\langle s\rangle\ a\ b\ c)$

Tabelle 2.1: Veranschaulichung der Generierung der drei Tokens b c d bei gegebenem Präfix $\langle s\rangle$ a durch ein Sprachmodell. In jedem Schritt wählt das Modell ein Token x_i aus V aus, sodass $\Pr(x_i|x_0,...,x_{i-1})$ maximiert wird. Dieses Token wird dann an das Ende der Kontextsequenz angehängt. Im nächsten Schritt wird derselbe Prozess wiederholt, jedoch basierend auf dem neuen Kontext.

Hier konzentrieren wir uns auf die reine Decoder-Transformer-Architektur, da sie eine der beliebtesten Modellarchitekturen ist, die in LLMs verwendet wird. Die Eingabesequenz von Tokens wird durch eine Sequenz von d_e-dimensionalen Vektoren $\{\mathbf{e}_0,...,\mathbf{e}_{m-1}\}$ dargestellt. $\mathbf{e}_i$ ist die Summe der Token-Einbettung von x_i und der Positionseinbettung von i. Der Hauptteil des Modells ist ein Stapel von Transformer-Blöcken (oder Schichten). Jeder Transformer-Block hat zwei gestapelte Unterschichten, eine für die Selbst-Aufmerksamkeits-Modellierung und eine für die FFN-Modellierung. Diese Unterschichten können unter Verwendung der Post-Norm-Architektur definiert werden

$$\text{output} \quad = \quad \text{LNorm}(F(\text{input}) + \text{input}) \tag{2.4}$$

oder der Pre-Norm-Architektur

$$\text{output} \quad = \quad \text{LNorm}(F(\text{input})) + \text{input} \tag{2.5}$$

wobei input und output die Eingabe und Ausgabe bezeichnen, die beide eine $m \times d$-Matrix sind. Die i-ten Zeilen von input und output können als kontextuelle Repräsentationen des i-ten Tokens in der Sequenz angesehen werden.

$F(\cdot)$ ist die Kernfunktion einer Unterschicht. Für FFN-Unterschichten ist $F(\cdot)$ ein mehrschichtiges FFN. Für Selbst-Aufmerksamkeits-Unterschichten ist $F(\cdot)$ eine Multi-Head-Selbst-Aufmerksamkeits-Funktion. Im Allgemeinen wird die Selbst-Aufmerksamkeit in Form einer QKV-Aufmerksamkeit ausgedrückt

$$\text{Att}_{\text{qkv}}(\mathbf{Q},\mathbf{K},\mathbf{V}) \quad = \quad \text{Softmax}(\frac{\mathbf{Q}\mathbf{K}^{\text{T}}}{\sqrt{d}} + \mathbf{Mask})\mathbf{V} \tag{2.6}$$

wobei $\mathbf{Q}$, $\mathbf{K}$ und $\mathbf{V} \in \mathbb{R}^{m\times d}$ jeweils die Abfragen, Schlüssel und Werte sind. Es ist wichtig zu beachten, dass bei der Vorhersage eines Tokens nur vorherige Tokens berücksichtigt werden. Daher wird eine Maskierungsvariable $\mathbf{Mask} \in \mathbb{R}^{m\times m}$ in die Selbst-Aufmerksamkeit integriert, um dies zu erreichen. Der Eintrag (i,k) von $\mathbf{Mask}$ hat einen Wert von 0, wenn $i \leq k$ ist, und andernfalls einen Wert von $-\inf$.

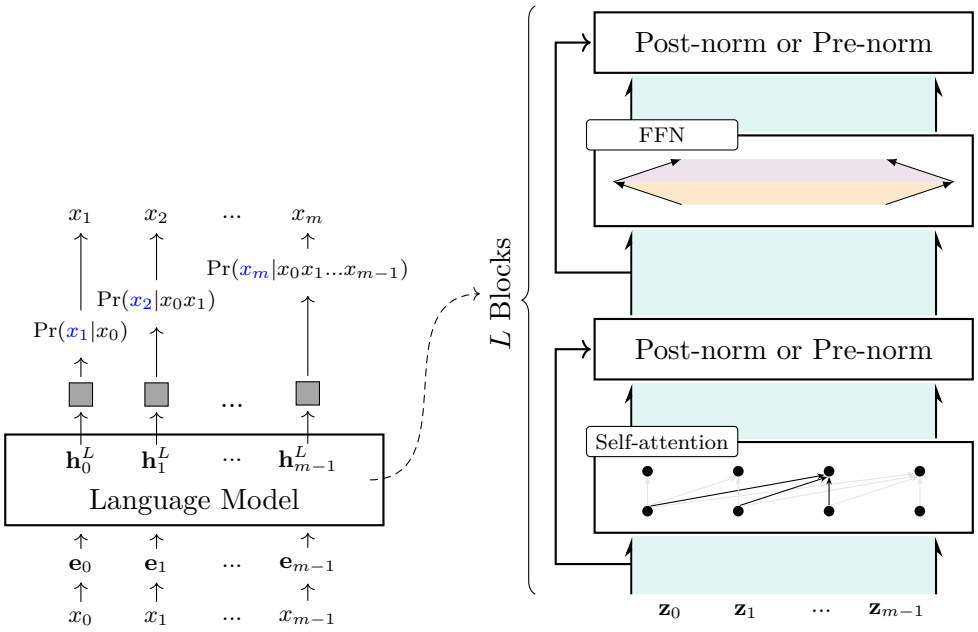


Abbildung 2.1: Die Transformer -Dekoder-Architektur für die Sprachmodellierung. Die zentralen Komponenten sind L gestapelte Transformer -Blöcke, die jeweils aus einer Selbst-Aufmerksamkeits-Unterschicht und einer FFN-Unterschicht bestehen. Um zu verhindern, dass das Modell auf den rechten Kontext zugreift, wird eine Maskierungsvariable in die Selbst-Aufmerksamkeit integriert. Die Ausgabeschicht verwendet eine Softmax-Funktion, um eine Wahrscheinlichkeitsverteilung für das nächste Token zu erzeugen, basierend auf der Sequenz der vorherigen Tokens. Während der Inferenz nimmt das Modell das zuvor vorhergesagte Token, um das nächste vorherzusagen, und wiederholt diesen Prozess, bis das Ende der Sequenz erreicht ist. $\{\mathbf{z}_0, ..., \mathbf{z}_{m-1}\}$ bezeichnen die Eingaben eines Transformer -Blocks, und $\{\mathbf{h}_0^L, ..., \mathbf{h}_{m-1}^L\}$ bezeichnen die Ausgaben des letzten Transformer -Blocks.

Gegeben eine Repräsentation $\mathbf{H} \in \mathbb{R}^{m \times d}$, kann die Multi-Head-Selbst-Aufmerksamkeits-Funktion wie folgt definiert werden

$$F(\mathbf{H}) = \text{Merge}(\text{head}_1, ..., \text{head}_\tau)\mathbf{W}^{\text{head}} \tag{2.7}$$

wobei Merge$(\cdot)$ eine Verkettung seiner Eingaben darstellt und $\mathbf{W}^{\text{head}} \in \mathbb{R}^{d \times d}$ eine Parametermatrix repräsentiert. head_j ist die Ausgabe der QKV-Aufmerksamkeit auf einem Unterraum der Repräsentation

$$\text{head}_j = \text{Att}_{\text{qkv}}(\mathbf{Q}^{[j]}, \mathbf{K}^{[j]}, \mathbf{V}^{[j]}) \tag{2.8}$$

$\mathbf{Q}^{[j]}$,$\mathbf{K}^{[j]}$ und $\mathbf{V}^{[j]}$ sind die Abfragen, Schlüssel und Werte, die über lineare Transformationen auf den j-ten Unterraum projiziert werden

$$\mathbf{Q}^{[j]} = \mathbf{H}\mathbf{W}_j^q \tag{2.9}$$
$$\mathbf{K}^{[j]} = \mathbf{H}\mathbf{W}_j^k \tag{2.10}$$
$$\mathbf{V}^{[j]} = \mathbf{H}\mathbf{W}_j^v \tag{2.11}$$

wobei $\mathbf{W}_j^q$, $\mathbf{W}_j^k$ und $\mathbf{W}_j^v \in \mathbb{R}^{d \times \frac{d}{\tau}}$ die Parametermatrizen der Transformationen sind.

Angenommen, wir haben L Transformer-Blöcke. Eine Softmax-Schicht wird auf die

Ausgabe des letzten Blocks aufgebaut. Die Softmax-Schicht gibt eine Sequenz von m Verteilungen über das Vokabular aus, wie folgt

$$\begin{bmatrix} \Pr(\cdot|x_0, ..., x_{m-1}) \\ \vdots \\ \Pr(\cdot|x_0, x_1) \\ \Pr(\cdot|x_0) \end{bmatrix} = \text{Softmax}(\mathbf{H}^L \mathbf{W}^o) \tag{2.12}$$

wobei $\mathbf{H}^L$ die Ausgabe des letzten Transformer-Blocks ist und $\mathbf{W}^o \in \mathbb{R}^{d \times |V|}$ die Parametermatrix ist.

Abbildung 2.1 zeigt die Transformer-Architektur für die Sprachmodellierung. Die Anwendung dieses Sprachmodells folgt einem autoregressiven Prozess. Jedes Mal nimmt das Sprachmodell ein Token x_{i-1} als Eingabe und sagt ein Token x_i voraus, das die Wahrscheinlichkeit $\Pr(x_i|x_0, ..., x_{i-1})$ maximiert. Es ist wichtig zu beachten, dass trotz unterschiedlicher Implementierungsdetails viele LLMs die gleiche oben beschriebene Architektur teilen. Diese Modelle werden als groß bezeichnet, weil sowohl ihre Tiefe als auch ihre Breite signifikant sind. Tabelle 2.2 zeigt die Modellgrößen für einige LLMs sowie deren Modellkonfigurationen.

2.1.2 Training von LLMs

Nehmen wir nun an, dass wir einen Trainingsdatensatz $\mathcal{D}$ haben, der K Sequenzen umfasst. Die Log-Likelihood jeder Sequenz $\mathbf{x} = x_0...x_m$ in $\mathcal{D}$ kann mit einem Sprachmodell berechnet werden

$$\mathcal{L}_\theta(\mathbf{x}) = \sum_{i=1}^{m} \log \Pr_\theta(x_i|x_0, ..., x_{i-1}) \tag{2.13}$$

Hier bezeichnet der Index θ, der an $\mathcal{L}(\cdot)$ und $\Pr(\cdot)$ angehängt ist, die Parameter des Sprachmodells. Dann wird das Ziel des Maximum-Likelihood-Trainings wie folgt definiert

$$\hat{\theta} = \arg\max_\theta \sum_{\mathbf{x} \in \mathcal{D}} \mathcal{L}_\theta(\mathbf{x}) \tag{2.14}$$

Das Training von Transformer -basierten Sprachmodellen mit dem oben genannten Ziel wird üblicherweise als ein Standard-Optimierungsprozess für neuronale Netze angesehen. Dies kann mithilfe von Gradientenabstiegsalgorithmen erreicht werden, die von handelsüblichen Deep-Learning-Toolkits umfassend unterstützt werden. Etwas überraschend wurden kontinuierlich bessere Ergebnisse erzielt, als Sprachmodelle zu rechenintensiveren Modellen weiterentwickelt und auf größeren Datensätzen trainiert wurden [Kaplan et al., 2020]. Diese Erfolge haben NLP-Forscher dazu veranlasst, sowohl die Trainingsdaten als auch die Modellgröße weiter zu erhöhen, um leistungsfähigere Sprachmodelle zu entwickeln.

Je größer Sprachmodelle jedoch werden, desto mehr stehen wir vor neuen Herausforderungen beim Training, die das Problem im Vergleich zum Training relativ kleiner Modelle erheblich verändern. Eine dieser Herausforderungen ergibt sich aus der Notwendigkeit

LLM	# of Parameters	Depth L	Width d	# of Heads (Q/KV)
GPT-1 [Radford et al., 2018]	0.117B	12	768	12/12
GPT-2 [Radford et al., 2019]	1.5B	48	1,600	25/25
GPT-3 [Brown et al., 2020]	175B	96	12,288	96/96
LLaMA2 [Touvron et al., 2023b]	7B	32	4,096	32/32
	13B	40	5,120	40/40
	70B	80	8,192	64/64
LLaMA3/3.1 [Dubey et al., 2024]	8B	32	4,096	32/8
	70B	80	8,192	64/8
	405B	126	16,384	128/8
Gemma2 [Team et al., 2024]	2B	26	2,304	8/4
	9B	42	3,584	16/8
	37B	46	4,608	32/16
Qwen2.5 [Yang et al., 2024]	0.5B	24	896	14/2
	7B	28	3,584	28/4
	72B	80	8,192	64/8
DeepSeek-V3 [Liu et al., 2024a]	671B	61	7,168	128/128
Falcon [Penedo et al., 2023]	7B	32	4,544	71/71
	40B	60	8,192	128/128
	180B	80	14,848	232/232
Mistral [Jiang et al., 2023a]	7B	32	4,096	32/32

Tabelle 2.2: Vergleich einiger LLMs hinsichtlich Modellgröße, Modelltiefe, Modellbreite und Anzahl der Heads (a/b bedeutet a Heads für Queries und b Heads für Keys und Values).

von großen verteilten Systemen zur Verwaltung der Daten, Modellparameter, Trainingsroutinen und so weiter. Die Entwicklung und Wartung solcher Systeme erfordert einen erheblichen Arbeitsaufwand sowohl in der Software- als auch in der Hardwareentwicklung sowie Fachwissen im Bereich Deep Learning. Ein verwandtes Problem ist, dass wir bei der Skalierung des Trainings mehr Rechenressourcen benötigen, um sicherzustellen, dass der Trainingsprozess in einer akzeptablen Zeit abgeschlossen werden kann. Beispielsweise werden im Allgemeinen Hunderte oder Tausende von GPUs benötigt, um ein LLM mit zig Milliarden Parametern von Grund auf zu trainieren. Diese Anforderung erhöht die Kosten für das Training solcher Modelle drastisch, insbesondere wenn man bedenkt, dass während der Entwicklung dieser Modelle viele Trainingsläufe erforderlich sind. Aus der Perspektive des Deep Learning kann der Trainingsprozess zudem instabil werden, wenn die neuronalen Netze sehr tief und/oder die Modellgröße sehr groß ist. Als Reaktion darauf müssen wir typischerweise die Modellarchitektur modifizieren, um LLMs an das Training im großen Maßstab anzupassen. In Abschnitt 2.2 werden wir diese Themen ausführlicher erörtern.

2.1.3 Feinabstimmung von LLMs

Sobald wir ein LLM vortrainiert haben, können wir es anwenden, um verschiedene NLP-Aufgaben durchzuführen. Traditionell werden Sprachmodelle als Komponenten anderer Systeme verwendet, zum Beispiel werden sie häufig zur Bewertung von Übersetzungen in statistischen maschinellen Übersetzungssystemen eingesetzt. Im Gegensatz dazu werden in der generativen KI LLMs als vollständige Systeme betrachtet und zur Lösung von NLP-Problemen eingesetzt, indem ihre generative Natur genutzt wird. Ein gängiger Ansatz besteht darin, die zu lösende Aufgabe in Textform zu beschreiben und die LLMs dann aufzufordern, auf der Grundlage dieser Beschreibung Text zu generieren. Dies ist eine standardmäßige Textgenerierungsaufgabe, bei der wir den Text ausgehend von einem gegebenen Kontext fortsetzen oder vervollständigen.

Formaler ausgedrückt, sei $\mathbf{x} = x_0...x_m$ eine Token-Sequenz des von Benutzern gegebenen Kontexts und $\mathbf{y} = y_1...y_n$ eine Token-Sequenz, die auf den Kontext folgt. Dann kann die Inferenz von LLMs als das Problem definiert werden, die wahrscheinlichste Sequenz $\mathbf{y}$ basierend auf $\mathbf{x}$ zu finden:

$$\begin{aligned} \hat{\mathbf{y}} &= \arg\max_{\mathbf{y}} \log \Pr(\mathbf{y}|\mathbf{x}) \\ &= \arg\max_{\mathbf{y}} \sum_{i=1}^{n} \log \Pr(y_i|x_0, ..., x_m, y_1, ..., y_{i-1}) \end{aligned} \tag{2.15}$$

Hier drückt $\sum_{i=1}^{n} \log \Pr(y_i|x_0, ..., x_m, y_1, ..., y_{i-1})$ im Wesentlichen dasselbe aus wie die rechte Seite von Gl. (2.2). Es modelliert die Log-Wahrscheinlichkeit der Vorhersage von Tokens ab Position $m + 1$, anstatt ab Position 0. In diesem und den folgenden Kapiteln werden wir separate Variablen $\mathbf{x}$ und $\mathbf{y}$ verwenden, um die Eingabe und Ausgabe eines LLM zu unterscheiden, obwohl sie als Teilsequenzen derselben Sequenz betrachtet werden können. Durch die Übernahme dieser Notation sehen wir, dass die Form der obigen Gleichung denen in anderen Textgenerierungsmodellen der NLP, wie z. B. neuronalen maschinellen Übersetzungsmodellen, sehr ähnlich ist.

Um zu veranschaulichen, wie LLMs angewendet werden, betrachten wir das Problem der Bestimmung der Grammatikalität eines gegebenen Satzes. Wir können eine Vorlage wie diese definieren

```
{*sentence*}
Question: Is this sentence grammatically correct?
Answer: ____
```

Hier repräsentiert __ den Text, den wir generieren wollen. {*sentence*} ist eine Platzhaltervariable, die durch den tatsächlichen, von den Benutzern bereitgestellten Satz ersetzt wird. Nehmen wir zum Beispiel an, wir haben einen Satz „John seems happy today.“. Wir können den Platzhalter {*sentence*} in der Vorlage durch diesen Satz ersetzen, um eine Eingabe für das Sprachmodell zu erhalten

> John seems happy today.
> Question: Is this sentence grammatically correct?
> Answer: ____

Um die Aufgabe auszuführen, erhält das Sprachmodell den Kontext $\mathbf{x}$ =„John seems happy today .\n Question : Is this sentence grammatically correct?\n Answer :“[4]. Es generiert dann den folgenden Text als Antwort, basierend auf dem Kontext. Zum Beispiel kann das Sprachmodell „Yes“ausgeben (d. h. $\mathbf{y}$ = „Yes“), wenn dieser Text derjenige mit der maximalen Vorhersagewahrscheinlichkeit für diesen Kontext ist.

Ebenso können wir weitere Vorlagen definieren, um andere Aufgaben zu bearbeiten. Zum Beispiel können wir einen englischen Satz ins Chinesische übersetzen, indem wir die folgende Vorlage verwenden

> {*sentence*}
> Question: What is the Chinese translation of this English sentence?
> Answer: ____

oder unter Verwendung einer anweisungsähnlichen Vorlage

> {*sentence*}
> Translate this sentence from English into Chinese.
> ____

oder unter Verwendung einer code-ähnlichen Vorlage.

> [src-lang] = English [tgt-lang] = Chinese [input] = {*sentence*}
> [output] = ____

Die obigen Vorlagen bieten eine einfache, aber effektive Methode, um ein einzelnes LLM zu „prompten“, verschiedene Aufgaben auszuführen, ohne die Struktur des Modells anzupassen. Dieser Ansatz erfordert jedoch, dass das LLM die Anweisungen oder Fragen erkennen und befolgen kann. Eine Möglichkeit, dies zu erreichen, besteht darin, Trainingsbeispiele mit Anweisungen und den entsprechenden Antworten in den Vortrainingsdatensatz zu integrieren. Obwohl diese Methode unkompliziert ist, ist das Erstellen und Trainieren von LLMs von Grund auf rechenintensiv. Darüber hinaus erfordert die effektive Nutzung von instruktionsfolgenden Daten für das Vortraining eine erhebliche Menge solcher Daten, aber das Sammeln von großen Mengen gelabelter Daten für alle interessierenden Aufgaben ist sehr schwierig.

Eine zweite Methode, die in der jüngsten Forschung zu einem De-facto-Standard geworden ist, ist die Anpassung von LLMs durch Feinabstimmung. Dadurch kann die in der Vortrainingsphase erlernte Fähigkeit zur Token-Vorhersage verallgemeinert werden, um

[4] \n ist ein Sonderzeichen für Zeilenumbrüche.

neue Aufgaben zu bewältigen. Die Idee hinter der Feinabstimmung ist, dass beim Vortraining ein gewisses allgemeines Sprachwissen erworben wurde, wir aber einen Mechanismus benötigen, um dieses Wissen für die Anwendung auf neue Aufgaben zu aktivieren. Um dies zu erreichen, können wir die Modellparameter mithilfe von instruktionsfolgenden Daten leicht feinabstimmen. Dieser Ansatz wird als Instruktions-Feinabstimmung bezeichnet.

Ein Beispiel für die Instruktions-Feinabstimmung, das durch eine Sequenz von Tokens dargestellt wird, kann als Tupel betrachtet werden, das aus einer Eingabe und der gewünschten Ausgabe besteht. Hier umfasst die Eingabe Anweisungen, Systeminformationen (oder ein System-Präfix) und alle anderen vom Benutzer bereitgestellten Informationen[5]. Zur Veranschaulichung betrachten Sie die folgenden Beispiele (blauer Text = Eingabe und unterstrichener Text = Ausgabe).

> Read the following sentence, and answer whether it is grammatically correct.
> LLMs are powerful models but are expensive to build.
> Yes

> LLMs are powerful models but are expensive to build.
> Does this sentence make sense grammatically?
> Answer Option
> - Yes
> - No
> Yes

> Yes or No? Yes = the following sentence is grammatically correct. No = it contains grammatical errors.
> LLMs are powerful models but are expensive to build.
> Yes

Alle diese Beispiele beschreiben dieselbe binäre Klassifikationsaufgabe, aber mit unterschiedlichen Anweisungen. Um die Diversität der Feinabstimmungsdaten zu erhöhen und LLMs zur Generalisierung zu befähigen, können wir weitere Aufgaben mithilfe von Anweisungen definieren.

[5] Systeminformationen beziehen sich auf eine Sequenz von Tokens, die am Anfang einer Eingabe hinzugefügt wird, um das Verhalten eines LLM zu steuern, wie zum Beispiel you are a helpful assistant and should not output toxic content.

Write a short story about three characters.
Character 1: A boy whose name is Andy.
Character 2: Andy's dog.
Character 3: Andy's friend, Jane.

In the heart of a bustling city, Andy, his loyal dog Rocket, and his friend Jane discovered a hidden garden beneath the glow of twilight. Surrounded by ancient trees and whispering winds, they shared dreams and secrets, forging an unbreakable bond. As the night deepened, they made a pact under the stars to always find magic in the mundane. Together, they realized that true adventure lay in the moments spent with friends, and every day was a treasure waiting to be discovered.

Show me a recipe for making ice cream.

Ingredients:
2 cups heavy cream; 1 cup milk; 3/4 cup sugar; 2 tsp vanilla extract;
Pinch of salt.
Steps:
1. Whisk all ingredients until sugar dissolves.
2. Chill mixture for 2 hours in the fridge.
3. Freeze in a shallow dish, stirring every 30 minutes, 3-4 times.
4. Freeze solid for 2 hours.
5. Serve your homemade ice cream.

If you buy 5 apples and each apple costs $1.20, how much do you spend in total?

$6.00

Write a Python program to calculate the sum of squares of the following numbers.
1 , 2 , 10 , -9 , 78

```
numbers = [1,2,10,-9 ,78]
sum_of_squares = sum(x**2 for x in numbers)
print(sum_of_squares)
```

Um die Fähigkeit zu erlangen, Anweisungen zu befolgen, ist eine gewisse Menge an Feinabstimmungsdaten erforderlich. Diese Daten können vielfältige Anweisungen und mögliche Antworten enthalten. Es wurde festgestellt, dass die Skalierung der Anzahl von Feinabstimmungsaufgaben vorteilhaft für die Verbesserung der Leistung von LLMs ist [Chung et al., 2022]. Beachten Sie, dass, obwohl mehr Feinabstimmungsdaten vorteilhaft sind, die Menge dieser Daten im Allgemeinen um Größenordnungen kleiner ist als die der Vortrainingsdaten. Zum Beispiel können LLMs mit zehntausenden oder hunderttausenden von

Beispielen feinabgestimmt werden, oder sogar mit weniger, wenn diese Beispiele von hoher Qualität sind [Zhou et al., 2023a; Chen et al., 2023b], während das Vortraining solcher Modelle Milliarden oder Billionen von Tokens erfordern kann, was zu erheblich größeren Rechenanforderungen und längeren Trainingszeiten führt [Touvron et al., 2023a].

Es ist auch erwähnenswert, dass wir nicht erwarten sollten, dass die Feinabstimmungsdaten alle nachgelagerten Aufgaben abdecken, auf die wir LLMs anwenden wollen. Ein gängiges Verständnis davon, wie der Ansatz des Vortrainings + Feinabstimmung funktioniert, ist, dass LLMs in der Vortrainingsphase Wissen zum Verstehen von Anweisungen und zum Generieren von Antworten erworben haben. Diese Fähigkeiten werden jedoch erst vollständig aktiviert, wenn wir eine Form der Überwachung (Supervision) einführen. Das allgemeine instruktionsfolgende Verhalten tritt auf, wenn wir die Modelle mit einer relativ geringen Menge an gelabelten Daten feinabstimmen. Als Ergebnis können wir ein gewisses Maß an Zero-Shot-Lernen erreichen: Die feinabgestimmten Modelle können neue Aufgaben bewältigen, für die sie nicht explizit trainiert oder feinabgestimmt wurden [Sanh et al., 2022; Wei et al., 2022a]. Diese Fähigkeit zum Zero-Shot-Lernen unterscheidet generative LLMs von früheren vortrainierten Modellen wie BERT, die hauptsächlich für spezifische Aufgaben feinabgestimmt werden.

Sobald wir eine Sammlung von durch Anweisungen beschriebenen Daten vorbereitet haben, ist der Feinabstimmungsprozess relativ einfach. Dieser Prozess kann als ein standardmäßiger Trainingsprozess wie das Vortraining betrachtet werden, jedoch auf einem viel kleineren Trainingsdatensatz. Sei $\mathcal{D}_{\text{tune}}$ der Feinabstimmungsdatensatz und $\hat{\theta}$ die durch Vortraining optimierten Modellparameter. Wir können Gl. (2.14) modifizieren, um das Ziel der Feinabstimmung zu erhalten

$$\tilde{\theta} \quad = \quad \arg\max_{\hat{\theta}^+} \sum_{\text{sample} \in \mathcal{D}_{\text{tune}}} \mathcal{L}_{\hat{\theta}^+}(\text{sample}) \tag{2.16}$$

Hier bezeichnet $\tilde{\theta}$ die optimalen Parameter. Die Verwendung der Notation $\hat{\theta}^+$ bedeutet, dass die Feinabstimmung mit den vortrainierten Parametern $\hat{\theta}$ beginnt.

Für jedes sample $\in \mathcal{D}_{\text{tune}}$ teilen wir es in ein Eingabesegment $\mathbf{x}_{\text{sample}}$ und ein Ausgabesegment $\mathbf{y}_{\text{sample}}$ auf, das heißt,

$$\text{sample} \quad = \quad [\mathbf{y}_{\text{sample}}, \mathbf{x}_{\text{sample}}] \tag{2.17}$$

Wir definieren dann die Verlustfunktion als

$$\mathcal{L}_{\hat{\theta}^+}(\text{sample}) \quad = \quad -\log \Pr_{\hat{\theta}^+}(\mathbf{y}_{\text{sample}} | \mathbf{x}_{\text{sample}}) \tag{2.18}$$

Mit anderen Worten, wir berechnen den Verlust über die Teilsequenz $\mathbf{y}_{\text{sample}}$ und nicht über die gesamte Sequenz. In einer praktischen Implementierung der Backpropagation für diese Gleichung wird die Sequenz $[\mathbf{y}_{\text{sample}}, \mathbf{x}_{\text{sample}}]$ im Forward-Pass wie gewohnt konstruiert. Im Backward-Pass werden die Fehlergradienten jedoch nur durch die Teile des Netzwerks zurückpropagiert, die $\mathbf{y}_{\text{sample}}$ entsprechen, während der Rest des Netzwerks unverändert bleibt. Betrachten Sie als Beispiel eine Sequenz

$$\underbrace{\langle s \rangle \text{ Square this number . 2 .}}_{\text{Context (Input)}} \quad \underbrace{\text{The result is 4 .}}_{\text{Prediction (Output)}}$$

Der Verlust wird nur für The result is 4 . berechnet und zurückpropagiert.

Die Instruktions-Feinabstimmung erfordert ebenfalls erhebliche Ingenieurarbeit. Um zufriedenstellende Ergebnisse zu erzielen, kann man mit verschiedenen Einstellungen der Lernrate, der Batch-Größe, der Anzahl der Feinabstimmungsschritte usw. experimentieren. Dies erfordert typischerweise viele Durchläufe der Feinabstimmung und Auswertungen. Die Kosten und der experimentelle Aufwand der Feinabstimmung bleiben kritisch und sollten nicht übersehen werden, auch wenn sie viel geringer sind als die der Vortrainingsphase.

Obwohl wir uns hier zur Veranschaulichung auf die Instruktions-Feinabstimmung konzentrieren, spielen Feinabstimmungstechniken eine wichtige Rolle bei der Entwicklung verschiedener LLMs und sind weitaus verbreiteter. Beispiele hierfür sind die Feinabstimmung von LLMs als Chatbots unter Verwendung von Dialogdaten und die Anpassung dieser Modelle zur Verarbeitung sehr langer Sequenzen. Die breite Anwendung der Feinabstimmung hat Forscher dazu veranlasst, diese Techniken zu verbessern, beispielsweise durch die Entwicklung effizienterer Feinabstimmungsalgorithmen. Obwohl die Forschung zur Feinabstimmung fruchtbar ist, geben wir in diesem Abschnitt nur einen Vorgeschmack auf die wichtigsten beteiligten Schritte. Wir werden in den folgenden Kapiteln detailliertere Diskussionen zu diesem Thema sehen.

2.1.4 Angleichung von LLMs an die Welt

Instruktions-Feinabstimmung bietet eine einfache Möglichkeit, LLMs an Aufgaben anzupassen, die gut definiert werden können. Dieses Problem lässt sich grob als Alignment-Problem kategorisieren. Hierbei bezeichnet Alignment einen Prozess, bei dem LLMs so gesteuert werden, dass ihr Verhalten mit menschlichen Absichten übereinstimmt. Die Steuerung kann durch gelabelte Daten, menschliches Feedback oder jede andere Form menschlicher Präferenzen erfolgen. Wir möchten zum Beispiel, dass LLMs nicht nur Anweisungen präzise befolgen, sondern auch unvoreingenommen, wahrheitsgemäß und harmlos sind. Daher müssen wir die Modelle im Hinblick auf menschliche Werte und Erwartungen überwachen. Ein gängiges Beispiel ist, dass ein LLM, wenn wir es fragen, wie man eine Waffe baut, eine Liste der wichtigsten Schritte dafür liefern könnte, wenn es nicht sorgfältig angepasst wurde. Ein verantwortungsbewusstes Modell sollte jedoch Anfragen nach schädlichen oder illegalen Informationen erkennen und deren Beantwortung vermeiden. Alignment ist in diesem Fall entscheidend, um sicherzustellen, dass LLMs verantwortungsbewusst und in Übereinstimmung mit ethischen Richtlinien handeln.

Ein verwandtes Konzept zum Alignment ist die KI-Sicherheit. Eines der obersten Ziele der KI ist es, intelligente Systeme zu schaffen, die sicher und sozial nützlich sind. Um dieses Ziel zu erreichen, sollten wir diese Systeme unter allen Bedingungen des realen Einsatzes robust, sicher und subjektiv halten, selbst bei Missbrauch oder schädlicher Nutzung. Bei LLMs kann die Sicherheit erhöht werden, indem sie durch angemessene menschliche Anleitung, wie z. B. von Menschen gelabelte Daten und Interaktionen mit Benutzern während der Anwendung, angeglichen werden.

Alignment ist schwierig, da menschliche Werte und Erwartungen vielfältig und wandelbar sind. Manchmal ist es schwer, genau zu beschreiben, was Menschen wollen, es sei denn, wir sehen die Reaktion der LLMs auf Benutzeranfragen. Dies macht Alignment nicht mehr nur zu einem Problem der Anpassung von LLMs an vordefinierte Aufgaben, sondern zu einem größeren Problem, sie durch Interaktionen mit der realen Welt zu trainieren.

Aufgrund der Bedenken hinsichtlich der Kontrolle von KI-Systemen hat die Forschung zum Alignment-Problem bei LLMs stark zugenommen. Typischerweise werden zwei Alignment-Schritte angewendet, nachdem die LLMs auf großen Mengen ungelabelter Daten vortrainiert wurden.

- Überwachtes Fine-Tuning (SFT). Dies beinhaltet die Fortsetzung des Trainings von vortrainierten LLMs mit neuen, aufgabenorientierten und gelabelten Daten. Eine häufig verwendete SFT-Technik ist das Instruction Fine-Tuning. Wie im vorherigen Unterabschnitt beschrieben, können sich LLMs durch das Lernen aus mit Anweisungen und Antworten versehenen Daten an die beabsichtigten Verhaltensweisen zum Befolgen von Anweisungen anpassen und werden dadurch in die Lage versetzt, verschiedene durch Anweisungen beschriebene Aufgaben auszuführen. Überwachtes Fine-Tuning kann als Befolgung des Paradigmas Pre-Training + Fine-Tuning angesehen werden und bietet eine relativ unkomplizierte Methode zur Anpassung von LLMs.

- Lernen aus menschlichem Feedback. Nachdem ein LLM das Pre-Training und das überwachte Fine-Tuning abgeschlossen hat, kann es verwendet werden, um auf Benutzeranfragen zu antworten, wenn es entsprechend gepromptet wird. Dieses Modell kann jedoch Inhalte generieren, die nicht faktengestützt, voreingenommen oder schädlich sind. Um das LLM besser auf die Benutzer auszurichten, besteht ein einfacher Ansatz darin, direkt aus menschlichem Feedback zu lernen. Beispielsweise werden Experten bei gegebenen Anweisungen und Eingaben von Benutzern gebeten zu bewerten, wie gut das Modell in Übereinstimmung mit ihren Präferenzen und Interessen antwortet. Dieses Feedback wird dann verwendet, um das LLM für eine bessere Ausrichtung weiter zu trainieren.

Eine typische Methode, um aus menschlichem Feedback zu lernen, besteht darin, es als ein Problem des bestärkenden Lernens (Reinforcement Learning, RL) zu betrachten, bekannt als bestärkendes Lernen durch menschliches Feedback (RLHF) [Ouyang et al., 2022]. Die RLHF-Methode wurde ursprünglich zur Lösung allgemeiner sequenzieller Entscheidungsprobleme vorgeschlagen [Christiano et al., 2017] und später erfolgreich bei der Entwicklung der GPT-Modellreihe eingesetzt [Stiennon et al., 2020]. Als Ansatz des bestärkenden Lernens ist das Ziel von RLHF, eine Policy zu lernen, indem eine Belohnung aus der Umgebung maximiert wird. Insbesondere werden bei RLHF zwei Komponenten aufgebaut:

- Agent. Ein Agent, auch LM-Agent genannt, ist das LLM, das wir trainieren wollen. Dieser Agent operiert durch Interaktion mit seiner Umgebung: Er empfängt einen Text von der Umgebung und gibt einen anderen Text aus, der an die Umgebung zurückgesendet wird. Die Policy des Agenten ist die durch das LLM definierte

Funktion, d. h. $\Pr(\mathbf{y}|\mathbf{x})$.

- Reward Model. Ein Belohnungsmodell ist ein Proxy für die Umgebung. Jedes Mal, wenn der Agent eine Ausgabesequenz erzeugt, weist das Belohnungsmodell dieser Ausgabesequenz einen numerischen Score (d. h. die Belohnung) zu. Dieser Score sagt dem Agenten, wie gut die Ausgabesequenz ist.

Bei RLHF müssen wir zwei Lernaufgaben durchführen: 1) das Lernen des Belohnungsmodells, was das Trainieren eines Belohnungsmodells unter Verwendung von menschlichem Feedback zur Ausgabe des Agenten beinhaltet, und 2) das Lernen der Policy, was die Optimierung einer Policy unter Anleitung des Belohnungsmodells mithilfe von Algorithmen des bestärkenden Lernens umfasst. Hier ist ein kurzer Überblick über die wichtigsten Schritte von RLHF.

- Erstellen einer anfänglichen Policy durch Pre-Training und anweisungsorientierte Feinabstimmung.

- Verwenden der Policy, um für jede Eingabe mehrere Ausgaben zu generieren, und anschließendes Sammeln von menschlichem Feedback zu diesen Ausgaben (z. B. durch Vergleiche der Ausgaben).

- Lernen eines Belohnungsmodells aus dem menschlichen Feedback.

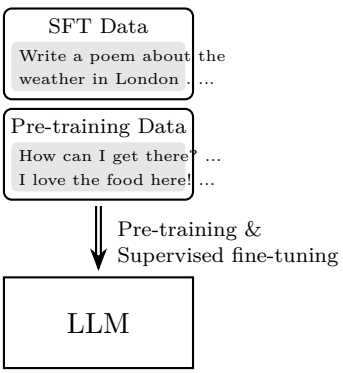
- Feinabstimmung der Policy mit der Supervision des Belohnungsmodells.

Abbildung 2.2 zeigt einen Überblick über RLHF. Da dieser Abschnitt nur als kurze Einführung in die Konzepte von LLMs dient, wird eine detaillierte Diskussion der RLHF-Techniken nicht enthalten sein. Stattdessen veranschaulichen wir die grundlegenden Ideen hinter RLHF anhand eines einfachen Beispiels.

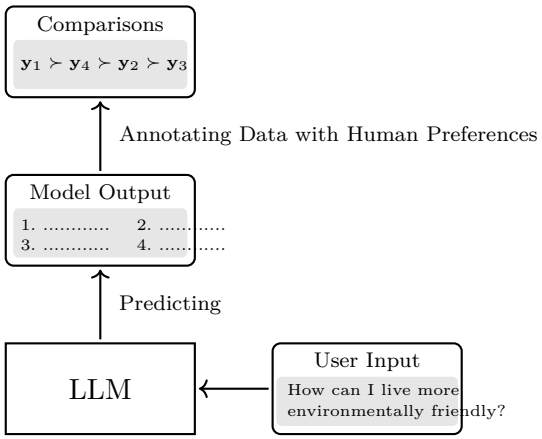
Angenommen, wir haben ein LLM durch Vortraining und Instruktions-Feinabstimmung trainiert. Dieses LLM wird eingesetzt, um auf Anfragen von Benutzern zu antworten. Zum Beispiel könnte ein Benutzer eingeben

> How can I live a more environmentally friendly life?

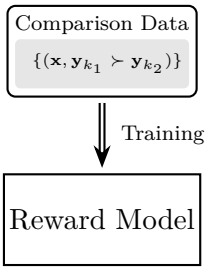
Wir verwenden das LLM, um 4 verschiedene Ausgaben (bezeichnet mit $\{\mathbf{y}_1, ..., \mathbf{y}_4\}$) durch Sampling des Ausgaberaums zu generieren



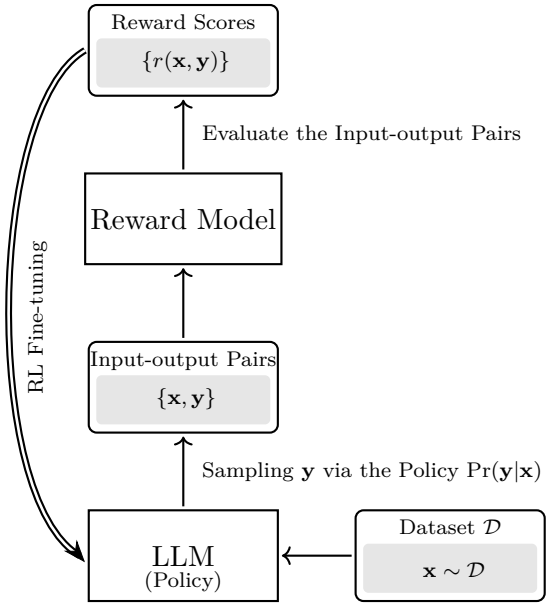
(a) Learning an Initial LLM



(b) Annotating Data with Human Preferences



(c) Training the Reward Model



(d) Training/Fine-tuning the Policy

Abbildung 2.2: Ein Überblick über RLHF. Es sind 4 Schlüsselschritte beteiligt: a) Training eines initialen LLM (d.h. einer Policy) durch Pre-Training und überwachtes Fine-Tuning; b) Sammeln von menschlichen Präferenzdaten durch Ranking der Ausgaben des LLM; c) Training eines Belohnungsmodells unter Verwendung der Ranking-Ergebnisse; d) RL-Feinabstimmung der Policy basierend auf dem Belohnungsmodell. Doppellinienpfeile bedeuten Training oder Feinabstimmung.

Output 1 ($\mathbf{y}_1$): Consider switching to an electric vehicle or bicycle instead of traditional cars to reduce carbon emissions and protect our planet.

Output 2 ($\mathbf{y}_2$): Adopt a minimalist lifestyle. Own fewer possessions to reduce consumption and the environmental impact of manufacturing and disposal.

Output 3 ($\mathbf{y}_3$): Go off-grid. Generate your own renewable energy and collect rainwater to become completely self-sufficient and reduce reliance on non-renewable resources.

Output 4 ($\mathbf{y}_4$): Support local farm products to reduce the carbon footprint of transporting food, while enjoying fresh, healthy food.

Anschließend bitten wir Annotatoren, diese Ausgaben zu bewerten. Eine einfache Möglichkeit besteht darin, jeder Ausgabe eine Bewertungspunktzahl zuzuordnen. In diesem Fall kann das Problem des Lernens des Belohnungsmodells als Aufgabe zum Trainieren eines Regressionsmodells formuliert werden. Das Vergeben numerischer Punktzahlen für LLM-Ausgaben ist jedoch keine leichte Aufgabe für Annotatoren. Es ist in der Regel schwierig, einen Annotationsstandard zu entwerfen, dem alle Annotatoren zustimmen und den sie leicht befolgen können. Eine alternative Methode, die bei der Entwicklung von LLMs beliebter ist, besteht darin, diese Ausgaben in eine Rangfolge zu bringen. Ein mögliches Ranking der obigen Ausgaben ist zum Beispiel

$$\mathbf{y}_1 \succ \mathbf{y}_4 \succ \mathbf{y}_2 \succ \mathbf{y}_3$$

Anschließend wird ein Belohnungsmodell unter Verwendung dieses Ranking-Ergebnisses trainiert. Im Allgemeinen ist ein Belohnungsmodell in RLHF ein Sprachmodell, das dieselbe Architektur wie das Ziel-LLM hat, aber eine kleinere Modellgröße aufweist. Gegeben die Eingabe $\mathbf{x}$ und die Ausgabe $\mathbf{y}_k$, konkatenieren wir sie zu einer Sequenz $\text{seq}_k = [\mathbf{x}, \mathbf{y}_k]$. Diese Sequenz wird von links nach rechts mittels erzwungener Dekodierung verarbeitet. Da jede Position beim Sprachmodellieren nur auf ihren linken Kontext zugreifen kann, kann die Ausgabe der obersten Transformer-Schicht an der ersten Position nicht als Repräsentation der Sequenz verwendet werden. Stattdessen wird ein spezielles Symbol (z. B. $\langle \backslash s \rangle$) am Ende der Sequenz hinzugefügt, und die entsprechende Ausgabe des Stapels von Transformer-Schichten wird als Repräsentation der gesamten Sequenz betrachtet. Eine Ausgabeschicht, wie z. B. eine lineare Transformationsschicht, wird auf dieser Repräsentation aufgebaut, um die Belohnung zu generieren, die mit $R(\text{seq}_k)$ oder $R(\mathbf{x}, \mathbf{y}_k)$ bezeichnet wird.

Wir trainieren dieses Belohnungsmodell mithilfe eines Ranking-Verlusts. Zum Beispiel kann eine paarweise Ranking-Verlustfunktion in der Form

$$\text{Loss}_\omega(\mathcal{D}_r) \quad = \quad -\mathbb{E}_{(\mathbf{x},\mathbf{y}_{k_1},\mathbf{y}_{k_2})\sim\mathcal{D}_r} \log(\text{Sigmoid}(R_\omega(\mathbf{x}, \mathbf{y}_{k_1}) - R_\omega(\mathbf{x}, \mathbf{y}_{k_2}))) \tag{2.19}$$

geschrieben werden, wobei ω die Parameter des Belohnungsmodells darstellt und $\mathcal{D}_r$ eine Menge von Tupeln aus einer Eingabe und einem Paar von Ausgaben repräsentiert. $(\mathbf{x}, \mathbf{y}_{k_1}, \mathbf{y}_{k_2}) \sim \mathcal{D}_r$ ist eine Sampling-Operation, die eine Stichprobe $(\mathbf{x}, \mathbf{y}_{k_1}, \mathbf{y}_{k_2})$ aus $\mathcal{D}_r$ mit einer gewissen Wahrscheinlichkeit zieht. Nehmen wir als Beispiel an, wir ziehen zuerst eine Modelleingabe $\mathbf{x}$ mit einer Gleichverteilung und dann ein Paar von Modellausgaben mit einer Wahrscheinlichkeit von $\mathbf{y}_{k_1} \succ \mathbf{y}_{k_2}$ gegeben $\mathbf{x}$ (bezeichnet als $\Pr(\mathbf{y}_{k_1} \succ \mathbf{y}_{k_2}|\mathbf{x})$). Die entsprechende Verlustfunktion ist gegeben durch

$$\begin{aligned} & \text{Loss}_\omega(\mathcal{D}_r) \\ = \quad & -\sum \Pr(\mathbf{x}) \cdot \Pr(\mathbf{y}_{k_1} \succ \mathbf{y}_{k_2}|\mathbf{x}) \cdot \log(\text{Sigmoid}(R_\omega(\mathbf{x}, \mathbf{y}_{k_1}) - R_\omega(\mathbf{x}, \mathbf{y}_{k_2}))) \\ = \quad & -\frac{1}{K} \sum \Pr(\mathbf{y}_{k_1} \succ \mathbf{y}_{k_2}|\mathbf{x}) \cdot \log(\text{Sigmoid}(R_\omega(\mathbf{x}, \mathbf{y}_{k_1}) - R_\omega(\mathbf{x}, \mathbf{y}_{k_2}))) \end{aligned} \tag{2.20}$$

wobei K die Anzahl der in das Sampling einbezogenen Modelleingaben darstellt. Obwohl die Form dieser Funktionen komplex erscheinen mag, ist ihre Idee einfach: Wir bestrafen das Modell, wenn das vorhergesagte Ranking von zwei Ausgaben vom von Menschen erstellten Ranking abweicht. Im Gegensatz dazu erhält das Modell einen Bonus, wenn das

vorhergesagte Ranking mit dem von Menschen erstellten Ranking übereinstimmt.

Wir können das Belohnungsmodell trainieren, indem wir den obigen Ranking-Verlust minimieren

$$\hat{\omega} \quad = \quad \arg\min_{\omega} \text{Loss}_{\omega}(\mathcal{D}_r) \tag{2.21}$$

Das resultierende Modell $R_{\hat{\omega}}(\cdot)$ kann verwendet werden, um jedes gegebene Paar aus Eingabe und Ausgabe zu bewerten. Beachten Sie, dass, obwohl das Belohnungsmodell mit einem Ranking-basierten Ziel trainiert wird, es zur Bewertung (Scoring) verwendet wird. Dies ermöglicht es, kontinuierliche Überwachungssignale zu liefern, was für das Training anderer Modelle sehr vorteilhaft ist.

Wir wenden uns nun dem Problem des Policy-Lernens zu. Ein häufig verwendetes Ziel ist die Maximierung der Belohnung für eine Menge von Eingabe-Ausgabe-Paaren. In Analogie zur Form von Gl. (2.16) erhalten wir ein einfaches Trainingsziel für das RL-Fine-Tuning

$$\tilde{\theta} \quad = \quad \arg\max_{\hat{\theta}^+} \mathbb{E}_{(\mathbf{x},\mathbf{y}_{\hat{\theta}^+})\sim\mathcal{D}_{\text{rlft}}} R_{\hat{\omega}}(\mathbf{x}, \mathbf{y}_{\hat{\theta}^+}) \tag{2.22}$$

wobei die optimalen Parameter $\tilde{\theta}$ durch Feinabstimmung (Fine-Tuning) der vortrainierten Parameter $\hat{\theta}$ erhalten werden. $\mathcal{D}_{\text{rlft}}$ ist der RL-Fine-Tuning-Datensatz. Für jede Stichprobe $(\mathbf{x}, \mathbf{y}_{\hat{\theta}^+})$ wird $\mathbf{x}$ aus einem vorbereiteten Datensatz von Eingabesequenzen gesampelt, und $\mathbf{y}_{\hat{\theta}^+}$ wird aus der durch die Policy gegebenen Verteilung $\Pr_{\hat{\theta}^+}(\mathbf{y}|\mathbf{x})$ gesampelt.

In der Praxis werden oft fortgeschrittenere Algorithmen des bestärkenden Lernens, wie die proximale Policy-Optimierung (PPO), verwendet, um ein stabileres Training sowie eine bessere Leistung zu erzielen. Wir überlassen die detaillierte Diskussion von Algorithmen des bestärkenden Lernens den folgenden Teilen dieses Buches, in denen RLHF ausgiebig für das Alignment verwendet wird.

Hier stellt sich eine interessante Frage: Warum betrachtet man das Lernen aus menschlichen Präferenzen nicht als ein standardmäßiges Problem des überwachten Lernens? Diese Frage ist eng mit unserer oben genannten Diskussion über die Schwierigkeit der Datenannotation verknüpft. Oft ist die Beschreibung menschlicher Werte und Ziele eine Herausforderung, und es ist für Menschen noch schwieriger, Ausgaben zu liefern, die gut angeglichen sind. Alternativ dazu bietet das Annotieren der Präferenzen einer gegebenen Liste von Modellausgaben eine einfachere Aufgabe. Dadurch können wir ein Modell erstellen, das menschliche Präferenzen versteht, welches dann als Belohnungsmodell für das Trainieren von Policies verwendet werden kann. Aus der Perspektive des maschinellen Lernens ist RLHF besonders nützlich für Szenarien, in denen das gewünschte Verhalten eines Agenten schwer zu demonstrieren, aber von Menschen leicht zu erkennen ist. Ein weiterer Vorteil von RLHF ist seine Fähigkeit, den Stichprobenraum zu erkunden. Durch den Einsatz von Sampling-Techniken können mit bestärkendem Lernen trainierte Modelle über den annotierten Datensatz hinausgehen, um zusätzliche Stichproben zu erkunden. Diese explorative Fähigkeit ermöglicht es RLHF, potenziell vorteilhafte Policies zu entdecken, die aus den gelabelten Daten allein nicht sofort ersichtlich sind.

2.1.5 Prompting von LLMs

Wir haben bisher gezeigt, dass LLMs verwendet werden können, um verschiedene Aufgaben durchzuführen, indem man ihnen entsprechende Prompts gibt. Es gibt keine Einschränkungen für diese Prompts, die beliebige Informationen enthalten können, die wir den LLMs fragen oder mitteilen möchten, wie z. B. Anweisungen in natürlicher Sprache und den Kontext von Konversationen. Da dieser Ansatz kein zusätzliches Training oder Tuning erfordert, wird die Anpassung von LLMs nach ihrer Entwicklung hocheffizient. Dies beeinflusst in gewissem Maße die Paradigmen im NLP: Wir müssen nicht mehr spezifische Systeme für einzelne Aufgaben entwickeln, sondern können stattdessen ein einziges, gut trainiertes LLM verwenden, um verschiedene Aufgaben durch Prompting zu erledigen. Ein reizvoller Aspekt des LLM-Promptings ergibt sich daraus: Benutzer können leicht „maßgeschneiderte“Systeme haben, indem sie ihre eigenen Prompts für LLMs entwerfen. Angesichts der wichtigen Rolle, die das Prompting bei LLMs spielt, ist prompting engineering zu einem sehr aktiven Forschungsbereich im NLP geworden.

Der Begriff prompt wird in der Literatur auf viele verschiedene Weisen verwendet. In diesem Kapitel bezieht sich dieser Begriff auf die gesamte Eingabe für LLMs, und daher verwenden wir die Begriffe prompt und Modelleingabe austauschbar. Bevor wir das Prompting weiter diskutieren, betrachten wir zunächst einige Beispiele, bei denen die Prompts komplexer sind als die in den vorherigen Unterabschnitten vorgestellten. Beachten Sie, dass dieser Unterabschnitt nicht darauf abzielt, qualitativ hochwertige Prompts zu schreiben, sondern vielmehr darauf, einige interessante Aspekte beim Prompting von LLMs hervorzuheben.

Eine der beliebtesten Methoden zur Verwendung von LLMs besteht darin, ihnen eine „Rolle“zuzuweisen, die sie bei der Generierung von Antworten spielen. Zum Beispiel können LLMs bei der Beantwortung von Fragen als Psychologe fungieren.

> Please explain what delayed gratification is?
>
> Note, you are a researcher with a deep background in psychology. Your own field of study is developmental psychology in children, with a focus on aspects of control and intelligence. You are very familiar with the concept of delayed gratification and have applied it to your own research. You have published papers on delayed gratification and conducted related experiments.
>
> Delayed gratification is the process of resisting an immediate reward in anticipation of receiving a more valuable reward in the future. It is a significant concept in developmental psychology, especially in relation to how children learn self-control and develop cognitive abilities. Research shows that the ability to delay gratification is associated with positive outcomes in academic performance, social competence, and emotional well-being. Studies and experiments in this area highlight the importance of cultivating this skill early in life to support long-term success and resilience.

Ein weiteres Beispiel ist die Verwendung von LLMs zur Erkennung und Korrektur von Fehlern wie syntaktischen oder semantischen Fehlern im Text. Für ein LLM, das sowohl auf

Code als auch auf Daten in natürlicher Sprache trainiert ist, können wir es zum Debuggen von Code verwenden[6].

```
Fix the bugs in this C language program.
#include <stdio.h>
int main() {
    printg("Hello, World!")
    return 0;
}
1. The function name 'printg' is incorrect . It should be 'printf'.
2. There is a missing semicolon at the end of the 'printf' function call.
Here is the corrected version of the program:
#include <stdio.h>
int main() {
    printf("Hello, World!");
    return 0;
}
```

Außerdem können LLMs so feinabgestimmt werden, dass sie als Gesprächspartner fungieren [7].

```
Hi! I'm an assistant. How can I help you?
Who won the FIFA World Cup 2022?
Argentina won the FIFA World Cup 2022.
Where was it held?
The 2022 FIFA World Cup was held in Qatar.
How many times has Argentina won the World Cup?
Argentina has won the FIFA World Cup three times.
...
```

Diese und frühere Beispiele haben gezeigt, dass durch Prompts mit klaren Anweisungen und Fragen angemessene Antworten generiert werden können. Wenn die Problemlösung jedoch Wissen erfordert, das nicht explizit spezifiziert ist, können LLMs Fehler machen, auch wenn die Anweisungen ausreichend klar und präzise sind. Eine Familie herausfordernder Aufgaben für LLMs umfasst arithmetisches Schlussfolgern und Alltagsverständnis (Commonsense Reasoning). Zum Beispiel können wir ein LLM bitten, Grundschul-Mathematikaufgaben zu lösen, die in natürlicher Sprache präsentiert werden.

[6] In diesem Beispiel wird der Code zur besseren Lesbarkeit nicht tokenisiert.

[7] Um ein LLM für einen mehrründigen Dialog feinabzustimmen, muss die Konversationshistorie im Kontext berücksichtigt werden, um die Antwort in der aktuellen Gesprächsrunde vorherzusagen. Dadurch wird der tatsächlich bei der Antwortgenerierung verwendete Prompt relativ länger als der bei einem einründigen Dialog verwendete.

> Jack has 7 apples. He ate 2 of them for dinner, but then his mom gave him 5 more apples. The next day, Jack gave 3 apples to his friend John. How many apples does Jack have left in the end?
>
> The answer is 10.

Die richtige Antwort sollte 7 sein, daher ist die Modellausgabe falsch.

Ein Ansatz zur Lösung solcher Probleme besteht darin, Lernen in Prompts zu integrieren, was als in-context learning oder (ICL) bezeichnet wird. Die Idee von ICL besteht darin, in Prompts die Lösungswege für Probleme zu demonstrieren und die Vorhersagen auf diesen Demonstrationen zu konditionieren. Hier ist ein Beispiel, bei dem ein ähnliches Problem und die entsprechende Antwort im Prompt präsentiert werden (grün = Demonstrationen).

> Tom has 12 marbles. He wins 7 more marbles in a game with his friend but then loses 5 marbles the next day. His brother gives him another 3 marbles as a gift. How many marbles does Tom have now?
>
> The answer is 17.
>
> Jack has 7 apples. He ate 2 of them for dinner, but then his mom gave him 5 more apples. The next day, Jack gave 3 apples to his friend John. How many apples does Jack have left in the end?
>
> The answer is 12.

Aber das LLM hat diesmal trotzdem Fehler gemacht. Ein Grund dafür könnte sein, dass das Lösen von Mathematikaufgaben nicht nur Problem-Antwort-Zuordnungen beinhaltet, sondern in größerem Maße auch die zugrunde liegende logische Schlussfolgerung in mehreren Schritten. Eine Methode zur Verbesserung der Inferenzfähigkeiten von LLMs ist das chain-of-thought prompting (COT prompting) [Wei et al., 2022c]. Beim COT-Prompting zerlegen wir komplexe Schlussfolgerungsprobleme in mehrere zwischengeschaltete Problemlösungsschritte. Diese Schritte werden in Prompts demonstriert, sodass LLMs dazu veranlasst werden können, zu lernen, die zugrunde liegenden Denkprozesse beim Lösen der Probleme nachzuahmen. COT-Prompting hat sich als sehr nützlich erwiesen, um gute Ergebnisse bei anspruchsvollen Aufgaben zu erzielen, wie z. B. dem GSM8K-Benchmark für mathematisches Schlussfolgern.

Betrachten Sie das obige Demonstrationsbeispiel Tom has 12 marbles. He wins 7 more marbles in a game with his friend but then loses 5 marbles the next day. His brother gives him another 3 marbles as a gift. How many marbles does Tom have now ?. Die entsprechenden logischen Schlussfolgerungsschritte können wie folgt strukturiert werden:

1. Tom beginnt mit 12 Murmeln.
2. Er gewinnt 7 weitere Murmeln in einem Spiel, also addieren wir diese zu seiner anfänglichen Anzahl: 12 Murmeln + 7 Murmeln = 19 Murmeln.
3. Am nächsten Tag verliert Tom 5 Murmeln, also subtrahieren wir diese von seiner

aktuellen Gesamtzahl: 19 Murmeln - 5 Murmeln = 14 Murmeln.

4. Sein Bruder gibt ihm 3 weitere Murmeln, also addieren wir diese zu seiner aktuellen Gesamtzahl: 14 Murmeln + 3 Murmeln = 17 Murmeln.

5. Nach all diesen Ereignissen hat Tom 17 Murmeln.

Wir können diese Schlussfolgerungsschritte in den Prompt einfügen und erhalten die richtige Antwort.

> Tom has 12 marbles. He wins 7 more marbles in a game with his friend but then loses 5 marbles the next day. His brother gives him another 3 marbles as a gift. How many marbles does Tom have now?
>
> Tom initially has 12 marbles. After winning a game, he gains 7 more, increasing his total to 19 marbles. He loses 5 marbles the next day, bringing his total down to 14 marbles. His brother gifts him 3 more marbles, increasing his total to 17 marbles. Therefore, Tom now has 17 marbles. So the answer is 17.
>
> Jack has 7 apples. He ate 2 of them for dinner, but then his mom gave him 5 more apples. The next day, Jack gave 3 apples to his friend John. How many apples does Jack have left in the end?
>
> Jack starts with 7 apples. He eats 2 apples for dinner, so we subtract 2 from 7, leaving him with 5 apples. His mom gives him 5 more apples, adding 5 to his current total. Now he has 5 + 5 = 10 apples. The next day, Jack gives 3 apples to his friend John. Subtracting 3 from 10, Jack is left with 7 apples. Therefore, Jack has 7 apples left. So the answer is 7.

Dieser Ansatz wird auch als one-shot COT prompting bezeichnet. Ebenso können wir weitere Demonstrationen hinzufügen, um einen few-shot COT prompting-Ansatz zu erhalten.

Darüber hinaus führt eine spezielle Art von COT, zero-shot COT, Anweisungen ein, die LLMs zum Nachdenken anregen, ohne Zwischenschritte des Schlussfolgerns in den Prompts zu präsentieren [Kojima et al., 2022]. Ein interessantes Beispiel ist, dass durch das Anhängen eines einfachen Satzes „Let's think step by step."am Ende eines Prompts LLMs Schlussfolgerungsschritte generieren können, um zur endgültigen Antwort zu gelangen.

Jack has 7 apples. He ate 2 of them for dinner, but then his mom gave him 5 more apples. The next day, Jack gave 3 apples to his friend John. How many apples does Jack have left in the end?
Let's think step by step.

1. Initial Quantity: Jack starts with 7 apples.
2. After Dinner: He eats 2 apples, so $7 - 2 = 5$ apples remain.
3. His Mom Gives More: His mom gives him 5 more apples, so $5 + 5 = 10$ apples are now with him.
4. Giving to John: The next day, Jack gives 3 apples to his friend John, so $10 - 3 = 7$ apples are left.

In the end, Jack has 7 apples left.

Zero-Shot-, One-Shot- und Few-Shot-Lernen sind gängige Konzepte im Bereich des In-Kontext-Lernens für LLMs und sind nicht auf das COT-Prompting beschränkt. Im Großen und Ganzen kann jedes Prompting, das nur einfache Anweisungen ohne Demonstrationen enthält, als eine Form des Zero-Shot-Lernens betrachtet werden. Diese Fähigkeit zum Zero-Shot-Lernen entsteht, wenn LLMs vor-trainiert und/oder feinabgestimmt werden. Außerdem werden One-Shot- und Few-Shot-Lernmethoden häufiger in Betracht gezogen, wenn LLMs die entsprechende Fähigkeit zum Zero-Shot-Lernen nicht erwerben. Diese Methoden sind daher wichtig für das In-Kontext-Lernen bei der Bewältigung neuer Aufgaben. Beispiele hierfür sind solche zur Durchführung verschiedener NLP-Aufgaben durch die Demonstration von aufgabenformatierten Beispielen. Siehe die folgenden Beispiele für die Klassifizierung von Gefühlssätzen und die Übersetzung von Phrasen mittels Few-Shot-Lernen.

Given the following text snippets, classify their sentiment as Positive, Negative, or Neutral.

Example 1: "I had an amazing day at the park!"
Sentiment: Positive

Example 2: "The service at the restaurant was terrible."
Sentiment: Negative

Example 3: "I think it's going to rain today."
Sentiment: Neutral

Text: "This movie was a fantastic journey through imagination."
Sentiment: Positive

Translate the following Chinese phrases into English.

Example 1: “你好”
Translation: “Hello”

Example 2: “谢谢你”
Translation: “Thank you”

Phrase to translate: “早上好”
Translation: “Good Morning”

Oben haben wir Beispiele vorgestellt, um die grundlegenden Fähigkeiten des In-Kontext-Lernens beim Prompting von LLMs zu veranschaulichen. Dieser Abschnitt enthält jedoch keine fortgeschritteneren Prompting-Techniken, um den Inhalt kurz und kompakt zu halten. Weitere Diskussionen zum Prompting finden sich in Kapitel 3.

2.2 Training in großem Maßstab

Als erster Schritt bei der Entwicklung von LLMs müssen wir diese Modelle mit großen Datenmengen trainieren. Die Trainingsaufgabe selbst ist Standard: Das Ziel ist die Maximierung der Likelihood, was mittels Der verlauf sinkt. erreicht werden kann. Wenn wir jedoch sowohl die Modellgröße als auch die Datenmenge skalieren, wird das Problem sehr herausfordernd, da große Modelle das Training im Allgemeinen instabil machen. In diesem Abschnitt erörtern wir verschiedene Aspekte des groß angelegten Trainings für LLMs, einschließlich Datenaufbereitung, Modellmodifikation und verteiltes Training. Wir diskutieren auch die Skalierungsgesetze für LLMs, die uns helfen, ihre Trainingseffizienz und -effektivität zu verstehen.

2.2.1 Datenaufbereitung

Die Bedeutung von Daten kann im Bereich des NLP nicht hoch genug eingeschätzt werden. Mit der Entwicklung größerer neuronaler Netzwerke steigt auch der Bedarf an Daten kontinuierlich. Beispielsweise kann die Entwicklung von LLMs Billionen von Tokens im Vortraining erfordern (siehe Tabelle 2.3), was um Größenordnungen mehr ist als bei herkömmlichen NLP-Modellen. Im Allgemeinen möchte man so viele Trainingsdaten wie möglich sammeln. Größere Trainingsdatensätze führen jedoch nicht zwangsläufig zu besseren Trainingsergebnissen, und die Entwicklung von LLMs wirft neue Probleme bei der Erstellung oder Sammlung dieser Datensätze auf.

Ein erstes Problem ist die Qualität der Daten. Hochwertige Daten gelten seit langem als entscheidend für das Training datengesteuerter NLP-Systeme. Die direkte Verwendung von Rohtext aus verschiedenen Quellen ist im Allgemeinen unerwünscht. Ein erheblicher Teil der Daten, die zum Trainieren neuerer LLMs verwendet werden, stammt beispielsweise aus dem Web-Scraping und kann Fehler und unangemessene Inhalte wie toxische Informationen und erfundene Fakten enthalten. Außerdem wird das Internet aufgrund des weit verbreiteten Einsatzes von KI mit maschinell erstellten Inhalten überschwemmt,

LLM	# of Tokens	Data
GPT3-175B [Brown et al., 2020]	0.5T	Webpages, Books, Wikipedia
Falcon-180B [Almazrouei et al., 2023]	3.5T	Webpages, Books, Conversations, Code, Technical Articles
LLaMA2-65B [Touvron et al., 2023a]	1.0T $\sim$ 1.4T	Webpages, Code, Wikipedia, Books, Papers, Q&As
PaLM-450B [Chowdhery et al., 2022]	0.78T	Webpages, Books, Conversations, Code, Wikipedia, News
Gemma-7B [Gemma Team, 2024]	6T	Webpages, Mathematics, Code

Tabelle 2.3: Trainingsdatenmengen einiger LLMs, angegeben in der Anzahl der Tokens.

was weitere Herausforderungen für die Verarbeitung und Nutzung von Web-Scraping-Daten mit sich bringt. Forscher haben herausgefunden, dass das Trainieren von LLMs mit ungefilterten Daten schädlich ist [Raffel et al., 2020]. Die Verbesserung der Datenqualität umfasst typischerweise die Integration von Filter- und Bereinigungsschritten in den Datenverarbeitungsprozess. Zum Beispiel zeigen Penedo et al. [2023], dass durch die Anwendung einer Reihe von Datenverarbeitungstechniken 90% ihrer durch Web-Scraping gewonnenen Daten für das LLM-Training entfernt werden können. Zusätzlich zu den umfangreichen Web-Scraping-Daten umfassen die Trainingsdaten für LLMs oft auch Bücher, wissenschaftliche Arbeiten, nutzergenerierte Daten aus sozialen Medien und so weiter. Die meisten der neuesten LLMs werden auf solchen kombinierten Datensätzen trainiert, die sich als wichtig für die hohe Leistungsfähigkeit der resultierenden Modelle erwiesen haben.

Ein zweites Problem ist die Diversität der Daten. Wir möchten, dass die Trainingsdaten so viele Datentypen wie möglich abdecken, damit sich die trainierten Modelle leicht an verschiedene nachgelagerte Aufgaben anpassen können. Es ist weithin anerkannt, dass sowohl die Qualität als auch die Diversität der Trainingsdaten eine sehr wichtige Rolle bei LLMs spielen. Ein interessantes Beispiel ist, dass die Einbeziehung von Programmiercode in die Trainingsdaten sich als vorteilhaft für LLMs erwiesen hat. Die Vorteile zeigen sich nicht nur in der Verbesserung der Programmierfähigkeiten von LLMs, sondern auch in der Verbesserung des logischen Denkens bei komplexen Problemen, insbesondere bei solchen, die COT-Prompting erfordern. Das Konzept der „Diversität“kann auch auf die sprachliche Vielfalt erweitert werden. Beispielsweise werden viele LLMs auf mehrsprachigen Daten trainiert, sodass wir mit einem einzigen Modell mehrere Sprachen verarbeiten können. Obwohl dieser Ansatz starke Fähigkeiten bei mehrsprachigen und sprachübergreifenden Aufgaben zeigt, hängt seine Leistung in bestimmten Sprachen stark vom Umfang und der Qualität der Daten für diese Sprachen ab. Es hat sich gezeigt, dass dies in einigen Fällen zu schlechten Ergebnissen bei ressourcenarmen Sprachen führt.

Ein drittes Problem ist der Bias in den Trainingsdaten. Dies ist kein spezifisches Problem von LLMs, sondern existiert in vielen NLP-Systemen. Ein häufiges Beispiel ist der Gender-Bias, bei dem LLMs eine Präferenz für ein Geschlecht gegenüber einem anderen zeigen. Dies kann teilweise auf ein Klassen-Ungleichgewicht in den Trainingsdaten zurückgeführt werden. Beispielsweise wird der Begriff Pflegekräfte häufiger mit Frauen assoziiert. Um den Bias in den Daten zu reduzieren, ist es üblich, die Kategorien verschiedener sprachlicher Phänomene wie Geschlecht, Ethnizität und Dialekte auszugleichen. Der Bias in den

Daten steht auch im Zusammenhang mit dem oben erwähnten Diversitätsproblem. Da beispielsweise viele LLMs mit englisch-zentrierten Daten trainiert und ausgerichtet werden, sind sie voreingenommen gegenüber den kulturellen Werten und Perspektiven, die in englischsprachigen Bevölkerungsgruppen vorherrschen. Die Erhöhung der sprachlichen Diversität in den Trainingsdaten kann diesen Bias etwas abmildern.

Ein weiteres Problem bei der Sammlung großer Datenmengen ist der Datenschutz. Wenn LLMs auf Daten aus umfangreichen Quellen trainiert werden, kann dies potenziell zu Risiken hinsichtlich der Offenlegung sensibler Informationen wie geistigem Eigentum und persönlichen Daten führen. Dies ist besonders besorgniserregend angesichts der Fähigkeit von LLMs, Muster aus den Daten, auf denen sie trainiert wurden, zu repräsentieren, was unbeabsichtigt das Auswendiglernen und Reproduzieren spezifischer Details beinhalten könnte. Ein einfacher Ansatz zum Schutz der Privatsphäre besteht darin, sensible Informationen zu entfernen oder zu anonymisieren. Beispielsweise können Anonymisierungstechniken angewendet werden, um persönlich identifizierbare Informationen aus den Trainingsdaten zu entfernen und so zu verhindern, dass LLMs aus solchen Daten lernen. In der Praxis ist es jedoch schwierig, alle sensiblen Daten zu löschen oder zu schwärzen. Daher arbeiten viele LLMs, insbesondere solche, die für den öffentlichen Dienst eingeführt wurden, typischerweise mit Systemen, die die potenzielle Offenlegung sensibler Daten erkennen können, oder sind so feinabgestimmt, dass sie bestimmte Anfragen ablehnen, die zu Informationslecks führen könnten.

2.2.2 Modellmodifikationen

Das Training von LLMs ist schwierig. Ein häufig auftretendes Problem ist, dass der Trainingsprozess mit zunehmender Größe der LLMs instabiler wird. Beispielsweise muss eine kleine Lernrate gewählt werden, um ein stabiles Training mit dem Gradientenabstieg zu erreichen, was jedoch zu deutlich längeren Trainingszeiten führt. Manchmal kann das Training, selbst bei sorgfältig gestalteter Trainingskonfiguration, an bestimmten Stellen während der Optimierung divergieren. Das Training von LLMs wird im Allgemeinen von vielen Faktoren beeinflusst, wie z. B. der Parameterinitialisierung, dem Batching und der Regularisierung. Hier konzentrieren wir uns auf gängige Modifikationen und Verbesserungen der Standard-Transformer -Architektur, die bei der Entwicklung trainierbarer LLMs als wichtig erachtet werden.

2.2.2.1 1. Layer-Normalisierung mit Residualverbindungen

Layer-Normalisierung wird verwendet, um das Training von tiefen neuronalen Netzen zu stabilisieren. Es ist ein Prozess, bei dem der Mittelwert subtrahiert und durch die Standardabweichung geteilt wird. Durch die Normalisierung der Schichtausgabe auf diese Weise können wir das Problem der Kovariatenverschiebung effektiv reduzieren und die Trainingsstabilität verbessern. In Transformer -Modellen wird die Layer-Normalisierung typischerweise zusammen mit Residualverbindungen verwendet. Wie in Abschnitt 2.1.1 beschrieben, kann ein Sub-Layer entweder auf der Post-Norm-Architektur basieren, bei der die Layer-Normalisierung direkt nach einem Residualblock durchgeführt wird, oder

auf der Pre-Norm-Architektur, bei der die Layer-Normalisierung innerhalb eines Residualblocks durchgeführt wird. Obwohl beide Architekturen in Transformer -basierten Systemen [Wang et al., 2019] weit verbreitet sind, hat sich die Pre-Norm-Architektur als besonders nützlich für das Training tiefer Transformer erwiesen. Daher basieren die meisten LLMs auf der Pre-Norm-Architektur, ausgedrückt als output = LNorm(F(input))+input.

Eine weit verbreitete Form der Layer-Normalisierungsfunktion ist gegeben durch

$$\text{LNorm}(\mathbf{h}) \quad = \quad \alpha \cdot \frac{\mathbf{h} - \mu}{\sigma + \epsilon} + \beta \tag{2.23}$$

wobei $\mathbf{h}$ ein d-dimensionaler reellwertiger Vektor ist, μ der Mittelwert aller Einträge von $\mathbf{h}$ ist und σ die entsprechende Standardabweichung ist. ϵ wird aus Gründen der numerischen Stabilität eingeführt. $\alpha \in \mathbb{R}^d$ und $\beta \in \mathbb{R}^d$ sind die Gain- und Bias-Terme.

Eine Variante der Layer-Normalisierung, die als Root Mean Square (RMS) Layer-Normalisierung bezeichnet wird, skaliert den Eingabevektor nur neu, zentriert ihn aber nicht neu [Zhang and Sennrich, 2019]. Die RMS-Layer-Normalisierungsfunktion ist gegeben durch

$$\text{LNorm}(\mathbf{h}) \quad = \quad \alpha \cdot \frac{\mathbf{h}}{\sigma_{\text{rms}} + \epsilon} + \beta \tag{2.24}$$

wobei σ_{rms} der quadratische Mittelwert von $\mathbf{h}$ ist, das heißt, $\sigma_{\text{rms}} = (\frac{1}{d}\sum_{k=1}^{d} h_k^2)^{\frac{1}{2}}$. Diese Layer-Normalisierungsfunktion wird in LLMs wie der LLaMA-Serie verwendet.

2.2.2.2 2. Aktivierungsfunktionen in FFNs

In Transformern sind FFN-Unterschichten darauf ausgelegt, Nichtlinearitäten in das Repräsentationslernen einzuführen, und es wurde festgestellt, dass sie nützlich sind, um die Degeneration der durch Selbst-Aufmerksamkeit gelernten Repräsentationen zu verhindern[8] [Dong et al., 2021]. Eine Standardform der in diesen Unterschichten verwendeten FFNs kann ausgedrückt werden als

$$\text{FFN}(\mathbf{h}) \quad = \quad \sigma(\mathbf{h}\mathbf{W}_h + \mathbf{b}_h)\mathbf{W}_f + \mathbf{b}_f \tag{2.25}$$

wobei $\mathbf{W}_h \in \mathbb{R}^{d \times d_h}$, $\mathbf{b}_h \in \mathbb{R}^{d_h}$, $\mathbf{W}_f \in \mathbb{R}^{d_h \times d}$ und $\mathbf{b}_f \in \mathbb{R}^d$ die Parameter sind und d_h die Dimension der versteckten Schicht ist. $\sigma(\cdot)$ ist die Aktivierungsfunktion der versteckten Schicht. Eine gängige Wahl für $\sigma(\cdot)$ ist die gleichgerichtete lineare Einheit (ReLU), gegeben durch

$$\sigma_{\text{relu}}(\mathbf{h}) \quad = \quad \max(0, \mathbf{h}) \tag{2.26}$$

In praktischen Implementierungen ist eine Erhöhung von d_h hilfreich, und daher wird es in LLMs oft auf eine größere Zahl gesetzt. Aber eine sehr große Dimension der versteckten Schicht stellt sowohl für das Training als auch für den Einsatz Herausforderungen dar. In diesem Fall spielt das Design der Aktivierungsfunktion eine relativ wichtigere Rolle

[8] Hier bezieht sich Degeneration auf das Phänomen, bei dem der Rang einer Matrix nach einer Verarbeitung reduziert wird.

in breiten FFNs. Es gibt mehrere Alternativen zur ReLU in LLMs. Eine davon ist die Gaußsche Fehler-Lineareinheit (GeLU), die als eine geglättete Version der ReLU angesehen werden kann. Anstatt die Ausgabe durch das Vorzeichen der Eingabe zu steuern, gewichtet die GeLU-Funktion ihre Eingabe mit dem Perzentil $\Pr(h \leq \mathbf{h})$. Hier ist h ein d-dimensionaler Vektor, dessen Einträge aus der Standardnormalverteilung Gaussian$(0, 1)$ gezogen werden[9]. Insbesondere ist die GeLU-Funktion definiert als

$$\begin{aligned} \sigma_{\text{gelu}}(\mathbf{h}) &= \mathbf{h} \Pr(h \leq \mathbf{h}) \\ &= \mathbf{h}\Phi(\mathbf{h}) \end{aligned} \tag{2.27}$$

wobei $\Phi(\mathbf{h})$ die kumulative Verteilungsfunktion von Gaussian$(0, 1)$ ist, die auf bequeme Weise implementiert werden kann [Hendrycks and Gimpel, 2016]. Die GeLU-Funktion wurde in mehreren LLMs, wie z.B. BERT, GPT-3 und BLOOM, übernommen.

Eine weitere Familie von Aktivierungsfunktionen, die in LLMs beliebt ist, sind auf der gesteuerten linearen Einheit (GLU) basierende Funktionen. Die Grundform von GLUs ist gegeben durch

$$\sigma_{\text{glu}}(\mathbf{h}) = \sigma(\mathbf{h}\mathbf{W}_1 + \mathbf{b}_1) \odot (\mathbf{W}_2 + \mathbf{b}_2) \tag{2.28}$$

wobei $\mathbf{W}_1 \in \mathbb{R}^{d \times d}$, $\mathbf{b}_1 \in \mathbb{R}^d$, $\mathbf{W}_2 \in \mathbb{R}^{d \times d}$ und $\mathbf{b}_2 \in \mathbb{R}^d$ Modellparameter sind. Unterschiedliche Wahlen für $\sigma(\cdot)$ führen zu unterschiedlichen Versionen von GLU-Funktionen. Wenn zum Beispiel $\sigma(\cdot)$ als die GeLU-Funktion definiert ist, erhalten wir die GeGLU-Funktion

$$\sigma_{\text{geglu}}(\mathbf{h}) = \sigma_{\text{gelu}}(\mathbf{h}\mathbf{W}_1 + \mathbf{b}_1) \odot (\mathbf{W}_2 + \mathbf{b}_2) \tag{2.29}$$

Diese Aktivierungsfunktion wurde erfolgreich in LLMs wie Gemma angewendet.

Als weiteres Beispiel betrachten wir $\sigma(\cdot)$ als die Swish-Funktion $\sigma_{\text{swish}}(\mathbf{h}) = \mathbf{h} \odot \text{Sigmoid}(c\mathbf{h})$ [Ramachandran et al., 2017]. Dann ist die SwiGLU-Funktion gegeben durch

$$\sigma_{\text{swiglu}}(\mathbf{h}) = \sigma_{\text{swish}}(\mathbf{h}\mathbf{W}_1 + \mathbf{b}_1) \odot (\mathbf{W}_2 + \mathbf{b}_2) \tag{2.30}$$

Sowohl die PaLM- als auch die LLaMA-Reihe basieren auf der SwiGLU-Funktion. Für weitere Diskussionen über GLUs wird der Leser auf die Arbeit von Shazeer [2020] verwiesen.

2.2.2.3 3. Entfernen von Bias-Termen

Ein weiteres beliebtes Modelldesign ist es, die Bias-Terme in den in LLMs verwendeten affinen Transformationen zu entfernen. Diese Vorgehensweise kann auf die Layer-Normalisierung, Transformationen der Eingaben für die QKV-Attention und FFNs angewendet werden. Beispielsweise können wir Gl. (2.25) modifizieren, um ein FFN ohne Bias-Terme zu erhalten

$$\text{FFN}(\mathbf{h}) = \sigma(\mathbf{h}\mathbf{W}_h)\mathbf{W}_f \tag{2.31}$$

[9] $\Pr(h \leq \mathbf{h})$ ist eine informelle Notation. Sie bezieht sich auf einen Vektor, bei dem jeder Eintrag das Perzentil für den entsprechenden Eintrag von $\mathbf{h}$ darstellt.

Chowdhery et al. [2022] berichten, dass das Entfernen von Bias-Termen die Trainingsstabilität von LLMs verbessert. Diese Methode wurde in mehreren neueren LLMs verwendet, wie zum Beispiel LLaMA und Gemma.

2.2.2.4 4. Weitere Probleme

Viele LLMs beinhalten auch Modifikationen ihrer Positionseinbettungsmodelle. Beispielsweise können sinusförmige Positionscodierungen durch rotierende Positionseinbettungen ersetzt werden, damit die gelernten LLMs längere Sequenzen besser verarbeiten können. Diese Modelle werden in Abschnitt 2.3 besprochen.

Es ist zu beachten, dass, obwohl Modellmodifikationen beim Training von LLMs üblich sind, die Stabilität des Trainings auf viele verschiedene Weisen verbessert werden kann. Beispielsweise hat sich gezeigt, dass eine Erhöhung der Batch-Größe im Laufe des Trainings für einige LLMs nützlich ist. Im Allgemeinen erfordert das Erreichen eines stabilen und effizienten groß angelegten LLM-Trainings sorgfältig konzipierte Setups, einschließlich Lernratenplanung, Optimiererauswahl, Trainingsparallelität, Training mit gemischter Präzision und so weiter. Einige dieser Aspekte sind hochgradig technisch, und daher benötigen wir typischerweise eine Reihe von Trainingsdurchläufen, um zufriedenstellende LLMs zu erhalten.

2.2.3 Verteiltes Training

Das Training von LLMs erfordert erhebliche Mengen an Rechenressourcen. Ein gängiger Ansatz zur Verbesserung der Trainingseffizienz ist die Verwendung von großen verteilten Systemen. Glücklicherweise wurden parallel zum Aufstieg der neuronalen Netze in der KI auf Deep Learning ausgerichtete Software und Hardware entwickelt, die die Implementierung von LLMs und die Durchführung von Berechnungen erleichtern. So kann man beispielsweise heute ein LLM leicht mit Deep-Learning-Software-Frameworks und einer Maschine mit mehreren GPUs feinabstimmen. Die Skalierung des Trainings von LLMs ist jedoch nach wie vor eine Herausforderung und erfordert erhebliche Anstrengungen bei der Entwicklung von Hardware- und Softwaresystemen für ein stabiles und effizientes verteiltes Training.

Ein wichtiger Aspekt des verteilten Trainings ist die Parallelität. Es gibt verschiedene Formen der Parallelität: Datenparallelität, Modellparallelität, Tensorparallelität und Pipeline-Parallelität. Trotz unterschiedlicher Methoden zur Verteilung der Berechnungen auf die Geräte basieren diese Parallelitätsmethoden auf einer ähnlichen Idee: Das Trainingsproblem kann in kleinere Aufgaben unterteilt werden, die gleichzeitig ausgeführt werden können. Das Thema der Parallelität beim Training von LLMs wurde ausgiebig untersucht [Narayanan et al., 2021; Fedus et al., 2022]. Hier skizzieren wir die grundlegenden Konzepte.

- Datenparallelismus. Diese Methode ist eine der am weitesten verbreiteten Parallelisierungsmethoden für das Training von neuronalen Netzen. Zur Veranschaulichung

betrachten wir den einfachsten Fall, in dem die Standard-Delta-Regel beim Gradientenabstieg verwendet wird

$$\theta_{t+1} \quad = \quad \theta_t - lr \cdot \frac{\partial L_{\theta_t}(\mathcal{D}_{\text{mini}})}{\partial \theta_t} \tag{2.32}$$

wobei die neuen Parameter θ_{t+1} durch Aktualisierung der letzten Parameter θ_t mit einem kleinen Schritt lr in Richtung des negativen Verlustgradienten erhalten werden. $\frac{\partial L_{\theta_t}(\mathcal{D}_{\text{mini}})}{\partial \theta_t}$ ist der Gradient des Verlusts in Bezug auf die Parameter θ_t und wird auf einem Minibatch von Trainingsbeispielen $\mathcal{D}_{\text{mini}}$ berechnet. Beim Datenparallelismus teilen wir $\mathcal{D}_{\text{mini}}$ in N kleinere Batches auf, die mit $\{\mathcal{D}^1, ..., \mathcal{D}^N\}$ bezeichnet werden. Dann verteilen wir diese Batches an N Worker, jeder mit einem entsprechenden Batch. Sobald die Daten verteilt sind, können diese Worker gleichzeitig arbeiten. Der Gradient des gesamten Minibatchs wird durch Aggregieren der von den Workern berechneten Gradienten erhalten, wie folgt

$$\frac{\partial L_{\theta_t}(\mathcal{D}_{\text{mini}})}{\partial \theta_t} \quad = \quad \underbrace{\frac{\partial L_{\theta_t}(\mathcal{D}^1)}{\partial \theta_t}}_{\text{Worker 1}} + \underbrace{\frac{\partial L_{\theta_t}(\mathcal{D}^2)}{\partial \theta_t}}_{\text{Worker 2}} + \cdots + \underbrace{\frac{\partial L_{\theta_t}(\mathcal{D}^N)}{\partial \theta_t}}_{\text{Worker } N} \tag{2.33}$$

In idealen Fällen, in denen die Worker gut koordiniert sind und der Kommunikationsaufwand gering ist, kann der Datenparallelismus eine nahezu N-fache Beschleunigung des Trainings erreichen.

- Modellparallelismus. Obwohl der Datenparallelismus einfach und effektiv ist, erfordert er, dass jeder Worker das gesamte LLM ausführt und den vollständigen Vorwärts- und Rückwärtsdurchlauf durchführt. Da LLMs immer größer werden, ist es manchmal nicht mehr möglich, ein LLM auf einem einzigen Gerät zu laden und auszuführen. In diesem Fall können wir das LLM in kleinere Komponenten zerlegen und diese Komponenten auf verschiedenen Geräten ausführen. Eine einfache Möglichkeit hierfür ist, aufeinanderfolgende Schichten im Schichtenstapel zu gruppieren und jede Gruppe einem Worker zuzuweisen. Die Worker arbeiten in der Reihenfolge der Schichten im Stapel, d.h. im Vorwärtsdurchlauf verarbeiten wir die Eingabe von den unteren zu den oberen Schichten, und im Rückwärtsdurchlauf propagieren wir die Fehlergradienten von den oberen zu den unteren Schichten. Betrachten wir zum Beispiel einen Transformer -Decoder mit L gestapelten Blöcken. Um die Rechenlast zu verteilen, wird jeder Block einem Worker zugewiesen. Siehe die folgende Abbildung für einen einzelnen Durchlauf der Vorwärts- und Rückwärtsdurchläufe dieses Modells.

Worker L	B_L (↑)	B_L (↓)
...	...	...
Worker 2	B_2 (↑)	B_2 (↓)
Worker 1	B_1 (↑)	B_1 (↓)

Hier bezeichnet B_l die Berechnung von Block l, und die Symbole ↑ und ↓ bezeichnen

jeweils den Vorwärts- und Rückwärtsdurchlauf. Beachten Sie, dass diese Parallelisierungsmethode die Worker zwingt, nacheinander zu laufen, sodass ein Worker warten muss, bis der vorherige Worker seine Arbeit beendet hat. Dies führt dazu, dass die Geräte die meiste Zeit im Leerlauf sind. In praktischen Systemen wird der Modellparallelismus im Allgemeinen zusammen mit anderen Parallelisierungsmechanismen verwendet, um die Auslastung der Geräte zu maximieren.

- Tensorparallelismus. Parallelismus kann auch in einem einzelnen Berechnungsschritt durchgeführt werden. Ein gängiges Beispiel ist das Aufteilen einer großen Parametermatrix in Teile (Chunks), die separate Multiplikation eines Eingabe-Tensors mit jedem dieser Teile und das anschließende Verketten der Ergebnisse dieser Multiplikationen, um die Ausgabe zu bilden. Betrachten wir zum Beispiel die Multiplikation der Repräsentation $\mathbf{h} \in \mathbb{R}^d$ mit der Parametermatrix $\mathbf{W}_h \in \mathbb{R}^{d \times d_h}$ in einer FFN-Subschicht (siehe Gl. (2.25)). Wir können die Matrix $\mathbf{W}_h \in \mathbb{R}^{d \times d_h}$ vertikal in eine Sequenz von M Submatrizen aufteilen

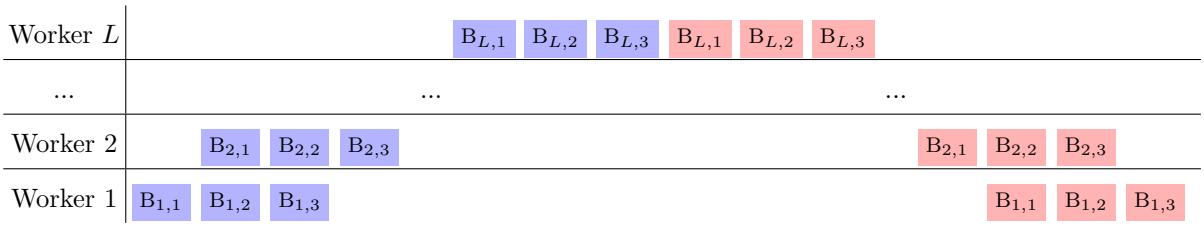
$$\mathbf{W}_h = \begin{bmatrix} \mathbf{W}_h^1 & \mathbf{W}_h^2 & ... & \mathbf{W}_h^M \end{bmatrix} \tag{2.34}$$

wobei jede Submatrix $\mathbf{W}_h^k$ die Form $d \times \frac{d_h}{M}$ hat. Die Multiplikation von $\mathbf{h}$ mit $\mathbf{W}_h$ kann ausgedrückt werden als

$$\begin{aligned} \mathbf{h}\mathbf{W}_h &= \mathbf{h}\begin{bmatrix} \mathbf{W}_h^1 & \mathbf{W}_h^2 & ... & \mathbf{W}_h^M \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{h}\mathbf{W}_h^1 & \mathbf{h}\mathbf{W}_h^2 & ... & \mathbf{h}\mathbf{W}_h^M \end{bmatrix} \end{aligned} \tag{2.35}$$

Wir können die Matrixmultiplikationen $\{\mathbf{h}\mathbf{W}_h^1, \mathbf{h}\mathbf{W}_h^2, ..., \mathbf{h}\mathbf{W}_h^M\}$ auf M Geräten separat durchführen. Dadurch verteilen wir eine große Matrixmultiplikation auf mehrere Geräte, von denen jedes einen relativ kleinen Speicher haben kann. Aus der Perspektive des Designs moderner GPUs bietet der Tensorparallelismus über GPUs einen zweistufigen, kachelbasierten Ansatz für paralleles Rechnen. Erstens zerlegen wir auf einer höheren Ebene eine Matrixmultiplikation in Submatrix-Multiplikationen, die direkt in den Speicher von GPUs passen. Dann führen wir auf einer niedrigeren Ebene diese Submatrix-Multiplikationen auf GPUs aus, unter Verwendung kachelbasierter paralleler Algorithmen, die speziell für GPUs optimiert sind.

- Pipeline-Parallelismus. Oben haben wir beim Modellparallelismus einen einfachen Ansatz beschrieben, um Gruppen von Modellkomponenten auf mehrere Geräte zu verteilen. Diese Methode ist jedoch ineffizient, da während der Verarbeitung jeweils nur ein Gerät aktiviert ist. Der Pipeline-Parallelismus löst dieses Problem, indem er Überlappungen zwischen den Berechnungen auf verschiedenen Geräten einführt [Harlap et al., 2018; Huang et al., 2019]. Dazu wird ein Batch von Beispielen in eine Anzahl von Mikro-Batches aufgeteilt, und diese Mikro-Batches werden dann wie gewohnt von jedem Worker verarbeitet. Sobald ein Mikro-Batch von einem Worker verarbeitet und an den nächsten weitergegeben wird, belegt der folgende Mikro-Batch sofort denselben Worker. Mit anderen Worten, wir erstellen eine Pipeline, in der verschiedene Berechnungsschritte überlappen können, wenn mehrere Aufträge an die Pipeline übergeben werden. Das Folgende zeigt eine Veranschaulichung des Pipeline-Parallelismus für die Verarbeitung von 3 Mikro-Batches.



Hier repräsentiert $B_{l,k}$ die Verarbeitung des k-ten Micro-Batches durch den l-ten Worker. Idealerweise möchten wir die Anzahl der Micro-Batches maximieren und somit die Leerlaufzeit der Worker minimieren. In der Praxis reduziert die Verwendung kleiner Micro-Batches jedoch oft die GPU-Auslastung und erhöht die Task-Switching-Kosten. Dies kann wiederum den Gesamtdurchsatz des Systems verringern.

Das ultimative Ziel der parallelen Verarbeitung ist es, ein lineares Wachstum der Effizienz zu erreichen, das heißt, die Anzahl der pro Zeiteinheit verarbeitbaren Samples steigt linear mit der Anzahl der Geräte. Verteiltes Training ist jedoch kompliziert und wird neben der gewählten Parallelitätsmethode von vielen Faktoren beeinflusst. Ein Problem, das oft mit verteilten Systemen in Verbindung gebracht wird, sind die Kommunikationskosten. Man kann sich ein verteiltes System als eine Gruppe von vernetzten Knoten vorstellen. Jeder dieser Knoten kann lokale Berechnungen durchführen oder Daten an andere Knoten weitergeben. Wenn es eine große Anzahl solcher Knoten gibt, wird es teuer, Daten über sie zu verteilen und zu sammeln. Manchmal werden die durch Parallelität erzielten Zeiteinsparungen durch den Kommunikations-Overhead eines großen Netzwerks aufgehoben. Ein weiteres Problem bei großen verteilten Systemen ist, dass die Synchronisation der Knoten zusätzliche Kosten verursacht. Wie es oft der Fall ist, benötigen einige Knoten möglicherweise länger für ihre Arbeit, was dazu führt, dass andere auf die langsamsten warten müssen. Obwohl wir asynchrones Training verwenden können, um die Heterogenität der Rechenressourcen zu bewältigen, kann dies zu veralteten Gradienten und einer nicht garantierten Konvergenz führen. Darüber hinaus steigt mit zunehmender Anzahl von Knoten im Netzwerk die Wahrscheinlichkeit, dass Knoten während des Trainings ausfallen. In diesem Fall müssen wir sicherstellen, dass das gesamte System fehlertolerant ist. In vielen praktischen Anwendungen müssen zur Erhöhung der Skalierbarkeit zusätzliche Aspekte berücksichtigt werden, darunter Architekturentwurf, Überlappung von Datenübertragung und Berechnung, Lastausgleich, Speicherbandbreite und so weiter.

Das Training von LLMs ist so rechenintensiv, dass Forscher und Ingenieure, obwohl verteiltes Training bereits eingesetzt wird, oft noch verschiedene Methoden zur Modellkomprimierung und Beschleunigung anwenden, um die Trainingseffizienz zu verbessern [Weng, 2021]. Ein Beispiel ist das Training mit gemischter Präzision, bei dem Daten mit niedriger Präzision (wie FP16- und FP8-Daten) für die Gradientenberechnung auf jedem einzelnen Knoten verwendet werden, und Daten mit einfacher oder doppelter Präzision (wie FP32/FP64-Daten) zur Aktualisierung des Modells [Micikevicius et al., 2018]. Eine Schlüsseloperation bei diesem Ansatz ist die Gradientenakkumulation, bei der Gradienten über Knoten hinweg akkumuliert und synchronisiert werden müssen. Aufgrund der Nicht-Assoziativität der Gleitkomma-Addition kann dies jedoch zu geringfügigen numerischen Unterschieden in den akkumulierten Gradienten auf verschiedenen Knoten führen, was die

Konvergenz des Modells und die endgültige Leistung beeinträchtigen kann. Dieses Problem ist umso deutlicher, je mehr Knoten am verteilten Training beteiligt sind, insbesondere da bei numerischen Berechnungen mit niedriger Präzision Überlauf- und Unterlaufprobleme sowie Inkonsistenzen zwischen verschiedenen Hardwaregeräten auftreten können. Daher muss das Design verteilter Systeme diese numerischen Berechnungsprobleme berücksichtigen, um zufriedenstellende Ergebnisse und Konvergenz zu gewährleisten.

2.2.4 Skalierungsgesetze

Der Erfolg von LLMs zeigt, dass das Training größerer Sprachmodelle mit mehr Ressourcen zu einer verbesserten Modellleistung führen kann. Forscher haben dies als Skalierungsgesetze von LLMs erklärt. Genauer gesagt beschreiben Skalierungsgesetze die Beziehungen zwischen der Leistung von LLMs und den Attributen des LLM-Trainings, wie der Modellgröße, dem für das Training verwendeten Rechenaufwand und der Menge der Trainingsdaten. Zum Beispiel zeigen Hestness et al. [2017], dass die Leistung von tiefen neuronalen Netzen eine potenzgesetzähnliche Funktion der Größe der Trainingsdaten ist. Am Anfang, wenn die Menge der Trainingsdaten nicht groß ist, verbessert sich die Leistung des Modells langsam. Danach, wenn mehr Trainingsdaten verwendet werden, tritt das Modell in eine Phase schneller Leistungsverbesserung ein, und die Leistungskurve ähnelt einer Potenzgesetz-Kurve. Schließlich wird die Leistungsverbesserung wieder langsam, und mehr Daten führen nicht zu signifikanten Gewinnen. Abbildung 2.3 zeigt ein Beispiel für solche Kurven.

Im Bereich NLP vertrat man traditionell die Ansicht, dass die Leistungssteigerungen ab einem bestimmten Punkt verschwinden, wenn das Training hochskaliert wird. Neuere Ergebnisse zeigen jedoch, dass, wenn wir das Problem in einem größeren Maßstab betrachten, die Skalierung des Trainings immer noch eine sehr effektive Methode ist, um stärkere LLMs zu erhalten. Zum Beispiel können sowohl Closed-Source- als auch Open-Source-LLMs von mehr Daten profitieren, obwohl bereits Billionen von Tokens für das Training verwendet wurden.

Mit der zunehmenden Skalierung des Modelltrainings zeigen LLMs neue Fähigkeiten, die als die emergente Fähigkeiten von LLMs bekannt sind. Zum Beispiel untersuchten Wei et al. [2022b] die Skalierungseigenschaften von LLMs über verschiedene Modellgrößen und Mengen an Rechenressourcen hinweg. Ihre Arbeit zeigt, dass einige Fähigkeiten entstehen, wenn wir die Modellgröße auf ein bestimmtes Niveau skalieren. Das Auftreten emergenter Fähigkeiten hat die Rolle des skalierten Trainings bei der Verbesserung der Leistung von LLMs demonstriert und Forscher in gewissem Maße motiviert, kontinuierlich zu versuchen, größere Modelle zu trainieren. Während immer größere und stärkere LMs erscheinen, reift unser Verständnis der Skalierungsgesetze weiter. Dies hilft Forschern, die Leistung von LLMs während des Trainings vorherzusagen und die minimalen Rechenressourcen abzuschätzen, die erforderlich sind, um ein bestimmtes Leistungsniveau zu erreichen.

Um zu verstehen, wie die Modellleistung mit verschiedenen Faktoren skaliert, die während des Trainings berücksichtigt werden, ist es üblich, die Modellleistung als eine Funktion dieser Faktoren auszudrücken. Zum Beispiel können wir im einfachsten Fall den Verlust oder Fehler eines LLM als Funktion einer einzelnen interessierenden Variablen ausdrücken.

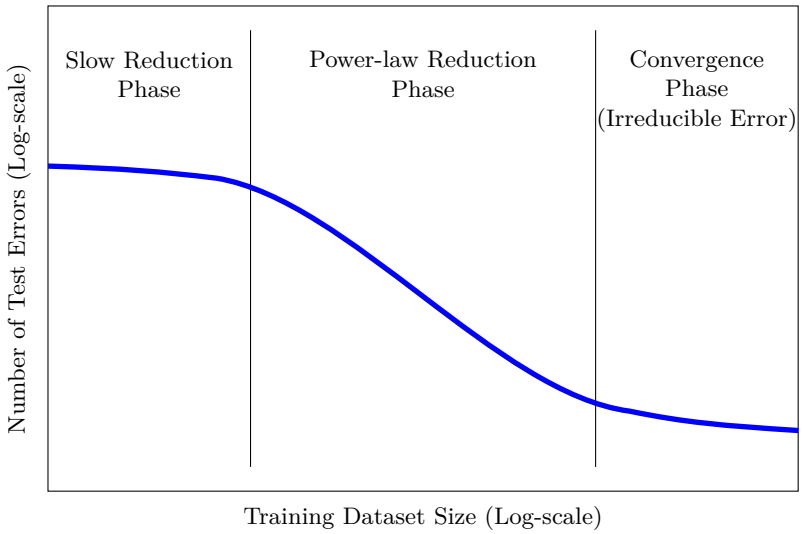


Abbildung 2.3: Ein Skalierungsgesetz des Testfehlers in Abhängigkeit von einer interessierenden Variablen (z. B. der Größe des Trainingsdatensatzes) [Hestness et al., 2017]. Die Kurve des Skalierungsgesetzes kann in drei Phasen unterteilt werden. Anfangs nimmt die Anzahl der Testfehler bei Verwendung von mehr Trainingsdaten langsam ab, jedoch nur für einen kurzen Zeitraum. In der zweiten Phase nimmt die Anzahl der Testfehler drastisch ab und die Kurve wird zu einer Potenzgesetz-Kurve. Danach verlangsamt sich die Fehlerreduktion in der dritten Phase wieder. Beachten Sie, dass es irreduzible Fehler gibt, die unabhängig von der Menge der Trainingsdaten nicht eliminiert werden können.

Es gibt jedoch keine universellen Skalierungsgesetze, die diese Beziehung beschreiben können. Stattdessen werden verschiedene Funktionen vorgeschlagen, um die Lernkurven von LLMs anzupassen.

Sei x die interessierende Variable (wie die Anzahl der Modellparameter) und $\mathcal{L}(x)$ der Verlust des Modells gegeben x (wie der Kreuzentropie-Verlust auf Testdaten). Die einfachste Form von $\mathcal{L}(x)$ ist ein Potenzgesetz

$$\mathcal{L}(x) \quad = \quad ax^b \tag{2.36}$$

wobei a und b Parameter sind, die empirisch geschätzt werden. Trotz ihrer Einfachheit hat diese Funktion die Skalierungsfähigkeit von Sprachmodellen und maschinellen Übersetzungssystemen in Bezug auf die Modellgröße (bezeichnet mit N) und die Größe des Trainingsdatensatzes (bezeichnet mit D) erfolgreich interpretiert [Gordon et al., 2021; Hestness et al., 2017]. Zum Beispiel stellten Kaplan et al. [2020] fest, dass sich die Leistung ihres Sprachmodells nach einer anfänglichen Übergangsphase als Potenzgesetz von entweder N oder D verbessert, und drückten diese Beziehungen mit $\mathcal{L}(N) = \left(\frac{N}{8.8\times10^{13}}\right)^{-0.076}$ und $\mathcal{L}(D) = \left(\frac{D}{5.4\times10^{13}}\right)^{-0.095}$ aus (siehe Abbildung 2.4).

Eine Verbesserung dieses Skalierungsgesetzes besteht darin, dem Potenzgesetz einen Term für den nicht reduzierbaren Fehler hinzuzufügen. Die Form von $\mathcal{L}(x)$ ist dann gegeben durch

$$\mathcal{L}(x) \quad = \quad ax^b + \epsilon_\infty \tag{2.37}$$

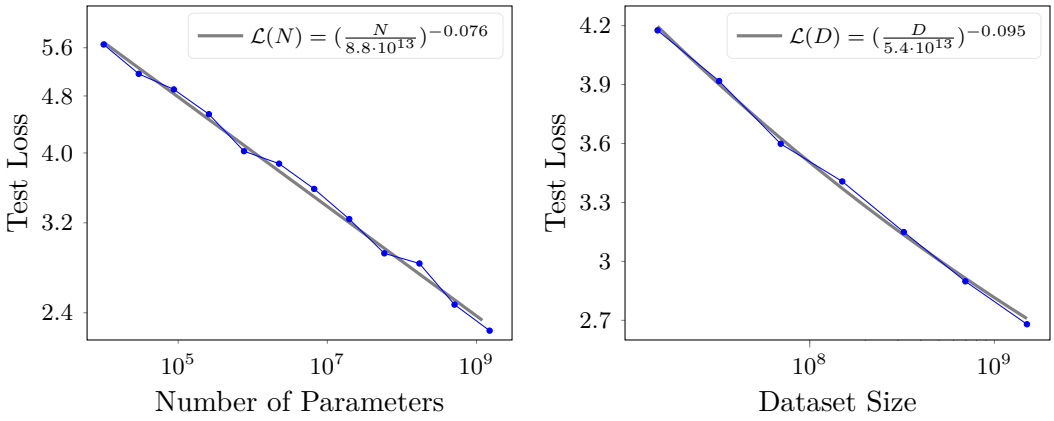


Abbildung 2.4: Test-Verlust in Abhängigkeit von der Modellgröße (N) und der Größe des Trainingsdatensatzes (D) (Datenpunkte sind zu Illustrationszwecken dargestellt). Wir stellen den Test-Verlust als Funktion von N dar, die als $\mathcal{L}(N) = \left(\frac{N}{8.8 \times 10^{13}}\right)^{-0.076}$ definiert ist, und als Funktion von D, die als $\mathcal{L}(D) = \left(\frac{D}{5.4 \times 10^{13}}\right)^{-0.095}$ definiert ist [Kaplan et al., 2020].

wobei ϵ_∞ der nicht reduzierbare Fehler ist, der den Fehler aufgrund unbekannter Variablen berücksichtigt, der auch bei $x \to \infty$ vorhanden ist. Gl. (2.37) ist eine der am weitesten verbreiteten Formen zur Gestaltung von Skalierungsgesetzen für LLMs. Zum Beispiel entwickelten Rosenfeld et al. [2020] ein Skalierungsgesetz, das sowohl die Modell- als auch die Datensatzskalierung umfasst, wie folgt

$$\mathcal{L}(N, D) \quad = \quad aN^b + cD^d + \epsilon_\infty \tag{2.38}$$

Ein Beispiel für eine solche Formulierung ist das Chinchilla-Skalierungsgesetz. Es besagt, dass der Testverlust pro Token die Summe der inversen Proportionsfunktionen von N und D ist, mit einem zusätzlichen nicht reduzierbaren Fehlerterm. Hoffmann et al. [2022] drücken dieses Skalierungsgesetz aus als

$$\mathcal{L}(N, D) \quad = \quad \underbrace{\frac{406.4}{N^{0.34}}}_{\text{Modellskalierung}} + \underbrace{\frac{410.7}{D^{0.28}}}_{\text{Datensatzskalierung}} + \underbrace{1.69}_{\text{Irreduzibler Fehler}} \tag{2.39}$$

Alle oben genannten Skalierungsgesetze basieren auf monotonen Funktionen. Daher können sie keine Funktionen mit Wendepunkten, wie z. B. Doppel-Abstiegs-Kurven, abdecken. Als Reaktion darauf haben Forscher anspruchsvollere Funktionen zur Anpassung der Lernkurven untersucht. Beispiele für solche Funktionen finden sich in den Arbeiten von Alabdulmohsin et al. [2022] und Caballero et al. [2023].

Die Bedeutung von Skalierungsgesetzen liegt darin, eine richtungsweisende Anleitung für die LLM-Forschung zu geben: Wenn wir uns noch im Bereich der Potenzgesetz-Kurve befinden, ist der Einsatz von mehr Ressourcen zum Trainieren größerer Modelle eine sehr vielversprechende Richtung. Während dieses Ergebnis große Forschungsgruppen und Unternehmen „zwingt“, mehr in Rechenressourcen für das Training größerer Modelle zu investieren, was sehr teuer ist, verschieben Skalierungsgesetze die Grenzen der KI kontinuierlich weiter. Andererseits hilft das Verständnis von Skalierungsgesetzen Forschern, Entscheidungen beim Training von LLMs zu treffen. Zum Beispiel kann bei gegebenen

Rechenressourcen die Leistung von LLMs vorhergesagt werden.

Eine letzte Anmerkung zu den Skalierungsgesetzen in diesem Abschnitt. Für LLMs bedeutet ein geringerer Testverlust nicht immer eine bessere Leistung bei allen nachgelagerten Aufgaben. Um LLMs anzupassen, gibt es mehrere Schritte wie Feinabstimmung und Prompting, die das Endergebnis beeinflussen können. Daher können die Skalierungsgesetze für verschiedene nachgelagerte Aufgaben in der Praxis unterschiedlich sein.

2.3 Modellierung langer Sequenzen

Wir haben bereits gesehen, dass bei groß angelegtem Training größere Sprachmodelle durch die Verwendung von mehr Daten und Rechenressourcen entwickelt werden können. Eine Skalierung kann jedoch auch in andere Richtungen erfolgen. Beispielsweise werden LLMs in vielen Anwendungen angepasst, um signifikant lange Sequenzen zu verarbeiten. Ein interessantes Beispiel ist, dass wir ein LLM auf umfangreichen Texten normaler Länge vortrainieren und es dann anwenden, um sehr lange Token-Sequenzen zu verarbeiten, die weit über die im Vortraining angetroffene Länge hinausgehen. Hier verwenden wir $\Pr(\mathbf{y}|\mathbf{x})$, um die Wahrscheinlichkeit der Textgenerierung zu bezeichnen, wobei $\mathbf{x}$ der Kontext und $\mathbf{y}$ der generierte Text ist. Es gibt grob drei Arten von Problemen bei der Modellierung langer Sequenzen.

- Textgenerierung basierend auf langem Kontext (d. h., $\mathbf{x}$ ist eine lange Sequenz). Zum Beispiel generieren wir eine kurze Zusammenfassung für einen sehr langen Text.
- Lange Textgenerierung (d. h., $\mathbf{y}$ ist eine lange Sequenz). Zum Beispiel generieren wir eine lange Geschichte basierend auf einigen Schlüsselwörtern.
- Lange Textgenerierung basierend auf langem Kontext (d. h., sowohl $\mathbf{x}$ als auch $\mathbf{y}$ sind lange Sequenzen). Zum Beispiel übersetzen wir ein langes Dokument vom Chinesischen ins Englische.

In jüngster Zeit haben sich NLP-Forscher verstärkt für die Anwendung und Evaluierung von LLMs bei Aufgaben interessiert, bei denen extrem lange Eingabetexte eine Rolle spielen. Stellen Sie sich ein LLM vor, das eine C++-Quelldatei mit Zehntausenden von Zeilen liest und die Funktionalität des Programms, das der Quelldatei entspricht, skizziert. Solche Modelle, die in der Lage sind, umfangreiche textuelle Kontexte zu verarbeiten, werden manchmal als LLMs mit langem Kontext bezeichnet. In diesem Abschnitt werden wir uns auf LLMs mit langem Kontext beschränken, aber die hier besprochenen Methoden können auch auf andere Probleme anwendbar sein.

Für Transformer ist die Verarbeitung langer Sequenzen rechenintensiv, da die Berechnungskosten der Selbst-Aufmerksamkeit quadratisch mit der Sequenzlänge wachsen. Dies macht es undurchführbar, solche Modelle für sehr lange Eingaben zu trainieren und einzusetzen. Zwei Forschungsstränge haben versucht, Transformer an die Sprachmodellierung mit langem Kontext anzupassen.

- Der erste untersucht effiziente Trainingsmethoden und Modellarchitekturen, um Self-Attention-Modelle aus Daten mit langen Sequenzen zu lernen.

- Der andere passt vortrainierte LLMs an, um lange Sequenzen mit geringem oder keinem Fine-Tuning-Aufwand zu verarbeiten.

Hier werden wir Ersteres kurz besprechen, da es in allgemeinen Diskussionen über effiziente Transformer-Architekturen zu finden ist [Tay et al., 2020; Xiao and Zhu, 2023]. Wir werden uns auf Letzteres konzentrieren und dabei populäre Methoden in aktuellen LLMs hervorheben. Wir werden auch die Stärken und Schwächen dieser Langsequenz-Modelle diskutieren.

2.3.1 Optimierung aus der Perspektive des Hochleistungsrechnens

Wir beginnen unsere Diskussion mit der Betrachtung von Verbesserungen an Standard-Transformer-Modellen aus der Perspektive des Hochleistungsrechnens. Die meisten dieser Verbesserungen, obwohl nicht speziell für LLMs entwickelt, wurden weithin auf verschiedene tiefenlernen-Modelle angewendet [Kim et al., 2023]. Ein häufig verwendeter Ansatz ist die Übernahme einer Implementierung von Transformern mit geringer Präzision. Zum Beispiel können wir 8-Bit- oder 16-Bit-Festkomma-Datentypen für arithmetische Operationen verwenden, anstelle von 32-Bit- oder 64-Bit-Gleitkomma-Datentypen. Die Verwendung dieser Datentypen mit geringer Präzision kann die Effizienz und den Speicherdurchsatz erhöhen, sodass längere Sequenzen einfacher verarbeitet werden können. Ein alternativer Ansatz besteht darin, Transformer mithilfe hardwarebewusster Techniken zu verbessern. Zum Beispiel kann auf modernen GPUs die Effizienz von Transformern durch die Verwendung von IO-bewussten Implementierungen der Selbst-Aufmerksamkeitsfunktion verbessert werden [Dao et al., 2022; Kwon et al., 2023].

Eine weitere Möglichkeit, lange Sequenzen zu verarbeiten, ist die Sequenzparallelität [Li et al., 2023b; Korthikanti et al., 2023]. Betrachten wir insbesondere das allgemeine Problem, die Aufmerksamkeit von der Abfrage $\mathbf{q}_i$ an Position i auf die Schlüssel $\mathbf{K}$ und die Werte $\mathbf{V}$ zu richten. Wir können $\mathbf{K}$ zeilenweise teilen und einen Satz von Teilmatrizen $\{\mathbf{K}^{[1]}, ..., \mathbf{K}^{[n_u]}\}$ erhalten, von denen jede einem Segment der Sequenz entspricht. Ähnlich können wir die Teilmatrizen von $\mathbf{V}$ erhalten, die mit $\{\mathbf{V}^{[1]}, ..., \mathbf{V}^{[n_u]}\}$ bezeichnet werden. Anschließend weisen wir jedes Paar von $\mathbf{K}^{[u]}$ und $\mathbf{V}^{[u]}$ einem Rechenknoten (z. B. einer GPU eines GPU-Clusters) zu. Die zugewiesenen Knoten können parallel ausgeführt werden, wodurch die Aufmerksamkeitsoperation parallelisiert wird.

Erinnern wir uns daran, dass die Ausgabe des Selbst-Aufmerksamkeitsmodells wie folgt geschrieben werden kann

$$\text{Att}_{\text{qkv}}(\mathbf{q}_i, \mathbf{K}, \mathbf{V}) = \sum_{j=0}^{m-1} \alpha_{i,j} \mathbf{v}_j \tag{2.40}$$

wobei $\alpha_{i,j}$ das Aufmerksamkeitsgewicht zwischen den Positionen i und j ist. In Transformern wird $\alpha_{i,j}$ durch die Normalisierung der neu skalierten Version des Skalarprodukts zwischen $\mathbf{q}_i$ und $\mathbf{k}_j$ erhalten. Sei $\beta_{i,j}$ der Aufmerksamkeits-Score zwischen $\mathbf{q}_i$ und $\mathbf{k}_j$. Wir

haben

$$\beta_{i,j} = \frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d}} + \text{Mask}(i,j) \tag{2.41}$$

wobei $\text{Mask}(i,j)$ die Maskierungsvariable für (i,j) ist. Dann definieren wir das Aufmerksamkeitsgewicht $\alpha_{i,j}$ als

$$\begin{aligned} \alpha_{i,j} &= \text{Softmax}(\beta_{i,j}) \\ &= \frac{\exp(\beta_{i,j})}{\sum_{j'} \exp(\beta_{i,j'})} \end{aligned} \tag{2.42}$$

Auf jedem Rechenknoten müssen wir diese Gleichungen implementieren. Gegeben die diesem Knoten zugewiesenen Schlüssel und Werte, ist die Berechnung des Zählers der rechten Seite von Gl. (2.42) (d. h. $\exp(\beta_{i,j})$) einfach, da alle erforderlichen Informationen auf dem Knoten gespeichert sind. Die Berechnung des Nenners der rechten Seite von Gl. (2.42) beinhaltet jedoch eine Summe von $\exp(\beta_{i,j'})$ über alle j', was die Übertragung von Daten zu und von anderen Knoten erfordert. Zur Veranschaulichung nehmen wir an, dass $\mathbf{v}_j$ und $\mathbf{k}_j$ auf Knoten u platziert sind. Wir können Gl. (2.42) umschreiben als

$$\alpha_{i,j} = \frac{\overbrace{\exp(\beta_{i,j})}^{\text{Knoten } u}}{\underbrace{\sum_{\mathbf{k}_{j'} \in \mathbf{K}^{[1]}} \exp(\beta_{i,j'})}_{\text{Knoten 1}} + \cdots + \underbrace{\sum_{\mathbf{k}_{j'} \in \mathbf{K}^{[u]}} \exp(\beta_{i,j'})}_{\text{Knoten } u} + \cdots + \underbrace{\sum_{\mathbf{k}_{j'} \in \mathbf{K}^{[n_u]}} \exp(\beta_{i,j'})}_{\text{Knoten } n_u}} \tag{2.43}$$

wobei die Notation $\mathbf{k}_{j'} \in \mathbf{K}^{[u]}$ darstellt, dass $\mathbf{k}_{j'}$ ein Zeilenvektor von $\mathbf{K}^{[u]}$ ist. In einer einfachen Implementierung führen wir zuerst die Summierungen $\{\sum_{\mathbf{k}_{j'} \in \mathbf{K}^{[u]}} \exp(\beta_{i,j'})\}$ separat auf den entsprechenden Knoten durch. Dann sammeln wir diese Summationsergebnisse von verschiedenen Knoten, um sie zu einem Endergebnis zu kombinieren. Dies entspricht einer kollektiven Operation im Kontext der parallelen Verarbeitung. Es gibt viele effiziente Implementierungen solcher Operationen, wie zum Beispiel die All-Reduce-Algorithmen. Daher kann die Summe aller $\exp(\beta_{i,j})$-Werte mithilfe optimierter Routinen in Toolkits für kollektive Kommunikation berechnet werden.

Gegeben die Aufmerksamkeitsgewichte $\{\alpha_{i,j}\}$, berechnen wir dann die Aufmerksamkeitsergebnisse unter Verwendung von Gl. (2.40). Das Problem kann umformuliert werden als

$$\text{Att}_{\text{qkv}}(\mathbf{q}_i, \mathbf{K}, \mathbf{V}) = \underbrace{\sum_{\mathbf{v}_{j'} \in \mathbf{V}^{[1]}} \alpha_{i,j'} \mathbf{v}_{j'}}_{\text{Knoten 1}} + \cdots + \underbrace{\sum_{\mathbf{v}_{j'} \in \mathbf{V}^{[u]}} \alpha_{i,j'} \mathbf{v}_{j'}}_{\text{Knoten } u} + \cdots + \underbrace{\sum_{\mathbf{v}_{j'} \in \mathbf{V}^{[n_u]}} \alpha_{i,j'} \mathbf{v}_{j'}}_{\text{Knoten } n_u} \tag{2.44}$$

Wie Gl. (2.43) kann auch Gl. (2.44) als Summationsprogramm in der parallelen Verarbeitung implementiert werden. Zuerst werden die gewichteten Summierungen der Werte auf verschiedenen Knoten gleichzeitig durchgeführt. Dann sammeln wir die Ergebnisse von diesen Knoten mittels kollektiver Operationen.

Beachten Sie, dass, obwohl sich dieser Abschnitt hauptsächlich auf die Modellierung langer Sequenzen konzentriert, ein Großteil der Motivation für die Sequenzparallelität aus den verteilten Trainingsmethoden tiefer Netzwerke stammt, wie in Abschnitt 2.2.3 diskutiert. Folglich kann die Implementierung dieser Methoden auf derselben Bibliothek für parallele Verarbeitung basieren.

2.3.2 Effiziente Architekturen

Eine Schwierigkeit bei der Anwendung von Transformern auf lange Sequenzen besteht darin, dass die Selbst-Aufmerksamkeit eine quadratische Zeitkomplexität in Bezug auf die Sequenzlänge aufweist. Darüber hinaus wird während der Inferenz ein Key-Value-Cache (kurz KV-Cache) verwaltet, dessen Größe mit der Verarbeitung weiterer Tokens zunimmt. Obwohl der KV-Cache linear mit der Sequenzlänge wächst, wird bei extrem langen Eingabesequenzen der Speicherbedarf erheblich, und es ist sogar undurchführbar, LLMs für solche Aufgaben einzusetzen. Infolgedessen weicht die Modellarchitektur von LLMs für lange Kontexte im Allgemeinen vom Standard-Transformer ab und wendet sich stattdessen der Entwicklung effizienterer Varianten und Alternativen zu.

Ein Ansatz besteht darin, anstelle der standardmäßigen Selbst-Aufmerksamkeit eine dünnbesetzte Aufmerksamkeit (sparse attention) zu verwenden. Diese Modellfamilie basiert auf der Idee, dass bei der Beachtung eines gegebenen Tokens nur eine kleine Anzahl von Tokens als wichtig erachtet wird und somit die meisten Aufmerksamkeitsgewichte zwischen den Tokens nahe Null sind. Folglich können wir die meisten Aufmerksamkeitsgewichte beschneiden und das Aufmerksamkeitsmodell in komprimierter Form darstellen. Zur Veranschaulichung betrachten wir das Selbst-Aufmerksamkeitsmodell

$$\mathrm{Att}_{\mathrm{qkv}}(\mathbf{Q},\mathbf{K},\mathbf{V}) \;=\; \alpha(\mathbf{Q},\mathbf{K})\mathbf{V} \tag{2.45}$$

wobei die Aufmerksamkeitsgewichtsmatrix $\alpha(\mathbf{Q},\mathbf{K}) \in \mathbb{R}^{m\times m}$ durch

$$\begin{aligned}\alpha(\mathbf{Q},\mathbf{K}) &= \mathrm{Softmax}\left(\frac{\mathbf{Q}\mathbf{K}^{\mathrm{T}}}{\sqrt{d}} + \mathbf{Mask}\right)\\ &= \begin{bmatrix} \alpha_{0,0} & 0 & 0 & \dots & 0\\ \alpha_{1,0} & \alpha_{1,1} & 0 & \dots & 0\\ \alpha_{2,0} & \alpha_{2,1} & \alpha_{2,2} & \dots & 0\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ \alpha_{m-1,0} & \alpha_{m-1,1} & \alpha_{m-1,2} & \dots & \alpha_{m-1,m-1}\end{bmatrix}\end{aligned} \tag{2.46}$$

erhalten wird. Jeder Zeilenvektor $\begin{bmatrix}\alpha_{i,0} & \dots & \alpha_{i,i} & 0 & \dots & 0\end{bmatrix}$ entspricht einer Verteilung, die die Aufmerksamkeit des i-ten Tokens auf jedes Token der Sequenz verteilt. Da Sprachmodelle die nächsten Tokens nur auf der Grundlage ihres linken Kontexts vorhersagen, schreiben wir die Ausgabe des Aufmerksamkeitsmodells an der Position i normalerweise

als

$$\begin{aligned}\text{Att}_{\text{qkv}}(\mathbf{q}_i, \mathbf{K}_{\leq i}, \mathbf{V}_{\leq i}) &= \begin{bmatrix}\alpha_{i,0} & ... & \alpha_{i,i}\end{bmatrix} \begin{bmatrix}\mathbf{v}_0 \\ \vdots \\ \mathbf{v}_i\end{bmatrix} \\ &= \sum_{j=0}^{i} \alpha_{i,j}\mathbf{v}_j \end{aligned} \tag{2.47}$$

wobei $\mathbf{K}_{\leq i} = \begin{bmatrix}\mathbf{k}_0 \\ \vdots \\ \mathbf{k}_i\end{bmatrix}$ und $\mathbf{V}_{\leq i} = \begin{bmatrix}\mathbf{v}_0 \\ \vdots \\ \mathbf{v}_i\end{bmatrix}$ die Schlüssel und Werte bis zur Position i sind.

In der ursprünglichen Version der Selbst-Aufmerksamkeit wird angenommen, dass $\begin{bmatrix}\alpha_{i,0} & ... & \alpha_{i,i}\end{bmatrix}$ dicht besetzt ist, das heißt, die meisten Werte sind ungleich Null. Bei der dünnbesetzten Aufmerksamkeit werden einige der Einträge von $\begin{bmatrix}\alpha_{i,0} & ... & \alpha_{i,i}\end{bmatrix}$ als ungleich Null betrachtet, und die restlichen Einträge werden bei der Berechnung einfach ignoriert. Angenommen, $G \subseteq \{0, ..., i\}$ ist die Menge der Indizes der Nicht-Null-Einträge. Für Sprachmodelle ist die Ausgabe des Modells mit dünnbesetzter Aufmerksamkeit an der Position i gegeben durch

$$\text{Att}_{\text{sparse}}(\mathbf{q}_i, \mathbf{K}_{\leq i}, \mathbf{V}_{\leq i}) = \sum_{j \in G} \alpha'_{i,j}\mathbf{v}_j \tag{2.48}$$

Hier werden $\{\alpha'_{i,j}\}$ über G normalisiert. Daher unterscheiden sich ihre Werte von den ursprünglichen Aufmerksamkeitsgewichten (tatsächlich gilt $\alpha'_{i,j} > \alpha_{i,j}$). Die Dünnbesetztheit des Modells wird durch die Größe von G bestimmt. Modelle mit dünnbesetzter Aufmerksamkeit unterscheiden sich in der Art und Weise, wie wir G definieren. Ein einfacher Ansatz besteht darin, G auf der Grundlage heuristisch entworfener Muster zu definieren. Ein weit verbreitetes Muster besteht beispielsweise darin, dass G ein Fenster von Tokens abdeckt, die sich in der Nähe der Position i befinden [Parmar et al., 2018].

Obwohl die dünnbesetzte Aufmerksamkeit die Berechnung durch die Verwendung von dünnbesetzten Operationen reduziert, haben solche Modelle immer noch erhebliche Einschränkungen, da wir während der Inferenz den gesamten KV-Cache (d. h. $\mathbf{K}_{\leq i}$ und $\mathbf{V}_{\leq i}$) beibehalten müssen. Wenn die Sequenz sehr lang ist, wird die Speicherung dieses Caches sehr speicherintensiv. Um dies zu beheben, können wir eine andere Form von Aufmerksamkeitsmodellen in Betracht ziehen, bei denen der KV-Cache nicht explizit beibehalten wird. Lineare Aufmerksamkeit ist ein solcher Ansatz [Katharopoulos et al., 2020]. Sie verwendet eine Kernelfunktion $\phi(\cdot)$, um jede Anfrage und jeden Schlüssel auf die Punkte $\mathbf{q}'_i = \phi(\mathbf{q}_i)$ bzw. $\mathbf{k}'_i = \phi(\mathbf{k}_i)$ zu projizieren. Durch Entfernen der Softmax-Funktion unter solchen Transformationen[10], ist die Form des resultierenden Aufmerksamkeitsmodells gegeben durch

[10] Im neuen Raum nach dieser Transformation kann die Softmax-Normalisierung in die einfache Skalierungsnormalisierung umgewandelt werden.

$$\begin{aligned} \text{Att}_{\text{qkv}}(\mathbf{q}_i, \mathbf{K}_{\leq i}, \mathbf{V}_{\leq i}) &\approx \text{Att}_{\text{linear}}(\mathbf{q}'_i, \mathbf{K}'_{\leq i}, \mathbf{V}_{\leq i}) \\ &= \frac{\mathbf{q}'_i \mu_i}{\mathbf{q}'_i \nu_i} \end{aligned} \tag{2.49}$$

wobei μ_i und ν_i Variablen sind, die in den rekurrenten Formen berechnet werden

$$\mu_i = \mu_{i-1} + \mathbf{k}'^{\mathrm{T}}_i \mathbf{v}_i \tag{2.50}$$

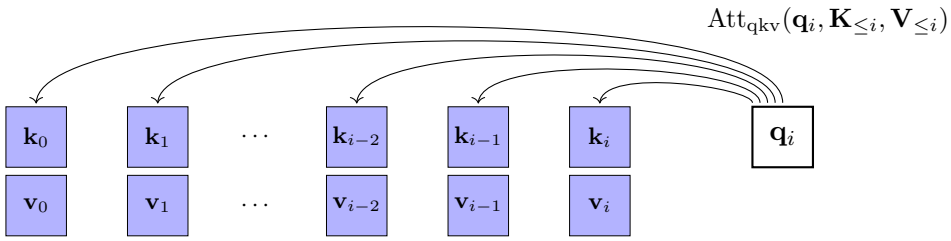
$$\nu_i = \nu_{i-1} + \mathbf{k}'^{\mathrm{T}}_i \tag{2.51}$$

μ_i und ν_i können als Repräsentationen der Historie bis zur Position i angesehen werden. Ein Vorteil dieses Modells ist, dass wir nicht alle vergangenen Anfragen und Werte behalten müssen. Stattdessen werden nur die neuesten Repräsentationen μ_i und ν_i verwendet. Daher sind die Berechnungskosten jedes Schritts konstant, und das Modell kann leicht erweitert werden, um lange Sequenzen zu verarbeiten.

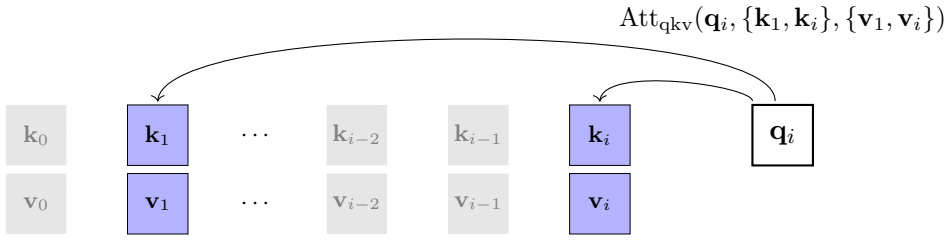
Tatsächlich ergibt sich dieser sequenzielle Ansatz zur Modellierung langer Sequenzen ganz natürlich, wenn man den Standpunkt rekurrenter Modelle einnimmt. Solche Modelle lesen jeweils ein Token (oder eine kleine Anzahl von Token), aktualisieren den rekurrenten Zustand mithilfe dieser Eingaben und verwerfen sie dann, bevor das nächste Token ankommt. Die Ausgabe bei jedem Schritt wird nur auf der Grundlage des rekurrenten Zustands erzeugt und nicht auf der Grundlage aller vorherigen Zustände. Der Speicherbedarf wird durch den rekurrenten Zustand bestimmt, der eine feste Größe hat. Rekurrente Modelle können in Echtzeit-Lernszenarien eingesetzt werden, in denen Daten in einem Stream eintreffen und Vorhersagen zu jedem Zeitschritt gemacht werden können. In der NLP ist die Anwendung rekurrenter Modelle auf die Sprachmodellierung einer der frühesten erfolgreichen Versuche, Repräsentationen von Sequenzen zu lernen. Obwohl der Transformer als grundlegende Architektur in LLMs verwendet wurde, sind rekurrente Modelle immer noch leistungsstarke Modelle, insbesondere für die Entwicklung effizienter LLMs. In jüngerer Zeit haben rekurrente Modelle in der Sprachmodellierung eine Renaissance erfahren und werden als vielversprechende Alternative zu Transformern wieder in Betracht gezogen [Gu and Dao, 2023]. Abbildung 2.5 zeigt einen Vergleich der in diesem Unterabschnitt besprochenen Modelle.

2.3.3 Cache und Speicher

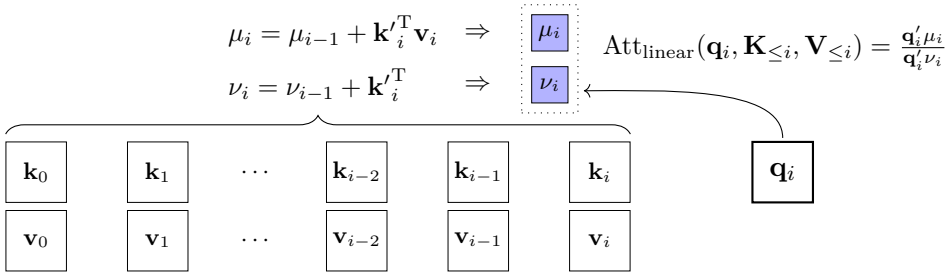
LLMs, die auf der Standard-Transformer -Architektur basieren, sind globale Modelle. Die Inferenz für diese Modelle beinhaltet das Speichern des gesamten linken Kontexts, um Vorhersagen für zukünftige Tokens zu treffen. Dies erfordert einen KV-Cache, in dem die Repräsentationen (d. h. Schlüssel und Werte) aller zuvor generierten Tokens gespeichert werden, und die Kosten für das Caching steigen mit fortschreitender Inferenz. Oben haben wir Methoden zur Optimierung dieses Caches durch effiziente Aufmerksamkeitsansätze wie Sparse Attention und lineare Attention besprochen. Eine weitere Idee, die sich möglicherweise mit der vorherigen Diskussion überschneidet, besteht darin, den Kontext explizit über ein zusätzliches Speichermodell zu kodieren.



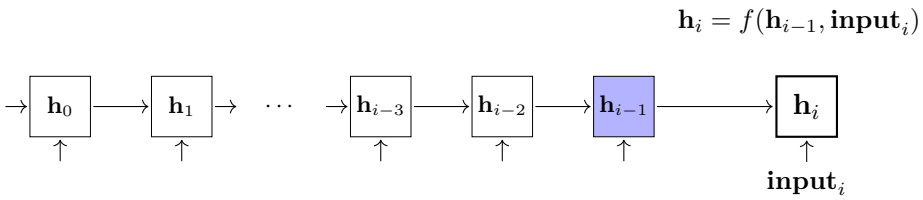
(a) Standard Self-attention



(b) Sparse Attention



(c) Linear Attention



(d) Recurrent Models

Abbildung 2.5: Illustrationen von Self-Attention, Sparse Attention, Linear Attention und rekurrenter Modelle. Blaue Kästchen = gecachte Zustände zur Erzeugung der Ausgabe an Position i. $f(\cdot)$ = eine rekurrente Zelle.

2.3.3.1 1. KV-Cache fester Größe

Ein einfacher Ansatz besteht darin, die Keys und Values unter Verwendung eines Speichermodells fester Größe darzustellen. Angenommen, wir haben einen Speicher Mem, der die kontextuellen Informationen speichert. Wir können die Aufmerksamkeitsoperation an Position i in einer allgemeinen Form schreiben

$$\text{Att}(\mathbf{q}_i, \text{Mem}) \quad = \quad \text{Att}_{\text{qkv}}(\mathbf{q}_i, \mathbf{K}_{\leq i}, \mathbf{V}_{\leq i}) \tag{2.52}$$

In diesem Modell ist Mem einfach der KV-Cache, d.h. Mem = $(\mathbf{K}_{\leq i}, \mathbf{V}_{\leq i})$. Daher wird die Größe von Mem durch i bestimmt. Wenn wir Mem als eine Variable mit fester Größe definieren, dann sind die Kosten für die Durchführung von $\text{Att}(\mathbf{q}_i, \text{Mem})$ fest. Es gibt mehrere alternative Wege, um Mem zu gestalten.

- Eine der einfachsten Methoden besteht darin, ein Fenster fester Größe für vorherige Schlüssel und Werte zu betrachten. Mem ist daher gegeben durch

$$\text{Mem} \quad = \quad (\mathbf{K}_{[i-n_c+1,i]}, \mathbf{V}_{[i-n_c+1,i]}) \tag{2.53}$$

  wobei n_c die Größe des Fensters bezeichnet. Die Notation $\mathbf{K}_{[i-n_c+1,i]}$ und $\mathbf{V}_{[i-n_c+1,i]}$ bezeichnen die Schlüssel und Werte über die Positionen von $i-n_c+1$ bis i.[11] Dieses Modell kann als eine Art lokales Aufmerksamkeitsmodell angesehen werden.

- Es ist auch möglich, Mem als ein Paar von zusammenfassenden Vektoren zu definieren, was zu einer komprimierteren Darstellung der Historie führt. Eine einfache Möglichkeit, die vorherigen Schlüssel und Werte zusammenzufassen, ist die Verwendung ihres gleitenden Durchschnitts. Zum Beispiel kann Mem als der ungewichtete gleitende Durchschnitt der vorherigen n_c Schlüssel und Werte definiert werden

$$\text{Mem} \quad = \quad \Big(\frac{\sum_{j=i-n_c+1}^{i} \mathbf{k}_j}{n_c}, \frac{\sum_{j=i-n_c+1}^{i} \mathbf{v}_j}{n_c}\Big) \tag{2.54}$$

  Alternativ können wir eine gewichtete Version des gleitenden Durchschnitts verwenden

$$\text{Mem} \quad = \quad \Big(\frac{\sum_{j=i-n_c+1}^{i} \beta_{j-i+n_c} \mathbf{k}_j}{\sum_{j=1}^{n_c} \beta_j}, \frac{\sum_{j=i-n_c+1}^{i} \beta_{j-i+n_c} \mathbf{v}_j}{\sum_{j=1}^{n_c} \beta_j}\Big) \tag{2.55}$$

  Hier sind $\{\beta_1, ..., \beta_{n_c}\}$ die Koeffizienten, die entweder als Modellparameter gelernt oder über Heuristiken bestimmt werden können. Zum Beispiel können sie auf ansteigende Koeffizienten gesetzt werden (d.h. $\beta_1 < \beta_2 < ... < \beta_{n_c-1} < \beta_{n_c}$), um Positionen, die näher an i liegen, ein größeres Gewicht zu geben. Wir können den gleitenden Durchschnitt erweitern, um alle Positionen bis zu i einzubeziehen. Dies führt zum kumulativen Durchschnitt der Schlüssel und Werte, gegeben in der Form

$$\text{Mem} \quad = \quad \Big(\frac{\sum_{j=0}^{i} \mathbf{k}_j}{i+1}, \frac{\sum_{j=0}^{i} \mathbf{v}_j}{i+1}\Big) \tag{2.56}$$

  Im Allgemeinen kann der kumulative Durchschnitt mit einer rekursiven Formel geschrieben werden

$$\text{Mem}_i \quad = \quad \frac{(\mathbf{k}_i, \mathbf{v}_i) + i \cdot \text{Mem}_{i-1}}{i+1} \tag{2.57}$$

[11] Formaler ausgedrückt schreiben wir $\mathbf{K}_{[i-n_c+1,i]} = \begin{bmatrix} \mathbf{k}_{i-n_c+1} \\ \vdots \\ \mathbf{k}_i \end{bmatrix}$ und $\mathbf{V}_{[i-n_c+1,i]} = \begin{bmatrix} \mathbf{v}_{i-n_c+1} \\ \vdots \\ \mathbf{v}_i \end{bmatrix}$. Manchmal bezeichnen wir $\mathbf{K}_{[i-n_c+1,i]}$ als $\{\mathbf{k}_{i-n_c+1}, ..., \mathbf{k}_i\}$ und $\mathbf{V}_{[i-n_c+1,i]}$ als $\{\mathbf{v}_{i-n_c+1}, ..., \mathbf{v}_i\}$ der Einfachheit halber.

wobei Mem_i und Mem_{i-1} die kumulativen Durchschnitte der aktuellen bzw. vorherigen Positionen bezeichnen. Ein Vorteil dieses Modells ist, dass wir während der Inferenz nur ein einziges Schlüssel-Wert-Paar speichern müssen, anstatt alle Schlüssel-Wert-Paare zu speichern. Beachten Sie, dass die oben genannten Speichermodelle mit rekurrenten Modellen verwandt sind und fortgeschrittenere Techniken verwendet wurden, um Alternativen zu Selbst-Aufmerksamkeits-Mechanismen in Transformern zu entwickeln [Ma et al., 2023].

- Der Speicher Mem kann auch ein neuronales Netzwerk sein. In jedem Schritt nimmt es sowohl die vorherige Ausgabe des Speichers als auch die aktuellen Zustände des Modells als Eingabe und erzeugt die neue Ausgabe des Speichers. Dieses neuronale Netzwerk kann als Funktion formuliert werden

$$\text{Mem} = \text{Update}(S_{\text{kv}}, \text{Mem}_{\text{pre}}) \tag{2.58}$$

Hier repräsentieren Mem und Mem_{pre} die Ausgaben des Speichers im aktuellen bzw. vorherigen Schritt. S_{kv} ist ein Satz von Schlüssel-Wert-Paaren, der die jüngsten Zustände des Modells darstellt. Diese Formulierung ist allgemein und ermöglicht es uns, verschiedene Speichermodelle zu entwickeln, indem wir unterschiedliche Update(·)- und S_{kv}-Konfigurationen auswählen. Wenn zum Beispiel S_{kv} nur das letzte Schlüssel-Wert-Paar $(\mathbf{k}_i, \mathbf{v}_i)$ enthält und Update(·) als eine rekurrente Zelle definiert ist, dann kann Gl. (2.58) als ein RNN-ähnliches Modell ausgedrückt werden

$$\text{Mem} = f((\mathbf{k}_i, \mathbf{v}_i), \text{Mem}_{\text{pre}}) \tag{2.59}$$

wobei $f(\cdot)$ eine rekurrente Zelle ist. Rekurrenz kann aus Effizienzgründen auch auf die Modellierung auf Segmentebene angewendet werden. Ein einfacher Ansatz besteht darin, die Sequenz in Segmente zu unterteilen und S_{kv} als ein Segment zu behandeln. Die Anwendung rekurrenter Modelle auf Update(·) führt zu Speichermodellen, die auf Segmenten operieren. Ein besonderes Beispiel ist, dass wir Update(·) als eine FIFO-Funktion definieren, die S_{kv} zum Speicher hinzufügt und das älteste Schlüssel-Wert-Segment aus dem Speicher entfernt, gegeben durch

$$\text{Mem} = \text{FIFO}(S_{\text{kv}}, \text{Mem}_{\text{pre}}) \tag{2.60}$$

Betrachten wir einen Speicher, der zwei Segmente umfasst, eines für das aktuelle Segment und eines für das vorherige Segment. Bei der Aufmerksamkeitsoperation kann jede Position auf die historischen Schlüssel-Wert-Paare in zwei nächstgelegenen aufeinanderfolgenden Segmenten zugreifen. Dies definiert im Wesentlichen einen lokalen Speicher, aber er und seine Varianten wurden in rekurrenten Modellen auf Segmentebene weit verbreitet eingesetzt [Dai et al., 2019; Hutchins et al., 2022; Bulatov et al., 2022].

- Die oben genannten Speichermodelle können erweitert werden, um mehrere Speicher zu umfassen. Ein Beispiel für diesen Ansatz ist der kompressive Transformer [Rae et al., 2019]. Er verwendet zwei verschiedene Speicher mit fester Größe: einen zur Modellierung des lokalen Kontexts (bezeichnet mit Mem) und den anderen zur Modellierung und Komprimierung der Langzeithistorie (bezeichnet mit CMem).

Der KV-Cache in diesem Modell ist die Kombination aus Mem und CMem. Die Aufmerksamkeitsfunktion kann geschrieben werden als

$$\text{Att}_{\text{com}}(\mathbf{q}_i, \text{Mem}, \text{CMem}) \quad = \quad \text{Att}_{\text{qkv}}(\mathbf{q}_i, [\text{Mem}, \text{CMem}]) \tag{2.61}$$

wobei [Mem, CMem] ein kombinierter Speicher aus Mem und CMem ist. Wie bei anderen Modellen auf Segmentebene arbeitet das kompressive Transformer-Modell auf Segmenten der Sequenz. Jedes Segment ist eine Sequenz von n_s aufeinanderfolgenden Tokens, und wir bezeichnen S^k_{kv} als die Schlüssel-Wert-Paare, die den Tokens des k-ten Segments entsprechen. Wenn ein neues Segment ankommt, wird Mem nach dem FIFO-Prinzip aktualisiert: Wir hängen die n_c Schlüssel-Wert-Paare in S^k_{kv} an Mem an und entfernen dann die n_s ältesten Schlüssel-Wert-Paare aus Mem, was gegeben ist durch

$$\text{Mem} \quad = \quad \text{FIFO}(S^k_{\text{kv}}, \text{Mem}_{\text{pre}}) \tag{2.62}$$

Die entfernten Schlüssel-Wert-Paare werden dann verwendet, um den kompressiven Speicher CMem zu aktualisieren. Diese n_s Schlüssel-Wert-Paare werden über ein Kompressionsnetzwerk in $\frac{n_s}{c}$ Schlüssel-Wert-Paare komprimiert. CMem ist eine FIFO-Warteschlange, die die komprimierten $\frac{n_s}{c}$ Schlüssel-Wert-Paare an das Ende der Warteschlange anhängt und die ersten $\frac{n_s}{c}$ Schlüssel-Wert-Paare der Warteschlange verwirft. Sie ist gegeben durch

$$\text{CMem} \quad = \quad \text{FIFO}(C^k_{\text{kv}}, \text{CMem}_{\text{pre}}) \tag{2.63}$$

wobei C^k_{kv} den Satz der komprimierten Schlüssel-Wert-Paare darstellt. Implizit im kompressiven Transformer-Modell ist, dass der lokale Kontext explizit mit minimalem Informationsverlust dargestellt werden sollte, während der Langzeitkontext stärker komprimiert werden kann.

- Wir haben bereits gesehen, dass sowohl globale als auch lokale Kontexte nützlich sind und mit Aufmerksamkeitsmodellen modelliert werden können. Diese Sichtweise motiviert die Erweiterung auf Aufmerksamkeitsmodelle zur Kombination von lokalen und Langzeitspeichern [Ainslie et al., 2020; Zaheer et al., 2020; Gupta and Berant, 2020]. Ein einfacher, aber weit verbreiteter Ansatz besteht darin, die ersten paar Tokens der Sequenz in die Aufmerksamkeit einzubeziehen, die als globale Tokens dienen. Dieser Ansatz wird normalerweise zusammen mit anderen dünnbesetzten Aufmerksamkeitsmodellen angewendet. Ein Vorteil der Einbeziehung globaler Tokens der Sequenz ist, dass es hilft, die Ausgabeverteilung der Softmax-Funktion, die bei der Berechnung der Aufmerksamkeitsgewichte verwendet wird, zu glätten und somit die Modellleistung bei sehr großer Kontextgröße stabilisiert [Xiao et al., 2024]. Ein Nachteil ist jedoch, dass die Verwendung eines globalen Speichers mit fester Größe zu Informationsverlust führen kann. Bei der Verarbeitung langer Sequenzen müssen wir den KV-Cache für ausreichende Repräsentationen des Kontexts vergrößern, was jedoch wiederum die Berechnungskosten erhöht.

Abbildung 2.6 zeigt Abbildungen der oben genannten Ansätze. Beachten Sie, dass,

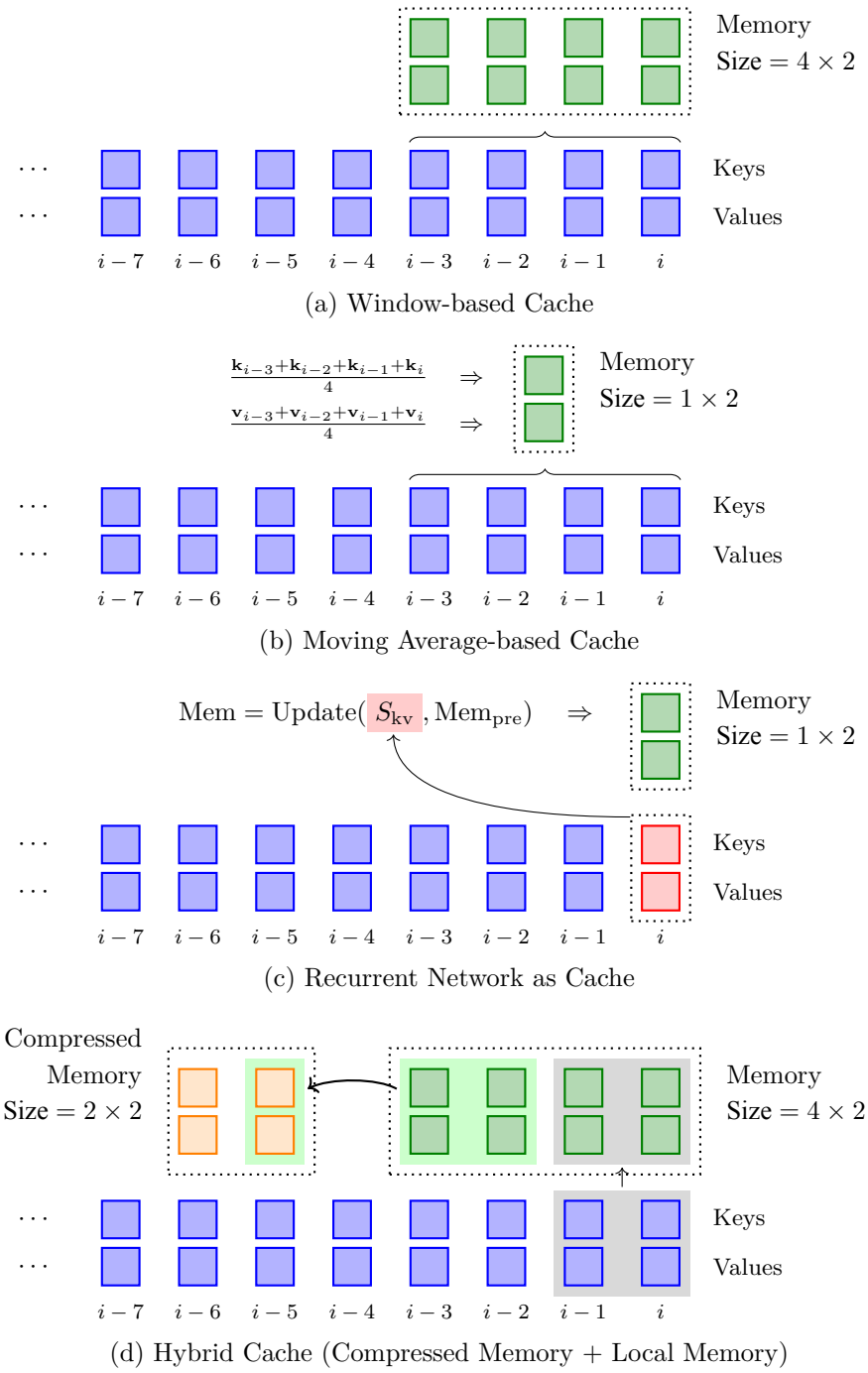


Abbildung 2.6: Darstellungen von KV-Caches fester Größe in LLMs. Blaue Kästchen stellen die während der LLM-Inferenz erzeugten Schlüssel und Werte dar, grüne Kästchen die im Primärspeicher gespeicherten oder kodierten Schlüssel und Werte und orangefarbene Kästchen die im komprimierten Speicher gespeicherten oder kodierten Schlüssel und Werte.

obwohl wir uns hier auf die Optimierung des KV-Caches konzentrieren, dieses Problem eng mit den im vorherigen Abschnitt besprochenen zusammenhängt. Alle bisher erwähnten Methoden können grob als effiziente Aufmerksamkeitsansätze kategorisiert werden, die in verschiedenen Transformer -Varianten weit verbreitet sind.

2.3.3.2 2. Speicherbasierte Modelle

Die oben diskutierte Modellierung von Speichern basierte auf Aktualisierungen des KV-Caches, und die resultierenden Modelle werden typischerweise als interne Speicher bezeichnet. Wir betrachten nun eine andere Familie von Modellen, die als externe Speicher bezeichnet werden und als unabhängige Modelle fungieren, um auf umfangreiche Kontexte für LLMs zuzugreifen. Viele solcher Modelle basieren auf speicherbasierten Methoden, die im maschinellen Lernen ausführlich diskutiert wurden [Bishop, 2006]. Ein gängiges Beispiel sind Nächste-Nachbarn-Algorithmen: Wir speichern Kontextrepräsentationen in einem Datenspeicher und versuchen, die ähnlichsten gespeicherten Repräsentationen zu finden, die zu einer gegebenen Abfrage passen. Die abgerufenen Kontextrepräsentationen werden dann verwendet, um die Aufmerksamkeit für diese Abfrage zu verbessern.

Hier betrachten wir die k-nächste Nachbarn (k-NN) Methode, die eine der populärsten speicherbasierten Methoden ist. Da unser Fokus in diesem Abschnitt auf der Sprachmodellierung liegt, definieren wir ein Sample im Datenspeicher als ein Schlüssel-Wert-Paar, das einem bestimmten Kontextzustand entspricht. Beachten Sie, dass „Kontext“hier ein weit gefasster Begriff ist und nicht nur ein Sequenzpräfix bei der Textgenerierung. Man könnte zum Beispiel den gesamten Datensatz als Kontext für die Vorhersage von Tokens betrachten. Dies ermöglicht es uns, die nächstgelegene Kontextsituation in einer Reihe von Sequenzen abzurufen, anstatt eines gegebenen Sequenzpräfixes. Obwohl wir uns in diesem Unterabschnitt auf die Kontextmodellierung für eine einzelne Sequenz beschränken werden, diskutieren wir einen relativ allgemeineren Fall.

Angenommen, wir haben einen Satz von Schlüsseln $\{\mathbf{k}_j\}$ mit entsprechenden Werten $\{\mathbf{v}_j\}$, und angenommen, wir speichern diese Schlüssel-Wert-Paare in einer Vektordatenbank[12]. Für jede Abfrage $\mathbf{q}_i$ finden wir ihre k nächsten Nachbarn, indem wir den Radius der Kugel, die bei $\mathbf{q}_i$ zentriert ist, vergrößern, bis sie k Datenpunkte in $\{\mathbf{k}_j\}$ enthält. Dies führt zu einem Satz von k Schlüsseln zusammen mit ihren entsprechenden Werten, bezeichnet mit $\text{Mem}_{k\text{nn}}$. Wie zuvor bezeichnen wir Mem als den lokalen Speicher für die Abfrage, wie z. B. den KV-Cache benachbarter Tokens. Unser Ziel ist es, die Abfrage $\mathbf{q}_i$ sowohl auf den lokalen Speicher Mem als auch auf den Langzeitspeicher $\text{Mem}_{k\text{nn}}$ zu richten. Es gibt natürlich mehrere Möglichkeiten, Mem und $\text{Mem}_{k\text{nn}}$ in das Aufmerksamkeitsmodell zu integrieren. Zum Beispiel könnten wir sie einfach zu einem einzigen KV-Cache $[\text{Mem}, \text{Mem}_{k\text{nn}}]$ kombinieren und $\mathbf{q}_i$ über die standardmäßige QKV-Aufmerksamkeit auf $[\text{Mem}, \text{Mem}_{k\text{nn}}]$ richten. Oder wir könnten Mem und $\text{Mem}_{k\text{nn}}$ in separaten Aufmerksamkeitsschritten verwenden. Ein Beispiel für solche Ansätze ist das von Wu et al. [2021] entwickelte Modell. Es kombiniert die beiden Arten von Aufmerksamkeit linear, gegeben durch

$$\begin{aligned} \text{Att}(\mathbf{q}_i, \text{Mem}, \text{Mem}_{k\text{nn}}) &= \mathbf{g} \odot \text{Att}_{\text{local}} + (1 - \mathbf{g}) \odot \text{Att}_{k\text{nn}} && (2.64) \\ \text{Att}_{\text{local}} &= \text{Att}(\mathbf{q}_i, \text{Mem}) && (2.65) \\ \text{Att}_{k\text{nn}} &= \text{Att}(\mathbf{q}_i, \text{Mem}_{k\text{nn}}) && (2.66) \end{aligned}$$

Hier ist $\mathbf{g} \in \mathbb{R}^d$ der Koeffizientenvektor, der die Ausgabe eines gelernten Gates sein kann.

[12] Eine Vektordatenbank oder ein Vektorspeicher ist eine Datenbank, die hochoptimierte Abrufschnittstellen zum Auffinden gespeicherter Vektoren bietet, die eng mit einem Abfragevektor übereinstimmen.

Angesichts des oben beschriebenen k-NN-basierten Speichermodells besteht die verbleibende Aufgabe darin, zu bestimmen, welche Schlüssel-Wert-Paare im Datenspeicher aufbewahrt werden. Bei Standardaufgaben der Sprachmodellierung betrachten wir die zuvor gesehenen Tokens in einer Sequenz als Kontext, sodass wir die Schlüssel und Werte all dieser Tokens in den Datenspeicher einfügen können. In diesem Fall ist das resultierende k-NN-basierte Aufmerksamkeitsmodell im Wesentlichen äquivalent zu einem spärlichen Aufmerksamkeitsmodell [Gupta et al., 2021].

Alternativ können wir den Kontext von einer Sequenz auf eine Sammlung von Sequenzen erweitern. Zum Beispiel könnten wir alle Schlüssel-Wert-Paare über die Sequenzen in einem Trainingsdatensatz sammeln und sie dem Datenspeicher hinzufügen, um einen größeren Kontext zu modellieren. So können LLMs Tokens basierend auf einem verallgemeinerten Kontext vorhersagen. Ein Problem bei diesem Ansatz ist, dass der Rechenaufwand hoch wäre, wenn viele Sequenzen beteiligt sind. Da diese Sequenzen Teil unserer Trainingsdaten sind, können wir vor dem Ausführen der LLMs einen Index für die Vektoren im Datenspeicher erstellen und optimieren. Dadurch kann der Abruf ähnlicher Vektoren sehr effizient sein, wie in den meisten Vektordatenbanken.

Tatsächlich können alle oben genannten Methoden als Instanzen eines abrufbasierten Ansatzes betrachtet werden. Anstatt Abrufergebnisse zur Verbesserung der Aufmerksamkeit zu verwenden, können wir diesen Ansatz auch auf andere Weisen anwenden. Eine Anwendung der k-NN-basierten Suche ist die k-NN-Sprachmodellierung (oder k-NN LM) [Khandelwal et al., 2020]. Die Idee ist, dass, obwohl versucht wird, den in der Selbst-Aufmerksamkeit verwendeten Kontext durch die Einbeziehung von nächsten Nachbarn beim Repräsentationslernen zu erweitern, in der Praxis ähnliche verborgene Zustände in Transformern oft sehr prädiktiv für ähnliche Tokens an nachfolgenden Positionen sind. Im k-NN-LM ist jeder Eintrag im Datenspeicher ein Schlüssel-Wert-Tupel $(\mathbf{z}, w)$, wobei $\mathbf{z}$ einen verborgenen Zustand des LLM an einer Position darstellt und w die entsprechende Vorhersage repräsentiert. Eine typische Methode zur Erstellung des Datenspeichers besteht darin, den Ausgabevektor des Transformer-Schichtstapels und den entsprechenden nächsten Token für jede Position jeder Sequenz in einem Trainingsdatensatz zu sammeln. Während der Inferenz haben wir eine Repräsentation $\mathbf{h}_i$ gegeben eines Präfixes. Gegeben diese Repräsentation, durchsuchen wir zuerst den Datenspeicher nach den k am besten passenden Datenelementen $\{(\mathbf{z}_1, w_1), ..., (\mathbf{z}_k, w_k)\}$. Hier werden $\{w_1, ..., w_k\}$ als Referenz-Tokens für die Vorhersage betrachtet und können somit zur Steuerung der Token-Vorhersage basierend auf $\mathbf{h}_i$ verwendet werden. Eine gängige Methode, Referenz-Tokens zu nutzen, besteht darin, eine Verteilung über das Vokabular V zu definieren,

$$\Pr_{knn}(\cdot|\mathbf{h}_i) \quad = \quad \text{Softmax}(\begin{bmatrix} -d_0 & \cdots & -d_{|V|} \end{bmatrix}) \tag{2.67}$$

wobei d_v dem Abstand zwischen $\mathbf{h}_i$ und $\mathbf{z}_j$ entspricht, wenn w_j dem v-ten Eintrag von V gleicht, und ansonsten 0 ist. Wir verwenden eine lineare Funktion mit einem Koeffizienten λ, die zwischen der abrufbasierten Verteilung $\Pr_{knn}(\cdot|\mathbf{h}_i)$ und der LLM-Ausgabeverteilung $\Pr_{\text{lm}}(\cdot|\mathbf{h}_i)$ interpoliert

$$\Pr(\cdot|\mathbf{h}_i) \quad = \quad \lambda \cdot \Pr_{knn}(\cdot|\mathbf{h}_i) + (1-\lambda) \cdot \Pr_{\text{lm}}(\cdot|\mathbf{h}_i) \tag{2.68}$$

Dann können wir wie üblich den nächsten Token y wählen, indem wir die Wahrscheinlichkeit $\Pr(y|\mathbf{h}_i)$ maximieren.

Wie bei Informationsabrufsystemen (IR) kann der Datenspeicher auch Texte verwalten und den Zugriff auf relevante Texte für eine Abfrage ermöglichen. Zum Beispiel können wir eine Sammlung von Textdokumenten in einer Suchmaschine mit Volltextindexierung speichern und diese dann nach Dokumenten durchsuchen, die zu einer gegebenen textbasierten Abfrage passen. Die Anwendung von IR-Techniken auf LLMs führt zu einem allgemeinen Rahmenwerk, das als abruf-erweiterte Generierung (RAG) bezeichnet wird. Das RAG-Framework funktioniert wie folgt. Wir verwenden den Kontext $\mathbf{x}$ als Abfrage und finden die k relevantesten Dokumententeile $\{\mathbf{c}_1, ..., \mathbf{c}_k\}$ aus dem Datenspeicher über effiziente IR-Techniken[13]. Diese Suchergebnisse werden mit dem ursprünglichen Kontext über eine Prompting-Vorlage $g(\cdot)$[14] kombiniert, was zu einer erweiterten Eingabe für das LLM führt

$$\mathbf{x}' = g(\mathbf{c}_1, ..., \mathbf{c}_k, \mathbf{x}) \tag{2.69}$$

Dann verwenden wir $\mathbf{x}'$ als Kontext und sagen den folgenden Text mit dem Modell $\Pr(\mathbf{y}|\mathbf{x}')$ voraus. Ein Vorteil von RAG ist, dass wir die Architektur von LLMs nicht ändern müssen, sondern stattdessen die Eingabe für LLMs über ein zusätzliches IR-System erweitern. Abbildung 2.7 zeigt einen Vergleich der Verwendung verschiedener externer Speicher in LLMs.

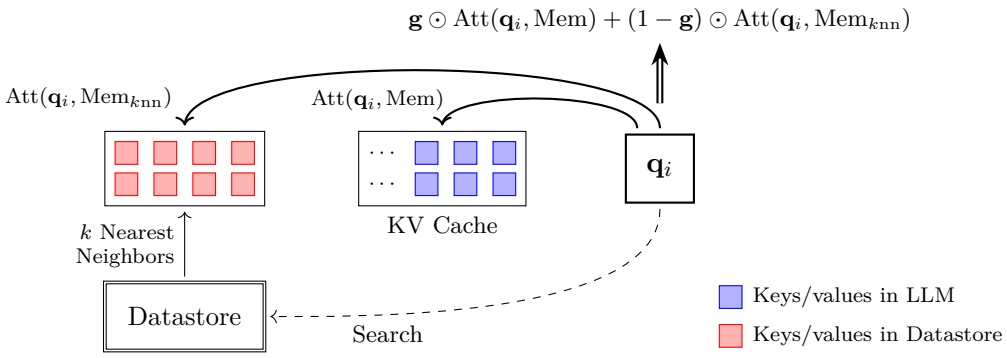
2.3.3.3 3. Speicherkapazität

Ein Speichermodell in LLMs, in Form eines einfachen Schlüssel-Wert-Caches oder eines Datenspeichers, kann im Großen und Ganzen als Encoder für kontextuelle Informationen angesehen werden. Idealerweise müssen wir, bevor wir sagen, dass ein Speichermodell für den gesamten Kontext bei der Token-Vorhersage repräsentativ ist, sicherstellen, dass das Modell jeden Teil des Kontexts genau darstellen kann. Der standardmäßige KV-Cache ist ein solches Modell, das die gesamte bisherige Historie vollständig speichert. In diesem Fall wird gesagt, dass das Modell eine ausreichende Kapazität zum Speichern des Kontexts besitzt. In vielen praktischen Anwendungen ist jedoch keine vollständige Speicherung erforderlich. Stattdessen besteht das Ziel darin, LLMs den Zugriff auf wichtige kontextuelle Informationen zu ermöglichen. Daher werden effiziente und komprimierte Speichermodelle entwickelt, wie in diesem Abschnitt beschrieben. Es ist zu beachten, dass es für ein Speichermodell mit geringer Kapazität umso schwieriger wird, wichtige kontextuelle Informationen zu erfassen, je länger die Sequenz ist. Daher ist es üblich, die Modellkapazität bei der Verarbeitung langer Kontexte einfach zu erhöhen.

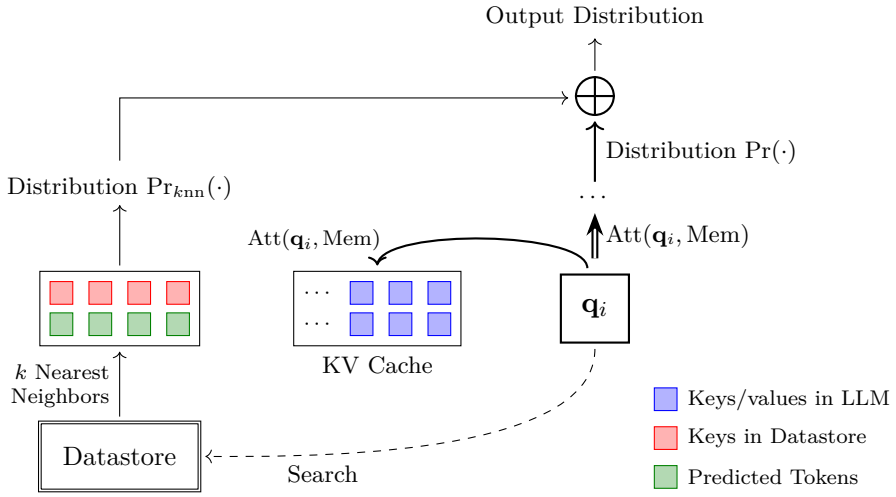
[13] In praktischen Anwendungen werden Abfragen typischerweise mit einem Abfragegenerierungssystem erstellt, das sie mit Variationen von Tokens und Abfrageabsichten erweitern kann.

[14] Zum Beispiel könnte die Vorlage lauten:

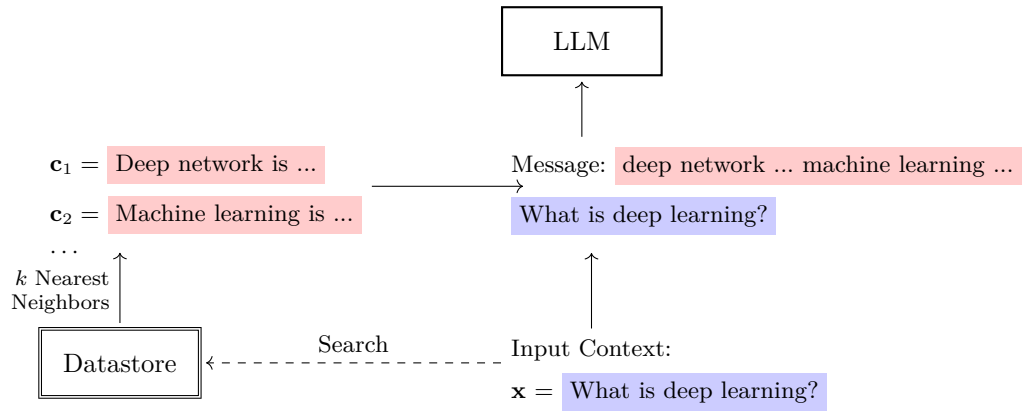
message = {*$\mathbf{c}_1$*} ... {*$\mathbf{c}_k$*}
input: {*$\mathbf{x}$*}
output: ____



(a) k-NN Search Augmented Attention



(b) k-NN Language Modeling



(c) Retrieval-augmented Generation

Abbildung 2.7: Illustrationen von externen Speichern (oder Datenspeichern) für die Sprachmodellierung.

Obwohl Modelle mit hoher Kapazität im Allgemeinen vorteilhaft sind, sind sie schwierig zu trainieren und bereitzustellen. Ein herausforderndes Szenario ist, dass die Tokens in einem Stream eintreffen und der Kontext kontinuierlich wächst. Die Entwicklung von

LLMs für solche Aufgaben ist schwierig, da wir Transformer auf extrem langen Sequenzen trainieren müssen. Eine mögliche Lösung für diese Schwierigkeit ist die Verwendung nichtparametrischer Methoden, wie zum Beispiel abrufbasierter Methoden. Zum Beispiel können wir, wie oben diskutiert, eine Vektordatenbank verwenden, um zuvor generierte Schlüssel-Wert-Paare zu speichern und so den Kontext durch dieses externe Speichermodell darzustellen. Obwohl dieser Ansatz die Herausforderung der Darstellung langer Kontexte in Transformern umgeht, sind der Aufbau und die Aktualisierung externer Speichermodelle rechenintensiv. Diese Modelle werden häufiger bei Problemen eingesetzt, bei denen der Kontext im Voraus gegeben und während der Inferenz festgelegt ist, und sind daher für die Modellierung von Streaming-Kontexten ungeeignet.

In Fällen, in denen die Größe des Kontexts kontinuierlich wächst, ist die Anwendung von Speichermodellen mit fester Größe ein häufig verwendeter Ansatz. Zum Beispiel kann in rekurrenten Modellen eine Sequenz beliebiger Länge in einem Satz von verborgenen Zuständen zusammengefasst werden, wodurch wir einen festen Rechenaufwand pro Schritt haben. Während rekurrente Modelle in frühen Anwendungen des Deep Learning auf NLP zunächst als nicht sehr gut im Umgang mit weitreichenden Abhängigkeiten bei der Sequenzmodellierung befunden wurden, haben jüngste Fortschritte gezeigt, dass ihre Varianten nun bei der Modellierung extrem langer Sequenzen wirksam sind. [Bulatov et al., 2022; Hutchins et al., 2022; Munkhdalai et al., 2024; Ma et al., 2024].

Es gibt keine allgemeingültige Definition von Speicherkapazität in LLMs. Ein einfacher Ansatz könnte berücksichtigen, wie viel Speicherplatz zur Aufbewahrung kontextueller Informationen verwendet wird. Zum Beispiel könnte die Speicherkapazität durch die Größe des KV-Caches in Transformern oder die in abrufbasierten Methoden verwendete Vektordatenbank definiert werden. Ein verwandtes Konzept ist die Modellkomplexität. Im maschinellen Lernen gibt es verschiedene Möglichkeiten, die Modellkomplexität eines Modells zu definieren. Eine der einfachsten Methoden ist das Zählen der Anzahl der Parameter. Es sollte jedoch betont werden, dass die hier diskutierten Speichermodelle hauptsächlich dazu dienen, Informationen zu speichern, anstatt trainierbare Parameter hinzuzufügen. Daher ist ein Modell mit großer Speicherkapazität nicht zwangsläufig komplexer. Dennoch ist die Bestimmung der Kapazität eines Speichermodells in der Praxis nicht einfach. Im Allgemeinen müssen wir den Kompromiss zwischen der Maximierung der Leistung und der Kontrolle des Speicherbedarfs steuern.

2.3.4 Teilung über Heads und Schichten hinweg

In Transformern ist der KV-Cache eine Datenstruktur, die dynamisch entlang mehrerer Dimensionen, wie Heads, Schichten und Sequenzlänge, angepasst werden kann. Betrachten wir zum Beispiel ein LLM mit L Schichten. Jede Schicht hat τ Attention-Heads, und jeder Head erzeugt eine d_h-dimensionale Ausgabe. Während der Inferenz speichern wir die Keys und Values für bis zu m Token. Die Raumkomplexität dieses Caching-Mechanismus beträgt $O(L \cdot \tau \cdot d_h \cdot m)$. Wie wir bereits gesehen haben, kann diese Komplexität reduziert werden, indem die Keys und Values für weniger Token zwischengespeichert werden. Bei der Gleitfenster-Attention zum Beispiel wird ein Fenster fester Größe verwendet, um die Keys und Values im lokalen Kontext zwischenzuspeichern. Und dieses Modell hat eine Raumkomplexität von $O(L \cdot \tau \cdot d_h \cdot m_w)$, wobei m_w die Größe des Fensters ist.

Zusätzlich zur Reduzierung von m können wir auch die Größe des KV-Cache entlang anderer Dimensionen verringern. Ein weit verbreiteter Ansatz ist, die gemeinsame Nutzung über Heads hinweg in der Multi-Head-Self-Attention zu ermöglichen. Erinnern wir uns aus Abschnitt 2.1.1, dass die Multi-Head-Self-Attention mehrere Sätze von Queries, Keys und Values verwendet (jeder Satz wird als Head bezeichnet), wobei jeder den QKV-Attention-Mechanismus wie gewohnt ausführt. Dies kann ausgedrückt werden als

$$\text{Ausgabe} \quad = \quad \text{Merge}(\text{head}_1, ..., \text{head}_\tau)\mathbf{W}^{\text{head}} \tag{2.70}$$

wobei $\text{head}_j \in \mathbb{R}^{d_h}$ unter Verwendung der Standard-QKV-Attention-Funktion berechnet wird

$$\text{head}_j = \text{Att}_{\text{qkv}}(\mathbf{q}_i^{[j]}, \mathbf{K}_{\leq i}^{[j]}, \mathbf{V}_{\leq i}^{[j]}) \tag{2.71}$$

Hier sind $\mathbf{q}_i^{[j]}$, $\mathbf{K}_{\leq i}^{[j]}$ und $\mathbf{V}_{\leq i}^{[j]}$ die Query, Keys und Values, die auf den j-ten Feature-Unterraum projiziert werden. Dieses Modell kann also so interpretiert werden, dass es Attention auf eine Gruppe von Feature-Unterräumen parallel durchführt (siehe Abbildung 2.8 (b)). Der KV-Cache muss die Keys und Values für all diese Heads beibehalten, das heißt, $\{(\mathbf{K}_{\leq i}^{[1]}, \mathbf{V}_{\leq i}^{[1]}), ..., (\mathbf{K}_{\leq i}^{[\tau]}, \mathbf{V}_{\leq i}^{[\tau]})\}$.

Eine Verfeinerung des Multi-Head-Attention-Modells, genannt multi-query attention (MQA), besteht darin, Keys und Values über die Heads hinweg gemeinsam zu nutzen, während die Queries für jeden Head einzigartig bleiben können [Shazeer, 2019]. Bei MQA gibt es einen einzigen Satz von Keys und Values $(\mathbf{K}_{\leq i}, \mathbf{V}_{\leq i})$. Zusätzlich gibt es τ Queries $\{\mathbf{q}_i^{[1]}, ..., \mathbf{q}_i^{[\tau]}\}$, die jeweils einem anderen Head entsprechen. Für jeden Head haben wir

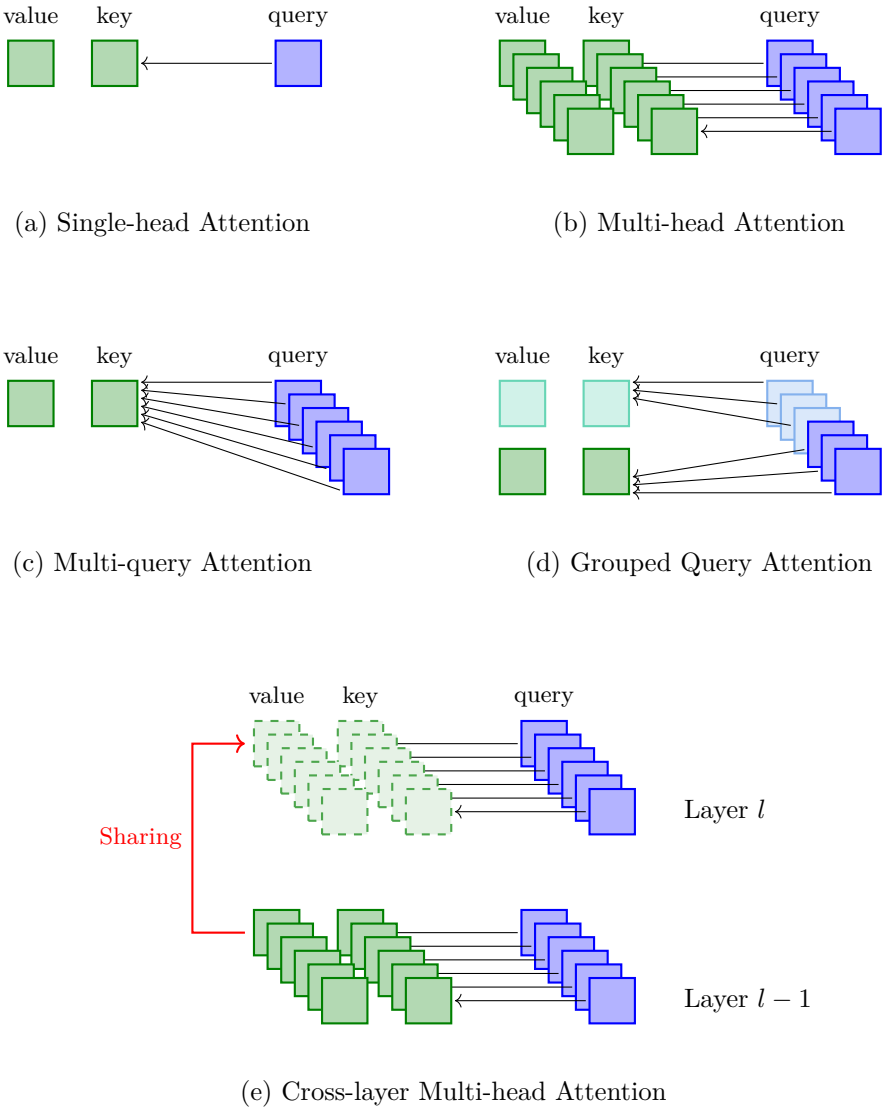
$$\text{head}_j = \text{Att}_{\text{qkv}}(\mathbf{q}_i^{[j]}, \mathbf{K}_{\leq i}, \mathbf{V}_{\leq i}) \tag{2.72}$$

Abbildung 2.8 (c) veranschaulicht dieses Modell. Durch die gemeinsame Nutzung von Keys und Values würde die Größe des KV-Cache $O(L \cdot d_h \cdot m)$ betragen.

Grouped query attention (GQA) ist eine natürliche Erweiterung der Multi-Head-Attention und MQA [Ainslie et al., 2023]. Bei GQA werden die Heads in n_g Gruppen unterteilt, wobei jede einem gemeinsam genutzten Satz von Keys und Values entspricht. Daher haben wir n_g Sätze von Keys und Values $\{(\mathbf{K}_{\leq i}^{[1]}, \mathbf{V}_{\leq i}^{[1]}), ..., (\mathbf{K}_{\leq i}^{[n_g]}, \mathbf{V}_{\leq i}^{[n_g]})\}$. Siehe Abbildung 2.8 (d) zur Veranschaulichung. Sei $g(j)$ die Gruppen-ID für den j-ten Head. Das GQA-Modell kann ausgedrückt werden als

$$\text{head}_j = \text{Att}_{\text{qkv}}(\mathbf{q}_i^{[j]}, \mathbf{K}_{\leq i}^{[g(j)]}, \mathbf{V}_{\leq i}^{[g(j)]}) \tag{2.73}$$

Die Größe des KV-Cache von GQA beträgt $O(L \cdot n_g \cdot d_h \cdot m)$. Ein Vorteil von GQA ist, dass wir durch Anpassen von n_g einen Kompromiss zwischen Recheneffizienz und Modellausdruckskraft eingehen können. Wenn $n_g = \tau$, wird das Modell zum Standard-Multi-Head-Attention-Modell. Im Gegensatz dazu wird es, wenn $n_g = 1$, zum MQA-Modell.



(a) Single-head Attention

(b) Multi-head Attention

(c) Multi-query Attention

(d) Grouped Query Attention

(e) Cross-layer Multi-head Attention

Abbildung 2.8: Veranschaulichung der QKV-Aufmerksamkeit basierend auf verschiedenen Multi-Head- und Sharing-Mechanismen. (a) = Single-Head-Aufmerksamkeit und (b-e) = Aufmerksamkeit mit mehreren Köpfen.

Die gemeinsame Nutzung kann auch über Schichten hinweg erfolgen. Eine solche Methode gehört zur Familie der Methoden mit geteilten Gewichten und geteilten Aktivierungen, die in Transformern ausgiebig verwendet wurden [Dehghani et al., 2018; Lan et al., 2020]. Zum Beispiel kann man KV-Aktivierungen oder Attention-Gewichte über Schichten hinweg gemeinsam nutzen, um sowohl den Rechenaufwand als auch den Speicherbedarf zu reduzieren [Xiao et al., 2019; Brandon et al., 2024]. Abbildung 2.8 (e) zeigt eine Veranschaulichung dieser Methode, wobei eine Query in einer Schicht direkt auf den KV-Cache einer untergeordneten Schicht zugreift.

2.3.5 Positionsextrapolation und -interpolation

Da Transformer -Schichten ordnungsunempfindlich gegenüber der Eingabe sind, benötigen wir eine Möglichkeit, Positionsinformationen in den Eingabe-Tokens zu kodieren. Dazu ist es üblich, Positions-Embeddings zu Token-Embeddings hinzuzufügen und diese kombinierten Embeddings als Eingabe in den Stapel der Transformer -Schichten einzuspeisen. In diesem Fall kann das Embedding an Position i wie folgt ausgedrückt werden:

$$\mathbf{e}_i \quad = \quad \mathbf{x}_i + \mathrm{PE}(i) \tag{2.74}$$

wobei $\mathbf{x}_i \in \mathbb{R}^d$ das Token-Embedding bezeichnet und $\mathrm{PE}(i) \in \mathbb{R}^d$ das Positions-Embedding. Im Allgemeinen ist das Token-Embedding $\mathbf{x}_i$ ein positionsunabhängiger Vektor, und daher wird das Positions-Embedding $\mathrm{PE}(i)$ verwendet, um den positionalen Kontext zu kodieren. Ein einfacher Ansatz besteht darin, $\mathrm{PE}(i)$ als lernbare Variable zu behandeln und es zusammen mit anderen Modellparametern zu trainieren. Auf diese Weise können wir für jede Position eine eindeutige Repräsentation lernen und so die an verschiedenen Positionen einer Sequenz erscheinenden Tokens unterscheiden.

Repräsentationen von Positionen mithilfe gelernter Vektoren können bei Aufgaben gut funktionieren, bei denen die Sequenzen zur Trainings- und Testzeit ähnliche Längen haben. In der Praxis legen wir jedoch während des Trainings oft Längenbeschränkungen für Sequenzen fest, um übermäßige Berechnungskosten zu vermeiden, möchten die trainierten Modelle aber während der Inferenz auf viel längere Sequenzen anwenden. In diesem Fall hat die Verwendung gelernter Positions-Embeddings offensichtliche Nachteile, da es keine trainierten Embeddings für Positionen gibt, die in der Trainingsphase nicht beobachtet wurden.

Ein alternativer Ansatz zur Modellierung von Positionsinformationen ist die Entwicklung von Positions-Embeddings, die generalisieren können: Einmal trainiert, kann das Embedding-Modell verwendet werden, um längere Sequenzen zu verarbeiten. Angenommen, wir trainieren ein Positions-Embedding-Modell an Sequenzen mit einer maximalen Länge von m_l und möchten das trainierte Modell auf eine Sequenz der Länge m ($m >> m_l$) anwenden. Wenn das Embedding-Modell auf den Bereich der Positionen beschränkt ist, die wir aus den Trainingsdaten beobachten können, wird dieses Modell einfach daran scheitern, neue Daten außerhalb dieses Bereichs zu verarbeiten. Siehe Abbildung 2.9 (a) für eine Veranschaulichung, bei der das gelernte Embedding-Modell Datenpunkte außerhalb des Trainingsbereichs nicht modellieren kann, wenn ihm die Fähigkeit zur Extrapolation fehlt.

Es gibt mehrere Ansätze, um Positions-Embedding-Modelle zu generalisieren. Sie können in zwei Klassen eingeteilt werden.

- Extrapolation. Das auf beobachteten Datenpunkten (d.h. Positionen) gelernte Modell kann direkt verwendet werden, um Datenpunkten, die außerhalb des ursprünglichen Bereichs liegen, aussagekräftige Werte zuzuordnen. Nehmen wir zum Beispiel an, wir haben eine Zahlenreihe $1, 2, ..., 10$ und wir möchten die Bedeutung einer neuen Zahl, 15, verstehen. Da wir wissen, dass diese Zahlen natürliche Zahlen sind, die zur Ordnung verwendet werden, können wir leicht schlussfolgern, dass 15 eine Zahl ist, die auf 10 folgt, obwohl 15 zuvor nicht beobachtet wurde. Abbildung 2.9 (b)

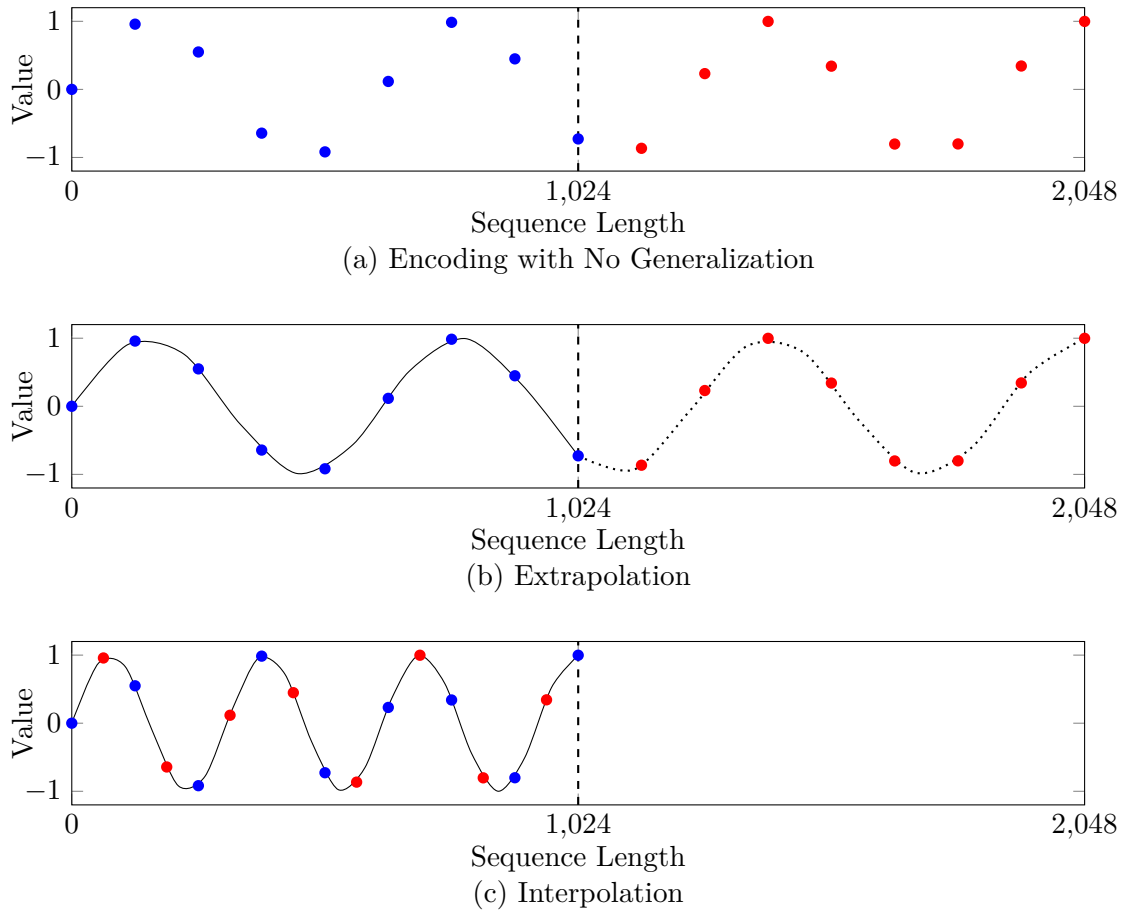


Abbildung 2.9: Darstellungen verschiedener Methoden für positionale Einbettungen für eine Reihe von Positionen. Blaue Punkte stellen die Positionen dar, die während des Trainings beobachtet wurden, und rote Punkte stellen die Positionen dar, die zur Testzeit neu beobachtet werden. In Teilabbildung (a) speichert das Kodierermodell nur die während des Trainings beobachteten Punkte und kann nicht generalisieren. In den Teilabbildungen (b) und (c) kann das Modell durch Extrapolation und Interpolation generalisieren.

zeigt ein Beispiel für diesen Ansatz, bei dem eine Funktion gelernt wird, um die Datenpunkte innerhalb eines bestimmten Bereichs anzupassen, und dann angewendet wird, um die Werte von Datenpunkten außerhalb dieses Bereichs zu schätzen.

- Interpolation. Dieser Ansatz bildet einen größeren Bereich von Datenpunkten auf den ursprünglichen Beobachtungsbereich ab. Nehmen wir zum Beispiel an, wir haben ein Modell, das für Zahlen im Bereich $[1, 10]$ ausgelegt ist. Wenn ein neuer Bereich von $[1, 20]$ gegeben ist, können wir diesen herunterskalieren, indem wir jede Zahl durch 2 teilen, wodurch alle Zahlen in den Bereich $[1, 10]$ passen. Diese Skalierung ermöglicht es uns, das auf dem Bereich $[1, 10]$ trainierte Modell zu verwenden, um Datenpunkte im erweiterten Bereich von $[1, 20]$ zu beschreiben. Siehe Abbildung 2.9 (c) für eine Veranschaulichung dieses Ansatzes.

Tatsächlich haben Positions-Embeddings in vielen Systemen ein gewisses Maß an Generalisierung erreicht. Zum Beispiel verwendet die sinusförmige Kodierung, die gängigste Methode für Positions-Embeddings, Sinus- und Kosinusfunktionen, die sich natürlich auf

Sequenzen beliebiger Länge erweitern lassen. Obwohl dieser Ansatz direkt und einfach erscheinen mag, funktioniert er nicht gut, wenn wir die zu verarbeitenden Sequenzen erheblich erweitern. In diesem Unterabschnitt werden wir mehrere alternative Methoden diskutieren, die entweder auf Extrapolation oder Interpolation basieren.

2.3.5.1 1. Aufmerksamkeit mit lernbaren Biases

Ein Problem bei Gl. (2.74) ist, dass das Einbettungsmodell jeden Token unabhängig behandelt und daher den Abstand zwischen verschiedenen Tokens ignoriert. Eine übliche Verbesserung dieses Modells, genannt relative positionale Einbettung, besteht darin, die paarweise Beziehung zwischen Tokens zu berücksichtigen [Shaw et al., 2018]. Die allgemeine Idee dahinter ist, den Versatz zwischen jedem Paar von Positionen zu ermitteln und ihn in das Self-Attention-Modell zu integrieren. Eine der einfachsten Formen der Self-Attention mit relativer positionaler Einbettung ist gegeben durch

$$\text{Att}_{\text{qkv}}(\mathbf{q}_i, \mathbf{K}_{\leq i}, \mathbf{V}_{\leq i}) = \sum_{j=0}^{i} \alpha(i,j)\mathbf{v}_j \tag{2.75}$$

$$\alpha(i,j) = \text{Softmax}(\frac{\mathbf{q}_i \mathbf{k}_j^{\mathrm{T}} + \text{PE}(i,j)}{\sqrt{d}} + \text{Mask}(i,j)) \tag{2.76}$$

Der einzige Unterschied zwischen diesem Modell und dem ursprünglichen Self-Attention-Modell besteht darin, dass in diesem neuen Modell ein Bias-Term $\text{PE}(i,j)$ zum Query-Key-Produkt hinzugefügt wird. Intuitiv kann $\text{PE}(i,j)$ als eine Distanzstrafe für das Positionspaar i und j interpretiert werden. Wenn sich i von j entfernt, nimmt der Wert von $\text{PE}(i,j)$ ab.

$\text{PE}(i,j)$ kann auf verschiedene Weisen definiert werden. Hier betrachten wir die T5-Version der relativen positionalen Einbettung, genannt der T5-Bias [Raffel et al., 2020]. Für jedes Paar aus Query $\mathbf{q}_i$ und Key $\mathbf{k}_j$ ist der Versatz zwischen ihnen definiert als[15]

$$d(i,j) = i - j \tag{2.77}$$

Ein einfacher Entwurf für den Bias $\text{PE}(i,j)$ besteht darin, dieselbe lernbare Variable für alle Query-Key-Paare mit demselben Versatz zu teilen, d.h., $\text{PE}(i,j) = u_{i-j}$, wobei u_{i-j} die Variable ist, die dem Versatz $i-j$ entspricht. Jedoch würde die einfache Zuweisung eines eindeutigen Wertes zu jedem Versatz dieses Modell auf beobachtete Versätze beschränken. Wenn $i-j$ größer ist als der maximal trainierte Versatz, kann das Modell nicht generalisieren.

Der T5-Bias verwendet stattdessen eine Generalisierung dieses Modells. Anstatt jedem Query-Key-Versatz einen eindeutigen Bias-Term zuzuweisen, gruppiert es unterschiedliche Versätze in „Buckets“, wobei jeder einem lernbaren Parameter entspricht. Genauer gesagt sind die Bias-Terme für $n_b + 1$ Buckets wie folgt gegeben.

[15] Für die Sprachmodellierung darf eine Query nur auf ihren linken Kontext achten, und somit gilt $i - j \geq 0$. Im allgemeineren Fall der Self-Attention, wo ein Token auf alle Tokens in der Sequenz achten kann, können wir negative Versätze haben, wenn $i < j$ ist.

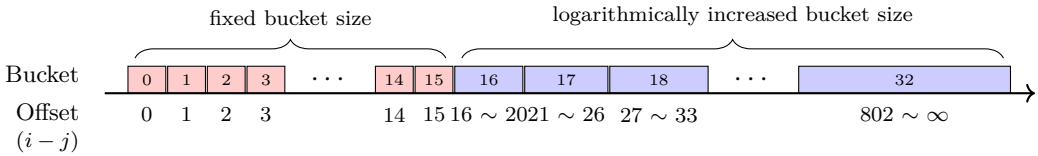


Abbildung 2.10: Veranschaulichung der Verteilung von Query-Key-Offsets in Buckets im T5-Modell ($n_b = 32$ und $\text{dist}_{\max} = 1024$). Die Kästen repräsentieren die Buckets. In der ersten Hälfte der Buckets wird eine feste Bucket-Größe verwendet. In der zweiten Hälfte der Buckets wird die Bucket-Größe logarithmisch erhöht. Der letzte Bucket enthält alle Query-Key-Offsets, die nicht von den vorherigen Buckets abgedeckt werden.

- Für die Buckets 0 bis $\frac{n_b+1}{2} - 1$ entspricht jeder Bucket einem Offset, d. h. Bucket 0 $\leftrightarrow$ Offset 0, Bucket 1 $\leftrightarrow$ Offset 1, Bucket 2 $\leftrightarrow$ Offset 2, und so weiter. Wir drücken dies als $b(\text{i} - \text{j}) = i - j$ aus.

- Für die Buckets $\frac{n_b+1}{2}$ bis n_b nimmt die Größe jedes Buckets logarithmisch zu. Beispielsweise kann die Bucket-Nummer für einen gegebenen Offset $i - j \geq \frac{n_b+1}{2}$ wie folgt definiert werden:

$$b(i-j) = \frac{n_b+1}{2} + \lfloor \frac{\log(i-j) - \log(\frac{n_b+1}{2})}{\log(\text{dist}_{\max}) - \log(\frac{n_b+1}{2})} \cdot \frac{n_b+1}{2} \rfloor \tag{2.78}$$

  wobei der Parameter $\text{dist}_{\max}$ typischerweise auf eine relativ große Zahl gesetzt wird, um den maximalen Offset anzugeben, dem wir begegnen könnten.

- Wenn $i - j > \text{dist}_{\max}$, platzieren wir $i - j$ im letzten Bucket. Mit anderen Worten, Bucket n_b enthält alle Offsets, die nicht den vorherigen Buckets zugewiesen wurden.

Zusammen können diese als die Funktion ausgedrückt werden

$$b(i-j) = \begin{cases} i-j & 0 \leq i-j < \frac{n_b+1}{2} \\ \min(n_b, \frac{n_b+1}{2} + \lfloor \frac{\log(i-j) - \log(\frac{n_b+1}{2})}{\log(\text{dist}_{\max}) - \log(\frac{n_b+1}{2})} \cdot \frac{n_b+1}{2} \rfloor) & i-j \geq \frac{n_b+1}{2} \end{cases} \tag{2.79}$$

Abbildung 2.10 zeigt eine Veranschaulichung dieser Buckets. Wir sehen, dass in der ersten Hälfte der Buckets jeder Bucket nur mit einem Wert von $i-j$ verknüpft ist, während in der zweiten Hälfte die Bucket-Größe zunimmt, wenn $i - j$ wächst. Der letzte Bucket ist dafür ausgelegt, Sequenzen von beliebig langen Längen zu verarbeiten.

Alle $\text{PE}(i, j)$ in einem Bucket teilen sich denselben Bias-Term $u_{b(i-j)}$. Durch Einsetzen

von $\text{PE}(i,j) = u_{b(i-j)}$ in Gl. (2.76) wird das Attention-Gewicht für $\mathbf{q}_i$ und $\mathbf{k}_j$ zu[16]

$$\alpha(i,j) \quad = \quad \text{Softmax}(\frac{\mathbf{q}_i\mathbf{k}_j^{\text{T}} + u_{b(i-j)}}{\sqrt{d}} + \text{Mask}(i,j)) \tag{2.81}$$

Die Parameter $\{u_0, ..., u_{n_b}\}$ werden während des Trainings als gemeinsame Parameter gelernt. Es sollte betont werden, dass dieses Modell auf lange Sequenzen generalisieren kann. Dies liegt daran, dass $\text{PE}(i,j)$s mit ähnlichen Query-Key-Versätzen denselben Parameter teilen, und diese Sharing-Strategie besonders wichtig ist, um eine gute Generalisierung zu erreichen, da große Query-Key-Versätze im Training selten sind. In der Praxis setzen wir n_b oft auf eine moderate Zahl, und es kann daher helfen, die Überanpassung von positionalen Einbettungsmodellen zu kontrollieren.

2.3.5.2 2. Aufmerksamkeit mit nicht-gelernten Biases

Modelle mit relativer positionaler Einbettung basieren auf einem Satz gelernter Biases für das Query-Key-Produkt in der Self-Attention. Ein alternativer Ansatz besteht darin, diesen Biases feste Werte mittels Heuristiken zuzuweisen, anstatt sie auf einem bestimmten Datensatz zu trainieren. Ein Vorteil dieses heuristikbasierten Ansatzes ist, dass er nicht auf einem Trainingsprozess beruht und daher direkt auf beliebige Sequenzen angewendet werden kann, sobald die Biases festgelegt sind.

Ein Beispiel für einen solchen Ansatz ist der Ansatz von Press et al. [2022], kurz attention with linear biases oder ALiBi genannt. Beim ALiBi-Ansatz wird der Bias-Term als der negativ skalierte Query-Key-Offset definiert

$$\begin{aligned} \text{PE}(i,j) \quad &= \quad -\beta \cdot (i-j) \\ &= \quad \beta \cdot (j-i) \end{aligned} \tag{2.82}$$

wobei β der Skalierungsfaktor ist. Indem wir diesen Term zum Query-Key-Produkt hinzufügen, erhalten wir eine neue Form der Aufmerksamkeitsgewichte

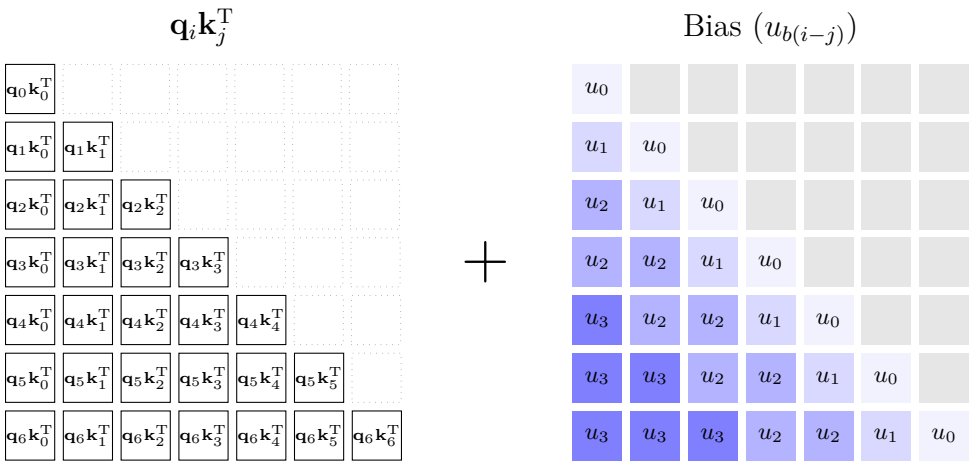
$$\alpha(i,j) \quad = \quad \text{Softmax}(\frac{\mathbf{q}_i\mathbf{k}_j^{\text{T}} + \beta \cdot (j-i)}{\sqrt{d}} + \text{Mask}(i,j)) \tag{2.83}$$

Dieses Modell kann so interpretiert werden, dass es eine feste Strafe zu $\mathbf{q}_i\mathbf{k}_j^{\text{T}}$ hinzufügt, wann immer sich j einen Schritt von i entfernt. Daher müssen wir es nicht an einen Bereich von Sequenzlängen anpassen und können es verwenden, um beliebig lange Sequenzen zu modellieren. Siehe Abbildung 2.11 für einen Vergleich des T5-Bias und des ALiBi-Bias.

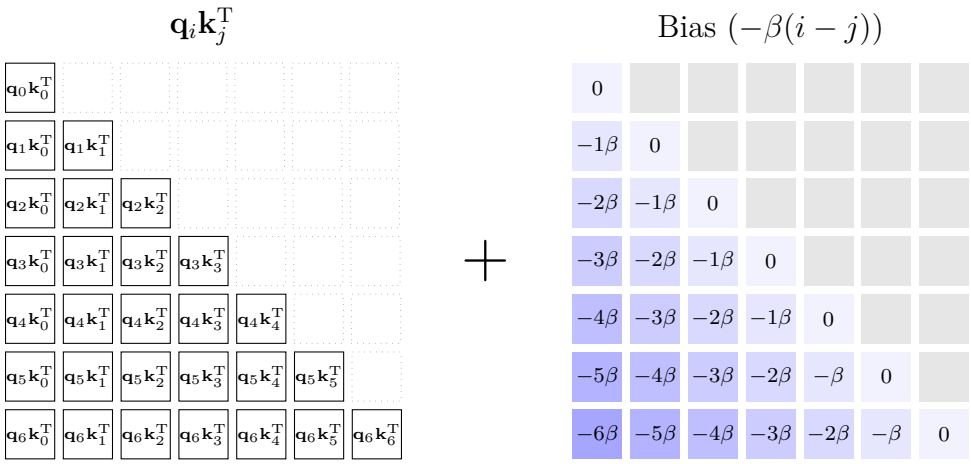
Im Allgemeinen sollte der Skalar β auf einem Validierungsdatensatz abgestimmt werden. Allerdings fanden Press et al. [2022] heraus, dass das Setzen von β auf Werte, die

[16] Beachten Sie, dass im T5-Modell von Raffel et al. [2020] die Neuskalierungsoperation für das Query-Key-Produkt entfernt wird. Das Attention-Gewicht $\alpha(i,j)$ ist dann gegeben durch

$$\alpha(i,j) \quad = \quad \text{Softmax}(\mathbf{q}_i\mathbf{k}_j^{\text{T}} + u_{b(i-j)} + \text{Mask}(i,j)) \tag{2.80}$$



(a) The T5 bias ($n_b = 3$ and $\mathrm{dist}_{\max} = 5$)



(b) The ALiBi bias

Abbildung 2.11: Query-Key-Produkte mit Biases (oben = der T5-Bias und unten = der ALiBi-Bias). Die Farbskala der Biases reicht von hellblau für kleine Absolutwerte bis dunkelblau für große Absolutwerte.

für Multi-Head-Attention geometrisch um einen Faktor von $\frac{1}{2^a}$ abnehmen, bei einer Vielzahl von Aufgaben gut funktioniert. Insbesondere ist für eine Self-Attention-Sub-Layer mit n_{head} Heads der Skalar für den k-ten Head gegeben durch

$$\beta_k = \frac{1}{2^{\frac{8}{k}}} \tag{2.84}$$

Der ALiBi-Ansatz bietet eine einfache Form von relativen positionalen Einbettungen. Es gibt andere ähnliche Methoden zur Gestaltung von Query-Key-Biases unter Verwendung des Offsets $i - j$. Tabelle 2.4 zeigt einen Vergleich solcher Biases. Nebenbei sei angemerkt, dass die Form der rechten Seite von Gl. (2.82) den Längenmerkmalen, die in konventionellen merkmalsbasierten Systemen verwendet werden, sehr ähnlich ist. In statistischen maschinellen Übersetzungssystemen werden solche Merkmale beispielsweise

Entry	Query-Key Bias (PE(i,j))
T5 [Raffel et al., 2020]	$u_{b(i-j)}$
ALiBi [Press et al., 2022]	$-\beta \cdot (\,i-j\,)$
Kerple [Chi et al., 2022]	$-\beta_1(\,i-j\,)^{\beta_2}$ (power)
	$-\beta_1 \log(1+\beta_2(\,i-j\,))$ (logarithmic)
Sandwich [Chi et al., 2023]	$\sum_{k=1}^{\bar{d}/2} \cos\left((\,i-j\,)/10000^{2k/\bar{d}}\right)$
FIRE [Li et al., 2024b]	$f(\psi(\,i-j\,)/\psi(\max(m_{\text{len}}, i)))$

Tabelle 2.4: Query-Key-Biases als relative positionale Einbettungen. β, β_1, β_2, $\bar{d}$ und m_{len} sind Hyperparameter. Im T5-Modell bezeichnet $b(i-j)$ den Bucket, der $i-j$ zugewiesen ist. Im FIRE-Modell ist $\psi(\cdot)$ eine monoton steigende Funktion, wie z. B. $\psi(x) = \log(cx+1)$, und $f(\cdot)$ ist ein FFN.

häufig verwendet, um Wortumordnungsprobleme zu modellieren, was zu Modellen führt, die gut über verschiedene Übersetzungsaufgaben hinweg generalisieren können [Koehn, 2010].

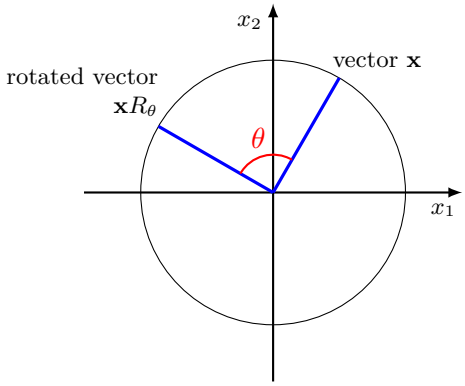
2.3.5.3 3. Rotatorische Positionseinbettung

Wie bei sinusförmigen Einbettungen basieren rotatorische Positionseinbettungen auf fest kodierten Werten für alle Dimensionen einer Einbettung [Su et al., 2024]. Zur Erinnerung: Im Modell der sinusförmigen Einbettung werden Positionen als Kombinationen von Sinus- und Kosinusfunktionen mit unterschiedlichen Frequenzen dargestellt. Diese Einbettungen werden dann zu den Token-Einbettungen addiert, um die Eingaben für den Transformer - Schichtenstapel zu bilden. Rotatorische Positionseinbettungen modellieren stattdessen den positionalen Kontext als Rotationen von Token-Einbettungen in einem komplexen Raum. Dies führt zu einem Modell, das in Form von multiplikativen Einbettungen ausgedrückt wird

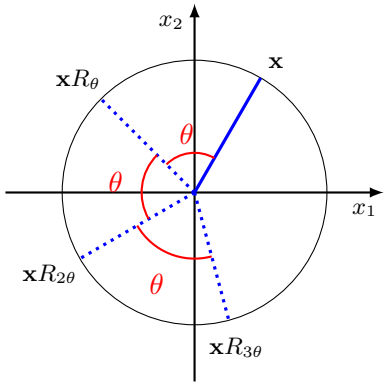
$$\mathbf{e}_i \quad = \quad \mathbf{x}_i R(i) \tag{2.85}$$

wobei $R(i) \in \mathbb{R}^{d \times d}$ die Rotationsmatrix ist, die die auf die Token-Einbettung $\mathbf{x}_i \in \mathbb{R}^d$ angewendeten Rotationen darstellt.

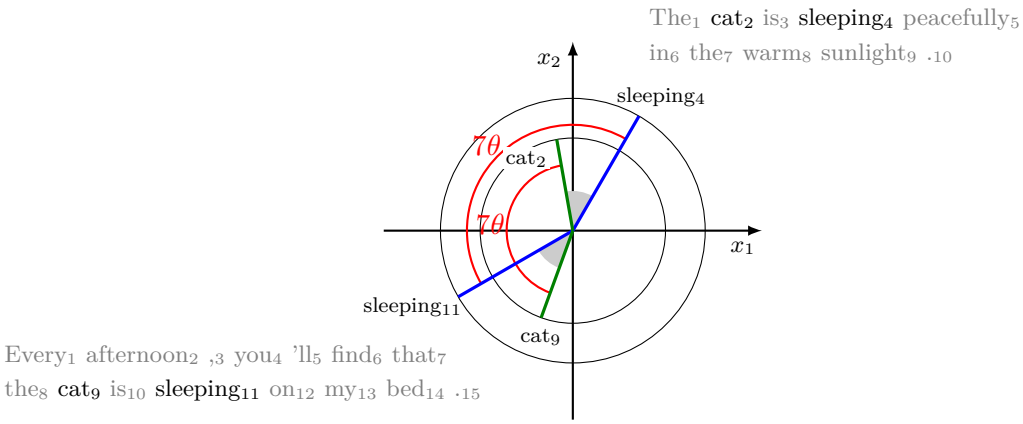
Der Einfachheit halber betrachten wir zunächst Einbettungen mit nur zwei Dimensionen und kehren später zu einer Diskussion der allgemeineren Formulierung zurück. Angenommen, wir haben eine 2-dimensionale Token-Einbettung $\mathbf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}$. Wir können sie als Vektor in einer Ebene darstellen, der im Ursprung $(0,0)$ beginnt und bei (x_1, x_2) endet. Eine Drehung dieses Vektors gegen den Uhrzeigersinn bezieht sich auf eine Operation, bei der der Vektor um den Ursprung bewegt wird, während seine Größe beibehalten wird, wie in Abbildung 2.12 (a) gezeigt. Der Grad der Drehung wird normalerweise durch einen bestimmten Winkel definiert, der mit θ bezeichnet wird. Die Drehung kann mathematisch



(a) Single-step Rotation



(b) Multi-step Rotation



(c) Angles between embeddings of two tokens at different positions

Abbildung 2.12: Illustrationen von Vektorrotationen in einer Ebene. Die Teilabbildungen (a) und (b) zeigen Rotationen eines Vektors in einem einzelnen bzw. in mehreren Schritten. Teilabbildung (c) zeigt die Einbettungen der token cat und sleeping in zwei verschiedenen Sätzen. Wir zeigen diese Sätze mit einem tiefgestellten Index, der jedem token zur Angabe seiner Position beigefügt ist. Wenn wir token als Vektoren darstellen, können wir Positionsinformationen durch die Rotation dieser Vektoren hinzufügen. Diese Rotation erhält die "Abstände" zwischen den Vektoren. Ist beispielsweise der Abstand zwischen cat und sleeping in beiden Sätzen gleich, so bleibt auch der Winkel zwischen ihren Einbettungen während der Rotation derselbe.

in der Form ausgedrückt werden

$$\begin{aligned} \mathrm{Ro}(\mathbf{x}, \theta) &= \mathbf{x}R_\theta \\ &= \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \\ &= \begin{bmatrix} \cos\theta \cdot x_1 - \sin\theta \cdot x_2 & \sin\theta \cdot x_1 + \cos\theta \cdot x_2 \end{bmatrix} \end{aligned} \tag{2.86}$$

wobei $R_\theta = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$ die Rotationsmatrix ist. Wenn zwei oder mehr Drehungen auf denselben Vektor angewendet werden, können wir den Vektor weiter drehen. Dies ergibt sich aus der Tatsache, dass die Zusammensetzung aufeinanderfolgender Drehungen

selbst eine Drehung ist. Formaler ausgedrückt, kann die t-malige Drehung eines Vektors um einen Winkel θ ausgedrückt werden als

$$\begin{aligned} \text{Ro}(\mathbf{x}, t\theta) &= \mathbf{x}R_{t\theta} \\ &= \begin{bmatrix} \cos t\theta \cdot x_1 - \sin t\theta \cdot x_2 & \sin t\theta \cdot x_1 + \cos t\theta \cdot x_2 \end{bmatrix} \end{aligned} \tag{2.87}$$

Wenn wir t als die Position eines durch $\mathbf{x}$ dargestellten Tokens in einer Sequenz interpretieren, dann stellen wir fest, dass die obige Gleichung ein einfaches Positionseinbettungsmodell definiert. Wie in Abbildung 2.12 (b) gezeigt, beginnen wir, das Token von Position 0 aus zu bewegen. Jedes Mal, wenn wir einen Schritt vorwärts gehen, wird der Vektor um den Winkel θ gedreht. Bei Erreichen der Position t ist die Darstellung des Tokens mit positionellem Kontext durch $\text{Ro}(\mathbf{x}, i\theta)$ gegeben. Da die Drehungen die Größe der Einbettung nicht verändern, bleibt die ursprüngliche „Bedeutung"des Tokens erhalten. Die Positionsinformation wird in die Einbettung injiziert, wenn sie gedreht wird.

Eine gängige Methode, die Vektordrehung zu verstehen, besteht darin, sie in komplexen Räumen zu definieren. Es ist einfach, jeden Vektor $\mathbf{x} = \begin{bmatrix} x_1 & x_2 \end{bmatrix}$ im 2D-euklidischen Raum $\mathbb{R}^2$ mittels einer bijektiven linearen Abbildung in eine komplexe Zahl $\mathbf{x}' = x_1 + \mathtt{i}x_2$ im komplexen Raum $\mathbb{C}$ umzuwandeln. Dann entspricht die Drehung von $\mathbf{x}$ um den Winkel $t\theta$ der Multiplikation mit $e^{\mathtt{i}t\theta}$. Da $e^{\mathtt{i}t\theta} = \cos t\theta + \mathtt{i} \sin t\theta$ gilt, kann die Rotationsoperation in der Form neu ausgedrückt werden

$$\begin{aligned} \mathbf{x}R_{t\theta} &\mapsto \mathbf{x}'e^{\mathtt{i}t\theta} \\ &= (x_1 + \mathtt{i}x_2)(\cos t\theta + \mathtt{i} \sin t\theta) \\ &= \cos t\theta \cdot x_1 - \sin t\theta \cdot x_2 + \mathtt{i}(\sin t\theta \cdot x_1 + \cos t\theta \cdot x_2) \end{aligned} \tag{2.88}$$

Hier bezeichnen wir die Token-Darstellung $\mathbf{x}'e^{\mathtt{i}t\theta}$ mit $C(\mathbf{x}, t\theta)$. Das Skalarprodukt der Darstellungen der Tokens an den Positionen t und s kann geschrieben werden als

$$\langle C(\mathbf{x}, t\theta), C(\mathbf{y}, s\theta) \rangle = (\mathbf{x}'\overline{\mathbf{y}'})e^{\mathtt{i}(t-s)\theta} \tag{2.89}$$

wobei $\overline{\mathbf{y}'}$ das komplex Konjugierte von $\mathbf{y}'$ ist. Wie man sehen kann, enthält das Ergebnis dieses Skalarprodukts einen Term $t - s$ und kann daher den Versatz zwischen den beiden Tokens modellieren.

Nun kehren wir zu den Darstellungen im 2D-euklidischen Raum zurück. Das Skalarprodukt von $\text{Ro}(\mathbf{x}, t\theta)$ und $\text{Ro}(\mathbf{y}, s\theta)$ kann als Funktion von $(t - s)\theta$ geschrieben werden

$$\begin{aligned} \text{Ro}(\mathbf{x}, t\theta)[\text{Ro}(\mathbf{y}, s\theta)]^{\mathrm{T}} &= \mathbf{x}R_{t\theta}[\mathbf{y}R_{s\theta}]^{\mathrm{T}} \\ &= \mathbf{x}R_{t\theta}[R_{s\theta}]^{\mathrm{T}}\mathbf{y}^{\mathrm{T}} \\ &= \mathbf{x}R_{(t-s)\theta}\mathbf{y}^{\mathrm{T}} \end{aligned} \tag{2.90}$$

Wenn wir angesichts dieses Ergebnisses $\text{Ro}(\mathbf{x}, t\theta)$ und $\text{Ro}(\mathbf{y}, s\theta)$ als Abfrage (Query) und Schlüssel (Key) betrachten, dann wird die Selbst-Aufmerksamkeits-Operation implizit die Modellierung des relativen positionalen Kontexts beinhalten.

Diese rotatorische Positionseinbettung kann auf mehrdimensionale Einbettungen erweitert werden. Für eine d-dimensionale Token-Einbettung $\mathbf{x} = \begin{bmatrix} x_1 & x_2 & ... & x_d \end{bmatrix}$ können wir sie als einen $\frac{d}{2}$-dimensionalen komplexen Vektor $\mathbf{x}' = \begin{bmatrix} x'_1 & x'_2 & ... & x'_{d/2} \end{bmatrix} = \begin{bmatrix} x_1 + \mathtt{i}x_2 & x_3 + \mathtt{i}x_4 & ... & x_{d-1} + \mathtt{i}x_d \end{bmatrix}$ behandeln, wobei jedes aufeinanderfolgende Paar von Elementen eine komplexe Zahl bildet. Dann ist die rotatorische Positionseinbettung im komplexen Raum gegeben durch

$$C(\mathbf{x}, t\theta) = \sum_{k=1}^{d/2} x'_k e^{\mathtt{i}t\theta_k} \vec{e}_k \tag{2.91}$$

wobei $\vec{e}_k$ der Standardbasisvektor mit einem einzigen Nicht-Null-Wert in der k-ten Koordinate und Nullen an allen anderen Stellen ist [Biderman et al., 2021].

Obwohl diese Formel einen komplizierten Ausdruck beinhaltet, ist ihre äquivalente Form im d-dimensionalen euklidischen Raum relativ einfach zu verstehen. Wir können sie schreiben als

$$\text{Ro}(\mathbf{x}, t\theta) = \begin{bmatrix} x_1 & x_2 & ... & x_d \end{bmatrix} \begin{bmatrix} R_{t\theta_1} & & & \\ & R_{t\theta_2} & & \\ & & \ddots & \\ & & & R_{t\theta_{d/2}} \end{bmatrix} \tag{2.92}$$

wobei $R_{t\theta_k} = \begin{bmatrix} \cos t\theta_k & \sin t\theta_k \\ -\sin t\theta_k & \cos t\theta_k \end{bmatrix}$. $\theta = \begin{bmatrix} \theta_1, ..., \theta_{d/2} \end{bmatrix}$ sind die Parameter zur Steuerung der Drehwinkel in verschiedenen Dimensionen. Typischerweise wird θ_k auf $10000^{-\frac{2(k-1)}{d}}$ gesetzt, was analog zur Einstellung bei sinusförmigen Einbettungen ist.

In einer praktischen Implementierung kann Gl. (2.92) in eine Form umgeschrieben werden, die ausschließlich auf dem elementweisen Produkt und der Addition von Vektoren beruht.

$$\text{Ro}(\mathbf{x}, t\theta) = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{d-1} \\ x_d \end{bmatrix}^{\text{T}} \odot \begin{bmatrix} \cos t\theta_1 \\ \cos t\theta_1 \\ \vdots \\ \cos t\theta_{d/2} \\ \cos t\theta_{d/2} \end{bmatrix}^{\text{T}} + \begin{bmatrix} -x_2 \\ x_1 \\ \vdots \\ -x_d \\ x_{d-1} \end{bmatrix}^{\text{T}} \odot \begin{bmatrix} \sin t\theta_1 \\ \sin t\theta_1 \\ \vdots \\ \sin t\theta_{d/2} \\ \sin t\theta_{d/2} \end{bmatrix}^{\text{T}} \tag{2.93}$$

Schließlich schreiben wir Gl. (2.85) um, um die Form der Einbettung an Position i zu erhalten

$$\mathbf{e}_i = \text{Ro}(\mathbf{x}_i, i\theta) \tag{2.94}$$

2.3.5.4 4. Positionsinterpolation

Bei der Positionsinterpolation ist unser Ziel, die Positionen in der neuen Sequenz so abzubilden, dass sie dem im Training beobachteten Bereich entsprechen. Angenommen, die Sequenzlänge für das Training reicht von 0 bis m_l. Wenn zur Testzeit $m > m_l$ ist, stellen wir die Positionen in $[0, m]$ so dar, dass unsere Repräsentationen in $[0, m_l]$ passen.

Zur Veranschaulichung betrachten wir das oben beschriebene rotierende Positionseinbettungsmodell. Die Einbettung jedes Tokens wird durch ein Modell $\text{Ro}(\mathbf{x}_i, i\theta)$ beschrieben, in dem $\theta = \left[\theta_1, ..., \theta_{d/2}\right]$ die Parameter sind. $\text{Ro}(\mathbf{x}_i, i\theta)$ kann in Form einer Linearkombination von zwei periodischen Funktionen dargestellt werden (siehe Gl. (2.93))

$$\cos i\theta = \left[\cos i\theta_1 \quad ... \quad \cos i\theta_{d/2}\right] \tag{2.95}$$

$$\sin i\theta = \left[\sin i\theta_1 \quad ... \quad \sin i\theta_{d/2}\right] \tag{2.96}$$

θ_k ist eine Exponentialfunktion von k und hat die Form

$$\theta_k = b^{-\frac{2(k-1)}{d}} \tag{2.97}$$

wobei b die Basis ist. Die Periode von $\cos i\theta_k$ und $\sin i\theta_k$ ist

$$T_k = 2\pi \cdot b^{\frac{2(k-1)}{d}} \tag{2.98}$$

Die Schlüsselidee hinter der Positionsinterpolation besteht darin, diese Periode anzupassen, sodass die neuen Positionen innerhalb des Bereichs $[0, m_l]$ kodiert werden können. Eine Möglichkeit, dies zu erreichen, besteht darin, T_k um $\frac{m}{m_l}$ zu skalieren, gegeben durch

$$T'_k = \frac{m}{m_l} \cdot 2\pi \cdot b^{\frac{2(k-1)}{d}} \tag{2.99}$$

Somit werden alle Punkte in $[0, m]$ in $[0, m_l]$ komprimiert. Diese lineare Skalierung kann einfach durch Modifizieren der Eingabe in das Einbettungsmodell realisiert werden [Chen et al., 2023c]. Das neue Modell mit linearer Positionsinterpolation ist gegeben durch

$$\text{Ro}'(\mathbf{x}_i, i\theta) = \text{Ro}(\mathbf{x}_i, \frac{m_l}{m} i\theta) \tag{2.100}$$

Eine andere Methode der Positionsinterpolation ist die Skalierung der Basis[17]. Angenommen, die Basis b wird mit λ skaliert. Wir möchten, dass die Periode dieses neuen Modells in der letzten Dimension von θ (d. h. Dimension $\frac{d}{2}$) der des linearen Positionsinterpolationsmodells entspricht. Dies kann ausgedrückt werden als

$$2\pi \cdot (\lambda b)^{\frac{2(\frac{d}{2}-1)}{d}} = \frac{m}{m_l} \cdot 2\pi \cdot b^{\frac{2(\frac{d}{2}-1)}{d}} \tag{2.101}$$

[17] Diese Methode wurde zuerst in https://www.reddit.com/r/LocalLLaMA/comments/14lz7j5/ntkaware_scaled_rope_allows_llama_models_to_have/ vorgeschlagen.

Durch Lösen dieser Gleichung erhalten wir

$$\begin{aligned} \lambda &= (\frac{m}{m_l})^{\frac{d}{2(\frac{d}{2}-1)}} \\ &= (\frac{m}{m_l})^{\frac{d}{d-2}} \end{aligned} \tag{2.102}$$

Dies ergibt ein Einbettungsmodell

$$\text{Ro}'(\mathbf{x}_i, i\theta) = \text{Ro}(\mathbf{x}_i, i\theta') \tag{2.103}$$

wobei

$$\theta' = \left[(\lambda b)^{-\frac{0}{d}}, (\lambda b)^{-\frac{2}{d}}, ..., (\lambda b)^{-\frac{d-2}{d}}\right] \tag{2.104}$$

Beachten Sie, dass die Skalierung der Basis eine ungleichmäßige Methode zur Skalierung der Perioden über verschiedene Dimensionen von θ bietet. Es hat sich gezeigt, dass diese Methode hilfreich ist, um LLMs auf längere Sequenzen zu erweitern, und es wurden mehrere Verbesserungen entwickelt [Peng et al., 2024; Ding et al., 2024].

2.3.6 Bemerkungen

In diesem Abschnitt haben wir eine Vielzahl von Methoden für die Sprachmodellierung mit langem Kontext vorgestellt. Wir schließen diesen Abschnitt ab, indem wir einige interessante Aspekte im Zusammenhang mit diesen Methoden diskutieren.

2.3.6.1 1. Bedarf an langem Kontext

Eines der ultimativen Ziele von LLMs mit langem Kontext ist, dass diese Modelle unendlichen Kontext präzise kodieren können. Der sogenannte unendliche Kontext bezieht sich eher auf die Tatsache, dass ein LLM kontinuierlich Wörter lesen kann. Dies motiviert LLMs, die extrem langen Kontext oder Streaming-Daten verarbeiten können. Wie in Abschnitt 2.3.3 erörtert, ist es üblich, Modelle mit fester Speichergröße zu verwenden, um kontinuierlich expandierenden Kontext zu verarbeiten. Viele solcher Systeme basieren auf rekurrenten Architekturen oder deren Varianten, da sie von Natur aus geeignet sind, Zeitreihenprobleme zu modellieren, bei denen die Auswirkungen vergangener Eingaben unbegrenzt andauern. Eine andere Möglichkeit, unendlichen Speicher zu erreichen, besteht darin, Alternativen zu Selbst-Aufmerksamkeits-Modellen zu entwickeln, zum Beispiel kann man Modelle mit kontinuierlichem Raum für die Aufmerksamkeit verwenden, um Kontext zu kodieren, was die Abhängigkeit von der Kontextlänge aufhebt [Martins et al., 2022].

Bei der Untersuchung von LLMs mit langem Kontext fragt man sich natürlich, welche Mechanismen die Verwendung von langem Kontext bei der Sprachmodellierung erklären könnten. Können wir die Darstellung von unendlichem Kontext in ein relativ kleines Modell komprimieren? Sind alle Kontext-Token nützlich, um die nächsten Token vorherzusagen? Wie bereiten sich LLMs auf die Token-Vorhersage vor, wenn sie den Kontext

sehen? Können wir im Voraus wissen, welche kontextuellen Informationen für die Vorhersage entscheidend sein werden? Allgemeine Antworten auf all diese Fragen sind nicht offensichtlich, aber sie inspirieren die weiterführende Forschung an erklärbaren Modellen, und es wurden einige interessante Ergebnisse gefunden. Zum Beispiel führten Deletang et al. [2024] umfangreiche Experimente durch, um zu zeigen, dass LLMs leistungsstarke In-Kontext-Kompressoren sind. Obwohl die Betrachtung von Vorhersagemodellen als Kompressionsmodelle im maschinellen Lernen seit langem untersucht wird, liefert dies auch Einblicke in unser Verständnis der LLM-Skalierungsgesetze. Pal et al. [2023] und Wu et al. [2024] untersuchten, ob die bis zum aktuellen Schritt gelernten Merkmale, obwohl nicht beabsichtigt, bereits ausreichen, um Token in den folgenden Schritten vorherzusagen. Beachten Sie, dass der Bedarf an langem Kontext in der Sprachmodellierung stark von dem Problem abhängt, das wir behandeln. Ein verwandtes Thema ist, wo LLMs angewendet und wie sie bewertet werden sollen. Zum Beispiel müssen wir bei Zusammenfassungsaufgaben möglicherweise nur einige wenige Schlüsselaspekte des Textes destillieren und uns darauf konzentrieren, während wir bei abrufähnlichen Aufgaben den gesamten Kontext „auswendig lernen"müssen, damit auf die relevanten Informationen zugegriffen werden kann. Wir werden das Thema der Evaluierung später in diesem Unterabschnitt erörtern.

2.3.6.2 2. Vortraining oder Anpassung von LLMs?

Das Training von LLMs erfordert erhebliche Rechenkosten. Obwohl es unkompliziert ist, LLMs auf langen Sequenzdaten zu trainieren, wird das Training bei großen Datensätzen rechentechnisch unhandlich. Es ist gängige Praxis, LLMs auf allgemeinen Datensätzen vorzutrainieren und sie dann mit geringem Feinabstimmungsaufwand anzupassen. Zum Beispiel können LLMs mit relativen oder rotatorischen Positionseinbettungen direkt auf großen Datenmengen in der Vortrainingsphase trainiert werden. Während die resultierenden Modelle in der Inferenzphase eine gewisse Fähigkeit zur Längenextrapolation aufweisen können, ist es möglicherweise effektiver, sie auf längeren Sequenzen feinabzustimmen.

Idealerweise würden wir LLMs mit Standard-Transformer -Architekturen vortrainieren und sie an neue Aufgaben anpassen. Dies ermöglicht es uns, viele handelsübliche LLMs zu verwenden und sie effizient für die Verarbeitung langer Sequenzen anzupassen. Wenn jedoch neue Architekturen eingeführt werden, scheint es unvermeidlich, dass wir diese Modelle von Grund auf neu trainieren müssen. Dies stellt praktische Schwierigkeiten bei der Entwicklung von LLMs mit langem Kontext dar, da wir keine gut entwickelten, vortrainierten Modelle nutzen können und sie stattdessen selbst trainieren müssen. Andererseits ist die Feinabstimmung immer noch eine effektive Methode, um LLMs mit bestimmten Architekturen anzupassen, die sich von denen des Vortrainings unterscheiden. Ein Beispiel sind Modelle, die mit externen Speichern erweitert werden. In diesen Modellen sind die vortrainierten LLMs fixiert, und der Fokus liegt darauf, wie diese LLMs mit den Speichermodellen zusammenarbeiten können. Bei RAG ist es beispielsweise üblich, LLMs feinabzustimmen, um ihre Verwendung von abrufgestützten Eingaben zu verbessern. Ein weiteres Beispiel für die Feinabstimmung von LLMs für die Langkontext-Modellierung besteht darin, ein LLM mit vollständigen Aufmerksamkeitsmodellen zu trainieren und diese dann in der Feinabstimmungsphase durch spärliche Aufmerksamkeitsmodelle zu ersetzen. Das vortrainierte LLM liefert Anfangswerte für die Modellparameter, die in einem anderen Modell verwendet werden, und dieses Modell wird dann wie gewohnt feinabgestimmt.

2.3.6.3 3. Evaluierung von Langkontext-LLMs

Die Evaluierung von Langkontext-LLMs ist wichtig, aber es handelt sich um eine neue Problemstellung in der Verarbeitung natürlicher Sprache (NLP). Die allgemeine Idee ist, dass wir, wenn wir einem LLM einen langen Kontext eingeben, anhand der Ausgabe des LLMs überprüfen können, ob es den gesamten Kontext versteht und ihn bei der Vorhersage nachfolgender Tokens verwendet. In der herkömmlichen NLP-Forschung zielen solche Evaluierungen oft darauf ab, die Fähigkeit von NLP-Modellen zu untersuchen, mit weitreichenden Abhängigkeiten umzugehen. Jedoch ist der Umfang der Kontexte, die in aktuellen LLMs verwendet werden, wesentlich größer als der, der noch vor einigen Jahren in NLP-Systemen verwendet wurde. Dies motiviert Forschende, neue Evaluierungs-Benchmarks und Metriken für Langkontext-LLMs zu entwickeln.

Ein Ansatz ist die Verwendung der Perplexitätsmetrik. Trotz ihrer scheinbaren Einfachheit tendiert diese Methode jedoch dazu, eher die Fähigkeit der LLMs widerzuspiegeln, lokalen Kontext anstatt globalen Kontext zu nutzen. Es ist daher naheliegend, Evaluierungsmethoden zu entwickeln, die spezifisch für Langkontext-LLMs sind. Beliebte Methoden umfassen verschiedene synthetische Aufgaben, bei denen künstlich erzeugte oder modifizierte Daten verwendet werden, um spezifische Fähigkeiten von Langkontext-LLMs zu bewerten. Bei den Aufgaben ‚Nadel im Heuhaufen'(needle-in-a-haystack)[18] und ‚Passkey-Abruf'(passkey retrieval) [Mohtashami and Jaggi, 2024; Chen et al., 2023c] müssen LLMs beispielsweise eine kleine, relevante Information aus einer großen Menge an gegebenem Text identifizieren und extrahieren. Die Annahme hierbei ist, dass ein LLM mit ausreichendem Gedächtnis sich an frühere Teile des Textes erinnern sollte, während es neue Informationen verarbeitet. Dieses LLM kann somit die relevanten Details, die möglicherweise spärlich und zwischen vielen irrelevanten Informationen verborgen sind, aus dem Text herausfiltern. Alternativ werden bei ‚Kopierspeicher'-Aufgaben (copy memory tasks, oder kurz Kopieraufgaben) LLMs dazu verwendet, den Eingabetext oder ein bestimmtes Segment mehrmals zu wiederholen. Diese Aufgaben wurden ursprünglich vorgeschlagen, um zu testen, inwieweit rekurrente Modelle zuvor gesehene Tokens beibehalten und abrufen können [Hochreiter and Schmidhuber, 1997; Arjovsky et al., 2016], und wurden für die Evaluierung aktueller LLMs übernommen [Bulatov et al., 2022; Gu and Dao, 2023].

Ein weiterer Ansatz zur Evaluierung von Langkontext-LLMs besteht darin, sie auf NLP-Aufgaben zu testen, die sehr lange Eingabesequenzen beinhalten. Beispiele hierfür sind die Zusammenfassung langer Dokumente oder mehrerer Dokumente, die Fragenbeantwortung für lange Dokumente, die Code-Vervollständigung und so weiter. Ein Vorteil dieses Ansatzes ist, dass er die Evaluierungen an den Erwartungen der Nutzer ausrichten kann.

Obwohl viele Methoden entwickelt wurden, gibt es noch keine allgemeingültige Methode zur Evaluierung von Langkontext-LLMs [Liu et al., 2024c]. Ein Problem besteht darin, dass sich die meisten dieser Methoden auf spezifische Aspekte von LLMs konzentrieren, anstatt auf ihre grundlegende Fähigkeit, sehr lange Kontexte zu modellieren. Auch wenn ein LLM das passende Textstück aus der Eingabe auswählen kann, können wir nicht sagen, dass es den gesamten Kontext wirklich versteht. Stattdessen erinnert es sich möglicherweise nur an einige wichtige Teile des Kontexts oder ruft sogar einfach die Antwort über das im Vortraining gelernte Modell ab. Darüber hinaus sind die in vielen

[18] https://github.com/gkamradt/LLMTest_NeedleInAHaystack

Aufgaben verwendeten Daten klein und relativ vorläufig, was zu Diskrepanzen zwischen den Evaluierungsergebnissen und der tatsächlichen Anwendungsleistung führt. Ein weiteres interessantes Problem ist, dass die Ergebnisse von LLMs von vielen anderen Faktoren und experimentellen Anordnungen beeinflusst werden; beispielsweise kann die Verwendung unterschiedlicher Prompts zu sehr unterschiedlichen Ergebnissen führen. Dies macht die Evaluierung noch schwieriger, da Verbesserungen möglicherweise nicht allein aus einer besseren Modellierung langer Kontexte resultieren und das Risiko besteht, unsere Ergebnisse zu überbewerten. Dennoch bleiben viele offene Fragen bei der Entwicklung und Evaluierung von Langkontext-LLMs. Zum Beispiel leiden diese Modelle immer noch unter Einschränkungen wie einer begrenzten Kontextlänge und hoher Latenz. Die Untersuchung dieser Probleme wird sich wahrscheinlich als wertvolle zukünftige Forschungsrichtung erweisen.

2.4 Zusammenfassung

In diesem Kapitel haben wir das Konzept von LLMs und verwandte Techniken erörtert. Dies kann als eine allgemeine, wenn auch nicht umfassende, Einführung in LLMs betrachtet werden, die die Grundlage für weitere Diskussionen über fortgeschrittenere Themen in den folgenden Kapiteln legt. Darüber hinaus haben wir zwei Möglichkeiten zur Skalierung von LLMs untersucht. Die erste konzentriert sich auf das groß angelegte Vortraining von LLMs, das für die Entwicklung hochmoderner Modelle von entscheidender Bedeutung ist. Die zweite konzentriert sich auf Methoden zur Anpassung von LLMs an lange Eingaben, einschließlich der Optimierung von Aufmerksamkeitsmodellen, dem Entwurf effizienterer und komprimierterer KV-Caches, der Einbindung von Speichermodellen und der Erforschung besserer Positionseinbettungen.

Die Stärke von LLMs liegt in ihrer Fähigkeit, die Beschränkungen des Trainings von NLP-Modellen für eine begrenzte Anzahl spezifischer Aufgaben zu durchbrechen. Stattdessen lernen LLMs aus großen Textmengen durch die einfache Aufgabe der Token-Vorhersage — wir sagen das nächste Token in einem Satz voraus, gegeben seine vorherigen Tokens. Eine allgemeine Ansicht ist, dass LLMs durch die sehr häufige Wiederholung dieser Token-Vorhersageaufgabe ein gewisses Wissen über die Welt und die Sprache erwerben können, das dann auf neue Aufgaben angewendet werden kann. Infolgedessen können LLMs dazu veranlasst werden, jede Aufgabe auszuführen, indem man sie als eine Aufgabe der Vorhersage nachfolgender Tokens basierend auf Prompts formuliert. Diese emergente Fähigkeit in Sprachmodellen ergibt sich aus mehreren Dimensionen, wie der Skalierung des Trainings, der Modellgröße und der Kontextgröße. Es ist unbestreitbar, dass Skalierungsgesetze derzeit das grundlegende Prinzip bei der Entwicklung großer Sprachmodelle sind, obwohl eine bloße Vergrößerung der Modellgröße sich noch nicht als ausreichend erwiesen hat, um AGI zu erreichen. Es wurde festgestellt, dass diese kontinuierlich skalierten LLMs Fähigkeiten im allgemeinen Sprachverständnis, in der Generierung und im Reasoning aufweisen. In jüngerer Zeit wurde festgestellt, dass die Skalierung der Rechenleistung zur Inferenzzeit ebenfalls zu signifikanten Verbesserungen bei komplexen Reasoning-Aufgaben führen kann [OpenAI, 2024].

Aufgrund ihrer erstaunlichen Leistungsfähigkeit haben LLMs erhebliches Interesse geweckt, sowohl in Bezug auf Techniken als auch auf Anwendungen. Infolgedessen hat die

explosionsartige Zunahme des Forschungsinteresses an LLMs auch zu einer Vielzahl neuer Techniken und Modelle geführt. Angesichts der rasanten Entwicklung des Feldes versuchen wir jedoch nicht, einen umfassenden Literaturüberblick über alle Aspekte von LLMs zu geben. Dennoch kann man sich Wissen über LLMs aus allgemeinen Übersichtsartikeln [Zhao et al., 2023; Minaee et al., 2024] oder fokussierteren Diskussionen zu spezifischen Themen [Ruan et al., 2024] aneignen.

Kapitel 3

Prompting

Im Kontext von LLMs bezeichnet Prompting die Methode, einem LLM eine spezifische Eingabe oder einen Hinweis zu geben, um eine gewünschte Ausgabe zu erzeugen oder eine Aufgabe auszuführen. Wenn wir beispielsweise möchten, dass das LLM einen Satz vom Englischen ins Chinesische übersetzt, können wir es wie folgt prompten

> Translate the text from English to Chinese.
>
> Text: The early bird catches the worm.
>
> Translation: ____

Prompting ist für LLMs von entscheidender Bedeutung, da es direkt beeinflusst, wie effektiv diese Modelle Benutzeranfragen verstehen und darauf reagieren. Ein gut gestalteter Prompt kann ein LLM dazu anleiten, genauere, relevantere und kontextuell passendere Antworten zu generieren. Darüber hinaus kann dieser Prozess iterativ verfeinert werden. Durch die Analyse der Antworten des LLM können Benutzer ihre Prompts anpassen, um sie besser auf ihre spezifischen Bedürfnisse abzustimmen. Angesichts der Bedeutung des Promptings bei der Anwendung von LLMs ist das Prompt-Design zu einer wesentlichen Fähigkeit für Benutzer und Entwickler geworden, die mit LLMs arbeiten. Dies führt zu einem aktiven Forschungsbereich, dem sogenannten Prompt-Engineering, in dem wir effektive Prompts entwerfen, um LLMs besser zu nutzen und ihre praktische Anwendbarkeit in realen Anwendungen zu verbessern.

Ein wichtiges Konzept im Zusammenhang mit Prompting ist das In-Kontext-Lernen. Beim Prompting eines LLM können wir dem Kontext neue Informationen hinzufügen, wie zum Beispiel Demonstrationen zur Problemlösung. Dies ermöglicht es dem LLM, aus diesem Kontext zu lernen, wie das Problem zu lösen ist. Hier ist ein Beispiel für das Prompting von LLMs mit einigen Demonstrationen, wie man Text basierend auf der Sentiment-Polarität klassifiziert.

> Here are some examples of text classification.
>
> Example 1: We had a delightful dinner together. → Label: Positive
>
> Example 2: I'm frustrated with the delays. → Label: Negative
>
> What is the label for "That comment was quite hurtful."?
>
> Label: ____

In-Kontext-Lernen wird oft als eine emergente Fähigkeit von LLMs angesehen, die nach dem Vortraining entsteht. Obwohl LLMs trainiert oder feinabgestimmt werden können, um neue Aufgaben auszuführen, bietet das In-Kontext-Lernen eine sehr effiziente Möglichkeit, diese Modelle ohne Trainings- oder Abstimmungsaufwand anzupassen. Vielleicht ist dies eine der bemerkenswertesten Eigenschaften von LLMs: Sie lernen während

des Vortrainings tatsächlich allgemeines Wissen über die Welt und die Sprache, das wir leicht auf neue Herausforderungen anwenden können. Darüber hinaus spiegelt das In-Kontext-Lernen den breiteren Trend wider, KI-Systeme allgemeiner anwendbar und benutzerfreundlicher zu gestalten. Anstatt dass spezialisierte Ingenieure Modelle für jede einzelne Aufgabe feinabstimmen müssen, können Benutzer auf eine intuitivere Weise mit LLMs interagieren, indem sie einfach Beispiele bereitstellen oder den Kontext nach Bedarf anpassen.

In diesem Kapitel konzentrieren wir uns auf Prompting-Techniken in LLMs. Wir beginnen mit der Betrachtung mehrerer interessanter Prompt-Designs, die im Prompt-Engineering häufig verwendet werden. Anschließend diskutieren wir eine Reihe von Verfeinerungen dieser Methoden. Schließlich untersuchen wir Ansätze zur Automatisierung des Prompt-Designs.

3.1 Allgemeines Prompt-Design

In diesem Abschnitt werden die Grundlagen des Prompt-Designs sowie Beispiele dafür vorgestellt, wie man Prompts für LLMs für verschiedene NLP-Aufgaben formuliert. Da die Wirksamkeit des Promptings stark von den verwendeten LLMs abhängt, variieren die Prompts oft zwischen verschiedenen LLMs, was es schwierig macht, eine umfassende Liste von Prompts für alle LLMs und nachgeschaltete Aufgaben bereitzustellen. Daher konzentriert sich diese Diskussion nicht auf ein bestimmtes LLM. Stattdessen ist es das Ziel, Leitlinien für das Prompt-Design zu geben.

3.1.1 Grundlagen

Der Begriff Prompt wird auf viele verschiedene Weisen verwendet. In diesem Kapitel definieren wir einen Prompt als den Eingabetext für ein LLM, der mit $\mathbf{x}$ bezeichnet wird. Das LLM generiert einen Text $\mathbf{y}$, indem es die Wahrscheinlichkeit $\Pr(\mathbf{y}|\mathbf{x})$ maximiert. In diesem Generierungsprozess fungiert der Prompt als Bedingung, unter der wir Vorhersagen treffen, und er kann jegliche Informationen enthalten, die helfen, das Problem zu beschreiben und zu lösen.

Ein Prompt kann unter Verwendung einer Prompt-Vorlage (oder kurz Vorlage) [Liu et al., 2023a] erstellt werden. Eine Vorlage ist ein Textstück, das Platzhalter oder Variablen enthält, wobei jeder Platzhalter mit spezifischen Informationen gefüllt werden kann. Hier sind zwei Vorlagen, um das LLM nach Wochenendvorschlägen zu fragen.

> Please give me some suggestions for a fun weekend.
> ___

> If {∗premise∗}, what are your suggestions for a fun weekend.
> ___

In der ersten Vorlage weisen wir das LLM einfach an, einige Vorschläge zurückzugeben. Die Vorlage ist also nur ein Textstück ohne Variablen. In der zweiten Vorlage muss die Variable {∗premise∗} von den Benutzern spezifiziert werden, um eine Prämisse für die Vorschläge zu liefern. Wenn wir zum Beispiel eingeben

$$\text{premise} \quad = \quad \text{the weather is nice this weekend}$$

dann können wir einen Prompt generieren

> If the weather is nice this weekend,
> what are your suggestions for a fun weekend.
> ____

Wir können auch eine Vorlage mit mehreren Variablen entwerfen. Hier ist ein Beispiel, in dem wir die beiden Sätze hinsichtlich ihrer semantischen Ähnlichkeit vergleichen.

> Here is a sentence
> {∗sentence1∗}
> Here is another sentence
> {∗sentence2∗}
> Compute the semantic similarity between the two sentences
> ____

Eine beliebte Methode zur Formatierung von Prompts ist es, jede Eingabe oder Ausgabe im Stil „Name:Inhalt“zu schreiben. Zum Beispiel können wir eine Konversation zwischen zwei Personen, John und David, beschreiben und das LLM verwenden, um die Konversation fortzusetzen. Eine Vorlage für solche Prompts ist gegeben durch

> John: {∗utterance1∗}
> David: {∗utterance2∗}
> John: {∗utterance3∗}
> David: {∗utterance4∗}
> John: {∗utterance5∗}
> David: {∗utterance6∗}
> John: {∗utterance7∗}
> David: ____

Das „Name:Inhalt“-Format kann verwendet werden, um die Aufgabe zu definieren, die das LLM ausführen soll. Da beispielsweise „Q“und „A“gebräuchliche Abkürzungen für „Frage“bzw. „Antwort“sind, können wir die folgende Vorlage für die Beantwortung von Fragen verwenden.

Q: {∗question∗}
A: ____

Dieses Format kann verwendet werden, um komplexere Aufgaben zu beschreiben. Das Folgende ist zum Beispiel ein Beispiel für die Bereitstellung einer Spezifikation für eine Übersetzungsaufgabe

Task: Translation
Source language: English
Target language: Chinese
Style: Formal text
Template: Translate the following sentence: {∗sentence∗}
____

In praktischen Systemen ist es üblich, solche Daten in Schlüssel-Wert-Paaren darzustellen und zu speichern, wie zum Beispiel im JSON-Format[1].

Wenn das Problem schwierig attributbasiert zu beschreiben ist, ist es üblicher, LLMs mit einer klaren und detaillierten Beschreibung anzuweisen. Es gibt viele Möglichkeiten, dies zu tun. Ein Beispiel ist, den LLMs eine Rolle zuzuweisen und ausreichend Kontext bereitzustellen. Das Folgende ist eine Vorlage, die ein LLM anweist, als Experte zu agieren und Fragen von Kindern zu beantworten.

You are a computer scientist with extensive knowledge in the field of deep learning.

Please explain the following computer-related concept to a child around 10 years old, using simple examples whenever possible.

{∗concept∗}

____

Hier wird der Text „You are a computer scientist ... deep learning. ” manchmal als Systeminformation bezeichnet und dient dazu, dem LLM zu helfen, den Kontext oder die Einschränkungen der Aufgabe zu verstehen, die es ausführen soll.

[1] Die JSON-Darstellung lautet

```
{
    Task: Translation”
    SSource language: Ënglish”
    Target language: Chinese”
    SStyle: Formal text”
    Template: Translate the following sentence: {∗sentence∗}”
}
```

3.1.2 In-Kontext-Lernen

Lernen kann während der Inferenz stattfinden. In-Kontext-Lernen ist eine solche Methode, bei der Prompts Demonstrationen zur Problemlösung beinhalten und LLMs aus diesen Demonstrationen lernen können, wie man neue Probleme löst. Da wir in diesem Prozess keine Modellparameter aktualisieren, kann In-Kontext-Lernen als eine Möglichkeit betrachtet werden, das im Vortraining erlernte Wissen effizient zu aktivieren und neu zu organisieren, ohne zusätzliches Training oder Fine-Tuning. Dies ermöglicht eine schnelle Anpassung von LLMs an neue Probleme und erweitert die Grenzen dessen, was vortrainierte LLMs ohne aufgabenspezifische Anpassungen erreichen können.

In-Kontext-Lernen kann durch den Vergleich von drei Methoden veranschaulicht werden: Zero-Shot-Lernen, One-Shot-Lernen und Few-Shot-Lernen. Zero-Shot-Lernen, wie der Name schon sagt, beinhaltet keinen traditionellen „Lernprozess". Stattdessen werden LLMs direkt angewendet, um neue Probleme zu lösen, die während des Trainings nicht beobachtet wurden. In der Praxis können wir Prompts wiederholt anpassen, um die LLMs anzuleiten, bessere Antworten zu generieren, ohne Problemlösungsschritte zu demonstrieren oder Beispiele bereitzustellen. Betrachten wir das folgende Beispiel. Angenommen, wir möchten ein LLM als Assistenten verwenden, der bei der Korrektur englischer Sätze helfen kann. Ein Zero-Shot-Lern-Prompt lautet wie folgt

SYSTEM	You are a helpful assistant, and are great at grammar correction.
USER	You will be provided with a sentence in English. The task is to output the correct sentence. Input: She don't like going to the park. Output: ____

Hier werden die grauen Wörter verwendet, um verschiedene Felder des Prompts anzuzeigen.

Beim One-Shot-Lernen erweitern wir diesen Prompt, indem wir eine Demonstration hinzufügen, wie Sätze korrigiert werden, wodurch das LLM aus dieser neu hinzugefügten Erfahrung lernen kann.

SYSTEM	You are a helpful assistant, and are great at grammar correction.
DEMO	You will be provided with a sentence in English. The task is to output the correct sentence. Input: There is many reasons to celebrate. Output: There are many reasons to celebrate.
USER	You will be provided with a sentence in English. The task is to output the correct sentence. Input: She don't like going to the park. Output: ____

Darüber hinaus können wir weitere Demonstrationen hinzufügen, um Few-Shot-Lernen zu ermöglichen.

SYSTEM	You are a helpful assistant, and are great at grammar correction.
DEMO1	You will be provided with a sentence in English. The task is to output the correct sentence. Input: There is many reasons to celebrate. Output: There are many reasons to celebrate.
DEMO2	You will be provided with a sentence in English. The task is to output the correct sentence. Input: Me and my friend goes to the gym every day. Output: My friend and I go to the gym every day.
USER	You will be provided with a sentence in English. The task is to output the correct sentence. Input: She don't like going to the park. Output: ____

Beim Few-Shot-Lernen stellen wir im Wesentlichen ein Muster bereit, das einige Eingaben den entsprechenden Ausgaben zuordnet. Das LLM versucht, diesem Muster bei der Erstellung von Vorhersagen zu folgen, vorausgesetzt, der Prompt enthält eine ausreichende, wenn auch in der Regel kleine, Anzahl von Demonstrationen. Es ist auch möglich, einfachere Muster zu verwenden, um dies zu erreichen. Beispielsweise kann man den folgenden Few-Shot-Lern-Prompt verwenden, um Wörter aus dem Chinesischen ins Englische zu übersetzen.

DEMO	现在	→	now
	来	→	come
	去	→	go
	男孩	→	boy
USER	女孩	→	____

Wenn das LLM leistungsstark genug ist, kann Few-Shot-Lernen es befähigen, komplexe Probleme wie mathematisches Schließen zu bewältigen. Betrachten wir zum Beispiel die folgende Aufgabe, zwei Zahlen zu summieren und dann die Summe durch ihr Produkt zu teilen.

DEMO	12	5	→	$(12+5)/(12\times 5)=0.283$
	3	1	→	$(3+1)/(3\times 1)=1.33$
	−9	4	→	$(-9+4)/(-9\times 4)=0.138$
	15	15	→	$(15+15)/(15\times 15)=0.133$
USER	19	73	→	____

In vielen praktischen Anwendungen hängt die Wirksamkeit des In-Kontext-Lernens stark von der Qualität der Prompts und den grundlegenden Fähigkeiten der vortrainierten LLMs ab. Einerseits benötigen wir einen erheblichen Aufwand an Prompt-Engineering, um geeignete Prompts zu entwickeln, die den LLMs helfen, effektiver aus Demonstrationen zu lernen. Andererseits können stärkere LLMs das In-Kontext-Lernen besser nutzen, um neue Aufgaben auszuführen. Angenommen, wir möchten ein LLM verwenden, um Wörter aus dem Inuktitut ins Englische zu übersetzen. Wenn dem LLM das Vortraining mit Inuktitut-Daten fehlt, wird sein Verständnis für Inuktitut schwach sein, und es wird für das Modell schwierig sein, bei der Übersetzung gute Leistungen zu erbringen, unabhängig davon, wie wir es prompten. In diesem Fall müssen wir das LLM mit mehr Inuktitut-Daten weiter trainieren, anstatt zu versuchen, bessere Prompts zu finden.

Es könnte interessant sein zu untersuchen, wie In-Kontext-Lernen während des Vortrainings entsteht und warum es während der Inferenz funktioniert. Ein einfaches Verständnis ist, dass LLMs ein gewisses Wissen über die Problemlösung erworben haben, aber es gibt viele mögliche Vorhersagen, die schwer zu unterscheiden sind, wenn die Modelle mit neuen Problemen konfrontiert werden. Die Bereitstellung von Demonstrationen kann die LLMs anleiten, den „richtigen" Pfaden zu folgen. Darüber hinaus haben einige Forscher versucht, In-Kontext-Lernen aus verschiedenen Perspektiven zu interpretieren, darunter Bayessche Inferenz [Xie et al., 2022], Gradientenabstieg [Dai et al., 2023; Von Oswald et al., 2023], lineare Regression [Akyürek et al., 2023], Meta-Lernen [Garg et al., 2022], und so weiter.

3.1.3 Strategien für das Prompt-Engineering

Die Gestaltung von Prompts ist in hohem Maße empirisch. Im Allgemeinen gibt es viele Möglichkeiten, ein LLM zur Ausführung derselben Aufgabe zu instruieren, und wir müssen eine Reihe von Trial-and-Error-Durchläufen durchführen, um einen zufriedenstellenden Prompt zu finden. Um gute Prompts effizienter zu schreiben, kann man bestimmte Strategien befolgen. Beispiele für gängige Prompting-Prinzipien sind unter anderem

- Die Aufgabe so klar wie möglich beschreiben. Wenn wir ein LLM anwenden, um ein Problem zu lösen, müssen wir eine präzise, spezifische und klare Beschreibung des Problems bereitstellen und das LLM anweisen, die Aufgabe wie erwartet auszuführen. Dies ist besonders wichtig, wenn wir möchten, dass die Ausgabe des LLMs bestimmte Erwartungen erfüllt. Nehmen wir zum Beispiel an, wir sind neugierig auf den Klimawandel. Ein einfacher Prompt, um das LLM um einige Informationen zu bitten, ist

  > Tell me about climate change.
  > ——

  Da diese Anweisung zu allgemein ist, kann das LLM eine Antwort generieren, die einen beliebigen Aspekt des Klimawandels behandelt, was möglicherweise nicht mit unseren spezifischen Interessen übereinstimmt. In diesem Fall können wir stattdessen Prompts verwenden, die spezifisch und detailliert sind. Ein solches Beispiel ist

> Provide a detailed explanation of the causes and effects of climate change, including the impact on global temperatures, weather patterns, and sea levels. Also, discuss possible solutions and actions being taken to mitigate these effects.
>
> ——

Nehmen wir nun an, wir möchten einem 10-jährigen Kind den Klimawandel erklären. Wir können den obigen Prompt weiter anpassen.

> Explain the causes and effects of climate change to a 10-year-old child. Talk about how it affects the weather, sea levels, and temperatures. Also, mention some things people are doing to help. Try to explain in simple terms and do not exceed 500 words.
>
> ——

- LLMs zum Nachdenken anleiten. LLMs haben überraschend gute Fähigkeiten zum „Denken“gezeigt. Ein gängiges Beispiel ist, dass gut entwickelte LLMs beeindruckende Leistungen bei Aufgaben des mathematischen Schlussfolgerns erzielt haben, die als anspruchsvoll gelten. Beim Prompt-Engineering muss die „Denk“-Fähigkeit von LLMs durch geeignetes Prompting aktiviert werden, insbesondere bei Problemen, die einen erheblichen Aufwand an Schlussfolgerungen erfordern. In vielen Fällen kann ein LLM, das angewiesen wird zu „denken“, völlig andere Ergebnisse liefern als dasselbe LLM, das angewiesen wird, die Aufgabe direkt auszuführen. Zum Beispiel fanden Kojima et al. [2022] heraus, dass das einfache Anhängen von „Let’s think step by step“an das Ende jedes Prompts die Leistung von LLMs bei mehreren Schlussfolgerungsaufgaben verbessern kann. LLMs können auf verschiedene Weisen zum „Denken“angeregt werden. Eine Methode besteht darin, LLMs anzuweisen, Schritte zum Schlussfolgern über das Problem zu generieren, bevor die endgültige Antwort erreicht wird. Betrachten wir zum Beispiel die Aufgabe, mathematische Probleme zu lösen. Siehe unten für einen einfachen Prompt für diese Aufgabe.

> You are a mathematician. You will be provided with a math problem. Please solve the problem.
>
> ——

Da das Lösen mathematischer Probleme einen detaillierten Schlussfolgerungsprozess erfordert, würden LLMs wahrscheinlich Fehler machen, wenn sie versuchen würden, die Antwort direkt zu erarbeiten. Daher können wir LLMs explizit auffordern, einem gegebenen Schlussfolgerungsprozess zu folgen, bevor sie zu einem Ergebnis kommen.

> You are a mathematician. You will follow these detailed reasoning steps when solving math problems.
>
> Step 1: Problem Interpretation.
> The mathematician carefully listens to your query and understands the intricate details of the mathematical challenge you have presented.
>
> Step 2: Strategy Formulation.
> Drawing upon their extensive knowledge, the mathematician chooses the most effective strategy tailored to the type of math problem, whether it is algebra, calculus, or geometry.
>
> Step 3: Detailed Calculation.
> With precision and expertise, the mathematician performs the necessary calculations step by step, adhering to all mathematical principles.
>
> Step 4: Solution Review.
> Before providing the final answer, the mathematician meticulously checks the calculations for accuracy and offers a concise explanation or rationale for the solution.
>
> You will be provided with a math problem. Please solve the problem.
>
> {∗problem∗}
>
> ——

Eine andere Methode, LLMs zum „Denken“anzuleiten, ist die mehrfache Interaktion mit LLMs. Zum Beispiel können wir als ersten Schritt LLMs anweisen, das Problem direkt zu lösen

> You will be provided with a math problem. Please solve the problem.
>
> {∗problem∗}
>
> ——

Nun haben wir eine erste Antwort auf das Problem. Als zweiten Schritt fordern wir LLMs auf, die Korrektheit der Antwort zu bewerten und sie gegebenenfalls zu überarbeiten, um eine bessere Lösung zu finden.

> You will be provided with a math problem, along with a solution. Evaluate the correctness of this solution, and identify any errors if present. Then, work out your own solution.
>
> Problem: {∗problem∗}
>
> Solution: {∗solution∗}
>
> ——

Die hier vorgestellten Prompts stehen in engem Zusammenhang mit einer langen Forschungsreihe zu Reasoning-Problemen in LLMs. Es ist unmöglich, eine vollständige Diskussion aller damit verbundenen Fragen zu liefern, da dieses Thema eine große Familie von Methoden abdeckt. Aber wir werden in Abschnitt 3.2 eine relativ detailliertere Diskussion darüber sehen, wie man das Prompting durch mehr Schlussfolgern verbessern kann.

- Bereitstellung von Referenzinformationen. Wie im vorherigen Abschnitt besprochen, können wir Beispiele in Prompts einfügen und es LLMs ermöglichen, durch In-Context-Lernen aus diesen Beispielen zu lernen, wie die Aufgabe auszuführen ist. Tatsächlich können wir angesichts der bemerkenswerten Fähigkeit des Sprachverständnisses von LLMs jede Art von Text in die Prompts einfügen, sodass diese Modelle auf der Grundlage angereicherter Kontexte vorhersagen können. In vielen Anwendungen haben wir verschiedene Informationen, die für Benutzeranfragen relevant sind. Anstatt LLMs für uneingeschränkte Vorhersagen zu verwenden, möchten wir oft, dass LLMs Ausgaben produzieren, die auf den relevanten Text beschränkt sind. Ein solches Beispiel ist RAG, bei dem der relevante Text für die Benutzeranfrage durch den Aufruf eines IR-Systems bereitgestellt wird, und wir fordern LLMs auf, Antworten auf der Grundlage dieses bereitgestellten relevanten Textes zu generieren. Der folgende Prompt zeigt ein Beispiel.

> You are an expert that can generate answers to input queries. You have now been provided with a query and the corresponding context information. Please generate an answer based on this context information. Note that you need to provide the answer in your own words, not just copy from the context provided.
>
> Context information: {∗IR-result∗}
>
> Query: {∗query∗}
>
> ——

Wenn die Kontextinformationen sehr zuverlässig sind, können wir LLMs sogar darauf beschränken, nur den bereitgestellten Text zur Beantwortung zu verwenden. Ein Beispiel-Prompt wird wie folgt gezeigt

> You are an expert tasked with generating answers from input queries. You have been provided with a query and corresponding context information, organized in a table where each row represents a useful record. Please generate an answer using only this context information. Ensure that you provide the answer in your own words.
>
> Context information: {∗table∗}
>
> Query: {∗query∗}
>
> ——

Bei der Behandlung von realen Problemen verfügen wir oft über Vorwissen und

zusätzliche Informationen über die Probleme, die helfen, bessere Antworten zu produzieren. Die Berücksichtigung solcher Informationen beim Prompting ist im Allgemeinen hilfreich, um das Ergebnis zu verbessern.

- Auf Prompt-Formate achten. Im Allgemeinen ist die Leistung von LLMs sehr empfindlich gegenüber den von uns eingegebenen Prompts. Manchmal kann eine kleine Änderung an einem Prompt zu einer großen Veränderung in der Modellausgabe führen. Ein interessantes Beispiel ist, dass die Änderung der Reihenfolge von Sätzen in einem Prompt dazu führen kann, dass LLMs unterschiedliche Ergebnisse generieren. Um Prompts leicht lesbar zu machen und Mehrdeutigkeiten zu reduzieren, ist es üblich, sie so zu formatieren, dass Klarheit gewährleistet ist. Ein Beispiel ist, dass wir mehrere Felder für Prompts definieren und in jedes Feld unterschiedliche Informationen eintragen. Ein anderes Beispiel ist, dass wir Prompts im Code-Stil für LLMs verwenden können, die sowohl natürliche Sprache als auch Code verstehen und generieren können. Siehe unten für einen Prompt im Code-Stil, der eine Übersetzung durchführt, bei der ein Beispiel präsentiert wird.

```
[English] = [I have an apple.]
[German] = [Ich habe einen Apfel.]
[English] = [I have an orange.]
[German] = ____
```

LLMs können Text in verschiedenen Formaten empfangen. Dies ermöglicht es uns, Steuerzeichen, XML-Tags und spezifische Formatierungen zu verwenden, um komplexe Daten darzustellen. Und es ist nützlich anzugeben, wie die Eingabe und Ausgabe formatiert oder strukturiert sein sollte. Zum Beispiel können wir Textabschnitte mit Anführungszeichen abgrenzen und LLMs entsprechend anweisen (z. B. indem wir einen Satz wie „der Eingabetext ist durch doppelte Anführungszeichen abgegrenzt “zum Prompt hinzufügen).

Oben haben wir nur einige Strategien zum Schreiben guter Prompts besprochen. Natürlich gibt es viele solcher Methoden, und man muss seine eigenen durch Übung entwickeln. Für weitere Informationen können interessierte Leser auf verschiedene Online-Dokumente zurückgreifen, wie zum Beispiel auf das Handbuch von OpenAI zu den Modellen der GPT-Serie[2].

3.1.4 Weitere Beispiele

In diesem Unterabschnitt betrachten wir weitere Beispiele für das Prompting von LLMs zur Durchführung verschiedener NLP-Aufgaben. Die Motivation hierbei ist nicht, Standard-Prompts für diese Aufgaben zu geben, sondern vielmehr, anhand einfacher Beispiele zu veranschaulichen, wie LLMs gepromptet werden können, um NLP-Probleme zu behandeln.

[2]Siehe https://platform.openai.com/docs/guides/prompt-engineering/six-strategies-for-getting-better-results.

3.1.4.1 Textklassifikation

Die Textklassifikation ist vielleicht eines der häufigsten Probleme im NLP. Viele Aufgaben lassen sich grob als die Zuweisung vordefinierter Label zu einem gegebenen Text kategorisieren. Hier betrachten wir das Problem der Polaritätsklassifikation in der Sentimentanalyse. Wir wählen die Polaritätsklassifikation zur Veranschaulichung, da sie eine der beliebtesten und am besten definierten Textklassifikationsaufgaben ist. In einem allgemeinen Aufbau der Polaritätsklassifikation müssen wir einen gegebenen Text in eine von drei Kategorien einteilen: negativ, positiv oder neutral. Nachfolgend finden Sie einen einfachen Prompt dafür (zur leichteren Lesbarkeit heben wir die Aufgabenbeschreibung im Prompt hervor).

> Analyze the polarity of the following text and classify it as positive, negative, or neutral.
>
> Text:
> The service at the restaurant was slower than expected, which was a bit frustrating.
>
> The polarity of the text can be classified as negative.

Um das Beispiel zu vervollständigen, zeigen wir die vom LLM generierte Antwort (unterstrichener Text).

Obwohl die Antwort korrekt ist, gibt das LLM diese Antwort nicht in Form von Labels, sondern als Text, der das Ergebnis beschreibt. Das Problem ist, dass LLMs darauf ausgelegt sind, Text zu generieren, aber nicht, Label zu Texten zuzuweisen, und Klassifikationsprobleme als Textgenerierungsprobleme behandeln. Daher benötigen wir ein weiteres System, um die Ausgabe des LLM auf den Label-Raum abzubilden (nennen wir es Label-Mapping), das heißt, wir extrahieren „negativ“aus „The polarity of the text can be classified as negative“. Dies ist in den meisten Fällen trivial, da wir Label-Wörter durch einfache Heuristiken identifizieren können. Gelegentlich drücken LLMs die Klassifikationsergebnisse jedoch nicht mit diesen Label-Wörtern aus. In diesem Fall wird das Problem komplizierter, da wir eine Methode benötigen, um den generierten Text oder die Wörter auf vordefinierte Label-Wörter abzubilden.

Eine Methode, um Ausgabe-Label von LLMs zu induzieren, besteht darin, das Problem als Lückentext-Aufgabe neu zu formulieren. Zum Beispiel zeigt das Folgende einen lückentextartigen Prompt für die Polaritätsklassifikation.

> Analyze the polarity of the following text and classify it as positive, negative, or neutral.
>
> Text:
> The service at the restaurant was slower than expected, which was a bit frustrating.
>
> The polarity of the text is negative

Wir können LLMs verwenden, um den Text zu vervollständigen und die Lücke mit dem am besten geeigneten Wort zu füllen. Idealerweise wünschen wir uns, dass das eingefügte Wort positive, negative oder neutral ist. Es ist jedoch nicht garantiert, dass LLMs diese Label-Wörter generieren. Eine Methode, um dieses Problem zu lösen, besteht darin, die Vorhersage auf den Satz von Label-Wörtern zu beschränken und dasjenige mit der höchsten Wahrscheinlichkeit auszuwählen. Dann wird das Ausgabe-Label gegeben durch

$$\text{Label} \quad = \quad \arg\max_{y \in Y} \Pr(y|\mathbf{x}) \tag{3.1}$$

wobei y das in die Lücke eingefügte Wort bezeichnet und Y den Satz der Label-Wörter $\{\text{positive}, \text{negative}, \text{neutral}\}$ bezeichnet.

Eine weitere Methode, LLMs zur Generierung von Labels zu verwenden, besteht darin, die Ausgabe mit Prompts einzuschränken. Zum Beispiel können wir LLMs anweisen, innerhalb eines kontrollierten Satzes von Wörtern Vorhersagen zu treffen. Hier ist ein Beispiel.

Analyze the polarity of the following text and classify it as positive, negative, or neutral.

Text:
The service at the restaurant was slower than expected, which was a bit frustrating.

What is the polarity of the text?

Just answer: positive, negative, or neutral.

Negative

Die Sentimentanalyse ist ein häufiges NLP-Problem, das von LLMs wahrscheinlich durch Vortraining oder Fine-Tuning gut verstanden wurde. Daher können wir LLMs mit einfachen Anweisungen auffordern, die Aufgabe auszuführen. Bei neuen Klassifikationsproblemen kann es jedoch erforderlich sein, zusätzliche Details zur Aufgabe anzugeben, wie z. B. die Klassifikationsstandards, damit die LLMs korrekt arbeiten können. Dazu können wir eine detailliertere Beschreibung der Aufgabe hinzufügen und/oder Klassifikationsbeispiele in den Prompts demonstrieren. Zur Veranschaulichung betrachten Sie das folgende Beispiel.

> Analyze the polarity of the following text and classify it as positive, negative, or neutral. Here's what each category represents:
>
> Positive: This indicates that the text conveys a positive emotion or attitude. For example, texts expressing happiness, satisfaction, excitement, or admiration are considered positive.
>
> Negative: This refers to a text that expresses a negative emotion or attitude. It encompasses feelings of sadness, anger, frustration, or criticism.
>
> Neutral: Neutral sentiment is used to describe texts that do not exhibit clear positive or negative emotions but instead convey informational, factual, or indifferent tones.
>
> Text:
> The service at the restaurant was slower than expected, which was a bit frustrating.
>
> What is the polarity of the text?
>
> Negative

Obwohl es unkompliziert erscheint, LLMs für Klassifikationsprobleme zu verwenden, gibt es immer noch Probleme, die nicht gut gelöst wurden. Zum Beispiel bleibt es bei einer großen Anzahl von Kategorien eine Herausforderung, LLMs effektiv zu prompten. Beachten Sie, dass bei einem sehr schwierigen Klassifikationsproblem und einer gewissen Menge an gelabelten Daten auch ein Fine-Tuning von LLMs oder die Übernahme von Architekturen wie „BERT + Klassifikator"wünschenswert ist.

3.1.4.2 Informationsextraktion

Viele NLP-Probleme können als Probleme der Informationsextraktion betrachtet werden, bei denen es um die Identifizierung oder Extraktion spezifischer Informationen aus unstrukturiertem Text geht. Diese Informationen können benannte Entitäten, Beziehungen, Ereignisse und andere relevante Datenpunkte umfassen. Das Ziel der Informationsextraktion ist es, Rohdaten in ein Format umzuwandeln, das leicht analysiert und in verschiedenen nachgelagerten Anwendungen verwendet werden kann.

Da die Informationsextraktion ein breites Spektrum an Problemen abdeckt, können wir sie hier nicht alle diskutieren. Stattdessen beginnen wir mit der Aufgabe der Eigennamenerkennung (Named Entity Recognition) –einer Aufgabe, die seit langem ein zentrales Thema im NLP ist. Die Eigennamenerkennung ist ein Prozess, der Schlüsselinformationen in Texten erkennt und in bestimmte Gruppen klassifiziert. Diese Schlüsselinformationen, bekannt als benannte Entitäten, umfassen typischerweise Eigennamen und werden in verschiedene Klassen wie Personen, Orte, Organisationen, Daten, Geldbeträge und Prozentsätze kategorisiert. Betrachten wir zunächst ein einfaches Beispiel für die Extraktion von Personennamen aus einem Text[3].

[3]Der Text stammt von https://edition.cnn.com/travel

Identify all person names in the provided text.

Text:
Is the UK really doing that badly or have travel trends shifted, post-pandemic? For Tom Jenkins, CEO of the European Tourism Organisation, it's the latter. "I think the UK is doing perfectly well but we'll see more people going to Europe," he says of 2024, adding that there's "allegedly a slight plateau of American demand for the UK."

The person name identified in the provided text is: Tom Jenkins

Anschließend können wir alle benannten Entitäten extrahieren.

Identify and classify all named entities in the provided text into categories such as person names, locations, dates, and organizations. List each entity with its type on one line.

Text:
Is the UK really doing that badly or have travel trends shifted, post-pandemic? For Tom Jenkins, CEO of the European Tourism Organisation, it's the latter. "I think the UK is doing perfectly well but we'll see more people going to Europe," he says of 2024, adding that there's "allegedly a slight plateau of American demand for the UK."

1. Tom Jenkins - Person Name
2. European Tourism Organisation - Organization
3. UK - Location
4. Europe - Location
5. 2024 - Date

Auf der Grundlage dieser identifizierten benannten Entitäten können wir den Text mit anderen Aufgaben der Informationsextraktion weiterverarbeiten. Zum Beispiel können wir die Beziehungen zwischen diesen benannten Entitäten identifizieren (dies wird als relation extraction bezeichnet). Der entsprechende Prompt wird wie folgt gezeigt.

> Given a text and a list of named entities identified within it, analyze and describe the relationships between these entities. Explain how each entity is contextually related to others.
>
> Text:
> Is the UK really doing that badly or have travel trends shifted, post-pandemic? For Tom Jenkins, CEO of the European Tourism Organisation, it's the latter. "I think the UK is doing perfectly well but we'll see more people going to Europe," he says of 2024, adding that there's "allegedly a slight plateau of American demand for the UK."
>
> Named Entity:
> Tom Jenkins, European Tourism Organisation, UK, Europe, 2024
>
> Tom Jenkins - European Tourism Organisation: Tom Jenkins is the CEO of the European Tourism Organisation. This indicates a professional association where Tom Jenkins likely has a significant role in directing the organization's activities, especially those related to tourism in Europe.
>
> ...

Wenn LLMs auf das Befolgen von Anweisungen (instruction following) für Aufgaben der Informationsextraktion feinabgestimmt wurden, ist es relativ einfach, verschiedene Aufgaben der Informationsextraktion durchzuführen. Zum Beispiel ist das Folgende eine Prompt-Vorlage für die Informationsextraktion.

> You will be provided with a text. Your task is to {∗task-description∗}
>
> Text: {∗text∗}
>
> ———

Hierbei ist task-description die Anweisung für die Aufgabe. Beispiele für die Anweisung sind

> extract the keywords from the text.
>
> extract the key events from the text.
>
> detect the coreference and anaphoric links between entities from the text.

und so weiter. Selbstverständlich können wir eine detailliertere Beschreibung der Aufgabe bereitstellen, um LLMs genauer zu prompten.

3.1.4.3 Textgenerierung

Im Bereich der Verarbeitung natürlicher Sprache (NLP) wird der Begriff Textgenerierung weit gefasst, um eine breite Palette verschiedener Probleme abzudecken, von denen sich

viele auf die Generierung von Text auf der Grundlage eines gegebenen Textes konzentrieren. Diese Probleme lassen sich grob in zwei Klassen einteilen: 1) das fortlaufende Schreiben auf der Grundlage des Eingabetextes (nennen wir es Textvervollständigung) und 2) die Umwandlung des Eingabetextes in einen anderen Text (nennen wir es Texttransformation). Hier illustrieren wir die Prompts für Ersteres und betrachten die Prompts für Letzteres in der folgenden Diskussion.

Ein einfaches Beispiel für die Textvervollständigung ist das Weiterschreiben einer Geschichte, die mit einigen vorgegebenen Sätzen beginnt.

> You are a writer who is great at writing stories. You are provided with a few sentences to start with. Please continue the story from the provided sentences.
>
> Start of the story:
> One sunny afternoon, Lily discovered a tiny, golden key hidden under a rock in her backyard. Curious, she picked it up and wondered what it might unlock. Just then, her cat, Whiskers, began to meow excitedly, leading her towards the old oak tree at the edge of the garden.
>
> Lily followed Whiskers, her heart pounding with excitement and a touch of nervousness. The old oak tree had always seemed mysterious to her, its gnarled branches casting long shadows even on the brightest days. As they reached the tree, Whiskers darted ahead and started pawing at something buried beneath a pile of autumn leaves ...

Ebenso können wir eine Konversation erstellen, wenn der Gesprächskontext gegeben ist.

> You are provided with a conversation between a tourist and a taxi driver. Please complete the conversation.
>
> Tourist: Could you take me to the downtown museum, please?
>
> Taxi Driver: Of course! First time in the city?
>
> Tourist: Yes, any must-see places around?
>
> Taxi Driver: Definitely check out the botanical gardens after the museum. They're nearby and quite beautiful.
>
> Tourist: Great, thanks for the tip!
>
> Taxi Driver: No problem. Enjoy your visit!

Eine weitere Anwendung der Textvervollständigung ist die Generierung von Text auf der Grundlage von Benutzeranforderungen. Die Anforderungen können Stil, Ton, Länge und jeden spezifischen Inhalt umfassen, den der Text enthalten sollte. Zum Beispiel benötigen Benutzer möglicherweise einen formellen Bericht, eine kreative Geschichte oder ein Stück Programmiercode. Hier ist ein Prompt zur Generierung eines regulierten chinesischen Gedichts mit fünf Zeichen.

You are a poet. Please write a traditional Chinese poem. Here is the requirement.

Theme: Spring's rejuvenation

Structure: Five-character regulated poem

Emotion: Convey a sense of happiness and renewal through vivid spring imagery.

春意盎然

翠柳风中舞,
红花雨后新。
山明水又绿,
天宽鸟自频。

Wenn das LLM darauf trainiert ist, Sprache und Code zu generieren, können wir es auffordern, Aufgaben zur Code-Vervollständigung auszuführen. Hier ist ein Beispiel.

Please write a Python function to calculate the average of a list of numbers.

```
def calculate_average(numbers):
    if numbers:    # Check if the list is not empty
    return sum(numbers) / len(numbers)
    else:
    return 0    # Return 0 if the list is empty
```

3.1.4.4 Texttransformation

Typische Aufgaben der Texttransformation umfassen maschinelle Übersetzung, Zusammenfassung, Text-Stiltransfer und so weiter. Die zur Durchführung dieser Aufgaben verwendeten Prompts unterscheiden sich nicht wesentlich von den zuvor vorgestellten. Hier präsentieren wir einige Beispiele, um diese Prompts zu veranschaulichen.

Translate the following text from English to Spanish.

Text:

The quick brown fox jumps over the lazy dog.

El rápido zorro marrón salta sobre el perro perezoso.

Summarize the following article in no more than 50 words:

Article:

In recent years, urban areas have been facing significant challenges related to sustainability and environmental management. Cities are experiencing higher rates of pollution, increased traffic congestion, and greater demands on infrastructure. This growth has led to numerous environmental issues, including elevated levels of air and water pollution, increased waste production, and strained public services ...

Urban areas are grappling with sustainability challenges, such as rising pollution, traffic congestion, and infrastructure demands ...

Rewrite this text in a formal tone.

Text:

Hey, what's up? Long time no see!

Hello, how have you been? It has been quite some time since we last met!

3.1.4.5 Beantwortung von Fragen

Das Format der Fragebeantwortung ist von Natur aus einfach. Zu einer gegebenen Frage gibt es eine entsprechende Antwort. Zum Beispiel erwarten wir beim Open-Domain-Fragebeantworten, dass das System eine Antwort auf eine vom Benutzer gestellte Frage zurückgibt. Prompt-Vorlagen für die allgemeine Fragebeantwortung können sein:

{∗question∗}

____

Question: {∗question∗}

Answer: ____

Die Beantwortung von Fragen ist in der natürlichen Sprachverarbeitung (NLP) wichtig, da viele Probleme als Aufgaben zur Beantwortung von Fragen formuliert werden können. Insbesondere werden viele neuere Schlussfolgerungsaufgaben in Form der Fragebeantwortung definiert. Zum Beispiel besteht im MMLU-Benchmark [Hendrycks et al., 2021] jedes Beispiel aus einer Multiple-Choice-Frage, und LLMs müssen die richtige Antwort auswählen. Siehe unten für ein Beispiel-Prompt zur Beantwortung einer Frage in diesem Datensatz.

$(Z, *)$ is a group with $a * b = a + b + 1$ for all a, b in Z. The inverse of a is

(A) 0

(B) -2

(C) $a - 2$

(D) $(2 + a) * -1$

D

Ein weiterer weit verbreiteter Benchmark ist der GSM8K-Datensatz [Cobbe et al., 2021]. Er besteht aus Tausenden von mathematischen Textaufgaben für die Grundschule. Für jedes Problem verwenden wir LLMs, um eine Lösung in natürlicher Sprache zurückzugeben. Zum Beispiel ist das Folgende ein Prompt, das zur Lösung eines Problems in GSM8K verwendet wird.

DEMO Q: Jerry's two daughters play softball on different teams. They each have 8 games this season. Each team practices 4 hours for every game they play. If each game lasts for 2 hours, how many hours will Jerry spend at the field watching his daughters play and practice altogether?

A: Jerry will spend 8 games * 2 hours per game $=\ll 8*2=16\gg$ 16 hours watching one daughter play her games. He will spend $16*2=\ll 16*2=32\gg$ 32 hours watching both daughters play their games. He will spend 8 games * 4 hours of practice $=\ll 8*4=32\gg$ 32 hours watching one daughter practice. He will spend $32*2=\ll 32*2=64\gg$ 64 hours watching both daughters practice. He will spend a total of 32 hours watching games + 64 hours watching practice $=\ll 32+64=96\gg$ 96 hours. #### 96

Q: Mary bought six apples from the store. From the apples she bought, for each that Mary ate, she planted two trees from the remaining ones. How many apples did Mary eat?

A: She planted eight trees. This means she used half of that amount of apples, which is 8 trees /2 trees/apple $=\ll 8/2=4\gg$ 4 apples. That means that she planted four of the six apples she bought, leaving only 6 apples − 4 apples $=\ll 6-4=2\gg$ 2 apples to be eaten. #### 2

USER Q: Boris has 100 apples. Beck has 23 fewer apples than Boris. If Boris gives Beck 10 apples, how many fewer apples does Beck have than Boris now?

A: Boris starts with 100 apples and gives Beck 10 apples, leaving him with 100 apples −10 apples $=\ll 100-10=90\gg$ 90 apples. Initially, Beck has 23 fewer apples than Boris, giving him a starting count of $100-23=\ll 100-23=77\gg$ 77 apples. After receiving 10 apples from Boris, Beck's total comes to 77 apples +10 apples $=\ll 77+10=87\gg$ 87 apples. Consequently, Beck now has $90-87=\ll 90-87=3\gg$ 3 fewer apples than Boris. #### 3

Hier wird ein Few-Shot-Prompt verwendet. Das LLM lernt aus diesen Demonstrationen von Problem-Lösungs-Paaren nicht nur die Art der Problemlösung, sondern auch die Art der Formatierung der Ausgabe. Zum Beispiel folgt das Endergebnis der Berechnung dem #### Token, und ≪ ... ≫ kommentiert die detaillierten Berechnungsschritte (genannt Berechnungsanmerkung)[4].

[4]Während der Vorhersage wird ein Taschenrechner verwendet, wenn wir ≪ ... ≫ sehen. Genauer gesagt, sobald das LLM auf „=" in einem ≪ ... ≫ stößt, berechnet der Taschenrechner den Ausdruck auf der linken Seite von „=". Diese Methode hilft, die Berechnungsfehler von LLMs zu reduzieren.

3.2 Fortgeschrittene Prompting-Methoden

Bisher haben wir in diesem Kapitel die grundlegenden Konzepte des LLM-Promptings eingeführt und eine Reihe von Prompts für NLP-Aufgaben vorgestellt. Nun betrachten wir verschiedene Techniken zur Steigerung der Effektivität des Promptings.

3.2.1 Gedankenkette

Wir sind dem Konzept der Gedankenkette (CoT) in diesem und früheren Kapiteln bereits mehrfach begegnet [Wei et al., 2022c; Chowdhery et al., 2022]. CoT-Methoden bieten eine einfache Möglichkeit, LLMs dazu zu veranlassen, schrittweise Schlussfolgerungen für komplexe Probleme zu generieren und so Aufgaben auf eine menschenähnlichere Weise anzugehen. Anstatt direkt zu einer Schlussfolgerung zu gelangen, weisen die CoT-Methoden LLMs an, Argumentationsschritte zu generieren oder aus Demonstrationen detaillierter Argumentationsprozesse zu lernen, die in den Prompts bereitgestellt werden. Um CoT zu veranschaulichen, betrachten wir das Problem der algebraischen Berechnung, wie es in der Literatur häufig beschrieben wird. Angenommen, wir erhalten ein algebraisches Problem

Calculate the average of the numbers 2, 4, and 6.

Wir können dies als Frage betrachten und ein LLM auffordern, sie zu beantworten.

> Q: Please calculate the average of the numbers 2, 4, and 9.
> A: The answer is 6.

Es scheint für das LLM schwierig zu sein, direkt eine korrekte Antwort zu geben. Eine einfache Verbesserung besteht darin, Demonstrationen ähnlicher Probleme in den Prompt aufzunehmen, sodass das LLM aus diesen Demonstrationen lernen kann.

> Q: Please calculate the average of the numbers 1, 3, 5, and 7.
> A: The answer is 4.
>
> Q: Please calculate the average of the numbers 2, 4, and 9.
> A: The answer is 7.

Das Problem hierbei ist, dass es für das LLM schwierig bleibt, die richtige Antwort herzuleiten, obwohl wir ein ähnliches Frage-Antwort-Paar gezeigt haben. Bei CoT können LLMs nicht nur aus der Korrespondenz zwischen Fragen und Antworten lernen, sondern sie können auch mehr von detaillierten Problemlösungsschritten profitieren, die zur Ableitung der Antworten verwendet werden. Dazu können wir einige Argumentationsschritte in den Prompt integrieren, um einen CoT-Prompt zu erhalten.

Q: Please calculate the mean square of the numbers 1, 3, 5, and 7.

A: Calculate the square of each number: $1^2 = 1$, $3^2 = 9$, $5^2 = 25$, and $7^2 = 49$. Sum the squares, $1 + 9 + 25 + 49 = 84$. There are 4 numbers in total. Divide the sum by the number of items, $84/4 = 21$. The answer is 21.

Q: Please calculate the average of the numbers 2, 4, and 9.

A: Calculate $2 + 4 + 9$, which equals 15. There are three numbers. Divide the total sum by the count, resulting in $15/3 = 5$. The answer is 5.

Hier heben wir die Argumentationsschritte grün hervor. Indem ein detaillierter Argumentationsprozess für ein ähnliches Problem bereitgestellt wird, lernt das LLM zu schlussfolgern und generiert dadurch einen Problemlösungspfad, der zur richtigen Antwort führt.

Die Verwendung von CoT-Prompting hat mehrere Vorteile. Erstens ermöglicht CoT den LLMs, komplexe Probleme in kleinere, sequenzielle Argumentationsschritte zu zerlegen. Dies spiegelt gewissermaßen menschliches Problemlösungsverhalten wider, was es besonders effektiv für Aufgaben macht, die detaillierte, mehrstufige Schlussfolgerungen erfordern. Zweitens macht CoT den Argumentationsprozess transparenter und interpretierbarer. Da alle Argumentationsschritte sichtbar sind, können wir verstehen und interpretieren, wie eine Schlussfolgerung erreicht wurde. Drittens, wenn Benutzer die Logik hinter dem Argumentationsprozess nachvollziehen können, werden sie den Vorhersagen eines LLM eher vertrauen. Dies ist besonders wichtig bei der Anwendung von LLMs in Bereichen wie Medizin, Bildung und Finanzen. Viertens ist CoT ein Ansatz des In-Kontext-Lernens und somit auf die meisten gut trainierten, handelsüblichen LLMs anwendbar. Darüber hinaus bietet CoT effiziente Möglichkeiten, LLMs an verschiedene Arten von Problemen anzupassen. Es kann sogar zu kreativeren Lösungen inspirieren, indem es verschiedene alternative Argumentationspfade erkundet, die bei einer direkten Schlussfolgerung möglicherweise nicht offensichtlich sind.

Die oben beschriebene Methode erfordert die Bereitstellung eines oder mehrerer Beispiele für CoT-Argumentation, was typischerweise als Few-Shot-CoT-Methode bezeichnet wird. Im Gegensatz dazu erfordert die Zero-Shot-CoT-Methode keine solchen Beispiele. Stattdessen veranlasst sie LLMs zu schrittweisen Schlussfolgerungen, indem spezifische Anweisungen in die Prompts integriert werden. Unten sehen Sie beispielsweise einen Zero-Shot-CoT-Prompt.

Q: Please calculate the average of the numbers 2, 4, and 9.

A: Let's think step-by-step.

We have three numbers: 2, 4, and 9. Add these numbers together, $2 + 4 + 9 = 15$. Determine how many numbers there are, which in this case is three. The average is calculated by dividing the total sum by the number of elements. Completing the division gives $15/3 = 5$. So the answer is 5.

Der Anweisung „Let's think step by step"folgend wird das LLM aufgefordert, detaillierte Argumentationsschritte zu generieren. Wie in der Arbeit von Kojima et al. [2022] erörtert, kann das Prompting mit solchen Anweisungen dazu führen, dass LLMs nur die Argumentationsschritte ohne eine klare Schlussfolgerung generieren. In diesem Fall kann eine zweite Runde des Promptings verwendet werden, um die Antwort aus diesen Argumentationsschritten zu extrahieren. Beispielsweise erstellen Kojima et al. [2022] einen zweiten Prompt, der sowohl die Eingabe als auch die Ausgabe der ersten Prompting-Runde kombiniert. Mithilfe dieser kombinierten Eingabe kann das LLM seinen Argumentationsprozess fortsetzen und dann die richtige Antwort generieren. Darüber hinaus ist es möglich, LLMs mit anderen Anweisungen als „Let's think step by step"zum Schlussfolgern aufzufordern, wie z. B. „Let's think logically"und „Please show me your thinking steps first ".

Obwohl wir CoT-Methoden anhand eines algebraischen Argumentationsproblems veranschaulicht haben, können diese Methoden auf eine Vielzahl unterschiedlicher Probleme angewendet werden. Typische Problemlösungsszenarien für CoT umfassen mathematisches Schließen, logisches Schließen, Alltagsverständnis, symbolisches Schließen, Codegenerierung und so weiter. Siehe Abbildung 3.1 für weitere Beispiele zur Anwendung von CoT in verschiedenen Aufgaben.

CoT ist heute eines der aktivsten Gebiete des Prompt-Engineerings. Dies hat nicht nur zu einer verbesserten Leistung beim LLM-Prompting geführt, sondern auch die Tür zu einer breiten Palette von Methoden zur Untersuchung und Überprüfung der Schlussfolgerungsfähigkeiten von LLMs geöffnet. Obwohl wir uns in diesem Abschnitt auf die Grundidee von CoT konzentriert haben, kann sie auf verschiedene Weisen verbessert werden. Beispielsweise können wir den Argumentationsprozess als ein Problem der Suche durch viele mögliche Pfade betrachten, von denen jeder aus mehreren Zwischenzuständen (d. h. Argumentationsschritten) bestehen kann. Im Allgemeinen wünschen wir uns, dass der Suchraum gut definiert und ausreichend groß ist, damit wir mit größerer Wahrscheinlichkeit das optimale Ergebnis finden. Aus diesem Grund zielt ein Bereich der aktuellen LLM-Forschung darauf ab, bessere Strukturen zur Darstellung von Argumentationsprozessen zu entwerfen, die es LLMs ermöglichen, komplexere Schlussfolgerungsherausforderungen zu bewältigen. Diese Strukturen umfassen baumbasierte Strukturen [Yao et al., 2024], graphbasierte Strukturen [Besta et al., 2024] und so weiter. Durch die Verwendung dieser kompakten Darstellungen von Argumentationspfaden können LLMs eine breitere Palette von Entscheidungspfaden erkunden, analog zum System-2-Denken[5]. Ein weiterer Forschungszweig konzentriert sich auf das Prompting von LLMs mit mehrrundigen Interaktionen. Dies beinhaltet die Zerlegung komplexer Probleme in Teilprobleme, die Überprüfung und Verfeinerung von Modellausgaben, den Einsatz von Modell-Ensembling und so weiter. Beachten Sie, dass diese Methoden und die damit verbundenen Probleme nicht auf CoT beschränkt sind. Tatsächlich werden sie oft als allgemeinere Ansätze zur Verbesserung von LLMs verwendet, während CoT als eine Möglichkeit angesehen werden kann, die Fähigkeiten von LLMs zu testen. Wir werden einige dieser Themen in den folgenden Unterabschnitten erörtern.

[5]System-1- und System-2-Denken, wie von Kahneman [2011] beschrieben, stellen zwei verschiedene Modi der kognitiven Verarbeitung dar. System 1 ist schnell, automatisch, intuitiv und emotional. Diese Denkweise funktioniert mühelos und schnell und leitet oft unsere täglichen Entscheidungen, Urteile und Eindrücke. System 2 ist langsam, überlegt und analytisch. Es wird aktiviert, wenn wir komplexe Berechnungen durchführen müssen.

Q: What home entertainment equipment requires cable?
Answer Choices: (a) radio shack (b) substation (c) television (d) cabinet
A: The answer must require cable. Of the above choices, only television requires cable. So the answer is (c).

Q: What type of person typically contracts illness?
Answer Choices: (a) hospital (b) head (c) sick person (d) elderly person (e) doctor's office
A: The answer must be a person susceptible to illness. Of the choices, ëlderly personïs correct because they typically have weaker immune systems.
So the answer is (d).

(a) CSQA

Q: Do hamsters provide food for any animals?
A: Hamsters are prey animals. Prey are food for predators. Thus, hamsters provide food for some animals. So the answer is yes.

Q: Can you use Microsoft Office without internet?
A: Yes, you can use Microsoft Office applications , although features like cloud storage and online collaboration require internet. So the answer is yes.

(b) StrategyQA

Q: Complete the rest of the sequence, making sure that the parentheses are closed properly. Input: [{
A: Let's think step by step.
0: empty stack
1: [; stack: [
2: { ; stack: [{
So the answer is }].

Q: Complete the rest of the sequence, making sure that the parentheses are closed properly. Input: < [[
A: Let's think step by step.
0: empty stack
1: < ; stack: <
2: [; stack: < [
3: [; stack: < [[
So the answer is]] >.

(c) Dyck languages

Q: Take the last letters of the words in Ëlon Muskänd concatenate them.
A: The last letter of Elon is n. The last letter of Musk is k. Concatenating them is nk. The answer is nk.

Q: Take the last letters of each word in Renata Mariela Mona Kristin and concatenate them.
A: The last letter of Renata is a. The last letter of Mariela is a. The last letter of Mona is a. The last letter of Kristin is n. Concatenating them is aaan. The answer is aaan.

(d) Last Letter Concatenation

Abbildung 3.1: CoT in vier verschiedenen Reasoning-Aufgaben, darunter CSQA, StrategyQA, Dyck-Sprachen und die Verkettung des letzten Buchstabens. Die CoT-Teile sind grün hervorgehoben.

Bevor wir unsere Diskussion über CoT abschließen, sollten wir ihre praktischen Einschränkungen betrachten. Eine davon ist die Notwendigkeit detaillierter, mehrstufiger Argumentationsdemonstrationen in Few-Shot-CoT-Szenarien, die sowohl automatisch als auch manuell schwer zu beschaffen sein können. Außerdem gibt es keine Standardmethode, um komplexe Probleme in einfachere Problemlösungsschritte zu zerlegen. Dies hängt oft stark von der Erfahrung des Benutzers ab. Darüber hinaus können Fehler in den Zwischenschritten auch die Genauigkeit der endgültigen Schlussfolgerung beeinträchtigen. Für eine weiterführende Diskussion über die Vor- und Nachteile von CoT wird der interessierte

Leser auf aktuelle Übersichtsarbeiten zu diesem Thema verwiesen [Chu et al., 2023; Yu et al., 2023; Zhang et al., 2023a].

3.2.2 Problemdekomposition

Wir haben gesehen, dass LLMs davon profitieren können, ein komplexes Problem zu lösen, indem sie es in einfachere Problemlösungsaufgaben zerlegen. Ein solcher Ansatz kann als Beispiel für ein breiteres Paradigma angesehen werden, das als Problemdekomposition bekannt ist und in der Psychologie und Informatik ausgiebig erforscht und diskutiert wurde. Aus psychologischer Sicht bezieht sich komplexe Problemlösung auf einen Prozess, bei dem ein Problem mithilfe von Wissen angegangen wird, das hilft, die Barrieren des Problems zu überwinden[6]. Es gibt im Allgemeinen keine standardmäßigen oder klaren Lösungswege für ein komplexes Problem. Es ist jedoch oft vorteilhaft, Strategien anzuwenden, die das Problem zerlegen, wodurch es einfacher wird, die entsprechenden Teilprobleme mit weniger Aufwand zu bewältigen. Betrachten wir zum Beispiel das Schreiben eines Blogs über die Risiken von KI. Wenn wir einem LLM einfach die Anweisung „Bitte schreiben Sie einen Blog über die Risiken von KI“geben, könnte das LLM einen Blog mit beliebigen Strukturen und Schreibstilen generieren. Eine bessere Methode könnte stattdessen darin bestehen, den Blog zu gliedern und detailliertere Informationen zu jedem Abschnitt bereitzustellen. Betrachten Sie den folgenden Prompt

[6] Eine relativ formale Definition findet sich in Frensch and Funke [2014]s Buch: komplexe Problemlösung tritt auf, um Barrieren zwischen einem gegebenen Zustand und einem gewünschten Zielzustand mittels verhaltensbezogener und/oder kognitiver, mehrstufiger Aktivitäten zu überwinden.

> You are a blog writer. Please follow the provided outline below to write a blog about the risks of AI.
>
> - Introduction
> Introduce AI, its relevance, and the importance of understanding its risks for youth.
> - Privacy Concerns
> Discuss how AI might compromise personal privacy through interactions online.
> - Misinformation
> Explore AI's role in spreading misinformation and influencing young people's decisions.
> - Cyberbullying
> Highlight how AI tools can be utilized in cyberbullying and the impact on mental health.
> - Tips for Safe AI Use
> Offer guidelines for responsible AI usage and promote critical thinking.
> - Conclusion
> Recap main points and encourage proactive engagement with AI ethics.
>
> ——

Hier geben wir den Titel und die Hauptpunkte für jeden Abschnitt an. Dann kann das LLM diese Struktur nutzen, um die Schreibaufgabe zu zerlegen, indem es Inhalte für diese Abschnitte ausfüllt. Beachten Sie, dass die Art und Weise, den Blog zu strukturieren, von Menschen vorgegeben oder sogar automatisch generiert werden kann. Zum Beispiel können wir das LLM verwenden, um zuerst die Gliederung zu erstellen und es dann bitten, dieser Gliederung zu folgen, um das Schreiben abzuschließen.

In der Informatik ist die Zerlegung komplexer Probleme eine häufig verwendete Strategie im Design von Software- und Hardwaresystemen. Ein bekanntes Beispiel ist das Divide-and-Conquer-Paradigma, das oft verwendet wird, um Algorithmen für Berechnungsprobleme zu entwerfen, die auf einfachere, besser handhabbare Probleme reduziert werden können. Betrachten wir zum Beispiel das Problem, festzustellen, ob ein Dokument die Risiken von KI behandelt. Wir können das LLM mit dem folgenden Prompt anweisen.

> You are provided with a text. Please determine whether it discusses the risks of AI.
>
> {∗document∗}
>
> ——

Wenn das Dokument lang ist, wird die Berechnung aufwendig sein. Alternativ können wir das Dokument in relativ kurze Segmente unterteilen und dieselbe Aufgabe für jedes Segment ausführen. Diese Segmente können parallel verarbeitet werden, um die Berechnungskosten weiter zu reduzieren. Als Nächstes bestimmen wir die Relevanz jedes Segments für das Thema der KI-Risiken. Die endgültige Ausgabe wird dann mit einem anderen Prompt generiert.

> Your task is to determine whether a text discusses the risks of AI. This text has been divided into segments, and you have obtained the relevancy of each segment to the topic of AI risks. Based on this, please provide your final result.
>
> Segment 1: {∗relevancy-to-the-topic1∗}
>
> Segment 2: {∗relevancy-to-the-topic2∗}
>
> Segment 3: {∗relevancy-to-the-topic3∗}
>
> ...
>
> ____

Kehren wir nun zu einer allgemeineren Diskussion der Problemdekomposition im Prompting zurück. Obwohl die Problemdekomposition auf verschiedene NLP-Probleme angewendet werden kann, wurde sie in letzter Zeit ausführlicher bei Reasoning-Aufgaben diskutiert und getestet. Bei komplexen Reasoning-Aufgaben benötigen wir oft einen mehrstufigen Reasoning-Pfad, um zu einer korrekten Schlussfolgerung zu gelangen. Wir können LLMs auf drei verschiedene Weisen nutzen, um dies zu erreichen. Erstens können LLMs direkt zur Schlussfolgerung gelangen. Mit anderen Worten, sie können ohne explizite Reasoning-Prozesse vorhersagen, und es gibt einen verborgenen und nicht interpretierbaren Reasoning-Mechanismus. Zweitens werden LLMs aufgefordert, einen mehrstufigen Reasoning-Pfad zu generieren, der zur Schlussfolgerung führt, wie bei CoT. Wir führen LLMs jedoch nur einmal aus, und alle Zwischenschritte im Reasoning werden in einer einzigen Vorhersage generiert. Drittens zerlegen wir das ursprüngliche Problem in eine Reihe von Teilproblemen, die entweder in separaten Durchläufen von LLMs behandelt oder mit anderen Systemen gelöst werden. Hier konzentrieren wir unsere Aufmerksamkeit auf den dritten Ansatz, der eng mit der Problemdekomposition zusammenhängt. Beachten Sie jedoch, dass eine umfassendere Diskussion all diese Ansätze abdecken könnte, während die ersten beiden in diesem Kapitel bereits in gewissem Maße diskutiert wurden.

Ein allgemeiner Rahmen für die Problemdekomposition umfasst zwei Elemente.

- Generierung von Teilproblemen. Dies beinhaltet die Zerlegung des Eingangsproblems in eine Reihe von Teilproblemen.
- Lösung von Teilproblemen. Dies beinhaltet die Lösung jedes Teilproblems und die Ableitung von Zwischen- und Endergebnissen durch Schlussfolgerungen.

Diese beiden Aspekte können auf unterschiedliche Weise modelliert werden, was zu verschiedenen Methoden der Problemdekomposition führt. Ein Ansatz besteht darin, sie

als separate Schritte in einem zweistufigen Prozess zu behandeln. Betrachten wir zum Beispiel die zu Beginn dieses Unterabschnitts beschriebene Aufgabe des Blogschreibens. Im ersten Schritt zerlegen wir das gesamte Problem auf einmal in Teilprobleme (d.h., wir gliedern den Blog). Im zweiten Schritt lösen wir die Teilprobleme entweder sequenziell oder in einer anderen Reihenfolge (d.h., wir füllen den Inhalt für jeden Abschnitt nach Bedarf aus). Die endgültige Ausgabe dieses Prozesses kombiniert die Ergebnisse der Lösung jedes Teilproblems. Obwohl diese Methode einfach und unkompliziert ist, geht sie davon aus, dass das Problem kompositionell ist, was sie besser für Aufgaben wie Schreiben und Codegenerierung geeignet macht.

Viele reale Probleme erfordern jedoch komplexes Reasoning. Ein wesentliches Merkmal dieser Probleme ist, dass die Reasoning-Schritte nicht festgelegt sein müssen. Der Reasoning-Pfad kann für verschiedene Probleme variieren, und jeder Schritt des Reasonings kann von den Ergebnissen vorheriger Schritte abhängen. In solchen Fällen ist es nicht wünschenswert, eine feste Generierung von Teilproblemen im Voraus zu verwenden. Stattdessen sollten Teilprobleme dynamisch basierend auf dem Eingabeproblem generiert werden und, wenn möglich, während des Reasoning-Prozesses spontan erstellt werden. Dies macht die Problemdekomposition im Vergleich zur Entwicklung von Divide-and-Conquer-Algorithmen anspruchsvoller. Idealerweise würden wir sowohl die Systeme zur Generierung von Teilproblemen als auch zur Lösung von Teilproblemen gemeinsam entwerfen. Ein praktischerer und weit verbreiteter Ansatz ist jedoch, separate Modelle für diese Aufgaben zu verwenden. Eine unkomplizierte Möglichkeit, dies zu erreichen, besteht darin, ein LLM für diese Aufgaben anzupassen, indem man das Modell entweder durch Prompting oder durch Tuning anpasst.

Hier betrachten wir eine auf der obigen Idee basierende Methode, die als Least-to-Most-Prompting [Zhou et al., 2023b] bezeichnet wird. Die Motivation für diese Methode ergibt sich aus den Herausforderungen bei der Lösung schwieriger Reasoning-Probleme –jener, die nicht einfach durch Verallgemeinerung aus einigen wenigen Beispielen gelöst werden können. Für diese Probleme ist eine effektivere Problemlösungsstrategie, einer progressiven Abfolge von Teilproblemen zu folgen, die systematisch zur Schlussfolgerung führen. Genauer gesagt wird bei der Least-to-Most-Prompting-Methode die Generierung von Teilproblemen durch das Anweisen eines LLM mit Anweisungen und/oder Demonstrationen durchgeführt. Unten sehen Sie beispielsweise einen 2-Shot-Prompt zur Generierung von Teilproblemen beim Least-to-Most-Prompting.

TASK Your task is to decompose a problem into several sub-problems. You will be given a few examples to illustrate how to achieve this.

DEMO Q: In a community, 5% of the population are infants, 15% are children, 40% are adults, and 40% are seniors. Which group makes up the largest portion of the population?

A: To answer the question "Which group makes up the largest portion of the population?", we need to know: "How many percent are infants?", "How many percent are children?", "How many percent are adults?", "How many percent are seniors?".

Q: Alice, Bob, and Charlie brought beads for their group project in their craft class. Alice has twice as many beads as Bob, and Bob has five times as many beads as Charlie. If Charlie has 6 beads, how many beads can they use for their craft project?

A: To answer the question "How many beads can they use for their craft project?", we need to know: "How many beads does Bob have?", "How many beads does Alice have?".

USER Q: The environmental study conducted from 2015 to 2020 revealed that the average temperature in the region increased by 2.3 degrees Celsius. What was the duration of the environmental study?

A: To answer the question "What was the duration of the environmental study?", we need to know: "When did the environmental study start?", "When did the environmental study end?".

Durch das Lernen aus den Beispielen kann das LLM zwei Teilprobleme zur Beantwortung des neuen Problems „Wie lange dauerte die Umweltstudie?"(in blau und orange hervorgehoben) generieren. Mit diesen Teilproblemen lösen wir sie nacheinander. Für jedes Teilproblem nehmen wir alle zuvor generierten QA-Paare als Kontext und erzeugen dann die Antwort. Für das obige Beispiel müssen wir das erste Teilproblem beantworten, indem wir das LLM wie folgt anweisen:

The environmental study conducted from 2015 to 2020 revealed that the average temperature in the region increased by 2.3 degrees Celsius.

SUB-PROB1 Q: When did the environmental study start?

A: The environmental study started in 2015.

Sobald wir die Antwort auf das erste Teilproblem haben, gehen wir zum zweiten über. Diesmal fügen wir sowohl das erste Teilproblem als auch seine entsprechende Antwort in die Eingabe ein.

The environmental study conducted from 2015 to 2020 revealed that the average temperature in the region increased by 2.3 degrees Celsius.

SUB-PROB1 Q: When did the environmental study start?

A: The environmental study started in 2015.

SUB-PROB2 Q: When did the environmental study end?

A: The environmental study ended in 2020.

Schließlich verwenden wir das LLM, um das ursprüngliche Problem zu lösen, gegeben die Antworten auf alle Teilprobleme.

The environmental study conducted from 2015 to 2020 revealed that the average temperature in the region increased by 2.3 degrees Celsius.

SUB-PROB1 Q: When did the environmental study start?

A: The environmental study started in 2015.

SUB-PROB2 Q: When did the environmental study end?

A: The environmental study ended in 2020.

FINAL Q: What was the duration of the environmental study?

A: The duration of the environmental study was 5 years.

Die Least-to-Most-Methode bietet einen grundlegenden Ansatz, um LLMs anzuweisen, Teilprobleme separat zu generieren und zu lösen. Wir können sie auf verschiedene Weisen verbessern. Eine einfache Verbesserung besteht darin, verschiedene fortgeschrittene Prompting-Techniken anzuwenden, die keine Änderungen am Rahmen der Problemdekomposition erfordern. Zum Beispiel können wir CoT in das Prompting integrieren, um die Reasoning-Leistung bei der Generierung und Lösung von Teilproblemen zu verbessern.

Eine weitere Verbesserung besteht darin, Methoden zur besseren Zerlegung von Problemen und zur Organisation von Problemlösungspfaden zu erforschen. Um diese Ansätze zu beschreiben, verwenden wir das Symbol p_0, um das Eingabeproblem zu bezeichnen, und die Symbole $\{p_1, ..., p_n\}$, um die Teilprobleme zu p_0 zu bezeichnen. Beim Least-to-Most-Prompting zerlegen wir p_0 in $\{p_1, ..., p_n\}$, gegeben durch

$$\{p_1, ..., p_n\} \quad = \quad G(p_0) \tag{3.2}$$

wobei $G(\cdot)$ die Funktion der Teilproblemgenerierung bezeichnet. Dann lösen wir die Teilprobleme $\{p_1, ..., p_n\}$ sequenziell, was zu einer Sequenz von Antworten $\{a_1, ..., a_n\}$ führt. Zur Beantwortung des i-ten Teilproblems p_i schließen wir sowohl das ursprüngliche Problem p_0 als auch alle zuvor gesehenen Problem-Antwort-Paare in den Kontext für die

Vorhersage ein. Die Antwort a_i ist gegeben durch

$$a_i = S_i(p_i, \{p_0, p_{<i}, a_{<i}\}) \tag{3.3}$$

wobei $p_{<i} = \{p_1, ..., p_{i-1}\}$ und $a_{<i} = \{a_1, ..., a_{i-1}\}$. $S_i(\cdot)$ bezeichnet die Funktion, die das Teilproblem p_i unter Berücksichtigung des Kontexts $\{p_0, p_{<i}, a_{<i}\}$ löst. Der letzte Schritt besteht darin, die Antwort auf das ursprüngliche Problem p_0 zu generieren, was in ähnlicher Weise wie in Gl. (3.3) ausgedrückt werden kann.

$$a_0 = S_0(p_0, \{p_{\leq n}, a_{\leq n}\}) \tag{3.4}$$

Eine Möglichkeit, dieses Modell zu verfeinern, besteht darin, die Funktion $G(\cdot)$ so zu modifizieren, dass das Modell dynamisch Antworten generieren kann. Anstatt alle Teilprobleme auf einmal zu generieren, können wir jedes von ihnen während der Problemlösung generieren [Dua et al., 2022]. Dazu können wir Gl. (3.2) ersetzen durch

$$p_i = G_i(p_0, \{p_{<i}, a_{<i}\}) \tag{3.5}$$

So erhalten wir ein Modell zur Generierung von Teilproblemen, das schrittweise arbeitet. In jedem Schritt i generieren wir zuerst das Teilproblem p_i, indem wir ein LLM mit dem ursprünglichen Problem p_0 und der Problemlösungshistorie $\{p_{<i}, a_{<i}\}$ anweisen. Anschließend generieren wir die Antwort a_i für dieses Teilproblem unter Verwendung desselben oder eines anderen LLM, basierend auf denselben kontextuellen Informationen (siehe Gl. (3.3)). Diese Methode erweitert effektiv die Reasoning-Kapazität von LLMs, indem sie ihnen ermöglicht, Teilprobleme in Zwischenschritten des Reasonings dynamisch zu generieren und zu lösen. Infolgedessen sind die Reasoning-Pfade nicht im Voraus festgelegt, und die Modelle können ihre Reasoning-Strategien während der Problemlösung wählen und anpassen.

Eine weitere Möglichkeit, das obige Modell zu verbessern, besteht darin, sich auf die Entwicklung besserer Löser für Teilprobleme zu konzentrieren. In unserer bisherigen Diskussion haben wir $S_i(\cdot)$ auf LLMs beschränkt, die angewiesen werden, das Teilproblem p_i zu lösen. Tatsächlich können wir diese Funktion auf jedes System erweitern, das in der Lage ist, das Teilproblem zu behandeln. Zum Beispiel könnte $S_i(\cdot)$ Aufrufe an IR-Systeme tätigen, was uns den Zugriff auf eine breitere Palette von Daten zur Problemlösung ermöglicht. Ein weiteres Beispiel ist die Verwendung von $S_i(\cdot)$ als Taschenrechner, um Ergebnisse bei mathematischen Problemlösungen genau zu berechnen. Wenn das Teilproblem p_i komplex ist und mehrere Zwischenschritte zur Problemlösung erfordert, ist es auch möglich, p_i weiter in kleinere Teilprobleme zu zerlegen. Zum Beispiel kann $S_i(\cdot)$ als rekursives Programm definiert werden, das Teilprobleme generiert und löst. Dies integriert Rekursion in die Problemlösung und ermöglicht es uns, Probleme durch iterative Zerlegung zu lösen. Als Ergebnis können wir eine hierarchische Struktur für die Problemlösung definieren [Khot et al., 2023].

Wenn wir die obige Formulierung etwas weiter verallgemeinern, können wir sie als ein Problem des Reinforcement Learning betrachten. Eine typische Methode besteht darin, einen Problemlösungsprozess als einen Entscheidungsprozess zu modellieren. In jedem Schritt dieses Prozesses wird eine Aktion basierend auf dem aktuellen Zustand ausgeführt.

Diese Aktionen können alle Funktionen zur Generierung und Lösung von Teilproblemen umfassen (d.h. $G_i(\cdot)$ und $S_i(\cdot)$). Somit entspricht die Aktionssequenz einem Problemlösungspfad. Da die Diskussion von Reinforcement-Learning-Problemen den Rahmen dieses Kapitels sprengt, überspringen wir die genaue Beschreibung dieser Lernaufgabe. Dennoch ist die Entwicklung eines Agenten oder Controllers, der bestimmt, wann und wie ein Teilproblem generiert und gelöst werden soll, ebenfalls eine naheliegende Wahl.

Im NLP steht die Problemdekomposition in Beziehung zu einer langen Forschungstradition zur Multi-Hop-Fragebeantwortung [Mavi et al., 2024]. Diese Aufgabe erfordert, dass das System Informationen aus mehreren Textstücken sammelt und kombiniert, um eine genaue Antwort auf eine komplexe Frage zu geben. Um beispielsweise die Frage „Was ist die Hauptstadt des Landes, in dem Albert Einstein geboren wurde?“zu beantworten, müssen wir wissen „Wo wurde Albert Einstein geboren?“und „Was ist die Hauptstadt von Deutschland?“. Frühere Arbeiten in diesem und verwandten Bereichen haben das Thema der Problemdekomposition untersucht, obwohl die Methoden möglicherweise nicht auf LLMs basierten. Eine beliebte Methode besteht beispielsweise darin, ein zusätzliches neuronales Modell zu entwickeln, um einfachere Fragen zu generieren, die verschiedene Aspekte der ursprünglichen Frage behandeln [Andreas et al., 2016; Talmor and Berant, 2018; Min et al., 2019]. Dieser Fragengenerator kann Fragen im Stapel- oder sequenziellen Modus erstellen.

Im weiteren Sinne steht die Problemdekomposition auch im Zusammenhang mit dem Thema der Kompositionalität im NLP [Drozdov et al., 2022; Press et al., 2023]. Zum Beispiel bilden wir beim semantischen Parsen Sätze in natürlicher Sprache auf strukturierte Bedeutungsrepräsentationen ab, indem wir sie in ihre Bestandteile zerlegen und die Sätze auf der Grundlage der Bedeutungen dieser Teile und der Regeln, die zu ihrer Kombination verwendet werden, verstehen. In frühen Studien auf diesem Gebiet galten stark kompositionelle Sätze als einfacher für Testsysteme, da es relativ unkompliziert ist, solche Sätze zu zerlegen und die Bedeutungen ihrer Teile zusammenzusetzen. Die Aufgabe wird jedoch viel schwieriger, wenn mehr Generalisierung für die Modellierung der Kompositionalität in neuen Daten erforderlich ist. In diesem Fall möchten wir, dass Systeme verbesserte Fähigkeiten zur kompositionellen Generalisierung aufweisen. In jüngerer Forschung zu LLMs wurde dieses Thema häufig im Zusammenhang mit kompositionellen Reasoning-Aufgaben diskutiert, wie zum Beispiel SCAN[7], da es als ein wichtiger Aspekt beim Testen der Sprachverständnis- und Reasoning-Fähigkeiten von LLMs angesehen wird. Dies stellt auch neue Aufgaben für die Entwicklung und Untersuchung von Methoden zur Problemdekomposition dar.

Bei LLMs ist eine interessante Anwendung der Problemdekomposition die Werkzeugnutzung. In einigen Fällen ist es notwendig, externe Werkzeuge in LLMs zu integrieren, um auf genaue Daten zuzugreifen, die während des Trainings oder Fine-Tunings nicht verfügbar waren. Zum Beispiel können LLMs mit APIs integriert werden, um Echtzeitdaten wie Wetteraktualisierungen, Börsenkurse oder Nachrichten-Feeds abzurufen, was es ihnen ermöglicht, aktuelle Antworten auf Benutzeranfragen zu geben. Bei der Verwendung

[7]Die SCAN-Aufgaben (Simplified versions of the CommAI Navigation tasks) sind darauf ausgelegt, die Fähigkeit von LLMs zur kompositionellen Generalisierung zu bewerten [Lake and Baroni, 2018]. Sie beinhalten die Übersetzung von Befehlen in natürlicher Sprache in eine Aktionssequenz. Zum Beispiel kann ein Befehl „jump opposite left and walk thrice“in die Aktionssequenz „LTURN LTURN JUMP WALK WALK WALK“übersetzt werden.

von Werkzeugen können LLM-Vorhersagen Markierungen enthalten, die anzeigen, wo und wie externe APIs aufgerufen werden sollen. Dies erfordert die Zerlegung des Problems in Teilprobleme, wobei einige von den LLMs und andere von externen Werkzeugen behandelt werden. Ausführlichere Diskussionen zu diesem Thema werden in Abschnitt 3.2.5 vorgestellt.

3.2.3 Selbstverfeinerung

In vielen Fällen können die Vorhersagen von LLMs ungenau oder falsch sein. Da aktuelle LLMs Aufgaben wie Verfeinerung und Korrektur durchführen können, ist es sinnvoll, Methoden zu erforschen, mit denen diese Modelle ihre Ausgaben selbst verfeinern können. Selbstverfeinerung ist ein häufiges Phänomen bei menschlichen psychologischen Aktivitäten und im täglichen Verhalten. Wenn ein Designer beispielsweise ein Produkt entwirft, erstellt er möglicherweise zuerst einen grundlegenden Prototyp und verfeinert dann das Design nach Evaluierung und Tests, um die Benutzererfahrung und Funktionalität zu verbessern. Die Verfeinerung kann mehrmals wiederholt werden, bis das Design zufriedenstellend ist. Die Idee des Vorhersagens-und-dann-Verfeinerns findet sich auch im NLP. Ein frühes Beispiel ist der Tagger von Brill [Brill, 1992], bei dem ein anfängliches POS-Tagging-Ergebnis eines Satzes iterativ mithilfe eines regelbasierten Systems verfeinert werden kann. Im Zeitalter des Deep Learning können auch zahlreiche Arbeiten zu Sequenz-zu-Sequenz-Problemen, wie Grammatikkorrektur und Textumschreibung, als Beispiele für dieses Thema angesehen werden.

Wir können LLMs dazu auffordern, eine Selbstverfeinerung durchzuführen. Betrachten wir ein einfaches Beispiel, bei dem wir zuerst ein LLM verwenden, um einen Text aus dem Chinesischen ins Englische zu übersetzen, und dann dasselbe Modell verwenden, um die Übersetzung zu verfeinern. So können wir das Modell zur Übersetzung auffordern:

> Translate the following text from Chinese to English:
>
> Chinese:
> 一系列考古发现奠定红山文化在中华文明起源研究中的重要地位。最新公布的研究成果认为，大约从距今 5800 年开始，中华大地上各个区域相继出现较为明显的社会分化，进入文明起源的加速阶段。
>
> English:
> A series of discoveries have cemented the significant role of the Hongshan culture in studies on the origins of Chinese civilization. The latest research findings suggest that, starting from around 5800 years ago, many regions across China began to experience noticeable social differentiations, entering an accelerated phase in the origin of civilization.

Dann können wir es auffordern, die Übersetzung zu verfeinern:

> You are provided with a Chinese text along with its English translation. Please refine the translation to improve its accuracy and fluency if necessary.
>
> Chinese:
> 一系列考古发现奠定红山文化在中华文明起源研究中的重要地位。最新公布的研究成果认为，大约从距今 5800 年开始，中华大地上各个区域相继出现较为明显的社会分化，进入文明起源的加速阶段。
>
> English:
> A series of discoveries have cemented the significant role of the Hongshan culture in studies on the origins of Chinese civilization. The latest research findings suggests that, starting from around 5800 years ago, many regions across China began to experience noticeable social differentiations, entering an accelerated phase in the origin of civilization.
>
> Refinement:
> A series of archaeological discoveries have cemented the significant role of the Hongshan culture in studies on the origins of Chinese civilization. The latest research findings suggest that, starting from around 5800 years ago, various regions across China began to experience noticeable social differentiations, entering an accelerated phase in the origin of civilization.

Hier sind Änderungen an der Eingabeübersetzung grün hervorgehoben. Dieser Prozess kann iterativ sein: Bei jedem Schritt verwenden wir die Übersetzung aus dem vorherigen Schritt als Eingabe und fordern das LLM auf, eine potenziell bessere Übersetzung zu erstellen.

Im obigen Beispiel wurde die Verfeinerung der anfänglichen Modellausgabe durch die Aufforderung des LLM mit einer einfachen Anweisung wie „Bitte verfeinern Sie es!“erreicht. Die Verfeinerung basiert jedoch ausschließlich auf der Fähigkeit des LLM, Anweisungen zu befolgen, und es gibt keine Anleitung oder Überwachung, wie und wo die Modellausgabe verbessert werden kann. Ein effektiverer Ansatz wäre es, Feedback zu spezifischen Aspekten zu berücksichtigen, die einer Verfeinerung bedürfen. Zum Beispiel können wir das LLM mit „Bitte korrigieren Sie alle grammatikalischen Fehler in der Übersetzung “auffordern, damit sich das Modell während der Verfeinerung stärker auf die Korrektur grammatikalischer Fehler konzentrieren kann.

Ein allgemeiner Rahmen für die Selbstverfeinerung mit LLMs umfasst drei Schritte [Madaan et al., 2024].

- Vorhersage. Wir verwenden ein LLM, um die anfängliche Modellausgabe zu erzeugen.
- Feedback-Sammlung. Wir holen Feedback zur Modellausgabe ein.
- Verfeinerung. Wir verwenden das LLM, um die Modellausgabe auf der Grundlage des Feedbacks zu verfeinern.

Die letzten beiden Schritte können mehrmals wiederholt werden, was zu einem iterativen Selbstverfeinerungsprozess führt. In diesem Rahmen ist der Erhalt von genauem und detailliertem Feedback entscheidend, da es den nachfolgenden Verfeinerungsschritt leitet. Im Allgemeinen kann Feedback auf verschiedene Weisen erhalten werden [Fernandes et al., 2023; Pan et al., 2024]. Wir können die Vorhersagen des Modells manuell untersuchen, um Probleme zu identifizieren. Außerdem können wir ein Feedback-Modell verwenden, um automatisch Feedback zu generieren. Zum Beispiel kann ein auf einigen gelabelten Daten trainiertes Belohnungsmodell helfen, die Modellausgabe zu bewerten und einen Score zurückzugeben, der ihre Qualität anzeigt.

Da der Fokus dieses Kapitels auf dem Prompting von LLMs liegt, betrachten wir die Verwendung von LLMs zur Bereitstellung von Feedback. Zur Veranschaulichung betrachten wir ein Beispiel für die Generierung guter Antworten auf Benutzerfragen:

- Zuerst fordern wir ein LLM auf, die Eingabefrage wie gewohnt zu beantworten.

> Generate a response to the question: "What are some major environmental concerns today?"
>
> Response:
>
> Major environmental concerns today include climate change, air and water pollution, deforestation, loss of biodiversity, and ocean degradation.

- Anschließend fordern wir dasselbe LLM mit dem folgenden Prompt erneut auf, um Feedback zur generierten Antwort zu erhalten.

You will be provided with a question and its corresponding response. Evaluate the accuracy and depth of this response, and identify any factual inaccuracies or areas where additional details could enhance understanding.

Question:

What are some major environmental concerns today?

Response:

Major environmental concerns today include climate change, air and water pollution, deforestation, loss of biodiversity, and ocean degradation.

Feedback:

The response provided is accurate and highlights key environmental concerns. However, it would benefit from more specific and detailed descriptions of these concerns. Here are suggestions for improvement:

1. Include specific examples of environmental issues.
2. Add information about major pollutants.
3. Specify the regions most affected.

- Anhand des vom LLM erzeugten Feedbacks fordern wir es auf, die Antwort zu verfeinern.

You will be provided with a question, its corresponding response, and some feedback on this response.

Question:

What are some major environmental concerns today?

Response:

Major environmental concerns today include climate change, air and water pollution, deforestation, loss of biodiversity, and ocean degradation.

Feedback:

The response provided is accurate and highlights key environmental concerns. However, it would benefit from more specific and detailed descriptions of these concerns. Here are suggestions for improvement:

1. Include specific examples of environmental issues.
2. Add information about major pollutants.
3. Specify the regions most affected.

Now refine the response based on the provided feedback.

Refined Response:

Key environmental concerns include climate change, evidenced by rising global temperatures and severe weather patterns; significant air and water pollution in urban and industrial areas; rampant deforestation in regions like the Amazon, causing biodiversity loss; and ocean degradation, highlighted by coral reef bleaching and widespread overfishing.

Idealerweise möchten wir, dass ein starkes LLM alle drei Schritte ohne zusätzliches Training durchführt. Andererseits, wenn wir genügend gelabelte Daten für die betreffende Aufgabe haben, können wir die Leistung des LLM durch überwachtes Lernen verbessern. Zum Beispiel können wir das LLM feinabstimmen, um es besser an Verfeinerungsaufgaben anzupassen, oder alternativ aufgabenspezifische Modelle verwenden, die nicht notwendigerweise auf LLMs basieren müssen [Welleck et al., 2023; Schick et al., 2023]. Im weiteren Sinne kann die Verbesserung von LLMs für Selbstverfeinerungsaufgaben als ein Anpassungsproblem (Alignment) betrachtet werden. Zum Beispiel wurde festgestellt, dass einige Selbstkorrekturfähigkeiten durch RLHF aktiviert werden können [Ganguli et al., 2023]. Die Diskussion dieser Themen geht jedoch über den Rahmen dieses Kapitels hinaus. Weitere Diskussionen finden sich in Kapitel 4.

In LLMs steht die Selbstverfeinerung im Zusammenhang mit mehreren Konzepten, die die psychologischen Aspekte dieser Modelle offenbaren, wie etwa die Fähigkeit zur Selbstreflexion. Eine Ansicht ist, dass, wenn LLMs zur Selbstreflexion fähig sind, ihre Vorhersagen genauer werden und sogar selbstkorrigierende Fähigkeiten besitzen können. Diese Selbstreflexion kann auf verschiedene Weisen aktiviert werden, zum Beispiel indem man diese LLMs auffordert, sich auf tiefergehendes und sorgfältigeres Denken einzulassen, oder indem man Beispiele bereitstellt, aus denen die Modelle lernen und reflektieren können. Zur

Veranschaulichung betrachten wir hier die Methode deliberate-then-generate (DTG), die in der Arbeit von Li et al. [2023a] vorgestellt wird, bei der LLMs zum Nachdenken (deliberate) angeregt werden. Bei DTG erhalten wir eine anfängliche Modellausgabe, die Fehler enthalten kann. LLMs werden dann aufgefordert, die Fehlertypen dieser Modellausgabe zu identifizieren und eine verbesserte Ausgabe zu liefern. Unten finden Sie eine Vorlage für das DTG-Prompting für Übersetzungsaufgaben vom Chinesischen ins Englische.

Given the Chinese sentence: {∗source∗}
The English translation is: {∗target∗}
Please first detect the type of error, and then refine the translation.
Error Type:

Wir zielen darauf ab, zuerst den Fehlertyp (rot) vorherzusagen und dann eine verfeinerte Übersetzung (blau) zu erstellen. Dieser Prozess des Nachdenkens wird durch die Anweisung „Bitte erkennen Sie zuerst den Fehlertyp und verfeinern Sie dann die Übersetzung"geleitet. Es ermutigt LLMs, zunächst eine durchdachte Analyse durchzuführen und dann bessere Ergebnisse zu liefern. Da die Vorhersage des Fehlertyps und die Verfeinerung in einem einzigen Durchlauf von LLMs durchgeführt werden, integriert diese Methode beide Schritte des Feedbacks und der Verfeinerung in einen Prozess.

In den obigen Prompts gehen wir davon aus, dass das von uns verwendete LLM in der Lage ist, die Eingabeübersetzung zu überprüfen und ihre Fehlertypen korrekt zu identifizieren. Dies wirft jedoch neue Schwierigkeiten auf, da das Modell möglicherweise nicht gut darin ist, Fehler in Übersetzungen zu finden. Dies wird wiederum zu zusätzlichem Aufwand für die Feinabstimmung oder das Prompt-Engineering führen. Eine einfachere Methode besteht also darin, die Last der Fehleridentifikation zu reduzieren und LLMs nur zum Nachdenken zu verwenden. Dazu können wir die Eingabeübersetzung durch eine zufällige Übersetzung ersetzen und einen Standardfehlertyp zuweisen. Ein Beispiel für einen solchen Prompt ist unten dargestellt.

Given the Chinese sentence:
一系列考古发现奠定红山文化在中华文明起源研究中的重要地位。
The English translation is:
A variety of innovative techniques have redefined the importance of modern art in contemporary cultural studies.
Please first detect the type of error, and then refine the translation.
Error Type: Incorrect Translation

In diesem Beispiel wird die Eingabeübersetzung nicht von LLMs generiert, sondern stattdessen zufällig aus dem Datensatz ausgewählt. Es handelt sich also einfach um eine

falsche Übersetzung für den Quellsatz, und wir können den Fehlertyp entsprechend festlegen. Die LLMs generieren dann eine neue Übersetzung, indem sie sowohl den Quellsatz als auch die falsche Übersetzung als Eingabe verwenden. Das Design dieses Prompts kann auch als Aktivierung der Lernfähigkeiten von LLMs durch „negative Evidenz"[Marcus, 1993] betrachtet werden, wodurch sie in die Lage versetzt werden, durch kontrastive Analyse zu reflektieren und bessere Ergebnisse zu erzielen. Dennoch stützt sich diese Methode nicht auf Feedback und kann die Leistung einer einzelnen LLM-Vorhersage durch einfaches Prompting verbessern.

Beachten Sie, dass, obwohl DTG nicht-iterativ ist, iteratives Lernen und Verfeinern im NLP häufig verwendet werden. Ein Vorteil dieser iterativen Ansätze ist, dass sie menschliches Lernen und Problemlösen nachahmen, bei denen kontinuierliches Feedback und Anpassungen zu schrittweise verbesserten Ergebnissen führen. Iterative Methoden können auf eine Reihe von LLM-Prompting-Problemen angewendet werden. Zum Beispiel kann man bei der Problemzerlegung in jedem Schritt neue Teilprobleme und ihre Lösungen in den Kontext einbeziehen, und so können sich LLMs schrittweise der Lösung des ursprünglichen Problems annähern. Andererseits werfen iterative Methoden mehrere Probleme auf, die bei nicht-iterativen Methoden nicht auftreten, zum Beispiel können Fehler in früheren Schritten die nachfolgende Problemlösung negativ beeinflussen, und die Bestimmung, wann die Iteration beendet werden soll, erfordert oft zusätzlichen technischen Aufwand.

3.2.4 Ensembling

Modell-Ensembling für die Textgenerierung wurde in der NLP-Literatur ausführlich diskutiert. Die Idee dabei ist, die Vorhersagen von zwei oder mehr Modellen zu kombinieren, um eine bessere Vorhersage zu erzeugen. Diese Technik ist direkt auf LLMs anwendbar. Zum Beispiel können wir eine Reihe von LLMs sammeln und jedes davon auf dieselbe Eingabe anwenden. Die endgültige Ausgabe ist eine kombinierte Vorhersage dieser Modelle.

Beim LLM-Prompting ist es ebenfalls möglich, die Leistung durch die Kombination von Vorhersagen auf der Grundlage verschiedener Prompts zu verbessern. Angenommen, wir haben ein LLM und eine Sammlung von Prompts, die dieselbe Aufgabe behandeln. Wir können dieses LLM mit jedem der Prompts ausführen und dann die Vorhersagen kombinieren. Nachfolgend finden Sie beispielsweise drei verschiedene Prompt-Vorlagen zur Textvereinfachung.

```
Make this text simpler.
{*text*}
___
```

```
Condense and simplify this text.
{*text*}
___
```

Rewrite for easy reading.
{∗text∗}
——

Jeder dieser Prompts führt zu einer anderen Vorhersage, und wir können alle drei Vorhersagen berücksichtigen, um die endgültige zu generieren.

Formal seien $\{\mathbf{x}_1, ..., \mathbf{x}_K\}$ K Prompts zur Durchführung derselben Aufgabe. Gegeben ein LLM $\Pr(\cdot|\cdot)$, können wir die beste Vorhersage für jedes $\mathbf{x}_i$ mit $\hat{\mathbf{y}}_i = \arg\max_{\mathbf{y}_i} \Pr(\mathbf{y}_i|\mathbf{x}_i)$ finden. Diese Vorhersagen können zu einer „neuen"Vorhersage kombiniert werden:

$$\hat{\mathbf{y}} \quad = \quad \text{Combine}(\hat{\mathbf{y}}_1, ..., \hat{\mathbf{y}}_K) \tag{3.6}$$

Hier ist Combine$(\cdot)$ das Kombinationsmodell, das auf verschiedene Weisen gestaltet werden kann. Zum Beispiel können wir die beste Vorhersage durch Abstimmung oder durch Identifizierung derjenigen mit der größten Überlappung mit anderen auswählen. Eine weitere Methode zur Modellkombination ist die Durchführung einer Modellmittelung während der Token-Vorhersage. Sei $\hat{y}_j$ das vorhergesagte Token im j-ten Schritt für die Modellkombination. Die Wahrscheinlichkeit, $\hat{y}_j$ vorherzusagen, ist gegeben durch

$$\hat{y}_j \quad = \quad \arg\max_{y_j} \sum_{k=1}^{K} \log \Pr(y_j|\mathbf{x}_k, \hat{y}_1, ..., \hat{y}_{j-1}) \tag{3.7}$$

Beim Ensembling für das LLM-Prompting ist es im Allgemeinen vorteilhaft, vielfältige Prompts zu verwenden, damit die Kombination ein breiteres Spektrum potenzieller Antworten erfassen kann. Diese Praxis ist im Ensemble-Lernen üblich, da Vielfalt dazu beiträgt, Verzerrungen und Fehler auszugleichen, die für ein einzelnes Modell oder eine einzelne Konfiguration spezifisch sein können. Aus Bayes'scher Sicht können wir den Prompt $\mathbf{x}$ als latente Variable betrachten, gegeben das interessierende Problem p. Dies ermöglicht es, die prädiktive Verteilung von $\mathbf{y}$ gegeben p als die über alle möglichen Prompts marginalisierte Verteilung $\Pr(\mathbf{y}|\mathbf{x})$ zu schreiben

$$\Pr(\mathbf{y}|p) \quad = \quad \int \Pr(\mathbf{y}|\mathbf{x}) \Pr(\mathbf{x}|p) d\mathbf{x} \tag{3.8}$$

Das Integral berechnet die Gesamtwahrscheinlichkeit von $\mathbf{y}$, indem alle möglichen Werte von $\mathbf{x}$ berücksichtigt werden, gewichtet nach ihren Wahrscheinlichkeiten gegeben p. Hier ist $\Pr(\mathbf{y}|\mathbf{x})$ durch das LLM gegeben, und $\Pr(\mathbf{x}|p)$ ist die A-priori-Verteilung der Prompts für das Problem. Dies ist ein gutes Modell, da das Integral die Unsicherheit bei der Wahl von $\mathbf{x}$ effektiv berücksichtigt und sicherstellt, dass die endgültige prädiktive Verteilung $\Pr(\mathbf{y}|p)$ robust ist und alle potenziellen Variationen und Verzerrungen in den Prompts umfasst. Die direkte Berechnung dieses Integrals kann jedoch aufgrund des potenziell unendlichen Raums von $\mathbf{x}$ rechentechnisch undurchführbar sein. Ein Ansatz zur Lösung dieses Problems ist der Einsatz von Methoden wie dem Monte-Carlo-Sampling, das das Integral unter Verwendung einer überschaubaren, endlichen Anzahl von Prompts approximiert.

Obwohl die Bayes'sche Behandlung mathematisch gut definiert ist, ist es in der NLP

gängige Praxis, einen nicht-informativen oder uniformen Prior anzunehmen und sich stattdessen auf die Konstruktion eines Satzes vielfältiger Prompts zu konzentrieren. Folglich kann die Ausgabe mit einem einfachen Kombinationsmodell berechnet werden, wie in Gl. (3.6) beschrieben. Das Problem der Erstellung hochwertiger, vielfältiger Prompts wurde im Rahmen von CoT und anderen Bereichen des In-Context-Lernens untersucht. Der Großteil der Forschung konzentriert sich auf die Einbeziehung einer Vielzahl von Demonstrationsbeispielen über verschiedene Prompts hinweg. Hier listen wir einige dieser Methoden auf.

- Für ein gegebenes Problem erstellen wir manuell eine Reihe von Demonstrationen und verwenden unterschiedliche für verschiedene Prompts.
- Für ein gegebenes Problem verwenden wir LLMs, um automatisch Demonstrationen und Prompts zu generieren.
- Für einen gegebenen Prompt erstellen wir verschiedene Prompts, indem wir die Reihenfolge der Demonstrationen im Prompt ändern.
- Für einen gegebenen Prompt verwenden wir LLMs, um eine Reihe ähnlicher Prompts zu generieren.
- Für einen gegebenen Prompt wandeln wir ihn in andere Formen um, z. B. indem wir ihn in andere Sprachen übersetzen.

In der Praxis können wir diese Methoden natürlich kombinieren, um eine größere Vielfalt zu erreichen. Eine zugrunde liegende Annahme hierbei ist, dass vielfältige Prompts zu vielfältigen Modellausgaben führen können. Dies ist insbesondere der Fall, wenn das zu behandelnde Problem relativ neu und schwierig ist. Bei stärkeren und robusteren LLMs ist die Varianz in der Ausgabe für ähnliche Prompts möglicherweise nicht groß. In diesem Fall kann der Vorteil der Einbeziehung mehrerer Prompts bescheiden sein.

Neben der Bereitstellung vielfältiger Prompts für LLMs besteht ein weiterer Ansatz darin, die inhärente Varianz in den Ausgaben von LLMs zu nutzen. Eine einfache Möglichkeit, mehrere Ausgaben zu generieren, besteht darin, Ausgaben aus dem Hypothesenraum zu samplen. Dies ist für LLMs unkompliziert, da ihre Suchalgorithmen von Natur aus geeignet sind, mehrere Ergebnisse zu produzieren. Wenn beispielsweise Beam-Search verwendet wird, können wir leicht alle vollständigen Hypothesen im Beam sammeln, um mehrere Ausgabekandidaten zu generieren. Außerdem können wir die Suchalgorithmen modifizieren, um leistungsfähigere Sampling-Methoden zu implementieren und ein breiteres Spektrum an hochwertigen Hypothesen zu erkunden.

Die nächste Frage ist, wie vielfältige Modellvorhersagen kombiniert werden können. Dies führt uns zurück zum Problem der Modellkombination, wie in Gl. (3.6) beschrieben. Hier betrachten wir die self-consistency-Methode, die nicht die Vorhersage mit der höchsten Wahrscheinlichkeit ausgibt, sondern diejenige, die am besten mit anderen Vorhersagen übereinstimmt [Wang et al., 2022a; 2023b]. Zuerst wird ein LLM wie üblich mit CoT gepromptet und generiert durch Sampling mehrere Argumentationspfade. Dann

wird die Häufigkeit jeder Antwort über diese Argumentationspfade gezählt. Die endgültige Ausgabe ist die Antwort mit der höchsten Zählung. Unten werden drei verschiedene Vorhersagen gezeigt, die durch Prompting eines LLM mit CoT erzielt wurden.

Vorhersage 1 (Korrekt):

> Q: Three friends play a game where they flip a fair coin. Each friend flips the coin once. What is the probability that exactly one of them flips heads?
>
> A: Let's think step by step.
>
> Each coin flip is an independent event with a 50% chance of landing heads (H) and 50% tails (T). Consider the possible outcomes for three flips: HHH, HHT, HTH, HTT, THH, THT, TTH, TTT. Out of these, only HHH, HHT, HTH, and THH are relevant. We are only interested in scenarios with exactly one head: HTT, THT, and TTH. There are 3 such outcomes out of 8 total, giving a probability of 3/8.

Vorhersage 2 (Korrekt):

> Q: Three friends play a game where they flip a fair coin. Each friend flips the coin once. What is the probability that exactly one of them flips heads?
>
> A: Let's think step by step.
>
> Using a binomial probability formula, where p (success or heads) is 0.5, and each of the 3 trials (flips) is independent, we look for exactly one success (head). The formula for exactly k successes in n trials is given by the binomial coefficient formula $\binom{n}{k}p^k(1-p)^{n-k}$. Plugging in the values, we get $\binom{3}{1}(0.5)^1(0.5)^{3-1} = 3 \cdot 0.5 \cdot 0.25 = 0.375$.

Vorhersage 3 (Falsch):

> Q: Three friends play a game where they flip a fair coin. Each friend flips the coin once. What is the probability that exactly one of them flips heads?
>
> A: Let's think step by step.
>
> Since each coin flip has a 50% chance of landing heads and a 50% chance of landing tails, and we are looking for the case where only one of the three flips results in heads, we might initially think this is simply a matter of picking which flip is the heads. There are three flips, so one might incorrectly assume that the chance of picking one specific outcome like this would be 1 out of 3. Thus, they might conclude that the probability of exactly one head is $1/3 = 33.3\%$.

Vorhersagen 1 und 2 identifizieren korrekt die drei Fälle, in denen genau einmal Kopf geworfen wird, und erhalten beide eine Wahrscheinlichkeit von 37,5%. Die Argumentation in Vorhersage 3 berücksichtigt nicht die Gesamtzahl der möglichen Ergebnisse bei drei

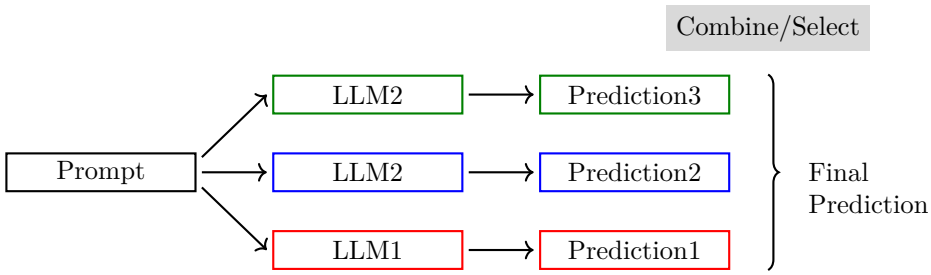
Münzwürfen und liefert daher eine falsche Antwort von 33,3%. Daher wählen wir 37,5% als endgültige Antwort, da dies der Konsens ist.

Self-Consistency bietet ein Kriterium zur Bestimmung der besten Vorhersage aus einem Pool von Kandidaten. Da der Prompt und das Modell bei dieser Methode feststehen, handelt es sich nicht streng genommen um eine Prompt-Ensembling-Methode. Stattdessen kann sie als ein Beispiel für Output-Ensembling-Methoden angesehen werden, auch bekannt als Hypothesenauswahlmethoden, die in der NLP, insbesondere bei Textgenerierungsproblemen, seit langem erforscht werden [Xiao et al., 2013]. Bei diesen Methoden werden mehrere Ausgaben durch Variation von Modellarchitekturen oder Parametern erzeugt. Jede Ausgabe wird dann nach einem bestimmten Kriterium bewertet, und die Ausgaben werden auf der Grundlage dieser Bewertungen neu geordnet. Es gibt verschiedene Möglichkeiten, die Bewertungsfunktion zu definieren, wie z. B. die Messung der Übereinstimmung zwischen einer Ausgabe und anderen und die Verwendung eines stärkeren Modells zur Neubewertung jeder Ausgabe[8]. Abbildung 3.2 zeigt einen Vergleich verschiedener Ensembling-Methoden für LLMs.

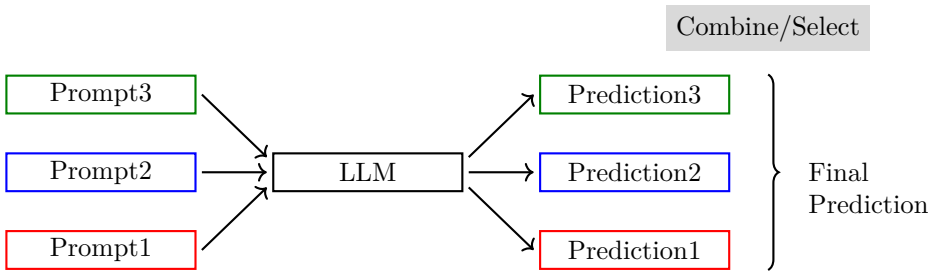
Lassen Sie uns nun kurz die Methoden, die wir bisher in diesem Abschnitt besprochen haben, wie Problemdekomposition und Self-Refinement, rekapitulieren. Es ist offensichtlich, dass diese Methoden die Entscheidungsfindung verbessern, indem sie mehr „Auswahlmöglichkeiten“in den Argumentationsprozess einführen. In gewissem Maße beinhalten sie alle die Bewertung und das Geben von Feedback zu den Ergebnissen von LLMs. Zum Beispiel müssen wir beim Self-Refinement Vorschläge zur Verbesserung der Vorhersage von LLMs machen, und beim Output-Ensembling wählen wir die optimale Ausgabe aus einem Pool von Kandidaten aus. In diesem Sinne fallen diese Methoden unter die breitere Kategorie der Predict-then-Verify-Ansätze, bei denen Vorhersagen zunächst gemacht, dann verifiziert und verfeinert werden. Das grundlegende Problem hierbei ist die Verifizierung und Bewertung der Argumentationsergebnisse oder der Zwischenschritte. Dieses Problem steht in gewissem Zusammenhang mit dem Problem des Trainings von Belohnungsmodellen in RLHF, obwohl RLHF einen anderen Aspekt behandelt. Tatsächlich wurde die Entwicklung von Verifizierern bei der Argumentation mit LLMs erforscht und implementiert. Die meiste Arbeit konzentriert sich, anstatt heuristikbasierte Inferenzzeitalgorithmen zu entwickeln, auf das Erlernen von Verifizierern auf überwachte Weise. Eine unkomplizierte Methode besteht darin, Verifizierer als binäre Klassifikatoren zu trainieren, z. B. zur Klassifizierung einer Antwort als richtig oder falsch, obwohl diese Verifizierer typischerweise als Bewertungsmodelle verwendet werden. Gegeben einen Argumentationspfad für ein Problem, können die Verifizierer verwendet werden, um entweder den gesamten Pfad (sogenannte ergebnisbasierte Ansätze) [Cobbe et al., 2021] oder jeden einzelnen Argumentationsschritt (sogenannte prozessbasierte Ansätze) [Uesato et al., 2022; Lightman et al., 2024] zu bewerten.

[8] Eine Interpretation von Self-Consistency besteht darin, sie als einen Suchprozess mit minimalem Bayes-Risiko zu betrachten. Sie sucht nach der besten Ausgabe, indem sie das Bayes-Risiko minimiert. Genauer gesagt wird eine Risikofunktion $R(\mathbf{y}, \mathbf{y}_r)$ für jedes Paar von Ausgaben (bezeichnet mit $(\mathbf{y}, \mathbf{y}_r)$) definiert, die die Kosten für das Ersetzen von $\mathbf{y}$ durch $\mathbf{y}_r$ darstellt. Gegeben eine Menge von Ausgaben Ω, ist das Risiko einer Ausgabe $\mathbf{y} \in \Omega$ gegeben durch

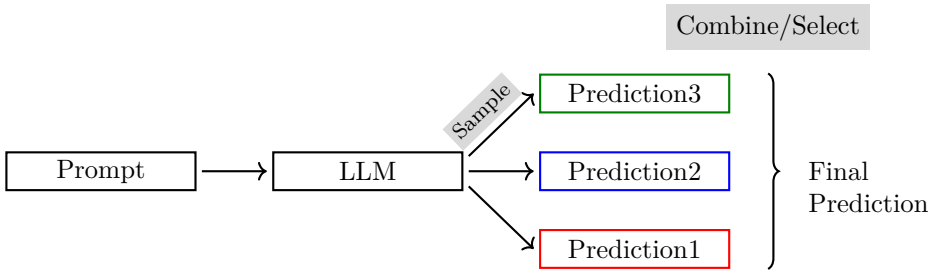
$$\begin{aligned} \text{Risk}(\mathbf{y}) &= \mathbb{E}_{\mathbf{y}_r \sim \Pr(\mathbf{y}_r|\mathbf{x})} R(\mathbf{y}, \mathbf{y}_r) \\ &= \sum_{\mathbf{y}_r \in \Omega} R(\mathbf{y}, \mathbf{y}_r) \cdot \Pr(\mathbf{y}_r|\mathbf{x}) \end{aligned} \tag{3.9}$$



(a) Model Ensembling



(b) Prompt Ensembling



(c) Output Ensembling

Abbildung 3.2: Ensembling-Methoden für LLMs. Beim Standard-Modell-Ensembling (a) werden mehrere LLMs mit unterschiedlichen Architekturen oder Parametern verwendet. Jedes LLM erhält denselben Prompt und erzeugt eine Vorhersage. Diese Vorhersagen werden kombiniert, um die endgültige Vorhersage zu generieren. Beim Prompt-Ensembling (b) verwendet man ein LLM und mehrere Prompts. Das LLM erzeugt für jeden Prompt eine Vorhersage, und diese Vorhersagen werden wie üblich kombiniert. Beim Output-Ensembling (c) sampelt das LLM für einen gegebenen Prompt mehrere Vorhersagen über den Vorhersageraum. Dies kann als eine Methode angesehen werden, um die Leistung des LLM selbst zu steigern. Es ist zu beachten, dass diese Ensembling-Methoden kombiniert werden können, um die Diversität der Vorhersagen zu erhöhen. Zum Beispiel können wir sowohl Prompt-Ensembling als auch Output-Ensembling verwenden, um diversere Vorhersagen zu erhalten.

3.2.5 RAG und Werkzeugnutzung

RAG wird im Allgemeinen dann eingesetzt, wenn Standard-LLMs, die sich ausschließlich auf vortrainiertes Wissen stützen, in der Genauigkeit und Tiefe des generierten Textes Mängel aufweisen. Durch die Nutzung externer Datenbanken und Dokumente kann RAG die Qualität der Antworten erheblich verbessern und sicherstellen, dass sie sowohl kontextuell relevant als auch faktisch korrekt sind. Ein solcher Ansatz ist besonders nützlich in

Szenarien, die eine hohe faktische Genauigkeit und aktuelle Informationen erfordern, wie zum Beispiel bei der Beantwortung komplexer Fragen.

Das Konzept von RAG wurde in den vorherigen Abschnitten und Kapiteln bereits mehrfach erwähnt. Der Vollständigkeit halber skizzieren wir hier die wichtigsten Schritte, die bei RAG involviert sind.

- Wir bereiten eine Sammlung von Texten vor, die als zusätzliche Wissensquelle behandelt wird, auf die wir zugreifen können.
- Wir rufen relevante Texte für eine gegebene Anfrage ab.
- Wir geben sowohl die abgerufenen Texte als auch die Anfrage in ein LLM ein, das dann aufgefordert wird, die endgültige Vorhersage zu erstellen.

Die Schritte 1 und 2 können durch die Verwendung eines externen Informationswiedergewinnungssystems implementiert werden. Zum Beispiel können wir die Textsammlung in einer Vektordatenbank speichern und dann die ähnlichsten Texte durch vektorbasierte Suchtechniken abrufen. Da die Informationswiedergewinnung nicht im Mittelpunkt dieses Kapitels steht, gehen wir davon aus, dass solche Systeme als Standardkomponenten verfügbar sind und direkt verwendet werden können.

Hier stellen wir vor, wie man LLMs dazu bringt, abgerufene Texte zu nutzen. Zur Veranschaulichung betrachten wir ein Beispiel, bei dem LLMs verwendet werden, um die folgende Frage zu beantworten.

Where will the 2028 Olympics be held?

Wir können diese Frage einfach in eine Online-Suchmaschine eingeben. Diese gibt dann die relevanten Textabschnitte zurück, die im Internet gefunden wurden, zum Beispiel:

> (Wikipedia)
>
> The 2028 Summer Olympics, officially the Games of the XXXIV Olympiad and commonly known as Los Angeles 2028 or LA28, is an upcoming international multi-sport event scheduled to take place from July 14-30, 2028, in the United States. ...
>
> (The Sporting News)
>
> In 2028, Los Angeles will become the third city, following London and Paris respectively, to host three Olympics after hosting the Summer Games in 1932 and 1984. It will also be the first time the United States has hosted an Olympic Games since the 2002 Winter Games in Salt Lake City. ...
>
> ...

Wir können diese abgerufenen Texte als zusätzlichen Kontext verwenden und ein LLM auffordern, eine Antwort auf der Grundlage dieser Texte zu generieren. Unten sehen Sie ein Beispiel für einen RAG-Prompt.

Your task is to answer the following question. To help you with this, relevant texts are provided. Please base your answer on these texts.

Question:
Where will the 2028 Olympics be held?

Relevant Text 1:
The 2028 Summer Olympics, officially the Games of the XXXIV Olympiad and commonly known as Los Angeles 2028 or LA28 ...

Relevant Text 2:
In 2028, Los Angeles will become the third city, following London and Paris respectively, to host three Olympics after ...

...

The 2028 Olympics will be held in Los Angeles.

Dieser Prompt geht davon aus, dass die bereitgestellten Texte für die Frage relevant sind, und erwartet, dass das LLM eine den Texten getreue Antwort generiert. Das Informationswiedergewinnungssystem kann jedoch manchmal irrelevante oder falsche Texte liefern, was das LLM dazu verleiten kann, eine falsche Antwort zu geben. Eine einfache Möglichkeit, dieses Problem zu beheben, besteht darin, die Genauigkeit des Informationswiedergewinnungssystems zu verbessern. Dennoch können, wie bei den meisten KI-Systemen, Fehler auftreten. Daher ist es auch notwendig, die Robustheit des LLM zu erhöhen, damit es auch bei ungenauen Eingaben vernünftige Vorhersagen treffen kann. Unten sehen Sie einen neuen Prompt, der das LLM dazu befähigt, sich stärker an die Fakten zu halten und ihm erlaubt, die Beantwortung von Fragen zu verweigern, wenn die bereitgestellten Informationen ungenau sind.

Your task is to answer the following question. To help you with this, relevant texts are provided. Please base your answer on these texts.
Please note that your answers need to be as accurate as possible and faithful to the facts. If the information provided is insufficient for an accurate response, you may simply output No answer!.

Question:
Where will the 2028 Olympics be held?

Relevant Text 1:
The 2024 Summer Olympics, officially the Games of the XXXIII Olympiad and branded as Paris 2024, were an international multi-sport event ...

...

No answer!

In diesem Beispiel weigert sich das LLM zu antworten, weil die bereitgestellten Informationen unzureichend und für die Frage irrelevant sind.

Sowohl RAG als auch die Feinabstimmung sind gängige Methoden zur Anpassung von LLMs mithilfe aufgabenspezifischer Daten. Standard-RAG ist trainingsfrei und kann direkt auf LLMs angewendet werden. Um RAG weiter zu verbessern, ist es auch möglich, LLMs feinabzustimmen, obwohl dies einen gewissen Trainingsaufwand erfordert. Zum Beispiel können wir LLMs mithilfe von von Menschen annotierten Daten feinabstimmen, um sie darin zu schulen, die Beantwortung zu verweigern. Beachten Sie, dass RAG, obwohl die oben gezeigten Beispiele einfach erscheinen, nicht trivial ist. Aus der Perspektive des Prompt-Engineerings können unterschiedliche Anwendungsfälle unterschiedliche Prompts erfordern, obwohl unser etwas „gieriges"Ziel darin besteht, eine universelle Prompting-Strategie zu entwickeln, die sich an verschiedene Aufgaben anpassen kann. In vielen Fällen müssen wir kontrollieren, wie stark wir uns auf den abgerufenen Kontext verlassen, um Vorhersagen zu treffen. Manchmal müssen LLMs Antworten strikt aus den bereitgestellten Texten ableiten, während sie zu anderen Zeiten möglicherweise Antworten unter Verwendung ihres vortrainierten Wissens generieren müssen, wenn die bereitgestellten Texte unzureichend sind. Es gibt viele Aspekte von RAG, wie z. B. Verbesserungen an den Abrufsystemen, die in diesem Kapitel nicht behandelt werden können. Interessierte Leser können sich für weitere Informationen auf Übersichtsarbeiten zu RAG-Techniken beziehen [Li et al., 2022; Gao et al., 2023c].

Ein Grund, warum wir RAG hier diskutieren, ist, dass es im Großen und Ganzen als ein Beispiel für den allgemeinen Rahmen der Problemdekomposition betrachtet werden kann (siehe Abschnitt 3.2.2). RAG teilt die Problemlösung in zwei Schritte auf. Im ersten Schritt sammeln wir relevante und unterstützende Informationen für eine gegebene Abfrage aus verschiedenen Wissensquellen. Im zweiten Schritt verwenden wir LLMs, um Antworten auf der Grundlage der gesammelten Informationen zu generieren. Wenn wir das Konzept der Problemdekomposition weiter ausdehnen, werden wir feststellen, dass viele Aufgaben, die die Verwendung externer Systeme oder Werkzeuge erfordern, als ähnliche Probleme behandelt werden können. Ein solches Beispiel ist die Werkzeugnutzung in LLMs. In vielen Anwendungen müssen LLMs externe Datenbanken, APIs und sogar Simulationswerkzeuge einsetzen, um genaue Antworten zu generieren. Zum Beispiel können LLMs auf Echtzeitdaten von den Finanzmärkten zugreifen, um aktuelle Anlageberatung zu geben, oder sich in Gesundheitsdatenbanken integrieren, um personalisierte medizinische Einblicke zu bieten. Diese Integration erweitert die Fähigkeiten von LLMs, indem sie ihnen erlaubt, mit externen Systemen zu interagieren und diese in einigen Kontexten zu beeinflussen oder zu steuern. Folglich fungieren LLMs mehr als autonome Agenten und nicht als bloße Textgeneratoren [Franklin and Graesser, 1996].

Das Thema der Werkzeugnutzung ist breit und umfangreich. Hier beschränken wir unsere Diskussion auf Aufgaben, die durch den Aufruf externer APIs zur Lösung einiger Teilprobleme erleichtert werden können [Parisi et al., 2022; Gao et al., 2023b]. Betrachten wir noch einmal das Beispiel, ein LLM zu bitten, zu antworten: „Wo werden die Olympischen Spiele 2028 stattfinden?". Angenommen, das LLM kann auf ein Websuch-Tool zugreifen. Wir können das LLM dann auffordern, die Frage mithilfe der Websuche zu beantworten, wie folgt:

> Your task is to answer the following question. You may use external tools, such as web search, to assist you.
>
> Question:
> Where will the 2028 Olympics be held?
>
> The information regarding this question is given as follows:
> `{tool: web-search, query: "2028 Olympics"}`
> So the answer is: Los Angeles

Hier zeigt `{tool: web-search, query: "2028 Olympics"}` eine Anfrage an das Websuchsystem mit der Abfrage „2028 Olympics“an. Wenn das LLM diese Zeichenkette sieht, führt es eine Websuche durch und verwendet das Ergebnis, um die Zeichenkette zu ersetzen. In den nachfolgenden Schritten der Vorhersage verwendet das LLM dieses Websuchergebnis dann als Kontext, um die richtige Antwort zu erzeugen.

Betrachten wir ein weiteres Beispiel, bei dem wir das LLM bitten, ein mathematisches Problem zu lösen.

> Problem:
> A swimming pool needs to be filled with water. The pool measures 10 meters in length, 4 meters in width, and 2 meters in depth. Calculate the volume of the pool in cubic meters and then determine how many liters of water are needed to fill it (considering 1 cubic meter equals 1000 liters).
>
> Solution:
> To solve this problem, the LLM needs to first calculate the volume of the pool by using the formula for the volume of a rectangular prism: Length× Width × Depth. Therefore, The volume is 10 m × 4 m × 2 m = `{tool: calculator, expression: 10 * 4 * 2}` m^3. Next, to find out how many liters of water are needed, the LLM multiplies the volume in cubic meters by 1000 (since 1 cubic meter equals 1000 liters). Thus, 80 ×1000 = `{tool: calculator, expression: 80*1000}` liters.

Hier löst die Zeichenkette `{tool: calculator, expression: 10 * 4 * 2}` den Aufruf eines mathematischen Interpreters aus, um das Ergebnis des Ausdrucks zu berechnen. Beachten Sie, dass das Ergebnis (d. h. 80) `{tool: calculator, expression: 10 * 4 * 2}` ersetzen wird und in den folgenden Token-Vorhersagen referenziert werden kann. Zum Beispiel wird im letzten Schritt der Problemlösung 80 anstelle von `{tool: calculator, expression: 10 * 4 * 2}` verwendet.

Ein wesentlicher Unterschied zwischen den hier gezeigten Beispielen für die Werkzeugnutzung und den zuvor besprochenen RAG-Beispielen besteht darin, dass bei der Werkzeugnutzung externe Funktionen während der Inferenz aufgerufen werden können. Im Gegensatz dazu werden bei RAG die abgerufenen Texte vor Beginn des Vorhersageprozesses bereitgestellt. Aus der Perspektive der Sprachmodellierung tun sie jedoch tatsächlich dasselbe: Bevor das Endergebnis generiert wird, verwenden wir externe Werkzeuge, entweder

manuell oder automatisch, um ausreichenden und relevanten Kontext zu erhalten. Eine übergeordnete Interpretation dieser Ansätze ist, dass beide auf einem „Agenten“beruhen, der bestimmen kann, wo und wie externe Funktionen aufgerufen werden sollen, um den für die Vorhersage notwendigen Kontext zu generieren.

Ein Problem bei der Werkzeugnutzung ist, dass die ursprünglichen LLMs nicht darauf trainiert sind, die notwendigen Markierungen für die Werkzeugnutzung zu generieren. Daher müssen wir die LLMs feinabstimmen, um sie an diese Aufgaben anzupassen [Schick et al., 2024]. Da sich dieses Kapitel auf das Prompting konzentriert, werden wir die Details dieses Feinabstimmungsprozesses nicht darstellen. Vereinfacht gesagt, müssen wir zuerst Daten annotieren. Für jedes Feinabstimmungsbeispiel ersetzen wir Teile der Ausgabe, die die Verwendung externer Werkzeuge erfordern, durch vordefinierte Befehle oder Markierungen. Dann verwenden wir diese annotierten Daten, um die Parameter des LLMs wie gewohnt feinabzustimmen. Dadurch kann das LLM die Fähigkeit erlangen, Befehle zum Aufrufen externer Werkzeuge zu generieren. Während der Inferenz können wir diese Befehle zur Werkzeugnutzung in den Modellausgaben ausführen, um Unterstützung von externen Werkzeugen zu erhalten.

3.3 Prompt-Lernen

Bisher haben wir in diesem Kapitel mehrere grundlegende Prompting-Strategien und deren verschiedene Verfeinerungen betrachtet. Allerdings wurden alle von uns besprochenen Prompts manuell entworfen. Dies führt zu einer Reihe von Problemen: Erstens ist der Entwurf hochwertiger Prompts von Natur aus schwierig und erfordert erheblichen manuellen Aufwand. Beispielsweise sind oft umfangreiche Experimente mit verschiedenen Prompts erforderlich, um die effektivsten zu identifizieren. Da verschiedene LLMs möglicherweise besser auf bestimmte Arten von Prompts reagieren, kann die Entwicklung universell wirksamer Prompts noch ressourcenintensiver sein. Zweitens stützt sich der manuelle Prompt-Entwurf stark auf menschliches Fachwissen, was die Vielfalt der Ansätze einschränken und potenziell wirksame Prompts übersehen kann, die für Menschen nicht sofort ersichtlich sind. Drittens können von Menschen erstellte Prompts komplex und redundant sein, was zu längeren Eingaben für LLMs und höheren Rechenkosten führt.

In diesem Abschnitt erörtern wir Techniken für das automatisierte Prompting. Diese Methoden zielen darauf ab, Prompts automatisch zu erstellen, zu optimieren und darzustellen, damit die nachgelagerten Aufgaben effektiver und effizienter bewältigt werden können. Insbesondere betrachten wir hier drei Aspekte.

- Wie können wir den Prozess des Entwerfens und Optimierens von Prompts für LLMs automatisieren?
- Gibt es über Zeichenketten hinaus andere Formen der Darstellung von Prompts, und wie können wir solche Darstellungen lernen?
- Wie können wir Prompts prägnanter und kompakter gestalten und dadurch ihre Komplexität und Länge reduzieren?

Es ist zu beachten, dass es viele Kontexte gibt, in denen wir diese Aspekte untersuchen können. Beispielsweise könnten wir festlegen, dass Prompts speziell für ein bestimmtes LLM entwickelt werden oder dass die Entwicklung unabhängig vom verwendeten LLM erfolgt. Diese Kontexte können zu unterschiedlichen Methoden und Anwendungsszenarien führen, aber diese Methoden können sich in gewisser Weise überschneiden. In der folgenden Diskussion werden wir mehrere verschiedene Szenarien behandeln und die Verbindungen zwischen den verschiedenen Methoden erörtern.

3.3.1 Prompt-Optimierung

Da das Prompt-Design schwierig und arbeitsintensiv ist, ist es wünschenswert, Modelle des maschinellen Lernens zu verwenden, um den optimalen Prompt für eine bestimmte Aufgabe zu finden (dies wird als automatisches Prompt-Design oder Prompt-Optimierung bezeichnet). Dieser Ansatz kann im Großen und Ganzen als ein Beispiel für automatisiertes maschinelles Lernen (AutoML) betrachtet werden, das darauf abzielt, die Notwendigkeit eines von Experten gesteuerten manuellen Designs von Modellen des maschinellen Lernens zu reduzieren oder zu eliminieren. Obwohl unser Fokus hier auf dem Design von Prompts liegt, sind Prompts selbst diskrete Strukturen. Daher ist das Entwerfen von Prompts dem Entwerfen von Modellen des maschinellen Lernens, wie z. B. diskreten Modellarchitekturen, sehr ähnlich. Eines der vielleicht engsten verwandten Gebiete ist die Suche nach neuronalen Architekturen (NAS), bei der die optimalsten neuronalen Netzwerke durch die Erkundung eines Raums möglicher neuronaler Netzwerke identifiziert werden [Zoph and Le, 2016; Elsken et al., 2019]. Wenn wir die Prompt-Optimierung als einen Suchprozess betrachten, können wir ein allgemeines Framework für die Prompt-Optimierung beschreiben, das die folgenden Komponenten umfasst:

- Prompt-Suchraum. Dies definiert alle möglichen Prompts, die die Algorithmen untersuchen können. Zum Beispiel kann man einige Seed-Prompts bearbeiten, um eine Reihe vielfältiger Kandidaten-Prompts zu generieren.
- Leistungsschätzung. Sobald ein Prompt ausgewählt wurde, muss er bewertet werden. Eine einfache Methode besteht beispielsweise darin, ihn in ein LLM einzugeben und dessen Leistung anhand eines Validierungsdatensatzes zu messen.
- Suchstrategie. Der Suchprozess ist im Allgemeinen derselbe wie der, der in vielen KI-Systemen verwendet wird. In jedem Schritt untersucht das System eine Reihe vielversprechender Prompts im Suchraum und bewertet diese. Dieser Prozess wird fortgesetzt, während weitere Prompts untersucht werden. Das Ergebnis der Suche ist der leistungsstärkste Prompt, der bis zum Abschluss der Suche beobachtet wurde.

Dies ist ein sehr allgemeines Framework, und verschiedene Systeme zur Prompt-Optimierung können sich im Design jeder Komponente unterscheiden. Ein weit verbreiteter Ansatz ist die Verwendung von LLMs als Grundlage für die Entwicklung dieser Komponenten. Anfänglich werden einige Prompts bereitgestellt. Anschließend wird der folgende Prozess iteriert, bis ein Abbruchkriterium erfüllt ist: 1) Die Prompts werden auf einem Validierungsdatensatz ausgewertet; 2) ein Kandidatenpool wird gepflegt, indem nur die vielversprechendsten Prompts beibehalten werden; und 3) neue Prompts werden erstellt, indem

LLMs eingesetzt werden, um ähnliche Prompts aus diesem Kandidatenpool abzuleiten. Ein Vorteil dieses Ansatzes ist, dass er es uns ermöglicht, handelsübliche LLMs zu verwenden, um die oben genannten Aufgaben ohne die Notwendigkeit einer umfangreichen Systementwicklung durchzuführen. Um dies zu erreichen, können wir LLMs prompten oder feinabstimmen, um sie an diese Aufgaben anzupassen. Hier betrachten wir die Methode von Zhou et al. [2023c] zur Veranschaulichung der LLM-basierten Prompt-Optimierung. Sie umfasst die folgenden Schritte.

- Initialisierung. Sei C der Pool der Kandidaten-Prompts, die wir untersuchen wollen. Der erste Schritt besteht darin, anfängliche Prompts zu C hinzuzufügen. Dies kann auf verschiedene Weisen geschehen. Eine einfache Methode ist es, solche Prompts für eine gegebene Aufgabe manuell zu erstellen. In vielen Fällen jedoch, in denen Menschen nur begrenztes Wissen darüber haben, wie man effektive Prompts für die Aufgabe schreibt, wird die Entwicklung von Prompts zu einer Herausforderung. In diesen Fällen ist es wünschenswert, LLMs zur Generierung von Prompts zu verwenden. Zum Beispiel können wir LLMs direkt anweisen, Prompts zu erstellen, indem wir ihnen eine Beschreibung der Aufgabe geben.

  > You are given a task to complete using LLMs. Please write a prompt to guide the LLMs.
  >
  > {∗task-description∗}
  >
  > ____

  Diese Methode ist einfach, erfordert aber dennoch eine von Menschen bereitgestellte Beschreibung der Aufgabe. Eine alternative Methode besteht darin, LLMs zu verwenden, um Prompts anhand von Beispielen für die Eingabe und Ausgabe der Aufgabe zu generieren. Hier ist eine Prompt-Vorlage.

  > You are provided with several input-output pairs for a task. Please write an instruction for performing this task.
  >
  > Input: {∗input1∗} Output: {∗output1∗}
  > Input: {∗input2∗} Output: {∗output2∗}
  > ...
  >
  > ____

  Auf diese Weise können LLMs die entsprechende Anweisung für die Aufgabe aus den bereitgestellten Ein- und Ausgaben ableiten.

- Bewertung. Sobald wir den Kandidatenpool C erhalten haben, müssen wir die Prompts in C bewerten. Eine Methode besteht darin, jeden Prompt in ein LLM einzuspeisen und die Ergebnisse bei der nachgelagerten Aufgabe zu bewerten. Zum Beispiel können wir die Ausgabe des LLM für eine gegebene Eingabe anhand einer vordefinierten Metrik bewerten oder alternativ die Log-Likelihood der Ausgabe als Maß für die Qualität des Prompts verwenden.

- Pruning. Wenn C eine große Anzahl von Prompts enthält, ist es sinnvoll, die wenig aussichtsreichen Prompts darin zu entfernen (prunen), um so den Rechenaufwand in den nachfolgenden Schritten zu reduzieren. Dies ist ein Standard-Pruning-Problem. Angesichts der Bewertungspunktzahl für jeden Prompt besteht eine einfache Methode darin, nur einen bestimmten Prozentsatz der Prompts beizubehalten und den Rest zu verwerfen.

- Erweiterung. Die Erweiterung ist eine Schlüsseloperation in Suchalgorithmen, die verwendet wird, um verschiedene Zustände im Suchraum zu erkunden. Die Erweiterungsoperation kann hier als eine Funktion definiert werden

$$C' = \text{Expand}(C, f) \tag{3.10}$$

  wobei C' die Menge der neuen Prompts ist, die aus C unter Verwendung des Modells f generiert werden. Wenn wir f als ein LLM betrachten, können wir die Erweiterungsoperation durchführen, indem wir f anweisen, neue und relevante Prompts basierend auf C zu generieren. Unten folgt ein Beispiel.

  > Below is a prompt for an LLM. Please provide some new prompts to perform the same task.
  >
  > Input: {∗prompt∗}
  >
  > ——

  Dann ersetzen wir C durch C'. Die Schritte der Bewertung, des Prunings und der Erweiterung können wiederholt werden, sodass wir nach und nach ein breiteres Spektrum an Prompts erkunden können.

Bei der Prompt-Optimierung spielt der Expansionsschritt eine Schlüsselrolle, da er definiert, wie wir den Suchraum erkunden, und unser Ziel ist es, mit minimalem Aufwand optimale Ergebnisse zu finden. Eine Verbesserung dieses Schrittes besteht darin, das Problem als eine Paraphrasierungsaufgabe zu behandeln. Eine einfache Methode ist die Anwendung von handelsüblichen Paraphrasierungssystemen, die entweder auf LLMs oder anderen Modellen basieren, um Eingabe-Prompts in semantisch äquivalente Formen umzuwandeln [Jiang et al., 2020]. Alternativ können wir für jeden token spezifische Bearbeitungsoperationen wie Einfügungen und Änderungen definieren. Ein gegebener Prompt kann durch Anwendung dieser Operationen in neue Prompts umgewandelt werden [Prasad et al., 2023]. Außerdem können weitere Auswertungen und Bereinigungen angewendet werden, um Prompts von geringer Qualität herauszufiltern. Zusätzlich zur Formulierung der Prompt-Generierung als Paraphrasierungsproblem können wir die Qualität der Prompts während der Expansion durch das Lernen aus Feedback verbessern [Pryzant et al., 2023]. Dieser Ansatz steht in gewissem Zusammenhang mit dem in Abschnitt 3.2.3 erörterten Problem der Selbstverfeinerung. Ein LLM kann verwendet werden, um Feedback zu einem Eingabe-Prompt zu generieren, der dann auf der Grundlage dieses Feedbacks überarbeitet wird. Dieser Feedback-und-Revisionszyklus kann mehrmals wiederholt werden, bis das Ergebnis konvergiert oder das gewünschte Ergebnis erreicht ist.

Ein weiterer Ansatz zur Prompt-Optimierung ist die Anwendung klassischer Optimierungstechniken. Zum Beispiel kann das Problem als ein evolutionäres Berechnungsproblem formuliert werden, bei dem Prompts als Kandidaten behandelt werden, die sich von Generation zu Generation weiterentwickeln, während die Optimierung fortschreitet [Guo et al., 2024]. Da in verwandten Bereichen viele leistungsstarke Optimierungsalgorithmen entwickelt wurden, können diese direkt auf dieses Problem angewendet werden.

In der Praxis könnten wir versucht sein, bestehende LLM-APIs zu verwenden, um die oben beschriebenen Schritte zu implementieren. Ein solcher Ansatz wäre jedoch stark von den Inferenz- und In-Kontext-Lernfähigkeiten der LLMs abhängig. Wenn diese LLMs nicht leistungsstark sind und es ihnen an Anpassung an die Aufgaben mangelt, können sie Fehler in die Suche einbringen, zum Beispiel durch die Generierung falscher Prompts während der Expansion. In solchen Fällen ist es vorzuziehen, Modelle zu trainieren, die besser für die Aufgaben geeignet sind. Ein Ansatz in dieser Forschungsrichtung greift auf das bestärkende Lernen zurück, das weithin zur Lösung von diskreten Entscheidungs- und Optimierungsproblemen eingesetzt wird. Zum Beispiel entwickelten Deng et al. [2022] einen Prompt-Generator durch die Integration eines FFN-basierten Adapters in ein LLM. Der Prompt-Generator wird als typisches Policy-Netzwerk trainiert, aber nur die Parameter des Adapters werden aktualisiert, während die übrigen Parameter des Modells unverändert bleiben. Während des Trainings wird die Belohnung durch das Testen der generierten Prompts mit einem anderen LLM ermittelt, ähnlich der oben beschriebenen Bewertungsmethode. Sobald das Training abgeschlossen ist, wird der Prompt-Generator dann eingesetzt, um neue Prompts zu generieren.

Beachten Sie, dass in unserer Diskussion hier Prompts einfach als Sequenzen von tokens betrachtet werden, und die Ausgabe der Prompt-Optimierung eine solche Sequenz ist. Im engeren Sinne haben Prompts jedoch komplexe Strukturen und umfassen verschiedene Felder wie Benutzereingabe, Anweisung und Demonstration. Obwohl die von uns besprochenen Ansätze meist allgemein sind, hat sich ein Großteil der Arbeit in der Prompt-Optimierung auf das Erlernen besserer Anweisungen für das Prompting konzentriert. Insbesondere ist das Ziel, Anweisungen zu generieren, die LLMs auf der Grundlage einer gegebenen Aufgabe effektiv leiten. Natürlich kann das Konzept der Prompt-Optimierung auch auf das Erlernen anderer Teile von Prompts erweitert werden. Zum Beispiel gab es ein erhebliches Forschungsinteresse daran zu lernen, wie man Demonstrationen in CoT auswählt oder generiert [Liu et al., 2022; Rubin et al., 2022; Zhang et al., 2023b]. Einer der Unterschiede zwischen dem Erlernen von Anweisungen und dem Erlernen von Demonstrationen besteht darin, dass die Erzeugung hochwertiger Demonstrationen mit LLMs relativ einfach ist und der Fokus beim Erlernen von Demonstrationen typischerweise darauf liegt, wie man geeignete Demonstrationen aus einem Pool von Kandidaten auswählt. Im Gegensatz dazu besteht die Schwierigkeit beim Erlernen von Anweisungen teilweise darin, dass vortrainierte LLMs nicht geeignet sind, die Qualität von Anweisungen vorherzusagen, und das Testen dieser Anweisungen auf nachgelagerten Aufgaben rechenintensiv ist. Dies macht die Anwendung der Optimierungsmethoden kostspielig, und die Erforschung einer Vielzahl von Anweisungen stellt eine erhebliche Herausforderung dar.

3.3.2 Weiche Prompts

Obwohl die Entwicklung von Prompts in natürlicher Sprache, sei es manuell oder automatisch, ein unkomplizierter und weit verbreiteter Ansatz ist, birgt er einige Probleme. Ein Problem besteht darin, dass Prompts in natürlicher Sprache komplex und lang sein können, was zu einer erheblichen Rechenlast bei der Verarbeitung durch LLMs führt. In vielen Anwendungen müssen Benutzer eine Aufgabe möglicherweise wiederholt ausführen, und die wiederholte Eingabe desselben langen Prompts in die LLMs ist eindeutig ineffizient. Ein weiteres Problem besteht darin, dass Prompts in der regulären LLM-Eingabe typischerweise als diskrete Token-Sequenzen (sogenannte harte Prompts) dargestellt werden, die LLMs sie jedoch als niedrigdimensionale reellwertige Vektoren kodieren. Dies wirft die Frage auf, ob es kompaktere und effizientere Möglichkeiten gibt, Prompts darzustellen.

In diesem Unterabschnitt führen wir das Konzept der weichen Prompts ein, die als verborgene, verteilte Darstellungen von Prompts betrachtet werden können. Beim Prompting von LLMs geht es uns darum, Aufgaben oder Fragen zu kommunizieren, um die gewünschten Antworten zu erhalten. Wir können harte Prompts als explizite, vordefinierte Textsequenzen definieren, die Benutzer direkt in LLMs eingeben, um die Antworten zu steuern. Im Gegensatz dazu können wir uns weiche Prompts als implizite, anpassungsfähige Prompting-Muster vorstellen, die in LLMs eingebettet sind. Im Gegensatz zu harten Prompts, die in natürlicher Sprache ausgedrückt werden und für Menschen verständlich sein sollten, werden weiche Prompts in einem Format kodiert, das für das Modell verständlicher ist als für den Menschen. Zur Veranschaulichung betrachten wir einen einfachen Prompt

> Translate the sentence into Chinese.
>
> Consider it done!
>
> ——

Hier kann die Anweisung „Übersetze den Satz ins Chinesische“als harter Prompt angesehen werden, der durch die Token-Sequenz $c_1...c_5$ bezeichnet wird. Indem diese Tokens in ein LLM eingespeist werden, werden sie in eine Sequenz von reellwertigen Vektoren $\mathbf{h}_1...\mathbf{h}_5$ umgewandelt, wobei jeder einem Token entspricht. Wir können uns $\mathbf{h}_1...\mathbf{h}_5$ grob als einen weichen Prompt vorstellen, wie in Abbildung 3.3 dargestellt.

Obwohl das obige Beispiel zeigt, dass weiche Prompts durch die Umwandlung von harten Prompts erzeugt werden können, gibt es nicht notwendigerweise eine direkte Entsprechung zwischen ihnen. Tatsächlich müssen wir weiche Prompts nicht einmal mit sinnvollem Text interpretieren. Sie sind stattdessen einfach verborgene Zustände in LLMs und können als Standardparameter der Modelle durch kontinuierliche Optimierung gelernt werden. Eine solche Behandlung ermöglicht es uns, Prompting-Methoden jenseits von Text zu erforschen. Als weiterer Vorteil bieten weiche Prompts dichte, niedrigdimensionale und lernbare Darstellungen zur Kodierung, wie wir LLMs anleiten, spezifische Ausgaben zu erzeugen. Das Training und die Anwendung dieser Darstellungen erfordern deutlich geringere Rechenkosten als die Verarbeitung langer harter Prompts. Dieser Ansatz wäre von großem praktischen Wert bei LLM-Inferenzanwendungen, bei denen derselbe Prompt wiederholt verwendet wird.

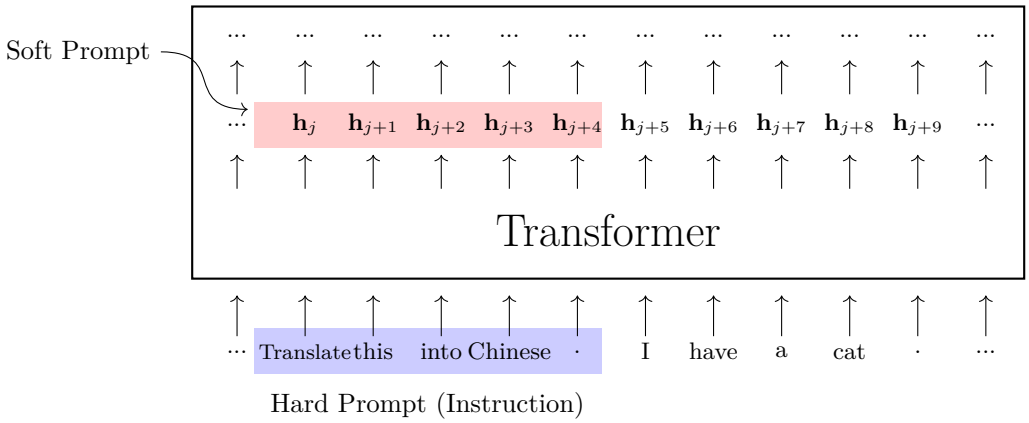


Abbildung 3.3: Veranschaulichung von harten und weichen Prompts. Hier ist der harte Prompt die Anweisung, die wir dem LLM zur Durchführung der Aufgabe eingeben. Das LLM kodiert diese Anweisung wie üblich, und die Zwischenrepräsentationen, die der Anweisung entsprechen, können als eine Art weicher Prompt betrachtet werden.

3.3.2.1 Anpassung von LLMs mit weniger Prompting

Eine naheliegende Möglichkeit, ein LLM für eine bestimmte Aufgabe anzupassen, besteht darin, das Modell einfach mithilfe von gelabelten Daten zu feintunen. Dies führt zu einer Vielzahl von LLM-Abstimmungsmethoden, wie z. B. dem überwachten Fine-Tuning, bei dem die Modellparameter aktualisiert werden, indem die Antworten auf gegebene Prompts mit Überwachungssignalen abgeglichen werden. Feingetunte LLMs betten aufgabenbezogene Informationen in die Modellparameter ein, und daher können diese Modelle korrekt reagieren, wenn sie mit ähnlichen Prompts wie beim Fine-Tuning konfrontiert werden.

Wenn wir diese Idee weiterverfolgen, können wir erwarten, dass LLMs während des Fine-Tunings so viel Wissen wie möglich über das Prompting einer Aufgabe aufnehmen. Folglich werden die Prompting-Informationen teilweise in den Modellparametern erfasst, und die feingetunten LLMs können die Aufgabe mit weniger Prompting ausführen. Hier betrachten wir eine einfache Form des Prompts, bei der nur eine Anweisung (bezeichnet mit $\mathbf{c}$) und eine Benutzereingabe (bezeichnet mit $\mathbf{z}$) enthalten sind. Ein Prompt kann durch das folgende Tupel ausgedrückt werden

$$\mathbf{x} \quad = \quad (\mathbf{c}, \mathbf{z}) \tag{3.11}$$

Gegeben sei ein Satz von Prompt-Antwort-Paaren $\mathcal{D} = \{(\mathbf{x}, \mathbf{y})\}$. Das Ziel des Fine-Tunings ist es, den Gesamtverlust über diesen Satz zu minimieren. Eine gängige Methode ist die Minimierung der negativen Log-Likelihood (d. h. die Maximierung der Log-Likelihood) in Bezug auf die Modellparameter θ:

$$\begin{aligned} \hat{\theta} &= \arg\max_{\theta} \sum_{(\mathbf{x},\mathbf{y})\in\mathcal{D}} \log \Pr_\theta(\mathbf{y}|\mathbf{x}) \\ &= \arg\max_{\theta} \sum_{(\mathbf{x},\mathbf{y})\in\mathcal{D}} \log \Pr_\theta(\mathbf{y}|\mathbf{c},\mathbf{z}) \end{aligned} \tag{3.12}$$

wobei $\Pr_\theta(\cdot|\cdot)$ die von einem LLM mit den Parametern θ vorhergesagte Wahrscheinlichkeit

ist[9].

Im Allgemeinen sollte die Anweisung in jedem Fine-Tuning-Beispiel den Richtlinien des Prompt-Designs folgen. Zum Beispiel sollte eine gute Anweisung so klar wie möglich sein und eine detaillierte Beschreibung der Aufgabe liefern. Die in der obigen Gleichung beschriebene Methode schränkt die Anweisung jedoch nicht auf eine bestimmte Form ein. Diese Flexibilität ermöglicht es uns, LLMs auf jede gewünschte Weise anzuweisen. Betrachten wir ein Beispiel, in dem wir LLMs anweisen wollen, einen englischen Satz ins Chinesische zu übersetzen. Natürlich können wir, wie bereits in diesem Kapitel erwähnt, LLMs mit der folgenden Anweisung anweisen

Translate the following sentence from English to Chinese.

Wenn wir die Anweisung einfacher gestalten wollen, können wir sie in eine einfachere Form umformulieren

Translate this into Chinese.

Wir können die Anweisung sogar als einzelne Phrase definieren

Translate!

Mit einem gewissen Fine-Tuning-Aufwand können wir LLMs so anpassen, dass sie jede dieser Anweisungen befolgen. Aus der Perspektive eines effizienten Promptings ergeben sich rechnerische Vorteile bei der Vereinfachung von Anweisungen im Prompting. Zum Beispiel können wir einfache Anweisungen wie „Übersetzen!“verwenden, um Aufgaben auszuführen, die typischerweise komplexere und detailliertere Anweisungen erfordern würden. Dies kann das nachfolgende Prompting während der Inferenz erheblich erleichtern. Andererseits kann das Fine-Tuning von LLMs mit übermäßig vereinfachten Anweisungen der Generalisierung der Modelle schaden. Da vereinfachte Anweisungen zu einem Informationsverlust führen können, ist es wahrscheinlicher, dass die LLMs die Fine-Tuning-Daten überanpassen und nicht über diese Anweisungen hinaus generalisieren können. In Szenarien, die sowohl komplexe als auch vereinfachte Anweisungen für das Fine-Tuning beinhalten, ist dieses Problem schwerwiegender, da die für das Fine-Tuning verfügbaren gelabelten Daten in der Regel begrenzt sind und die Berücksichtigung einer Vielzahl von Anweisungen kostspielig ist.

Eine alternative Methode zur Anpassung von LLMs an vereinfachte Anweisungen ist die Wissensdestillation. Als Beispiel betrachten wir die Kontext-Destillationsmethode [Snell et al., 2022]. Das Ziel dieser Methode ist es, ein Schülermodell zu lernen, das vereinfachte Anweisungen von einem gut trainierten, anweisungsfolgenden Lehrermodell nutzen kann. Abbildung 3.4 zeigt eine Veranschaulichung dieses Ansatzes. Der Aufbau des Lehrermodells folgt einem standardmäßigen Fine-Tuning-Prozess: Wir sammeln zunächst eine

[9] In der Praxis initialisieren wir θ mit den aus dem Pre-Training erhaltenen Parametern und passen θ dann moderat an, um sicherzustellen, dass die Ergebnisse nach dem Fine-Tuning nicht zu stark von den vortrainierten Ergebnissen abweichen.

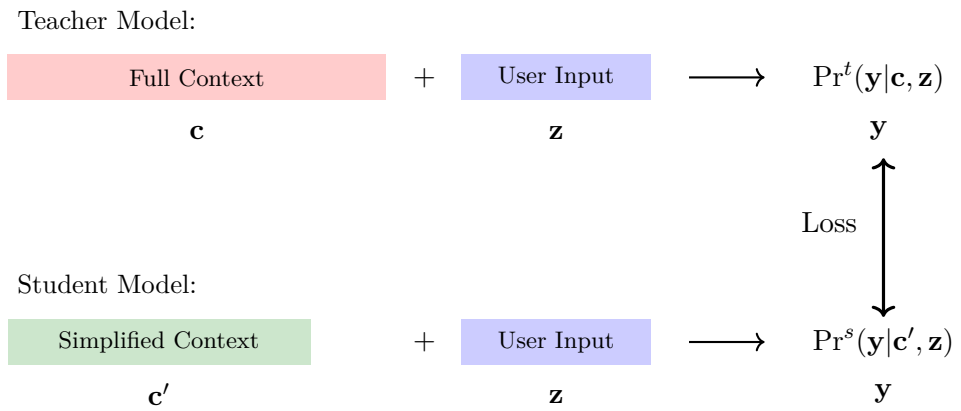


Abbildung 3.4: Veranschaulichung der Kontext-Destillation [Snell et al., 2022]. Das Lehrermodell ist ein Standard-LLM, das sowohl den Kontext als auch die Benutzereingabe als Modelleingabe verwendet und eine Vorhersage als Modellausgabe erzeugt. Anschließend vereinfachen wir den Kontext (z. B. durch Vereinfachung der Anweisung beim Prompting) und verwenden das Schülermodell, um Vorhersagen basierend auf dem vereinfachten Kontext und der Benutzereingabe zu treffen. Das Schülermodell wird trainiert, indem der Verlust zwischen den von den beiden Modellen erzeugten Vorhersagen minimiert wird.

bestimmte Menge an Daten, die Anweisungen, Benutzereingaben und korrekte Antworten enthalten, und trainieren dann ein vortrainiertes Modell mit diesem Datensatz weiter. Für den Aufbau des Schülermodells müssen wir einen neuen Datensatz $\mathcal{D}'$ erstellen, bei dem jeder Stichprobe ein Tupel ist, das aus einer Anweisung, einer entsprechenden vereinfachten Anweisung und einer Benutzereingabe besteht, bezeichnet mit $\mathbf{x}' = (\mathbf{c}, \mathbf{c}', \mathbf{z})$. Die Wissensdestillation wird durch Minimierung einer Verlustfunktion durchgeführt, die auf den Ausgaben des Lehrer- und Schülermodells definiert ist

$$\hat{\theta} \quad = \quad \arg\min_{\theta} \sum_{\mathbf{x}' \in \mathcal{D}'} \text{Loss}(\Pr^t(\cdot|\cdot),\ \Pr^s_\theta(\cdot|\cdot),\ \mathbf{x}') \tag{3.13}$$

wobei $\Pr^t(\cdot|\cdot)$ das vortrainierte Lehrermodell und $\Pr^s_\theta(\cdot|\cdot)$ das Schülermodell mit den Parametern θ bezeichnet. Um die Notation einfach zu halten, schreiben wir kurz $\text{Loss}(\Pr^t(\cdot|\cdot),\ \Pr^s_\theta(\cdot|\cdot),\ \mathbf{x})$ als Loss. Ein häufig verwendeter Verlust ist der Verlust auf Sequenzebene, der die folgende Grundform hat:

$$\text{Loss} \quad = \quad \sum_{\mathbf{y}} \Pr^t(\mathbf{y}|\mathbf{c},\mathbf{z}) \log \Pr^s_\theta(\mathbf{y}|\mathbf{c}',\mathbf{z}) \tag{3.14}$$

Diese Funktion ist jedoch rechnerisch nicht durchführbar, da sie eine Summierung über eine exponentiell große Anzahl von Ausgaben erfordert. Eine Variante dieser Methode besteht darin, das Schülermodell mit den vom Lehrermodell generierten Ausgaben zu trainieren. Für jede Stichprobe verwenden wir das Lehrermodell, um eine Ausgabe $\hat{\mathbf{y}} = \arg\max_{\mathbf{y}} \log \Pr^t(\mathbf{y}|\mathbf{c},\mathbf{z})$ zu erzeugen. Dann betrachten wir $\hat{\mathbf{y}}$ als das Lernziel, und die Verlustfunktion ist gegeben durch

$$\text{Loss} \quad = \quad \log \Pr^s_\theta(\hat{\mathbf{y}}|\mathbf{c}',\mathbf{z}) \tag{3.15}$$

Alternativ können wir die Abstände zwischen den von den beiden Modellen ausgegebenen Wahrscheinlichkeitsverteilungen minimieren [Askell et al., 2021]. Zum Beispiel kann die Verlustfunktion als die KL-Divergenz zwischen den beiden Ausgabeverteilungen definiert werden

$$\text{Loss} = \text{KL}(\text{P}^t \mid\mid \text{P}^s_\theta) \tag{3.16}$$

wobei

$$\text{P}^t = \text{Pr}^t(\cdot|\mathbf{c}, \mathbf{z}) \tag{3.17}$$
$$\text{P}^s_\theta = \text{Pr}^s_\theta(\cdot|\mathbf{c}', \mathbf{z}) \tag{3.18}$$

Obwohl wir uns auf die Wissensdestillation für Anweisungen beschränkt haben, sind die hier diskutierten Ansätze allgemein gültig. Durch das Lernen aus den Ausgaben des Lehrermodells kann das Wissen im Prompting in die Parameter des Schülermodells destilliert werden. Daher kann das destillierte Modell als eine Art weichen Prompt kodierend betrachtet werden. Diese Methode kann auf viele andere Probleme im Prompt-Lernen angewendet werden, wie z. B. das Komprimieren langer Kontexte und das Lernen weicher Prompts als spezifische Komponenten von LLMs.

3.3.2.2 Lernen von Soft Prompts für parametereffizientes Fine-Tuning

Die Aktualisierung aller Parameter ist eine gängige Methode, um LLMs an Aufgaben von Interesse anzupassen. Obwohl Fine-Tuning als rechengünstiger als das Vortraining gilt, ist seine Anwendung in der Praxis dennoch kostspielig. Dieses Problem motiviert die Entwicklung von parametereffizienten Fine-Tuning-Methoden, die darauf abzielen, die Anzahl der zu aktualisierenden Parameter zu minimieren.

Ein Ansatz, bekannt als prefix fine-tuning, besteht darin, eine Reihe von trainierbaren Vektoren oder Präfixen an den Anfang der Eingabe jeder Transformer-Schicht anzuhängen [Li and Liang, 2021]. Diese Präfixe können als Soft Prompts betrachtet werden, die als zusätzlicher Kontext dienen, um das Verhalten des Modells bei spezifischen Aufgaben zu steuern. Während des Fine-Tunings müssen wir nur die Präfixe lernen, um aufgabenspezifisches Wissen einzubetten. Daher ist diese Methode effizient, da sie nur einen kleinen Teil des Modells modifiziert, anstatt den gesamten Satz von Modellparametern anzupassen.

Genauer gesagt, sei die Eingabe einer Schicht in der Tiefe l mit $\mathbf{H}^l = \mathbf{h}^l_0\mathbf{h}^l_1...\mathbf{h}^l_m$ bezeichnet. Die Ausgabe der Schicht kann ausgedrückt werden als

$$\mathbf{H}^{l+1} = \text{Layer}(\mathbf{H}^l) \tag{3.19}$$

Beim Prefix-Fine-Tuning erweitern wir die Sequenz $\mathbf{h}^l_0\mathbf{h}^l_1...\mathbf{h}^l_m$, indem wir am Anfang einige Vektoren hinzufügen, die wir als $\mathbf{p}^l_0\mathbf{p}^l_1...\mathbf{p}^l_n$ bezeichnen. Daher kann $\mathbf{H}^l$ in der Form geschrieben werden

$$\mathbf{H}^l = \underbrace{\mathbf{p}^l_0\ \mathbf{p}^l_1\ ...\ \mathbf{p}^l_n}_{\text{trainierbar}} \quad \underbrace{\mathbf{h}^l_0\ \mathbf{h}^l_1\ ...\ \mathbf{h}^l_m}_{\text{Ausgabe der vorherigen Schicht}} \tag{3.20}$$

Die Ausgabe der Schicht sind die letzten $m+1$ Repräsentationen.

$$\begin{aligned} \overline{\mathbf{H}}^{l+1} &= \text{Layer}(\mathbf{H}^l)[-m-1:] \\ &= \mathbf{h}_0^{l+1}\mathbf{h}_1^{l+1}...\ \mathbf{h}_m^{l+1} \end{aligned} \tag{3.21}$$

wobei $[-m-1:]$ die Slicing-Operation bezeichnet, die die letzten $m+1$ Elemente einer Sequenz extrahiert. Gegeben $\overline{\mathbf{H}}^{l+1}$, kann die Eingabe der nächsten Schicht in der gleichen Form wie in Gl. (3.20) ausgedrückt werden:

$$\begin{aligned} \mathbf{H}^{l+1} &= \mathbf{p}_0^{l+1}\ \mathbf{p}_1^{l+1}\ ...\ \mathbf{p}_n^{l+1}\ \overline{\mathbf{H}}^{l+1} \\ &= \mathbf{p}_0^{l+1}\ \mathbf{p}_1^{l+1}\ ...\ \mathbf{p}_n^{l+1}\mathbf{h}_0^{l+1}\mathbf{h}_1^{l+1}...\ \mathbf{h}_m^{l+1} \end{aligned} \tag{3.22}$$

Hier kann jedes $\mathbf{p}_i \in \mathbb{R}^d$ als ein lernbarer Parameter betrachtet werden. Während des Trainings werden $\mathbf{p}_0^l\mathbf{p}_1^l...\mathbf{p}_n^l$ wie üblich trainiert, und die Parameter des ursprünglichen Transformer-Modells bleiben fixiert.

Abbildung 3.5 zeigt eine Illustration des Prefix-Fine-Tunings für eine Übersetzungsaufgabe. Hier werden nur die Präfixvektoren $\mathbf{p}_0^l$ und $\mathbf{p}_1^l$ aktualisiert, indem sie die Fehlergradienten aus der Ausgabe (d.h. der chinesischen Übersetzung) erhalten. Durch die Anpassung dieser Vektoren für die Übersetzungsaufgabe passt sich das Modell entsprechend an. Dadurch dienen $\mathbf{p}_0^l$ und $\mathbf{p}_1^l$ als Prompts, die das LLM aktivieren, um die Aufgabe auszuführen, ohne explizite Eingabe-Prompts wie „Übersetze den folgenden Satz vom Englischen ins Chinesische“zu benötigen. Zur Testzeit stellen wir die optimierten $\mathbf{p}_0^l$ und $\mathbf{p}_1^l$ der Schicht voran, und das LLM übersetzt dann den Eingabesatz. Beachten Sie, dass das Prefix-Fine-Tuning zusätzliche $L \times n \times d$ Parameter einführt, wobei L die Anzahl der Schichten, n die Anzahl der Präfixe und d die Dimensionalität jedes Präfixes ist. Diese Anzahl ist jedoch im Vergleich zur Gesamtzahl der Parameter im LLM viel kleiner, was den Fine-Tuning-Prozess hocheffizient macht.

Obwohl das Prefix-Fine-Tuning einfach ist, erfordert es dennoch Modifikationen an den LLMs. Alternativ ermöglicht uns die Trennung von Soft Prompts von den LLMs, die ursprüngliche Modellarchitektur beizubehalten, was den Einsatz für verschiedene Aufgaben effizienter macht, ohne dass das Kernmodell angepasst werden muss. Eine solche Methode ist das Prompt-Tuning [Lester et al., 2021]. Wie das Prefix-Fine-Tuning integriert das Prompt-Tuning trainierbare Vektoren, sodass sich LLMs durch Anpassung dieser Vektoren an gegebene Aufgaben anpassen können. Das Prompt-Tuning unterscheidet sich jedoch dadurch, dass es nur die Einbettungsschicht modifiziert.

Erinnern Sie sich daran, dass in LLMs jedes Eingabe-Token z_i durch eine Einbettung $\mathbf{e}_i$ repräsentiert wird. Diese Einbettungen werden im Allgemeinen durch ein Token-Einbettungsmodell gelernt und dann als die tatsächlichen Eingaben für die LLMs verwendet, wobei sie die symbolisch dargestellten Token ersetzen. Beim Prompt-Tuning werden eine Reihe von Pseudo-Einbettungen $\mathbf{p}_0...\mathbf{p}_n$ an den Anfang der Token-Einbettungssequenz hinzugefügt. Die tatsächliche Eingabe für die LLMs kann also ausgedrückt werden als

$$\underbrace{\mathbf{p}_0\ \mathbf{p}_1\ ...\ \mathbf{p}_n}_{\text{trainable}}\ \underbrace{\mathbf{e}_0\ \mathbf{e}_1\ ...\ \mathbf{e}_m}_{\text{token embeddings}}$$

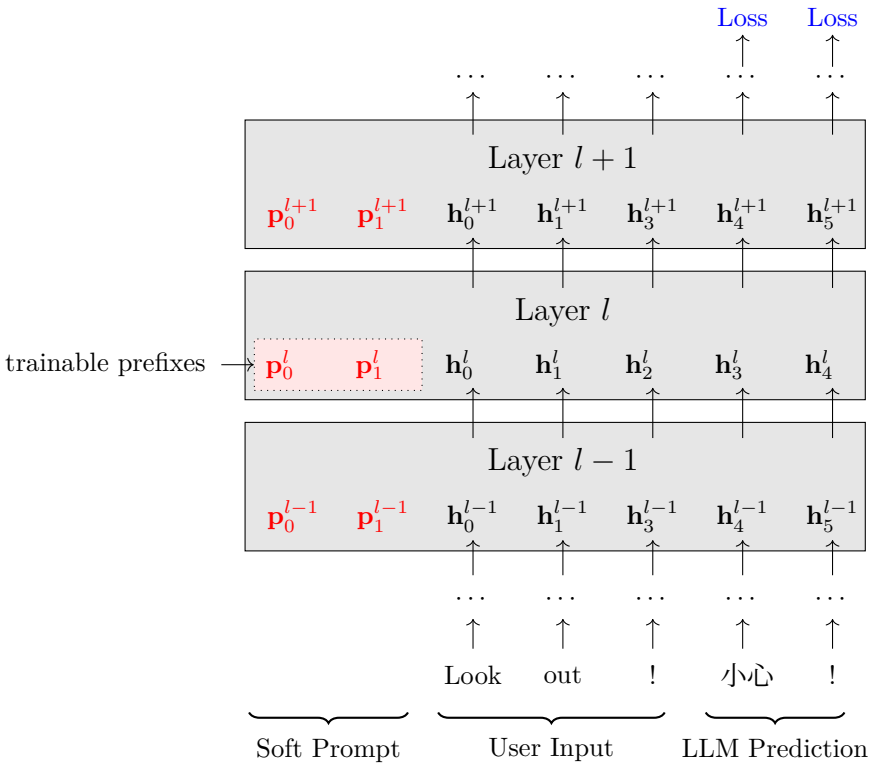


Abbildung 3.5: Veranschaulichung der Präfix-Feinabstimmung für eine Übersetzungsaufgabe (Vorsicht! → 小心!). Für jede Schicht fügen wir zwei Präfixe $\mathbf{p}_0^l$ und $\mathbf{p}_1^l$ am Anfang hinzu. Das LLM wird trainiert, um den Verlust bei den Vorhersagen basierend auf der Eingabe zu minimieren. Während dieses Prozesses werden nur die Präfixe optimiert, während die restlichen Parameter fixiert bleiben. Daher kann sich das Modell auf sehr effiziente Weise an die gegebene Aufgabe anpassen. Zur Inferenzzeit arbeitet das LLM mit optimierten Präfixen und kann die Aufgabe ohne die Notwendigkeit expliziter harter Prompts ausführen.

Beachten Sie, dass eine Pseudo-Einbettung nicht unbedingt einem Token in natürlicher Sprache entsprechen muss. Stattdessen können diese Einbettungen als „Soft-Prompt-Einbettungen "betrachtet werden, die dazu dienen, die LLMs zu konditionieren. Durch das Training von Soft-Prompt-Einbettungen auf aufgabenspezifischen Daten lernen sie, adaptiv mit den Token-Einbettungen $\mathbf{e}_0 ... \mathbf{e}_m$ zu interagieren und das Verhalten der LLMs zu steuern. Da das Prompt-Tuning die zugrunde liegenden Parameter der vortrainierten LLMs nicht verändert, gilt es als eine leichtgewichtige und effiziente Methode des Fine-Tunings, die die aufgabenspezifische Leistung verbessert und gleichzeitig ihre Generalisierungsfähigkeiten beibehält. Siehe Abbildung 3.6 für eine Illustration des Prompt-Tunings.

Da $\mathbf{p}_0$ $\mathbf{p}_1$ ... $\mathbf{p}_n$ selbst eine Sequenz ist, können wir Sequenzmodelle einsetzen, um sie besser darzustellen. Zum Beispiel kann ein Transformer-Modell diese Sequenz kodieren, und die resultierende Repräsentation kann dann als Eingabe für das LLM verwendet werden. Mit anderen Worten, wir können ein zusätzliches Modell zur Kodierung von Soft Prompts entwickeln. Eine weitere Möglichkeit, das Prompting zu verbessern, besteht darin, Soft und Hard Prompts zu kombinieren und so die Vorteile beider Arten zu nutzen [Liu et al., 2023b]. In der Einbettungssequenz können wir diese Prompts anordnen oder einstreuen. Dies würde zu unterschiedlichen Prompt-Mustern führen. Zum Beispiel ist ein einfaches Muster, das beide Arten von Prompts verwendet

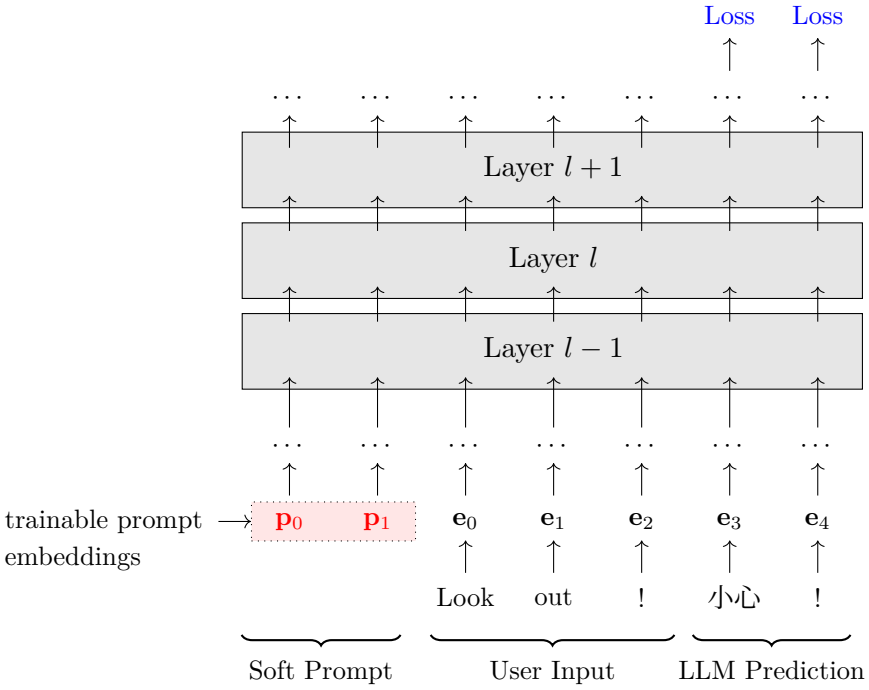
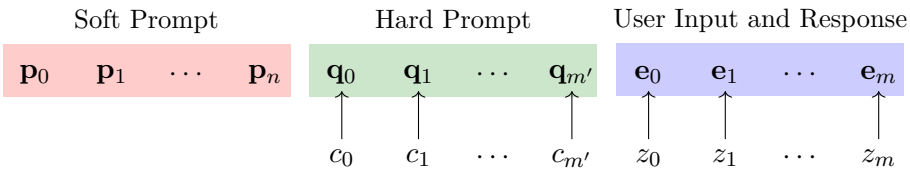
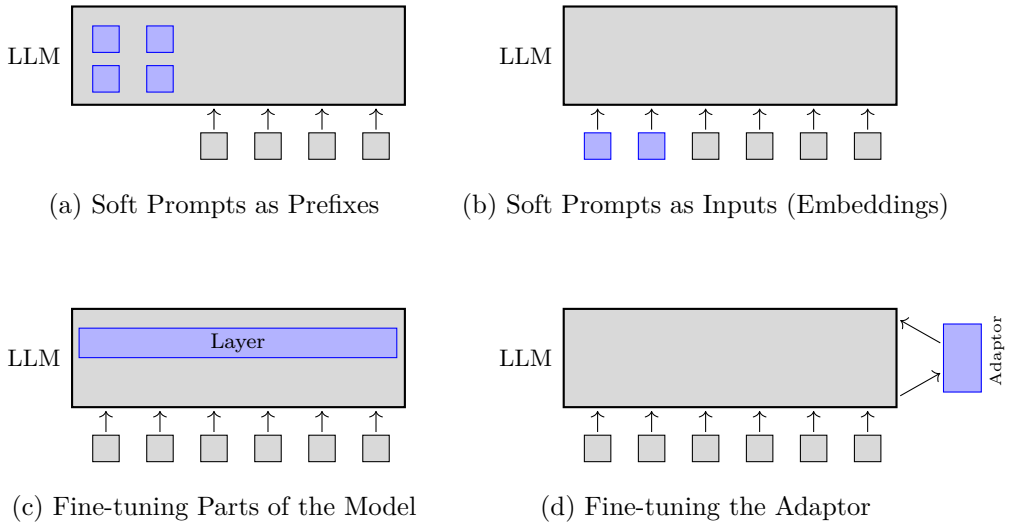


Abbildung 3.6: Veranschaulichung des Prompt-Tunings für eine Übersetzungsaufgabe (Achtung! → 小心!). Anstatt feste textuelle Prompts zu verwenden, sind Soft-Prompts lernbare Einbettungen, die an den Anfang der Einbettungssequenz hinzugefügt werden. Während des Fine-Tunings werden nur diese Prompt-Einbettungen optimiert, um das LLM effizient an die gegebene Aufgabe anzupassen. Nach der Optimierung werden die Prompt-Einbettungen verwendet, um das LLM anzuweisen, die Aufgabe auszuführen, wenn neue Daten eintreffen.



wobei $c_0 ... c_{m'}$ den Hard Prompt und $\mathbf{q}_0 ... \mathbf{q}_{m'}$ die entsprechende Einbettungssequenz bezeichnet.

Hier haben wir Methoden zum Einfügen von Soft Prompts in LLMs betrachtet. Wir überspringen jedoch die Details des Trainings dieser Soft Prompts und gehen davon aus, dass der Leser mit dem standardmäßigen überwachten Lernprozess vertraut ist, d.h. der Maximierung der Wahrscheinlichkeit der korrekten Modellausgabe bei gegebener Modelleingabe. Tatsächlich kann das Lernen von Soft Prompts mit vielen Aspekten des LLM-Fine-Tunings in Verbindung gebracht werden. Wenn wir es beispielsweise als ein Problem der Kontextkompression betrachten, können wir die zuvor beschriebenen Methoden der Wissensdestillation anwenden. In der Arbeit von Mu et al. [2024] werden Prompts komprimiert und als einige Pseudo-Token dargestellt, die an jede Eingabesequenz angehängt werden. Die Einbettungen dieser Pseudo-Token werden optimiert, um die Vorhersagen eines standardmäßig geprompteten Modells nachzuahmen. Mit anderen Worten, das Prompting-Wissen wird von einem Lehrermodell in die Pseudo-Token destilliert.



(a) Soft Prompts as Prefixes

(b) Soft Prompts as Inputs (Embeddings)

(c) Fine-tuning Parts of the Model

(d) Fine-tuning the Adaptor

Abbildung 3.7: Darstellungen zur Verwendung von Soft Prompts in LLMs. Hier sind abstimmbare Soft Prompts in Blau und Komponenten, deren Parameter während der Feinabstimmung fixiert sind, in Grau dargestellt. In Teilabbildung (a) sind Soft Prompts Präfixe, die jeder Schicht des LLM angehängt werden. In Teilabbildung (b) werden Soft Prompts als Eingabe-Einbettungen für das LLM verwendet. In den Teilabbildungen (c) und (d) werden Soft Prompts allgemein als Komponenten des Modells behandelt, die für die Aufgabenanpassung feinabgestimmt werden.

Im weiteren Sinne können viele parametereffiziente Fine-Tuning-Methoden als das Lernen einer Art von Soft Prompt betrachtet werden [Lialin et al., 2023]. Wenn wir einen Teil eines LLM für eine Aufgabe feinabstimmen, kann dieser Prozess im Wesentlichen als das Injizieren von aufgabenbezogenen Prompting-Informationen in diesen spezifischen Teil des Modells angesehen werden. Ein weiterer weit verbreiteter Ansatz für parametereffizientes Fine-Tuning ist das Hinzufügen einer Adapterschicht zwischen den bestehenden Modellschichten. Dieser Ansatz ermöglicht es uns, nur die Adapterschicht für spezifische Aufgaben feinabzustimmen, ohne die zugrunde liegende Architektur zu verändern oder das gesamte Modell neu zu trainieren. In diesem Sinne können Adapterschichten als Soft Prompts betrachtet werden, die Prompting- und aufgabenbezogene Informationen kodieren und mit dem ursprünglichen LLM interagieren, um ihm bei der Anpassung zu helfen. Zusammenfassend zeigt Abbildung 3.7 einen Vergleich verschiedener Methoden zur Verwendung von Soft Prompts in LLMs.

3.3.2.3 Lernen von Soft Prompts mit Komprimierung

Ein weiterer Ansatz zum Lernen von Soft Prompts betrachtet das Problem aus der Perspektive der Komprimierung. Betrachten wir als einfaches Beispiel das Problem, einen langen Kontext durch eine kontinuierliche Repräsentation zu approximieren [Wingate et al., 2022]. Angenommen, wir haben eine Benutzereingabe $\mathbf{z}$ und deren Kontext $\mathbf{c}$ (wie zum Beispiel lange Anweisungen und Demonstrationen). Nun wollen wir eine komprimierte Repräsentation des Kontexts entwickeln, die mit σ bezeichnet wird, sodass die Vorhersage basierend auf $\mathbf{z}$ und σ so nah wie möglich an der Vorhersage basierend auf $\mathbf{z}$ und $\mathbf{c}$ liegt.

Dieses Ziel kann in der folgenden Form ausgedrückt werden

$$\hat{\sigma} \quad = \quad \arg\min_{\sigma} s(\hat{\mathbf{y}}, \hat{\mathbf{y}}_{\sigma}) \tag{3.23}$$

wobei $\hat{\mathbf{y}} = \arg\max_{\mathbf{y}} \Pr(\mathbf{y}|\mathbf{c}, \mathbf{z})$ und $\hat{\mathbf{y}}_{\sigma} = \arg\max_{\mathbf{y}_{\sigma}} \Pr(\mathbf{y}|\sigma, \mathbf{z})$ die LLM-Vorhersagen für den vollständigen bzw. den komprimierten Kontext sind. Die Funktion $s(\cdot, \cdot)$ stellt typischerweise ein Verlust- oder Ähnlichkeitsmaß dar, mit dem Ziel, den Unterschied in den Vorhersagen zwischen den beiden Kontextrepräsentationen zu minimieren.

Ein allgemeiner Rahmen, um dies zu erreichen, ist die Wissensdestillation, bei der $\hat{\mathbf{y}}$ und $\hat{\mathbf{y}}_{\sigma}$ als die Vorhersagen des Lehrermodells bzw. des Schülermodells betrachtet werden können. Diese Formalisierung verbindet unsere Diskussion mit dem zuvor erörterten Problem der Kontextdestillation. Das Trainingsziel kann in Analogie zu den Gleichungen (3.15) und (3.16) hergeleitet werden. Ein einfaches Trainingsziel ist zum Beispiel gegeben durch

$$\hat{\sigma} \quad = \quad \arg\max_{\sigma} \log \Pr(\hat{\mathbf{y}}|\sigma, \mathrm{z}) \tag{3.24}$$

Alternativ können wir die KL-Divergenz zwischen den Ausgabeverteilungen minimieren, was zu folgendem Ergebnis führt

$$\hat{\sigma} \quad = \quad \arg\min_{\sigma} \; \mathrm{KL}(\Pr(\cdot|\mathbf{c}, \mathbf{z}) \;||\; \Pr(\cdot|\sigma, \mathbf{z})) \tag{3.25}$$

Der Unterschied zu den Modellen in den Gleichungen (3.15) und (3.16) besteht darin, dass hier der komprimierte Kontext als reellwertige Vektoren (genannt Prompt-Einbettungen) dargestellt wird, anstatt als normale Token. Durch die Anwendung der oben genannten Methoden destillieren wir den Kontext aus der Tokensequenz $\mathbf{c}$ in die Einbettungen σ. Beachten Sie, dass das Lehrermodell $\Pr(\cdot|\mathbf{c}, \mathbf{z})$ und das Schülermodell $\Pr(\cdot|\sigma, \mathbf{z})$ nicht unbedingt die gleiche Architektur oder Modelleinstellungen haben müssen. In der Praxis wünschen wir uns im Allgemeinen, dass das Lehrermodell leistungsfähiger ist, während das Schülermodell kleiner und effizienter sein sollte.

Obwohl die Komprimierung des vollständigen Kontexts in kontinuierliche Repräsentationen ein unkomplizierter Ansatz zum Erlernen von Soft Prompts ist, erfordert er ein Lehrer-Modell, das mit langen Eingabesequenzen umgehen kann. In vielen Fällen ist der Kontext jedoch so lang, dass die Anwendung eines LLMs zu kostspielig oder undurchführbar ist. Die Modellierung langer Eingabesequenzen lässt sich der breiten Familie von effizienten Methoden für Long-Context-LLMs zuordnen. Es wurden viele Techniken entwickelt, um dieses Problem zu lösen. Zum Beispiel kann man einen KV-Cache fester Größe verwenden, um die vergangenen Informationen bei jedem Schritt während der Inferenz zu speichern. Effiziente Transformer-Architekturen und Long-Context-LLMs wurden in diesem Buch intensiv diskutiert. Für detailliertere Diskussionen zu diesen Themen können interessierte Leser auf Kapitel 2verweisen.

Es gibt auch Methoden, die speziell dafür entwickelt wurden, langen Kontext in Soft Prompts zu komprimieren. Hier betrachten wir die Methode von Chevalier et al. [2023] als Beispiel. Die Grundidee ist, dass wir Soft Prompts schrittweise lernen, indem wir die Kontextrepräsentation fester Größe über die Kontextsequenz akkumulieren. Gegeben sei

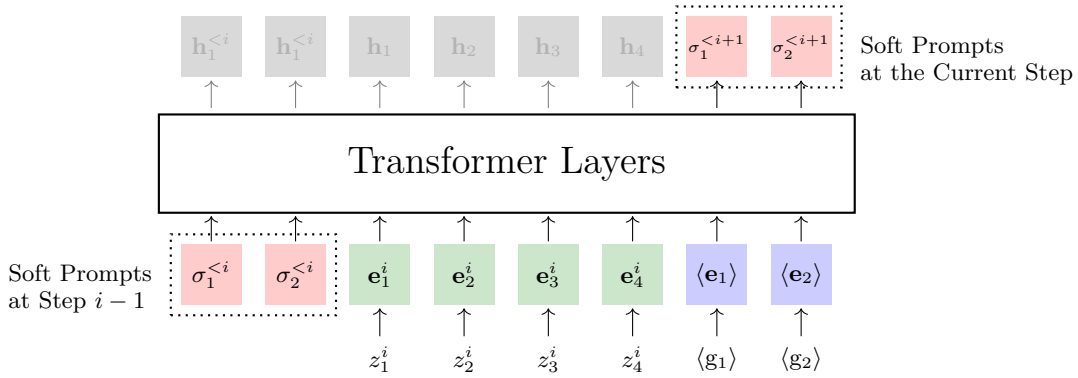


Abbildung 3.8: Darstellung der Komprimierung eines Kontextsegments in weiche Prompts ($\kappa = 2$ und $m_i = 4$). Die Eingabe für das LLM umfasst die weichen Prompts aus dem vorherigen Schritt ($\sigma_1^{<i}$ und $\sigma_2^{<i}$), die Token des Segments (z_1, z_2, z_3, und z_4) und die Zusammenfassungstoken ($\langle g_1 \rangle$ und $\langle g_2 \rangle$). Auf dieser Grundlage arbeitet das LLM wie gewohnt. Anschließend extrahieren wir die Ausgaben an der letzten Transformer-Schicht, die den Zusammenfassungstoken entsprechen. Diese Ausgaben können als die weichen Prompts betrachtet werden, die sich bis zu diesem Segment angesammelt haben.

ein langer Kontext, den wir zunächst in eine Anzahl von Segmenten $\mathbf{z}^1, ..., \mathbf{z}^K$ unterteilen. Anschließend verarbeiten wir diese Segmente nacheinander und erzeugen jedes Mal eine Repräsentation des bisher verarbeiteten Kontexts, die mit $\sigma^{<i+1}$ bezeichnet wird. Dazu werden einige Zusammenfassungs-Token $\langle g_1 \rangle, ..., \langle g_\kappa \rangle$ eingeführt. In jedem Schritt nehmen wir ein Segment $\mathbf{z}^i = z_1^i ... z_{m_i}^i$ zusammen mit der vorherigen Kontextrepräsentation $\sigma^{<i}$ und den Zusammenfassungs-Token $\langle g_1 \rangle, ..., \langle g_\kappa \rangle$ als Eingabe und verwenden ein LLM, um die entsprechende verborgene Repräsentationssequenz auf der letzten Transformer-Schicht zu erzeugen. Ein Beispiel für diesen Prozess ist in Abbildung 3.8 dargestellt.

Hier ist $\sigma^{<i}$ im Wesentlichen ein Speicher. Das Modell arbeitet auf eine RNN-ähnliche Weise. Jedes Mal nehmen wir ein Segment und aktualisieren diesen Speicher, indem wir sowohl den vorherigen Speicherzustand als auch das Segment kodieren. Daher ist das $\sigma^{<i}$, das beim letzten Segment erzeugt wird, eine Repräsentation der gesamten Kontextsequenz. Das Transformer-Modell zum Lernen dieser Repräsentationen kann ein Standard-LLM sein, aber wir müssen es feinabstimmen, um es an diese Kontextrepräsentationsaufgabe anzupassen.

Beachten Sie, dass wir hier Prompt und Kontext einfach als ähnliche Begriffe betrachten, obwohl sie nicht dasselbe sind. Obwohl wir den Begriff Prompt gewissermaßen „missbrauchen“, können wir ihn oft als eine Art von Kontext betrachten. Aus dieser Perspektive können die hier diskutierten Methoden auf allgemeine Textkomprimierungsprobleme angewendet werden.

3.3.3 Reduzierung der Prompt-Länge

Obwohl weiche Prompts dichte, verborgene Repräsentationen liefern, sind sie nicht direkt interpretierbar. Die mangelnde Interpretierbarkeit kann für Benutzer eine erhebliche Hürde darstellen, wenn sie versuchen zu verstehen, wie ihre Eingaben die LLM-Ausgaben beeinflussen. Darüber hinaus sind weiche Prompts zwar effizient für das Fine-Tuning und die Bereitstellung, aber sie sind unflexibel und erlauben keine einfachen Anpassungen

ohne umfangreiches Fine-Tuning oder Modifikationen. Diese Inflexibilität kann ihre Nützlichkeit in dynamischen Umgebungen einschränken, in denen häufige Prompt-Änderungen erforderlich sind.

Eine alternative Möglichkeit, effiziente Prompts zu entwickeln, besteht darin, den für das Prompting verwendeten Text zu vereinfachen. Nachfolgend finden Sie beispielsweise einen Prompt zur Beantwortung von Fragen zu den Themen Gesundheitswesen und Finanzen.

> The task involves developing a language model capable of understanding and responding to user inquiries across various domains, with a particular emphasis on healthcare and finance. Considering the broad range of potential queries, from the specifics of medical diagnoses to the nuances of financial regulations, the model must ensure a comprehensive understanding and accurate responses.
>
> Question:
>
> What are the best practices for using artificial intelligence in diagnosing cardiovascular diseases?
>
> ——

Wir können die Aufgabenbeschreibung vereinfachen, indem wir die unwichtigen Teile löschen.

> Die Aufgabe besteht darin, ein Sprachmodell zu entwickeln, das in der Lage ist, Benutzeranfragen ~~in verschiedenen Bereichen, mit besonderem Schwerpunkt~~ zum Gesundheitswesen und zu Finanzen zu verstehen und zu beantworten. ~~Angesichts der großen Bandbreite möglicher Anfragen, von den Einzelheiten medizinischer Diagnosen bis hin zu den Nuancen von Finanzvorschriften,~~ muss das Modell ein umfassendes Verständnis und genaue Antworten gewährleisten.

Wir können es auch als kürzeren Text paraphrasieren.

> Die Aufgabe besteht darin, ein auf das Gesundheitswesen und das Finanzwesen ausgerichtetes Sprachmodell zu entwickeln, das in der Lage ist, ein breites Spektrum von Benutzeranfragen zu verstehen und präzise zu beantworten.

Dieses Problem kann als ein klassisches NLP-Problem betrachtet werden –die Textvereinfachung. Daher können die verwendeten Methoden allgemein sein und sind nicht auf das Problem der Vereinfachung von Prompts beschränkt. Es gibt viele Möglichkeiten, dies zu erreichen. Eine einfache Methode besteht darin, einige Heuristiken zu definieren und redundante Wörter zu identifizieren, die ohne Verlust wesentlicher Informationen eliminiert werden können. Zum Beispiel können wir jeden Token in einer Sequenz hinsichtlich seines Beitrags zur Gesamtbedeutung untersuchen und diejenigen entfernen, die einen minimalen Wert liefern [Li et al., 2023c; Jiang et al., 2023b]. Eine andere Methode besteht darin,

das Problem als eine Sequenz-zu-Sequenz-Aufgabe zu formulieren. Mit gelabelten Daten zur Textvereinfachung können wir ein Encoder-Decoder-Modell trainieren, um jeden Eingabetext in seine vereinfachte Form umzuwandeln. Darüber hinaus ist es, da viele LLMs für Textvereinfachungsaufgaben feinabgestimmt und ausgerichtet wurden, unkompliziert, diese Modelle zur Vereinfachung von Prompts zu verwenden. Zum Beispiel können wir ein LLM auffordern, einen Text unter bestimmten Einschränkungen zu vereinfachen, wie z. B. die Begrenzung der Länge des vereinfachten Textes.

3.4 Zusammenfassung

In diesem Kapitel haben wir eine Vielzahl von Themen im Zusammenhang mit dem LLM-Prompting erörtert. Unsere Diskussion hat sich hauptsächlich auf zwei Aspekte konzentriert:

- Wie man grundlegende Prompts entwirft, um die Vorhersagen von LLMs zu steuern, und wie man diese Prompts für eine effektivere und effizientere Problemlösung verfeinert?
- Wie man den Entwurf und die Darstellung von Prompts automatisiert?

Lösungen für diese Probleme umfassen sowohl allgemeine Prompt-Designs als auch fortgeschrittenere Techniken wie CoT und Prompt-Learning, die in der jüngsten Forschung ausgiebig untersucht wurden.

Im NLP kann Prompting als eine Technologie betrachtet werden, die sich zusammen mit LLMs entwickelt hat und in gewissem Sinne die Tür zur praktischen Anwendung dieser Modelle in einer beeindruckenden Bandbreite von Problembereichen geöffnet hat. Tatsächlich lässt sich das Konzept der Prompts, wenn man es etwas erweitert, bis in die Anfänge des maschinellen Lernens und des NLP zurückverfolgen. Zum Beispiel verwenden viele NLP-Systeme handgefertigte Merkmale und Vorlagen, um bestimmte Aufgaben zu „prompten“. Stellen Sie sich vor, Sie entwickeln ein Merkmal, das anzeigt, ob ein Text formell oder informell ist. Wir können dieses Merkmal in ein maschinelles Übersetzungssystem einspeisen, um die Übersetzung von der Art des Eingabetextes abhängig zu machen.

Die weit verbreitete Verwendung des modernen Konzepts der Prompts begann mit dem Aufkommen großer vorabtrainierter Modelle im Bereich des NLP. Anfänglich wurden diese Modelle, wie zum Beispiel BERT, hauptsächlich durch Feinabstimmung an spezifische nachgelagerte Aufgaben angepasst. Forscher entdeckten jedoch bald, dass durch das Entwerfen spezifischer „Prompts“–das Hinzufügen bestimmter Wörter oder Sätze zur Eingabe –die Modelle dazu veranlasst werden konnten, auf spezifische Aufgaben ohne umfangreiche Feinabstimmung zu reagieren. Dies motivierte die NLP-Community, universelle Basismodelle zu entwickeln und anzuwenden, die durch Prompts dazu gebracht werden können, verschiedene Aufgaben zu bewältigen, ohne die zugrunde liegende Architektur und das Vorabtrainingsverfahren zu ändern.

Prompting-Ansätze wurden zuerst mit kleineren Modellen erprobt und zeigten später

beeindruckende Fähigkeiten bei großen Modellen wie GPT-3, die als Reaktion auf einfache Prompts bei verschiedenen Aufgaben qualitativ hochwertige Texte generieren konnten. Mit der Weiterentwicklung der Prompting-Technologie entwickelte sich das Prompt-Engineering zu einem wichtigen Forschungsbereich. Wie in diesem Kapitel erörtert, umfasst es im Großen und Ganzen das Entwerfen effektiver Prompts zur Maximierung der Modellleistung, was sowohl handgefertigte als auch automatisch generierte Prompts einschließt. Neuere Forschungen haben untersucht, wie die Effektivität des Promptings durch Techniken wie Few-Shot-Lernen, Zero-Shot-Lernen und CoT-Schlussfolgern verbessert werden kann, wodurch LLMs in einer Vielzahl von Szenarien effektiv arbeiten können. Eine allgemeine Diskussion über Prompting kann sehr breit gefächert sein, und wir können in diesem Kapitel nicht alle Details behandeln. Für fortgeschrittenere Techniken des Promptings kann der Leser auf aktuelle Übersichtsartikel verwiesen werden. Zu den Themen gehören In-Context-Lernen [Li, 2023; Dong et al., 2022], CoT [Chu et al., 2023; Yu et al., 2023; Zhang et al., 2023a], effizientes Prompting [Chang et al., 2024] und allgemeines Prompt-Engineering [Liu et al., 2023c; Chen et al., 2023a].

Es ist zu beachten, dass, obwohl wir idealerweise allgemeine Prompting-Methoden ohne Anpassung von Modellarchitekturen und -parametern entwickeln möchten, die Ergebnisse des Promptings im Allgemeinen stark von der Qualität und Größe der gegebenen LLMs abhängen. Bei leistungsfähigeren Modellen, wie kommerzialisierten Online-LLMs, können einfache Prompts ausreichen, um diese Modelle anzuweisen, Aufgaben korrekt auszuführen. In diesem Fall ist das Prompt-Engineering relativ einfach, obwohl wir immer noch gewisse Anstrengungen unternehmen müssen, damit die LLMs ordnungsgemäß funktionieren. Im Gegensatz dazu müssen wir, wenn die LLMs nicht leistungsfähig genug sind, die Prompts möglicherweise sorgfältig gestalten, um die gewünschten Ergebnisse zu erzielen. In vielen Fällen ist eine Feinabstimmung weiterhin notwendig, um die Modelle an anspruchsvolle Prompting-Strategien anzupassen.

Kapitel 4

Alignment

Alignment ist kein neues Konzept im NLP, aber seine Bedeutung variiert je nach Bereich und im Laufe der Zeit. Im traditionellen NLP bezieht sich der Begriff Alignment typischerweise auf Aufgaben, die entsprechende Elemente in zwei Mengen verknüpfen, wie zum Beispiel das Alignieren von Wörtern zwischen einem chinesischen und einem englischen Satz. Da LLMs in der NLP-Forschung immer wichtiger werden, wird dieser Begriff allgemeiner verwendet, um die Anpassung von Modellausgaben an menschliche Erwartungen zu bezeichnen. Das Problem, das durch Alignment gelöst werden soll, besteht darin, dass die Ausgabe eines Modells möglicherweise nicht mit den spezifischen Zielen oder Kontexten übereinstimmt, die von den Benutzern beabsichtigt sind. Zum Beispiel sind vorabtrainierte LLMs möglicherweise nicht in der Lage, Benutzeranweisungen zu befolgen, da sie nicht dafür trainiert wurden. Ein weiteres Beispiel ist, dass LLMs schädliche Inhalte generieren oder in ihren Trainingsdaten enthaltene Verzerrungen (Biases) aufrechterhalten können. Dies stellt neue Herausforderungen dar, um sicherzustellen, dass die Ausgaben von LLMs nicht nur korrekt und relevant, sondern auch ethisch vertretbar und nicht diskriminierend sind.

Das bloße Vorabtrainieren von LLMs kann zu einer Vielzahl von Alignment-Problemen führen. Unser letztendliches Ziel ist es, all diese Probleme zu lösen oder zu mildern, um sicherzustellen, dass LLMs sowohl präzise als auch sicher sind. Hier gibt es ein interessantes Problem: Da große Sprachmodelle auf riesigen Datenmengen trainiert werden, haben wir Grund zu der Annahme, dass, wenn wir über ausreichende Daten verfügen, die eine Vielzahl von Aufgaben abdecken und mit menschlichen Präferenzen übereinstimmen, das Vorabtraining die LLMs genau und sicher genug machen könnte, vielleicht sogar die Notwendigkeit des Alignments beseitigen würde. Die Realität ist jedoch, dass es nahezu unmöglich ist, Daten zu sammeln, die alle Aufgaben umfassen oder menschliche Präferenzen adäquat repräsentieren. Dies macht es schwierig, ein Modell-Alignment allein durch Vorabtraining zu erreichen, oder zumindest bleibt Alignment in dieser Phase ein sehr notwendiger und kritischer Schritt in der Entwicklung von LLMs.

In diesem Kapitel werden wir uns auf Alignment-Methoden für LLMs konzentrieren. Wir werden zunächst die allgemeinen Alignment-Aufgaben erörtern. Anschließend werden wir zwei weit verbreitete Ansätze betrachten, die als instruction alignment bzw. human preference alignment bekannt sind. Ersterer greift auf Techniken der überwachten Feinabstimmung zurück und leitet die LLMs an, Ausgaben zu generieren, die sich eng an die Anweisungen der Benutzer halten. Andererseits stützt sich letzterer typischerweise auf Techniken des Reinforcement Learning, bei denen die LLMs auf der Grundlage von menschlichem Feedback trainiert werden. Obwohl diese Methoden von unterschiedlichen Zielen motiviert sind, werden sie üblicherweise gemeinsam eingesetzt, um gut ausgerichtete (well-aligned) LLMs zu entwickeln.

4.1 Ein Überblick über die LLM-Abstimmung

Abstimmung kann auf verschiedene Weisen erreicht werden. Wir benötigen unterschiedliche Methoden zur LLM-Abstimmung, da dieses Problem an sich komplex und vielschichtig ist und eine Mischung aus technischen Überlegungen erfordert. Hier betrachten wir drei weit verbreitete Ansätze zur Abstimmung von LLMs.

Der erste Ansatz besteht darin, LLMs mit annotierten Daten einem Fine-Tuning zu unterziehen. Dieser Ansatz ist unkompliziert, da er lediglich das vorangegangene Training eines vortrainierten LLMs erweitert, um es an spezifische Aufgaben anzupassen. Ein Beispiel hierfür ist das supervised fine-tuning (SFT), bei dem das LLM auf einem Datensatz weiter trainiert wird, der aufgaben-spezifische Anweisungen gepaart mit ihren erwarteten Ausgaben umfasst. Der SFT-Datensatz ist im Allgemeinen viel kleiner als der ursprüngliche Trainingsdatensatz, aber diese Daten sind hochspezialisiert. Das Ergebnis von SFT ist, dass das LLM lernen kann, Aufgaben basierend auf Benutzeranweisungen auszuführen. Durch das Fine-Tuning des LLMs mit einem Satz von Frage-Antwort-Paaren kann das Modell beispielsweise auf spezifische Fragen antworten, auch wenn diese nicht direkt im SFT-Datensatz abgedeckt sind. Diese Methode erweist sich als besonders nützlich, wenn es relativ einfach ist, die Eingabe-Ausgabe-Beziehungen zu beschreiben, und unkompliziert, die Daten zu annotieren.

Der zweite Ansatz besteht darin, LLMs mithilfe von Belohnungsmodellen fein abzustimmen. Eine Schwierigkeit bei der Abstimmung besteht darin, dass menschliche Werte und Erwartungen komplex und schwer zu beschreiben sind. In vielen Fällen ist es selbst für Menschen eine Herausforderung zu artikulieren, was ethisch korrekt oder kulturell angemessen ist. Daher ist das Sammeln oder Annotieren von Feinabstimmungsdaten nicht so unkompliziert wie bei SFT. Darüber hinaus ist die Abstimmung von LLMs nicht nur eine Aufgabe der Datenanpassung, oder anders ausgedrückt, die begrenzten, von Menschen annotierten Stichproben sind oft unzureichend, um diese Verhaltensweisen umfassend zu beschreiben. Was wir hier wirklich benötigen, ist, dem Modell beizubringen, wie es bestimmen kann, welche Ausgaben eher den menschlichen Präferenzen entsprechen. Wir wollen beispielsweise nicht nur, dass die Ausgaben technisch korrekt sind, sondern auch, dass sie mit menschlichen Erwartungen und Werten übereinstimmen. Eine Idee ist, ein Belohnungsmodell analog zu einem menschlichen Experten zu entwickeln. Dieses Belohnungsmodell würde funktionieren, indem es das LLM belohnt, wann immer es Antworten generiert, die enger mit menschlichen Präferenzen übereinstimmen, ähnlich wie ein Lehrer einem Schüler Feedback gibt. Um ein solches Belohnungsmodell zu erhalten, können wir eine Bewertungsfunktion aus menschlichen Präferenzdaten trainieren. Das trainierte Belohnungsmodell wird dann als Leitfaden verwendet, um das LLM anzupassen und zu verfeinern. Dies rahmt die LLM-Abstimmungsaufgabe als eine Aufgabe des bestärkenden Lernens ein. Die resultierenden Methoden, wie zum Beispiel reinforcement learning from human feedback (RLHF), haben sich als besonders erfolgreich erwiesen, um LLMs an die Feinheiten menschlichen Verhaltens und sozialer Normen anzupassen.

Der dritte Ansatz besteht darin, die Abstimmung während der Inferenz anstatt während des Trainings oder der Feinabstimmung durchzuführen. Aus dieser Perspektive kann auch das Prompting in LLMs als eine Form der Abstimmung angesehen werden, die jedoch kein Training oder keine Feinabstimmung erfordert. So können wir ein LLM dynamisch und mit minimalen Kosten an verschiedene Aufgaben anpassen. Eine weitere Methode, die

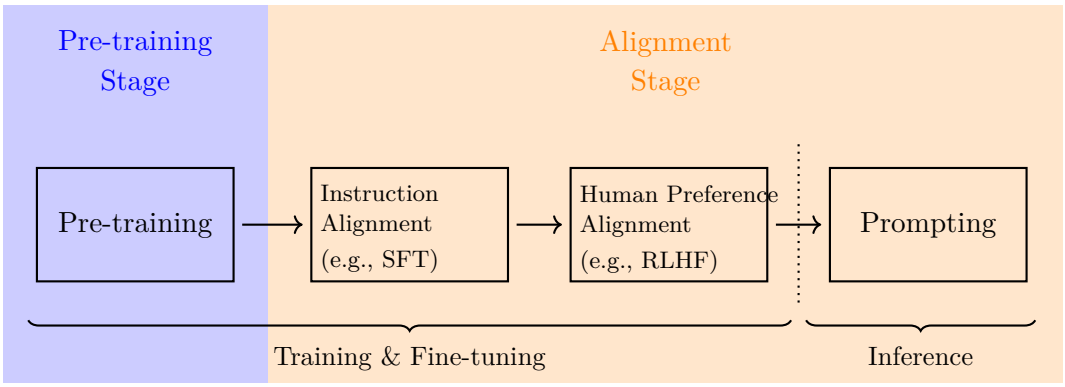


Abbildung 4.1: Schematische Darstellung der Pre-train-then-align-Methode zur Entwicklung von LLMs. In der Vortrainingsphase wird ein LLM anhand großer Datenmengen mittels der Vorhersage des nächsten Tokens trainiert. Anschließend wird das LLM in der Anpassungsphase an die Anweisungen, Absichten und Präferenzen des Benutzers angepasst. Dazu gehören die Anpassung an Anweisungen, die Anpassung an menschliche Präferenzen und das Prompting.

Abstimmung zur Inferenzzeit durchzuführen, besteht darin, die Ausgaben eines LLM neu zu bewerten. Zum Beispiel könnten wir ein Bewertungssystem entwickeln, um menschliches Feedback zu den Ausgaben des LLM zu simulieren (ähnlich einem Belohnungsmodell) und diejenigen zu priorisieren, die positiveres Feedback erhalten.

Die drei oben genannten Methoden werden typischerweise nacheinander angewendet, sobald das Vortraining abgeschlossen ist: Zuerst führen wir SFT durch, dann RLHF, und anschließend wird das LLM während der Inferenz auf irgendeine Weise gepromptet. Dies teilt die Entwicklung von LLMs grob in zwei Phasen ein — die Vortrainingsphase und die Abstimmungsphase. Abbildung 4.1 zeigt eine Veranschaulichung davon. Da Prompting-Techniken im vorigen Kapitel intensiv diskutiert wurden, werden wir uns im Rest dieses Kapitels auf Feinabstimmungs-basierte Abstimmungsmethoden konzentrieren.

4.2 Anpassung an Anweisungen

Ein Merkmal von LLMs ist, dass sie den von Benutzern bereitgestellten Prompts folgen können, um verschiedene Aufgaben auszuführen. In vielen Anwendungen besteht ein Prompt aus einer einfachen Anweisung und einer Benutzereingabe, und wir möchten, dass das LLM dieser Anweisung folgt, um die Aufgabe korrekt auszuführen. Diese Fähigkeit von LLMs wird auch als Anweisungsbefolgungsfähigkeit bezeichnet. Beispielsweise ist unten ein Prompt dargestellt, bei dem wir möchten, dass das LLM die wichtigsten Punkte extrahiert und eine prägnante Zusammenfassung für einen langen Artikel liefert.

Instruction	Summarize this text in three sentences.
Input	Daylight Savings Time (DST) - the process of moving clocks forward by one hour in the summer - was started in Germany in 1916 ...
Output	____

Diese Aufgabe erfordert, dass das LLM die Anweisung „Fassen Sie diesen Text in drei

Sätzen zusammen"versteht und die Zusammenfassung entsprechend durchführt. Allerdings werden LLMs typischerweise für die Vorhersage des nächsten Tokens trainiert und nicht für die Erzeugung von Ausgaben, die Anweisungen folgen. Die Anwendung eines vor-trainierten LLMs auf das obige Beispiel würde wahrscheinlich dazu führen, dass das Modell den Eingabeartikel weiterschreibt, anstatt die Hauptpunkte zusammenzufassen. Das Ziel der Anpassung an Anweisungen (oder Anweisungs-Feinabstimmung) ist es, das LLM so abzustimmen, dass es präzise auf die Anweisungen und Absichten des Benutzers reagiert. Der Rest dieses Abschnitts wird einige Probleme im Zusammenhang mit der Anpassung an Anweisungen erörtern, einschließlich der Feinabstimmung von LLMs zur Befolgung von Anweisungen, der Generierung oder Sammlung von Anweisungsdaten und der Verallgemeinerung der Anpassung an Anweisungen.

4.2.1 Überwachtes Fine-Tuning

Ein unkomplizierter Ansatz, um LLMs so anzupassen, dass sie Anweisungen befolgen, besteht darin, diese Modelle mithilfe von annotierten Eingabe-Ausgabe-Paaren einem Fine-Tuning zu unterziehen [Ouyang et al., 2022; Wei et al., 2022a]. Im Gegensatz zum standardmäßigen Sprachmodelltraining möchten wir hier nicht die Wahrscheinlichkeit maximieren, eine vollständige Sequenz zu generieren, sondern vielmehr die Wahrscheinlichkeit, den Rest der Sequenz gegeben ihres Präfixes zu generieren. Dieser Ansatz unterscheidet das anweisungsbasierte Fine-Tuning etwas vom Pre-Training. Die SFT-Daten sind eine Sammlung solcher Eingabe-Ausgabe-Paare (bezeichnet mit S), wobei jede Ausgabe die korrekte Antwort für die entsprechende Eingabeanweisung ist. Unten ist beispielsweise ein SFT-Datensatz aufgeführt:

x (instruction + user input)	**y** (output)
Summarize the following article. Article: In recent years, solar energy has seen unprecedented growth, becoming the fastest-growing ...	{∗summary∗}
Extract the main financial figures from the following earnings report. Report: The company reported a revenue of $10 million in the first quarter with a profit margin of 15% ...	Revenue: $10 million, Profit Margin: 15%
Classify the following email as spam or not spam. Text: Congratulations! You've won a $500 gift card. Click here to claim now.	Spam
Provide a solution to the following technical issue. Issue: my computer is running slow and often freezes.	First, check for ...

wobei die Anweisungen hervorgehoben sind. Dieser Datensatz enthält Anweisungen und die entsprechenden Ausgaben für mehrere verschiedene NLP-Probleme, sodass wir ein LLM so feinabstimmen können, dass es mehrere Aufgaben gleichzeitig bewältigen kann.

Sei $\mathbf{x} = x_0...x_m$ eine Eingabesequenz (z. B. Anweisung + Benutzereingabe) und $\mathbf{y} = y_1...y_n$ die entsprechende Ausgabesequenz. Beim SFT ist es das Ziel, die Wahrscheinlichkeit der Ausgabe $\mathbf{y}$ gegeben der Eingabe $\mathbf{x}$ zu maximieren. Betrachten wir ein LLM mit den

vortrainierten Parametern $\hat{\theta}$. Das Ziel des Fine-Tunings kann dann wie folgt formuliert werden:

$$\tilde{\theta} = \arg\max_{\hat{\theta}^+} \sum_{(\mathbf{x},\mathbf{y})\in\mathcal{D}} \log \Pr_{\hat{\theta}^+}(\mathbf{y}|\mathbf{x}) \tag{4.1}$$

wobei $\tilde{\theta}$ die durch das Fine-Tuning optimierten Parameter bezeichnet und $\hat{\theta}^+$ eine Anpassung von $\hat{\theta}$ darstellt. Hier werden wir den hochgestellten Index + weglassen und θ verwenden, um $\hat{\theta}^+$ darzustellen, um die Notation übersichtlich zu halten. Der Leser sollte jedoch bedenken, dass das Fine-Tuning mit den vortrainierten Parametern beginnt und nicht mit zufällig initialisierten Parametern.

Die Zielfunktion $\log \Pr_\theta(y_i|\mathbf{x}, \mathbf{y}_{<i})$ wird berechnet, indem die Log-Wahrscheinlichkeiten der Token in $\mathbf{y}$ summiert werden, bedingt durch die Eingabe $\mathbf{x}$ und alle vorherigen Token $\mathbf{y}_{<i}$:

$$\log \Pr_\theta(\mathbf{y}|\mathbf{x}) = \sum_{i=1}^{n} \log \Pr_\theta(y_i|\mathbf{x}, \mathbf{y}_{<i}) \tag{4.2}$$

Diese Formulierung ist äquivalent zur Minimierung des Kreuzentropie-Verlusts.

Beachten Sie, dass die Minimierung der bedingten Log-Wahrscheinlichkeit $\log \Pr_\theta(\mathbf{y}|\mathbf{x})$ kein Standardproblem des Sprachmodelltrainings ist. Wenn wir $\mathbf{x}$ und $\mathbf{y}$ zu einer einzigen Sequenz verketten, basiert eine allgemeinere Form der Sprachmodellierung auf der gemeinsamen Log-Wahrscheinlichkeit $\log \Pr_\theta(\mathbf{x}, \mathbf{y})$, das heißt, wir minimieren den Verlust über alle Token der Sequenz $\text{seq}_{\mathbf{x},\mathbf{y}} = [\mathbf{x}, \mathbf{y}]$. Wir können die Wahrscheinlichkeit dieser Sequenz mithilfe der Kettenregel schreiben

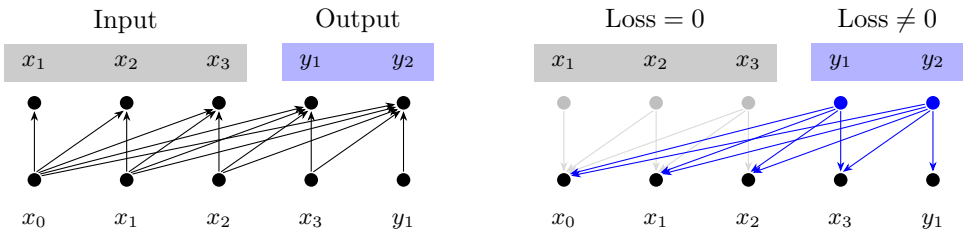
$$\begin{aligned} \log \Pr_\theta(\text{seq}_{\mathbf{x},\mathbf{y}}) &= \log \Pr_\theta(\mathbf{x}, \mathbf{y}) \\ &= \underbrace{\log \Pr_\theta(\mathbf{x})}_{\text{auf 0 gesetzt}} + \underbrace{\log \Pr_\theta(\mathbf{y}|\mathbf{x})}_{\text{Verlustberechnung}} \end{aligned} \tag{4.3}$$

Auf der rechten Seite der Gleichung befinden sich zwei Terme. Wir können den ersten Term $\log \Pr_\theta(\mathbf{x})$ einfach auf 0 setzen und uns ausschließlich auf den zweiten Term $\log \Pr_\theta(\mathbf{y}|\mathbf{x})$ für die Verlustberechnung konzentrieren. Dadurch kann das Training mit Standard-LLMs implementiert werden. Für die Sequenz $\text{seq}_{\mathbf{x},\mathbf{y}}$ führen wir zunächst wie gewohnt den Forward-Pass durch. Dann, während des Backward-Passes, zwingen wir den Verlust, der $\mathbf{x}$ entspricht, auf Null. Abbildung 4.2 zeigt eine Veranschaulichung dieses Prozesses.

Indem wir $\log \Pr_\theta(\text{seq}_{\mathbf{x},\mathbf{y}})$ als Zielfunktion verwenden, können wir SFT mit einer regulären Form des Sprachmodelltrainings beschreiben:

$$\tilde{\theta} = \arg\max_{\theta} \sum_{(\mathbf{x},\mathbf{y})\in\mathcal{D}} \log \Pr_\theta(\text{seq}_{\mathbf{x},\mathbf{y}}) \tag{4.4}$$

Das oben betrachtete Problem ist im Grunde ein einrundige Vorhersage-Problem, bei dem das LLM eine Antwort auf eine einzelne Eingabe generiert, ohne weitere Interaktion oder Rückmeldung vom Benutzer. Die Eingabe wird verarbeitet und die Ausgabe wird



(a) Forward Pass (b) Backward Pass

Abbildung 4.2: Illustration des überwachten Fine-Tunings für LLMs. Wir verketten die Eingabe und die Ausgabe zu einer einzigen Sequenz. Während des Forward-Passes führen wir das LLM wie gewohnt aus. Während des Backward-Passes berechnen wir den Verlust nur für den Ausgabeteil und setzen den Verlust für den Eingabeteil einfach auf 0.

in einem Durchgang generiert. Dies ist typisch für Szenarien, in denen eine einzelne Frage gestellt und eine einzelne Antwort gegeben wird, ohne Folgefragen oder Klärungen. In der Praxis müssen wir jedoch manchmal mehrrundige Vorhersageprobleme behandeln, zum Beispiel, wenn ein LLM einen Dialog über mehrere Runden führt. In diesem Setting generiert das LLM nicht nur Antworten auf der Grundlage der ursprünglichen Eingabe, sondern bezieht auch nachfolgende Eingaben ein, die frühere Interaktionen verfeinern oder erweitern könnten. Zum Beispiel können wir das LLM als Chatbot für das Gesundheitswesen einsetzen und eine Konversation mit dem Benutzer führen, wie hier:

User	I've been feeling very tired lately.
Chatbot	I'm sorry to hear that. Besides feeling tired, have you noticed any other symptoms?
User	Yes, I'm also experiencing headaches frequently.
Chatbot	How long have these symptoms been going on?
User	About a week now.
Chatbot	It might be good to check in with a healthcare professional. Would you like help setting up an appointment?
User	Yes, please. Can it be after work hours?
Chatbot	Sure, I can arrange that. There are slots available next Wednesday and Thursday after 5 PM. Which day works better for you?
	...

Bei dieser Aufgabe gibt es mehrere Konversationsrunden, von denen jede die Generierung einer Antwort auf der Grundlage der Anfrage oder Frage des Benutzers und des Konversationsverlaufs beinhaltet. Angenommen, wir haben K Konversationsrunden, die durch $\{\mathbf{x}^1, \mathbf{y}^1, \mathbf{x}^2, \mathbf{y}^2, ..., \mathbf{x}^K, \mathbf{y}^K\}$ bezeichnet werden. Hier bezeichnen $\mathbf{x}^k$ und $\mathbf{y}^k$ die Benutzeranfrage bzw. die Antwort für jede Runde k. Die Log-Wahrscheinlichkeit der Generierung der Antwort kann als $\log \Pr_\theta(\mathbf{y}^k | \mathbf{x}^1, \mathbf{y}^1, ..., \mathbf{x}^k)$ geschrieben werden. Unser Ziel ist es dann, die Summe dieser Log-Wahrscheinlichkeiten zu maximieren

$$\tilde{\theta} \quad = \quad \arg\max_{\theta} \sum_{k=1}^{K} \log \Pr_\theta(\mathbf{y}^k | \mathbf{x}^1, \mathbf{y}^1, ..., \mathbf{x}^k) \tag{4.5}$$

Eine unkomplizierte Implementierung davon beinhaltet die Berechnung der bedingten Wahrscheinlichkeit für jedes k. Dies erfordert jedoch, das LLM K-mal auszuführen, jedes Mal mit einem erweiterten Konversationsverlauf, um Vorhersagen zu treffen. Eine effizientere Methode besteht darin, die Verlustberechnung aller Antworten in einem einzigen Durchlauf des LLMs durchzuführen. Dazu stellen wir die Konversation als eine Sequenz $\text{seq}_{\mathbf{x}^1,\mathbf{y}^1,...,\mathbf{x}^K,\mathbf{y}^K} = [\mathbf{x}^1, \mathbf{y}^1, ..., \mathbf{x}^K, \mathbf{y}^K]$ (oder kurz seq) dar. Die Log-Wahrscheinlichkeit dieser Sequenz ist gegeben durch

$$\begin{aligned} \log \Pr_\theta(\text{seq}) &= \log \Pr_\theta(\mathbf{x}^1, \mathbf{y}^1, ..., \mathbf{x}^K, \mathbf{y}^K) \\ &= \underbrace{\log \Pr_\theta(\mathbf{x}^1)}_{\text{auf 0 gesetzt}} + \underbrace{\log \Pr_\theta(\mathbf{y}^1 | \mathbf{x}^1)}_{\text{Verlustberechnung}} + \cdots + \\ &\quad \underbrace{\log \Pr_\theta(\mathbf{x}^K | \mathbf{x}^1, \mathbf{y}^1, ..., \mathbf{y}^{K-1})}_{\text{auf 0 gesetzt}} + \\ &\quad \underbrace{\log \Pr_\theta(\mathbf{y}^K | \mathbf{x}^1, \mathbf{y}^1, ..., \mathbf{x}^K)}_{\text{Verlustberechnung}} \end{aligned} \tag{4.6}$$

Der Trick hierbei ist, dass wir den Verlust (Loss) für die Generierung von Benutzereingaben ignorieren, wie in Abbildung 4.3 dargestellt. Daher berechnen wir nur die Wahrscheinlichkeiten für die Generierung der Antworten gegeben ihrer Konversationshistorien, mit anderen Worten, der Wert auf der rechten Seite von Gl. (4.6) ist tatsächlich gleich dem Wert auf der rechten Seite von Gl. (4.5). Wie bei Gl. (4.4) kann das Training dieses mehrstufigen Vorhersagemodells durch die Maximierung der Log-Likelihood über einen Trainingsdatensatz $\mathcal{D}$ erreicht werden:

$$\tilde{\theta} \quad = \quad \arg\max_{\theta} \sum_{\text{seq} \in \mathcal{D}} \log \Pr_\theta(\text{seq}) \tag{4.7}$$

Obwohl die Implementierung der oben vorgestellten SFT-Methoden trivial erscheint, da sie im Grunde genommen mit dem regulären Training von Sprachmodellen identisch sind, gibt es in der Praxis dennoch Probleme, die berücksichtigt werden müssen. Zum Beispiel,

- SFT erfordert gelabelte Daten. Dadurch unterscheidet sich SFT deutlich vom Vortraining, bei dem Rohtext als Trainingsdaten verwendet wird und leicht verfügbar ist. Wie bei anderen Problemen des überwachten maschinellen Lernens sind die Datenannotation und -auswahl bei SFT keine einfachen Aufgaben. Im Allgemeinen möchten wir SFT-Daten entwickeln, die sowohl quantitativ umfangreich als auch qualitativ hochwertig sind und die für die Aufgaben, die das LLM ausführen wird, von hoher Relevanz sein sollten. Andererseits besteht die Notwendigkeit, LLMs mit weniger Daten feinabzustimmen, um die Kosten für Berechnungen und Datenerstellung zu minimieren. Oft hängt die Qualität von LLMs stark von den in SFT

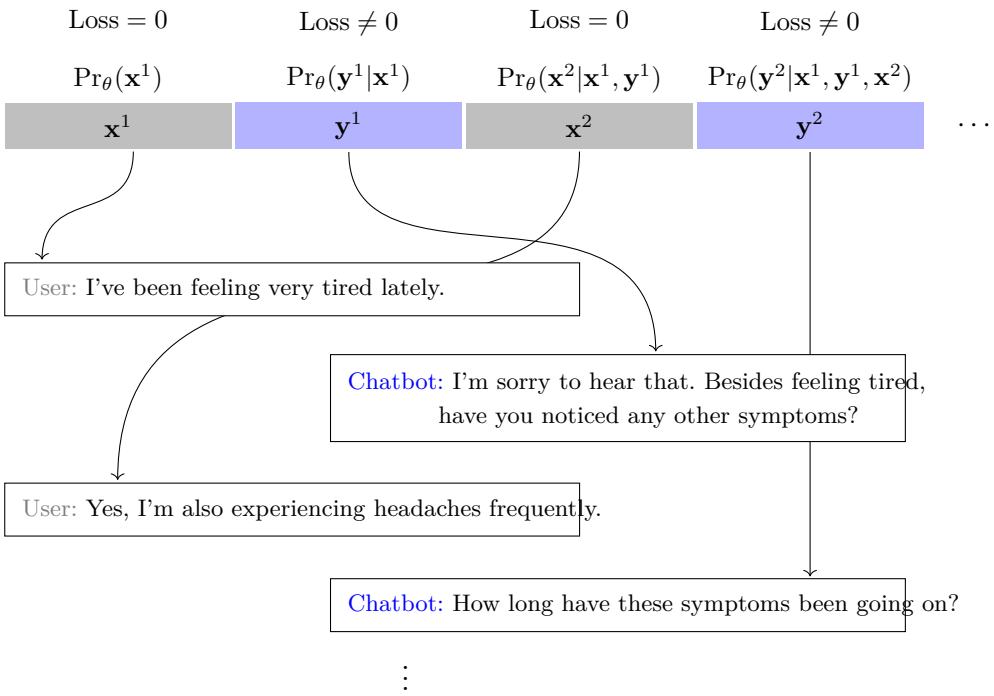


Abbildung 4.3: Veranschaulichung des überwachten Fine-Tunings für Konversationsmodelle. Hier fungiert das LLM als Chatbot, um auf jede Anfrage basierend auf dem Gesprächsverlauf zu antworten. Die Konversation entwickelt sich durch den Wechsel zwischen dem Benutzer und dem Chatbot. Beim SFT behandeln wir die gesamte Konversation wie bei Standard-LLMs als eine Sequenz, berechnen den Verlust jedoch nur für die Antworten des LLMs.

verwendeten Daten ab. Daher müssen solche Daten sorgfältig entwickelt und geprüft werden. Wie wir in späteren Unterabschnitten sehen werden, kann SFT durch fortgeschrittenere Techniken zur Datenerstellung effizienter und effektiver gestaltet werden.

- SFT ist für LLMs aufgrund ihrer Größe immer noch rechenintensiv. Daher ist die Wartung und Aktualisierung solcher Modelle ressourcenintensiv. Beispielsweise erfordert die Anwendung von Gradientenaktualisierungen auf Milliarden von Parametern innerhalb eines LLM erhebliche Rechenleistung und Speicher. Dies erfordert oft Hochleistungsrechenumgebungen, deren Betrieb kostspielig ist. Um diesen Herausforderungen zu begegnen, wurden verschiedene Optimierungsstrategien wie Pruning, Quantisierung und die Verwendung effizienterer Trainingsalgorithmen untersucht. Insbesondere besteht großes Interesse an parametereffizienten Feinabstimmungsmethoden, die darauf ausgelegt sind, eine Spitzenleistung ohne den Bedarf an umfangreichen Rechenressourcen aufrechtzuerhalten. Wir haben in Kapitel 3 gesehen, dass die Anwendung von Techniken wie Soft Prompts den Feinabstimmungsprozess effizienter gestalten kann. Für eine weiterführende Diskussion über parametereffiziente Methoden kann der Leser auf einschlägige Veröffentlichungen zu diesem Thema verweisen [Houlsby et al., 2019; Hu et al., 2022; Han et al., 2024].

- SFT kann als ein dem Vortraining folgender Nach-Trainingsschritt betrachtet werden. Es handelt sich um eine separate Trainingsphase, die darauf abzielt, die Vorteile

des ursprünglichen Vortrainings zu bewahren und gleichzeitig neue Anpassungen zu integrieren. Dies mag paradox erscheinen, da die Aktualisierung eines vortrainierten LLM mit weiteren Daten potenziell dazu führen kann, dass das Modell einen Teil seines Vorwissens vergisst. Stellen Sie sich ein Szenario vor, in dem wir eine große Menge an SFT-Daten haben und das LLM ausgiebig feinabstimmen. In diesem Fall könnte das LLM die Daten überanpassen, was wiederum die Generalisierungsleistung verringern oder katastrophales Vergessen verursachen kann. Eine gängige Strategie zur Minderung dieses Problems ist der Einsatz von Regularisierungs- und Frühstopp-Techniken. Ein weiterer praktischer Ansatz ist die Verwendung einer kleineren Lernrate, um die Gewichte des LLM behutsam anzupassen. Darüber hinaus kann auch die Feinabstimmung mit Daten aus verschiedenen Quellen und Problembereichen vorteilhaft sein. Dennoch wird der SFT-Schritt in der Praxis oft sorgfältig geprüft und erfordert erhebliche technische und experimentelle Anstrengungen zur Optimierung.

4.2.2 Erfassung von Feinabstimmungsdaten

Feinabstimmungsdaten sind so wichtig, dass sich viele neuere Arbeiten im Bereich LLM auf die Entwicklung verschiedener Datensätze für die Anweisungs-Feinabstimmung konzentriert haben. Wie bei den meisten Arbeiten im maschinellen Lernen gibt es im Allgemeinen zwei Ansätze zur Datenerfassung — die manuelle Datenerzeugung und die automatische Datenerzeugung.

4.2.2.1 Manuell erstellte Daten

Eine einfache Methode besteht darin, menschliche Annotatoren zu rekrutieren, um Eingabe-Ausgabe-Paare für die interessierenden Aufgaben zu erstellen. Im Gegensatz zur Datenannotation im herkömmlichen NLP, wie z. B. bei der Textklassifizierung, bei der Annotatoren gesammelten Texten einfach gemäß Richtlinien Etiketten zuweisen, erfordert die Erstellung von Feinabstimmungsdaten für LLMs mehr Schritte und Aufwand, was sie daher anspruchsvoller macht. Angenommen, wir möchten Feinabstimmungsdaten für die Aufgabe der maschinellen Übersetzung vom Englischen ins Chinesische erhalten. Der erste Schritt besteht darin, eine Prompt-Vorlage zu schreiben, um die Aufgabe zu beschreiben und das Problem klar zu formatieren. Zum Beispiel,

Instruction	Translate the text from English to Chinese.
User Input	{∗text∗}
Output	{∗translation∗}

Dann sammeln wir Paare von Quell- und Zieltexten (d. h. chinesische Texte und die entsprechenden Übersetzungen) und ersetzen die Variablen {∗text∗} und {∗translation∗}, um die Feinabstimmungsmuster zu generieren. Gegeben zum Beispiel ein Paar englischer und chinesischer Sätze

$$\underset{\{*\text{text}*\}}{\text{How's the weather today?}} \quad \rightarrow \quad \underset{\{*\text{translation}*\}}{\text{今天天气怎么样？}}$$

können wir mit der Prompt-Vorlage ein Feinabstimmungsmuster generieren, wie dieses

Instruction	Translate the text from English to Chinese.
User Input	How's the weather today?
Output	今天天气怎么样？

Das heißt,

$$\begin{aligned} \mathbf{x} &= \text{Translate the text from English to Chinese.\textbackslash n How's the weather today?} \\ \mathbf{y} &= \text{今天天气怎么样？} \end{aligned}$$

Wir können dieses $(\mathbf{x}, \mathbf{y})$-Paar verwenden, um das LLM feinabzustimmen, wie im vorherigen Unterabschnitt beschrieben.

Eine Schwierigkeit hierbei ist, dass es viele verschiedene Möglichkeiten gibt, Prompt-Vorlagen für dieselbe Aufgabe zu schreiben, und verschiedene Personen können Prompt-Vorlagen mit unterschiedlichen Qualitäten und Komplexitäten erstellen. Manchmal schreiben wir möglicherweise Prompt-Vorlagen mit übermäßig komplexen oder ausführlichen Anweisungen. Manchmal wissen wir vielleicht nicht einmal genau, was die Zielaufgabe ist und wie wir sie beschreiben sollen. Eine weit verbreitete Strategie besteht darin, Prompt-Vorlagen für bestehende NLP-Aufgaben zu erstellen, da es so viele gut etablierte NLP-Probleme und Benchmarks gibt [Bach et al., 2022; Wang et al., 2022b; Mishra et al., 2022]. In diesem Fall können den Annotatoren die ursprüngliche Aufgabenbeschreibung und viele Beispiele gegeben werden. Dann können sie auf ihre eigene Weise ausdrücken, wie sie das LLM dazu auffordern, die Aufgaben auszuführen. Beachten Sie, dass, obwohl eine solche Methode den Prozess der Erstellung und des Schreibens von Prompts erleichtern kann, wir immer noch Annotations-Frameworks und Crowdsourcing-Systeme benötigen, um die Arbeit zu verwalten und Qualitätskontrollen durchzuführen. Zum Beispiel müssen wir im Allgemeinen Annotationsrichtlinien und ein einheitliches Format für das Schreiben von Prompt-Vorlagen entwerfen, insbesondere wenn viele Annotatoren zur selben Aufgabe beitragen. Ein Vorteil der Ableitung von Prompts aus bestehenden NLP-Aufgaben besteht darin, dass, sobald die Prompt-Vorlagen entwickelt wurden, es einfach ist, Prompts unter Verwendung der annotierten Muster in den ursprünglichen Aufgaben zu generieren. Zum Beispiel können wir bei einem zweisprachigen Datensatz für die Englisch-Chinesisch-Übersetzung leicht eine Reihe von Feinabstimmungsbeispielen erstellen, indem wir die Lücken in der obigen Vorlage mit den Satzpaaren in diesem Datensatz füllen.

Ein anderer Ansatz besteht darin, die natürlich vorhandenen, im Internet verfügbaren Daten direkt zu nutzen. Ein gängiges Beispiel ist das Sammeln von Frage-Antwort-Paaren von QA-Websites, um LLMs für Open-Domain-QA-Aufgaben feinabzustimmen [Joshi et al., 2017]. Viele Benchmarks im QA-Bereich werden auf diese Weise erstellt, da es so viele Arten von Fragen gibt, dass es für eine kleine Gruppe von Menschen unmöglich

ist, sich alle auszudenken. Stattdessen kann die Verwendung von Daten von diesen Websites sicherstellen, dass die Feinabstimmungsdaten des LLM in Bezug auf Quantität und Qualität auf einem guten oder akzeptablen Niveau sind.

Neben der Nutzung bestehender Ressourcen ist eine weitere einfache Möglichkeit, einen Feinabstimmungsdatensatz zu entwickeln, das Crowdsourcing der Daten. Ein einfacher Ansatz besteht darin, den Benutzern zu erlauben, eine beliebige Frage einzugeben, woraufhin Antworten entweder manuell gegeben oder automatisch von einem LLM generiert und dann manuell annotiert und korrigiert werden. So ist es möglich, das reale Benutzerverhalten zu erfassen und folglich Eingaben und Ausgaben für eine große Anzahl von „neuen"Problemen zu sammeln, die traditionelle NLP-Aufgaben nicht abdecken.

Ein Problem im Zusammenhang mit der Erstellung von Feinabstimmungsdatensätzen ist, dass wir normalerweise möchten, dass die Daten so vielfältig wie möglich sind. Viele Studien haben ergeben, dass eine Erhöhung der Vielfalt der Feinabstimmungsdaten die Robustheit und Generalisierungsfähigkeit von LLMs verbessern kann. Aus diesem Grund besteht ein erhebliches Interesse daran, vielfältigere Prompts und Aufgaben in die Feinabstimmungsdatensätze von LLMs einzubeziehen. Wir werden eine weitere Diskussion über die Generalisierung der Feinabstimmung in Abschnitt 4.2.4 bereitstellen.

4.2.2.2 Automatisch generierte Daten

Eine Einschränkung der manuellen Datengenerierung besteht darin, dass Qualität und Vielfalt stark von menschlicher Erfahrung und Kreativität abhängen. Wenn wir also wollen, dass LLMs ein breites Spektrum an Aufgaben bewältigen, d. h. jede Anweisung effektiv ausführen können, ist es oft ineffizient, sich für das Fine-Tuning von LLMs auf von Menschen annotierte Daten zu verlassen. Darüber hinaus kann die Abdeckung solcher Daten begrenzt sein, und die Daten können sogar Verzerrungen enthalten, die von den Annotatoren selbst eingeführt wurden. Ein alternativer Ansatz ist die automatische Generierung von Daten. Zum Beispiel können wir eine Reihe von Fragen durch Crowdsourcing sammeln und ein gut abgestimmtes LLM einsetzen, um Antworten auf die Fragen zu generieren. Diese Frage-Antwort-Paare werden dann wie üblich als Stichproben für das Fine-Tuning verwendet. Diese Methode, obwohl sehr einfach, wurde ausgiebig angewendet, um umfangreiche Fine-Tuning-Daten für LLMs zu generieren.

Die oben beschriebene Art der Erzeugung synthetischer Fine-Tuning-Daten ähnelt denen, die bei der Datenaugmentation für NLP verwendet werden. Wenn wir ein LLM haben, können wir als Reaktion auf jede Eingabe eine Vorhersage erstellen. Die Wiederholung dieses Prozesses für verschiedene Eingaben ermöglicht es uns, eine ausreichende Anzahl von Stichproben für das Fine-Tuning zu erstellen. Eine solche Methode ist besonders nützlich für das Fine-Tuning neuer LLMs unter Verwendung eines gut abgestimmten LLMs. Ein Nachteil dieses Ansatzes ist jedoch, dass er auf von Menschen erstellten oder gesammelten Eingaben für die Datengenerierung beruht, die sich als ungeeignet für die Generalisierung von LLMs erweisen können. In vielen LLM-Anwendungen ergibt sich eine erhebliche Herausforderung aus der großen Bandbreite an Fragen und Anfragen der Benutzer, von denen viele in bestehenden NLP-Aufgaben und Datensätzen nicht abgedeckt sind. In diesen Fällen wird es notwendig, nicht nur die Vorhersagen, sondern auch die Eingaben selbst zu generieren.

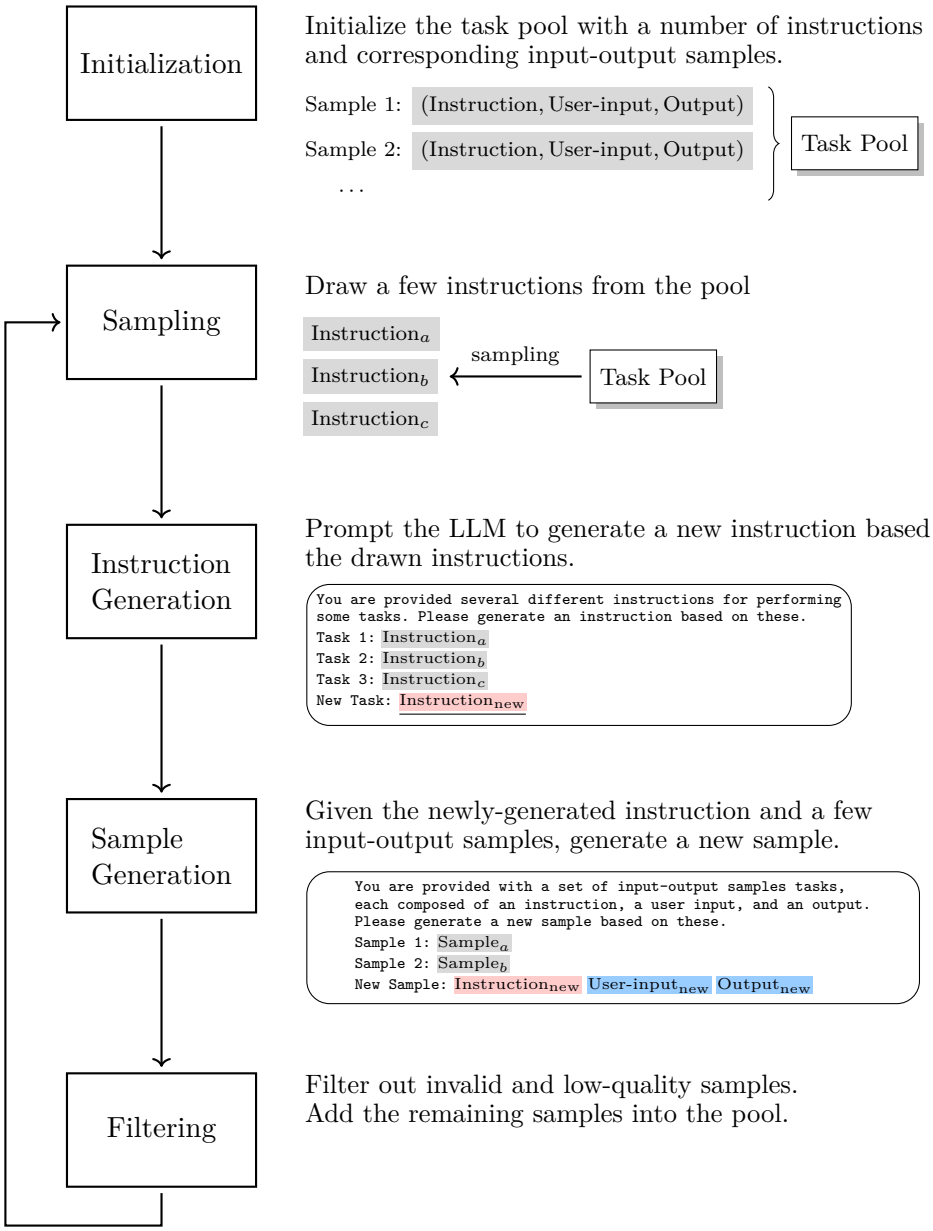


Abbildung 4.4: Darstellung von Self-Instruct [Wang et al., 2023b]. Diese Methode verwaltet einen Pool von Anweisungen und entsprechenden Eingabe-Ausgabe-Beispielen. Anfänglich enthält der Pool eine Reihe von manuell erstellten Anweisungen und Beispielen. Jedes Mal werden einige Anweisungen aus dem Pool entnommen. Ein LLM wird dann aufgefordert, auf der Grundlage der entnommenen Anweisungen neue Anweisungen und Beispiele zu generieren. Schließlich werden die neu generierten Anweisungen und Beispiele gefiltert und dem Pool hinzugefügt.

Hier betrachten wir self-instruct als Beispiel, um zu veranschaulichen, wie man Stichproben für das LLM-Fine-Tuning generiert [Wang et al., 2023d; Honovich et al., 2023]. Die Idee ist, dass wir ein LLM anweisen können, eine neue Anweisung zu erstellen, indem es von anderen Anweisungen lernt. Auf der Grundlage dieser Anweisung kann das LLM dann andere Felder (wie die Benutzereingabe) ausfüllen und die Vorhersagen erstellen. Abbildung 4.4 zeigt eine schematische Darstellung von Self-Instruct. Hier geben wir einen kurzen Überblick über die wichtigsten Schritte.

- Der Self-Instruct-Algorithmus verwaltet einen Pool von Aufgaben. Anfänglich enthält er eine Anzahl von handgefertigten Seed-Aufgaben, jede mit einer Anweisung und einem Eingabe-Ausgabe-Beispiel. Im Verlauf des Algorithmus werden von LLMs generierte Anweisungen und Beispiele diesem Pool hinzugefügt.

- In jedem Schritt wird eine kleine Anzahl von Anweisungen aus dem Anweisungspool entnommen. Zum Beispiel können wir zufällig einige von Menschen geschriebene Anweisungen und einige von LLMs generierte Anweisungen auswählen, um die Vielfalt zu gewährleisten.

- Die ausgewählten Anweisungen werden dann als Demonstrationsbeispiele verwendet. So kann das LLM aus diesen Beispielen im Kontext lernen und eine neue Anweisung erstellen. Unten sehen Sie eine Beispielvorlage für das Prompting des LLM.

> You are provided several different instructions for performing some tasks. Please generate an instruction based on these.
>
> Task 1: {instruction1}
>
> Task 2: {instruction2}
>
> Task 3: {instruction3}
>
> Task 4: {instruction4}
>
> New Task: ____

- Anhand der generierten Anweisung wird das LLM dann aufgefordert, das Beispiel zu vervollständigen, indem es die verbleibenden Eingabefelder ausfüllt und die entsprechende Ausgabe generiert. Unten sehen Sie eine Prompt-Vorlage.

> You are provided with a set of input-output samples, each composed of an instruction, a user input, and an output. Please generate a new sample based on these.
>
> Sample 1: {instruction1}
>
> Input: {user-input1}
>
> Output: {output1}
>
> Sample 2: {instruction2}
>
> Input: {user-input2}
>
> Output: {output2}
>
> New Sample: {new-instruction}
>
> ____

- Dieses neu generierte Beispiel wird durch einige heuristische Regeln geprüft (z. B. das Herausfiltern von Beispielen oder Anweisungen, die denen im Pool bereits ähneln). Wenn es besteht, werden das Beispiel und die Anweisung dem Pool hinzugefügt.

Dieser Generierungsprozess kann viele Male wiederholt werden, um eine ausreichende Anzahl von Stichproben für das Fine-Tuning zu erhalten. Beachten Sie, dass wir oben nur einfache Prompt-Vorlagen zur Generierung von Anweisungen und Fine-Tuning-Stichproben zeigen. Natürlich können wir bessere Vorlagen entwickeln, um vielfältigere und genauere Anweisungen und Fine-Tuning-Stichproben zu generieren. Zum Beispiel neigt das LLM bei bestimmten Aufgaben wie der Textklassifikation dazu, verzerrte Vorhersagen zu machen, z. B. gehören die meisten generierten Stichproben zu einer einzigen Klasse. In solchen Fällen können wir die Reihenfolge der Generierung der verschiedenen Felder anpassen. Genauer gesagt können wir die Ausgabe (d. h. die Klasse) mit einer A-priori-Wahrscheinlichkeit festlegen und das LLM anweisen, die Benutzereingabe basierend auf der Anweisung und der Ausgabe zu generieren. Diese Methode ähnelt dem input inversion, bei dem das LLM die Eingabe basierend auf der angegebenen Ausgabe generiert [Longpre et al., 2023].

Die Verwendung von LLM-generierten Anweisungen und Fine-Tuning-Stichproben ist eine gängige Methode zur Entwicklung von LLMs, insbesondere da die manuelle Entwicklung solcher Daten so teuer ist, dass sich die meisten Forschungsgruppen dies nicht leisten können. In mehreren gut abgestimmten LLMs enthalten ihre Fine-Tuning-Datensätze eine gewisse Menge an synthetischen Daten, was sich als nützlich erwiesen hat [Ouyang et al., 2022; Taori et al., 2023; Chiang et al., 2023]. Es gab weitere Studien zur Generierung synthetischer Daten für das LLM-Fine-Tuning. Zum Beispiel kann man durch die Einführung evolutionärer Algorithmen vielfältigere Anweisungen generieren [Xu et al., 2024] oder synthetische Daten als Supervisionssignale in einem fortschrittlicheren Fine-Tuning-Prozess verwenden [Chen et al., 2024b]. In jüngerer Zeit gab es auch ein beträchtliches Interesse an der Verwendung synthetischer Daten in der Vortrainingsphase [Gunasekar et al., 2023; Allal et al., 2024].

In vielen Anwendungen ist ein reales Szenario, dass wir für eine gegebene Aufgabe eine relativ kleine Menge an Fine-Tuning-Daten sammeln oder annotieren können, zum Beispiel können wir Experten beauftragen, Fragen für QA-Aufgaben in einem bestimmten Bereich zu erstellen. Aber die Menge und Vielfalt dieser Daten sind im Allgemeinen nicht ausreichend. In diesem Fall können wir Self-Instruct-Techniken verwenden, um vielfältigere Frage-Antwort-Paare zu generieren und so die Fine-Tuning-Daten zu erweitern. Dies bietet eine Möglichkeit, das LLM von einem Startsatz von Fine-Tuning-Stichproben aus zu bootstrappen. Beachten Sie, dass die Verwendung selbstgenerierter Daten eine gängige Praxis ist und seit langem im NLP angewendet wird. Zum Beispiel wurde dieser Ansatz erfolgreich beim Parsing und bei der maschinellen Übersetzung eingesetzt [Charniak, 1997; Sennrich et al., 2016].

4.2.3 Feinabstimmung mit weniger Daten

Mit der zunehmenden Bedeutung der anweisungsbasierten Feinabstimmung ist die Nachfrage nach umfangreichen, qualitativ hochwertigen Feinabstimmungsdaten stark gestiegen. Beispielsweise enthält der FLAN-Feinabstimmungsdatensatz, der aus 1.836 Aufgaben zusammengestellt wurde, 15 Millionen Beispiele [Longpre et al., 2023]. Die Feinabstimmung von LLMs mit solch großen Datensätzen ist in der Regel eine rechenintensive Aufgabe,

insbesondere da die Aktualisierung der großen Anzahl von Parametern in LLMs ressourcenintensiv ist. Ein Ansatz zur Minderung dieses Problems besteht darin, effiziente Modelltrainingsmethoden zu erforschen. Beispielsweise kann man parametereffiziente Methoden verwenden, um nur einen kleinen Teil des Modells zu aktualisieren. Viele Feinabstimmungsdatensätze enthalten jedoch eine große Menge an synthetischen Daten, bei denen Fehler und Verzerrungen dennoch unvermeidlich sind.

Ein weiterer Ansatz zur effizienten Feinabstimmung besteht darin, nur die relevantesten und wirkungsvollsten Beispiele für die Feinabstimmung zu berücksichtigen. So können wir die zu verarbeitende Datenmenge reduzieren und gleichzeitig die Qualität der Modellaktualisierungen beibehalten. Es gibt mehrere Methoden, um dies zu erreichen. Beispielsweise erstellten Zhou et al. [2023a] einen anweisungsfolgenden Datensatz mit nur 1.000 Beispielen, indem sie die Prompts sorgfältig gestalteten und Beispiele aus einer Vielzahl von NLP-Aufgaben sammelten. Sie zeigten, dass das mit diesem Datensatz feinabgestimmte LLaMa-65B-Modell mit Modellen, die einen wesentlich größeren Feinabstimmungsaufwand erforderten, konkurrenzfähig oder sogar besser sein konnte. Dies deutet darauf hin, dass LLMs an die Beantwortung verschiedener Aufgaben angepasst werden können, ohne dass notwendigerweise eine Feinabstimmung auf allen Arten von anweisungsfolgenden Daten erforderlich ist. Chen et al. [2024a] entwickelten ein auf dem GPT-3.5-Modell basierendes System zur Bewertung der Qualität jedes anweisungsfolgenden Beispiels. Daher konnten sie qualitativ hochwertige Beispiele aus bestehenden Datensätzen auswählen und zeigten eine bessere Feinabstimmungsleistung mit weniger Feinabstimmungsbeispielen. Forscher haben auch Methoden entwickelt, um Daten entweder mithilfe von Heuristiken auszuwählen oder herauszufiltern [Zhao et al., 2024; Ge et al., 2024] oder um Daten zu priorisieren, die den Feinabstimmungsprozess maßgeblicher beeinflussen [Xia et al., 2024]. Tatsächlich können die meisten dieser Methoden als Instanzen größerer Familien von Datenauswahl- und Filtermethoden angesehen werden. Und es ist oft der Fall, dass die Verwendung von qualitativ hochwertigeren (aber vielleicht weniger) Daten für das Training von NLP-Modellen vorteilhaft ist.

Die Erkenntnisse bei der anweisungsbasierten Feinabstimmung weichen etwas von den traditionellen Ansichten im NLP ab: Die Fähigkeit von Modellen, komplexe Probleme zu bewältigen, kann mit einer kleinen Menge annotierter Daten aktiviert werden, anstatt massive Mengen an überwachten Daten für ein umfangreiches Training zu erfordern. Eine mögliche Erklärung ist, dass die Fähigkeit, auf Anweisungen hin korrekte Antworten zu generieren, während des Vortrainings erlernt wurde, aber solche Anweisungs-Antwort-Zuordnungen während der Inferenz keine hohen Wahrscheinlichkeiten aufweisen. Durch die Feinabstimmung können die Modelle geringfügig angepasst werden, damit sie Anweisungen befolgen, was deutlich weniger Trainingsaufwand erfordert als das Vortraining. Dies steht in engem Zusammenhang mit der sogenannten superficial alignment hypothesis, die besagt, dass das Lernen hauptsächlich während des Vortrainings stattfindet und die anschließende Feinabstimmungs- oder Anpassungsphase nicht wesentlich zur zugrunde liegenden Wissensbasis eines LLM beiträgt [Zhou et al., 2023a]. Da die Kernfähigkeiten und das Wissen des Modells bereits durch das Vortraining etabliert sind, kann eine effektive Feinabstimmung zur Anpassung an die Benutzerbedürfnisse mit relativ geringem Trainingsaufwand für die Feinabstimmung erreicht werden. Dies impliziert die Möglichkeit, LLMs mit sehr wenigen Daten feinabzustimmen. In einer anderen Richtung ist es

möglicherweise nicht notwendig, die Feinabstimmung auf gepaarte Anweisungs-Antwort-Daten zu beschränken. Beispielsweise fanden Hewitt et al. [2024] heraus, dass das Befolgen von Anweisungen implizit erreicht werden kann, indem LLMs nur auf Antworten feinabgestimmt werden, ohne entsprechende Anweisungen.

Ein mit dieser Diskussion verwandtes Konzept ist die Stichprobeneffizienz. Eine Methode des maschinellen Lernens wird als stichprobeneffizient bezeichnet, wenn sie effektiv aus einer kleinen Anzahl von Trainingsbeispielen lernen kann. In diesem Sinne ist die anweisungsbasierte Feinabstimmung im Vergleich zum Vortraining stichprobeneffizient. Aus der Perspektive des maschinellen Lernens können stichprobeneffiziente Methoden als effiziente Wege zur Abtastung des Datenraums angesehen werden und sind vorteilhaft, da sie knappe Daten optimal nutzen. Daher können stichprobenbasierte Lerntechniken, wie viele Algorithmen des Verstärkenden Lernens, von diesen stichprobeneffizienten Ansätzen profitieren. Beispielsweise können wir bei der Anpassung an menschliche Präferenzen entweder Präferenzdaten über Belohnungsmodelle effizient abtasten [Liu et al., 2024b] oder die Stichprobeneffizienz beim Richtlinienlernen verbessern [Wang et al., 2024].

4.2.4 Instruktionsgeneralisierung

Bei vielen Problemen des maschinellen Lernens und der NLP ist das Trainieren eines Modells zur Generalisierung ein grundlegendes Ziel. Zum Beispiel erwarten wir bei der Textklassifizierung, dass unser Modell neue Texte, die während des Trainings nicht gesehen wurden, korrekt klassifiziert. Die Generalisierung stellt jedoch bei der Instruktions-Feinabstimmung zusätzliche Herausforderungen dar. Wir erwarten von instruktions-feinabgestimmten LLMs, dass sie nicht nur angemessene Antworten auf verschiedene Eingaben innerhalb einer Aufgabe generieren, sondern auch verschiedene Aufgaben, wie sie durch unterschiedliche Instruktionen beschrieben werden, präzise ausführen. Um dieses Problem zu veranschaulichen, betrachten wir ein LLM $\Pr(\mathbf{y}|\mathbf{c},\mathbf{z})$, wobei $\mathbf{c}$ eine Instruktion, $\mathbf{z}$ eine Benutzereingabe und $\mathbf{y}$ die entsprechende Modellausgabe (d. h. die Antwort) ist. Angenommen, die Leistung dieses Modells wird anhand einer Metrik bewertet, die als $\text{Performance}(\Pr(\mathbf{y}|\mathbf{c},\mathbf{z}))$ oder kurz $\mathrm{P}(\mathbf{c},\mathbf{z},\mathbf{y})$ geschrieben wird. Informell sagen wir, dass dieses Modell innerhalb einer gegebenen Aufgabe (angegeben durch die Instruktion $\mathbf{c}^*$) generalisieren kann, wenn es einen Wert ϵ geben kann, sodass die durchschnittliche Leistung bei neuen Eingaben über diesem Wert liegt:

$$\frac{1}{|\mathcal{Z}|}\sum_{\mathbf{z}'\in\mathcal{Z}}\mathrm{P}(\mathbf{c}^*,\mathbf{z}',\mathbf{y}')>\epsilon \tag{4.8}$$

wobei $\mathcal{Z}$ die Menge der neuen Eingaben ist und $\mathbf{z}'$ und $\mathbf{y}'$ eine Eingabe in dieser Menge bzw. die entsprechende Ausgabe sind.

Ebenso können wir sagen, dass dieses Modell aufgabenübergreifend generalisieren kann, wenn die durchschnittliche Leistung über alle Instruktions-Eingabe-Paare über einem gewissen ϵ liegt:

$$\frac{1}{|\mathcal{D}|}\sum_{(\mathbf{c}',\mathbf{z}')\in\mathcal{D}}\mathrm{P}(\mathbf{c}',\mathbf{z}',\mathbf{y}')>\epsilon \tag{4.9}$$

wobei $\mathcal{D}$ die Menge der neuen Instruktions-Eingabe-Paare ist.

Hier müssen wir uns mit Variationen in zwei Dimensionen befassen: Instruktion und Benutzereingabe. Dies macht das Generalisierungsproblem sehr komplex, da ein Modell intuitiv von einer großen Anzahl von Aufgaben und den damit verbundenen verschiedenen Eingabe-Ausgabe-Paaren lernen muss, um eine gute Generalisierung zu erreichen. Wie wir in diesem Buch bereits mehrfach erörtert haben, verursacht das Erreichen einer solchen Generalisierung wesentlich geringere Kosten als das Vortraining. Im Allgemeinen kann die Feinabstimmung von LLMs mit Instruktions-Antwort-Daten bis zu einem gewissen Grad dazu führen, dass Modelle bei neuen Aufgaben Instruktionen befolgen. Dennoch wird allgemein angenommen, dass noch gewisse Anstrengungen erforderlich sind, um LLMs anzupassen, damit sie Instruktionen umfassend verstehen und ausführen können.

Eine Möglichkeit, die Instruktions-Feinabstimmung zu generalisieren, besteht darin, die Vielfalt der Feinabstimmungsdaten zu erhöhen. In früheren Studien zur Instruktions-Feinabstimmung entwickelten Forscher viele Datensätze, die eine große Vielfalt an NLP-Aufgaben und unterschiedliche Instruktionen für jede Aufgabe abdecken [Wang et al., 2022b; Sanh et al., 2022; Longpre et al., 2023]. Durch die Umwandlung dieser Aufgaben in ein einheitliches Format kann man ein LLM mit einer ausreichend großen Anzahl von Beispielen feinabstimmen; so gab es beispielsweise mehrere Datensätze zur Instruktions-Feinabstimmung, die über 100 NLP-Aufgaben und 1 Million Beispiele umfassen. Diese frühen Datensätze konzentrieren sich jedoch hauptsächlich auf bestehende akademische Probleme und nicht auf solche, mit denen Benutzer in realen Anwendungen umgehen möchten. Viele neuere Arbeiten haben ihren Fokus auf die Bewältigung neuer und praktischerer Probleme verlagert. Zum Beispiel gab es ein beträchtliches Interesse an der Erstellung von Datensätzen, die umfangreiche und komplizierte Demonstrationen und Antworten von SOTA-Modellen auf reale Benutzeranfragen enthalten [Wang et al., 2023c; Teknium, 2023].

Vielleicht hat die Verwendung großer und vielfältiger Feinabstimmungsdatensätze ihren Ursprung in Versuchen, LLMs in verschiedenen Dimensionen zu skalieren. Tatsächlich wurden Skalierungsgesetze weithin genutzt, um die Entwicklung einer breiten Palette verschiedener instruktions-feinabgestimmter LLMs zu motivieren. Und es ist vernünftig, die Instruktions-Feinabstimmung zu skalieren, damit ein LLM breite Instruktionen befolgen kann. Aus der Perspektive der LLM-Anpassung ist die Skalierung der Instruktions-Feinabstimmung jedoch möglicherweise nicht effizient, um eine Generalisierung zu erreichen.

Ein Problem ist, dass die Instruktions-Feinabstimmung auf überwachtem Lernen beruht, das lernt, auf der Grundlage von Instruktions-Antwort-Zuordnungen zu generalisieren und Aufgaben auszuführen. Ein solcher Ansatz erfasst jedoch keine subtilen oder komplexen menschlichen Präferenzen (z. B. Ton, Stil oder subjektive Qualität), da diese schwer als explizite Instruktions-Antwort-Daten zu kodieren sind. Darüber hinaus wird die Generalisierungsleistung durch die Vielfalt und Qualität des Instruktions-Antwort-Datensatzes begrenzt. Angesichts dieser Einschränkungen möchten wir stattdessen Präferenzmodelle als zusätzlichen Feinabstimmungsschritt nach der Instruktions-Feinabstimmung einsetzen, damit die LLMs weiter generalisieren können (siehe Abschnitt 4.3).

Eine andere Sichtweise ist, dass einige Instruktions-Antwort-Zuordnungen bereits während des Vortrainings gelernt worden sein könnten und die vor-trainierten LLMs solche

Zuordnungen kodiert haben. Da wir jedoch oft nicht genau wissen, welche Daten beim Vortraining verwendet werden, ist es schwer zu beurteilen, ob wir solche Zuordnungen bei der Feinabstimmung lernen müssen. Eine verwandte Frage ist, ob die Out-of-Distribution-Generalisierung hauptsächlich während des Vortrainings oder der Feinabstimmung erreicht wird. Obwohl die direkte Beantwortung dieser Frage den Rahmen dieses Kapitels sprengen würde, wurde gezeigt, dass das Vortraining auf großen und vielfältigen Datensätzen die Out-of-Distribution-Leistung wirksam verbessert [Hendrycks et al., 2020; Radford et al., 2021; Gunasekar et al., 2023]. Dies wirft ein interessantes Problem auf: Wenn ein LLM in großem Umfang gut vor-trainiert wurde, ist die Feinabstimmung für die Out-of-Distribution-Generalisierung möglicherweise nicht so wesentlich, da das Modell bereits auf ausreichende Verteilungsvariationen gestoßen sein könnte. Dies veranlasst Forscher dazu, LLMs mit bescheidenem Aufwand feinabzustimmen oder neue Methoden zu erforschen, um das Befolgen von Instruktionen zu erreichen. Wie im vorherigen Unterabschnitt erörtert, kann das Befolgen von Instruktionen beispielsweise durch Feinabstimmung auf einer kleinen Anzahl sorgfältig ausgewählter Instruktions-Antwort-Paare erreicht werden [Zhou et al., 2023a] oder sogar durch Methoden, die nicht explizit dafür entwickelt wurden [Kung and Peng, 2023].

Die obige Diskussion liefert zwei unterschiedliche Strategien: Die eine erfordert die Skalierung von Feinabstimmungsdatensätzen für eine größere Vielfalt, die andere erfordert kleine, aber notwendige Feinabstimmungsdatensätze für eine effiziente LLM-Anpassung. In der Praxis hilft es jedoch oft, vielfältige Instruktionen einzubeziehen. In vielen Fällen müssen wir unser LLM für bestimmte Zwecke anpassen. Aber das LLM, das möglicherweise während des Vortrainings breite Zuordnungen zum Befolgen von Instruktionen kodiert hat, könnte dazu neigen, sich selbst bei bescheidener Feinabstimmung wie ein allgemeiner Instruktionsausführer zu verhalten. Ein interessantes Phänomen ist, dass bei der Feinabstimmung auf mathematischen Daten das resultierende LLM sich möglicherweise nicht auf mathematische Ausgaben spezialisiert. Stattdessen könnte dieses Modell normal auf allgemeine Instruktionen reagieren, zum Beispiel könnte es Gedichte generieren, wenn es dazu angewiesen wird [Hewitt, 2024]. Das ist keine schlechte Sache, aber es zeigt, dass LLMs ihre Natur, allgemeine Instruktionen zu befolgen, möglicherweise nicht leicht ändern. In diesem Fall können zusätzliche Anpassungen mit vielfältigeren Daten helfen, die Art und Weise anzupassen, wie das LLM Instruktionen befolgt, insbesondere für die Aufgaben, die wir angehen wollen.

4.2.5 Verwendung schwacher Modelle zur Verbesserung starker Modelle

Bisher haben wir eine Vielzahl von Methoden zur Instruktions-Feinabstimmung auf der Grundlage von gelabelten Daten untersucht. Eine der Einschränkungen vieler solcher Methoden besteht darin, dass die Daten von Menschen annotiert oder von starken LLMs generiert werden müssen, die genaue Überwachungssignale für die Feinabstimmung liefern können. In vielen Fällen ist das LLM, das wir zur Verfügung haben, jedoch bereits stark (oder zumindest in bestimmten Aspekten der Problemlösung vorteilhaft), und daher ist es nicht einfach, ein überlegenes Modell zur Überwachung zu finden. Selbst für menschliche Experten kann es schwierig oder manchmal sogar undurchführbar sein, korrekte und detaillierte Antworten zu geben, wenn das Problem komplex wird. Wenn Experten beispielsweise mit einem extrem langen Dokument konfrontiert werden, fänden sie es schwierig,

ohne eine erschöpfende und zeitaufwändige Überprüfung Inkonsistenzen, subtile Verzerrungen oder fehlende Schlüsselpunkte zu identifizieren.

An dieser Stelle könnte man fragen: Können wir schwache LLMs verwenden, um starke LLMs zu überwachen? Dies scheint eine erhebliche Herausforderung zu sein, könnte aber ein zukünftiges Szenario widerspiegeln, in dem wir KI-Systeme überwachen müssen, die intelligenter sind als Menschen oder jedes andere KI-System [Burns et al., 2023b]. Das Problem, kleinere, weniger komplexe Modelle zur Verbesserung des Trainings größerer, komplexerer Modelle zu verwenden, wird auch als das Problem der Schwach-zu-Stark-Generalisierung bezeichnet. Obwohl es noch keine ausgereiften Ansätze zur Schwach-zu-Stark-Generalisierung gibt, hat sich die Verwendung kleinerer Modelle zur Unterstützung stärkerer Modelle in mehreren Bereichen von LLMs als nützlich erwiesen.

Für die Instruktions-Feinabstimmung ist eine der einfachsten Möglichkeiten, schwache LLMs anzuwenden, diese Modelle zur Generierung synthetischer Feinabstimmungsdaten zu verwenden. Angenommen, wir haben eine Sammlung von Eingaben X, wobei jede Eingabe eine Anweisung und bei Bedarf eine Benutzereingabe enthält. Für jedes $\mathbf{x} \in X$ verwenden wir ein schwaches LLM $\Pr^w(\cdot)$, um eine Vorhersage $\hat{\mathbf{y}} = \arg\max_{\mathbf{y}} \Pr^w(\mathbf{y}|\mathbf{x})$ zu generieren. Dann kann das starke LLM $\Pr^s_\theta(\cdot)$ auf diesen generierten Vorhersagen trainiert werden (siehe Gl. (4.1)):

$$\tilde{\theta} \quad = \quad \arg\max_{\theta} \sum_{\mathbf{x} \in X} \log \Pr^s_\theta(\hat{\mathbf{y}}|\mathbf{x}) \tag{4.10}$$

wobei θ die Modellparameter sind.

Die obige Form transformiert das Feinabstimmungsproblem in ein Wissensdestillationsproblem, mit anderen Worten, wir destillieren Wissen vom schwachen Modell auf das starke Modell. Folglich können wir verschiedene Wissensdestillationsmethoden anwenden, um dieses Ziel zu erreichen. Die Erklärung der Schwach-zu-Stark-Feinabstimmung aus der Perspektive der Wissensdestillation ist jedoch nicht trivial. Eine große Sorge ist, dass das starke Modell lediglich die Fehler des schwachen Modells imitiert oder überanpasst und nicht generalisiert. Zum Beispiel kann das feinabgestimmte starke Modell immer noch keine schwierigen Probleme lösen, die das schwache Modell nicht genau vorhersagen kann. Glücklicherweise haben vorläufige Experimente in diesem Forschungsbereich positive und vielversprechende Ergebnisse gezeigt. Zum Beispiel fanden Burns et al. [2023a], dass die Feinabstimmung des starken, vor-trainierten GPT-4-Modells mit Überwachung auf GPT-2-Niveau die Generalisierung über mehrere NLP-Aufgaben hinweg verbessern konnte. Um zu messen, wie das schwache Modell die Generalisierung des starken Modells verbessert, definieren wir die folgenden Begriffe:

- Weak Performance ($\mathrm{P}_{\mathrm{weak}}$). Dies ist die Leistung des schwachen Modells auf dem Testdatensatz, die als Baseline-Leistung angesehen werden kann.
- Weak-to-strong Performance($\mathrm{P}_{\mathrm{weak}\to\mathrm{strong}}$). Dies ist die Leistung des starken Modells auf dem Testdatensatz, das mit dem schwachen Modell feinabgestimmt wurde.
- Strong Ceiling Performance ($\mathrm{P}_{\mathrm{ceiling}}$). Dies ist die Leistung des starken Modells auf dem Testdatensatz, das mit Ground-Truth-Daten feinabgestimmt wurde. Zum Beispiel stimmen wir das starke Modell mit von Menschen annotierten Vorhersagen fein

ab und betrachten das resultierende Modell als eine Obergrenze.

Dann kann die wiederhergestellte Leistungslücke (PGR) definiert werden als

$$\text{PGR} \quad = \quad \max\left\{0, \frac{\text{P}_{\text{weak}\to\text{strong}} - \text{P}_{\text{weak}}}{\text{P}_{\text{ceiling}} - \text{P}_{\text{weak}}}\right\} \tag{4.11}$$

Diese Metrik misst, wie viel der Leistungslücke zwischen dem Obergrenzen-Modell und dem schwachen Modell durch das Schwach-zu-Stark-Modell wiederhergestellt werden kann. Ein PGR von 1 zeigt an, dass die Schwach-zu-Stark-Feinabstimmung die Leistungslücke vollständig schließen kann, während ein PGR von 0 keine Verbesserung anzeigt. In der Arbeit von Burns et al. [2023a] wird gezeigt, dass PGR bei 22 NLP-Klassifizierungsaufgaben etwa 0.8 betragen kann. Es ist zu beachten, dass, obwohl das Potenzial der Schwach-zu-Stark-Feinabstimmung vielversprechend ist, das Erreichen einer substanziellen Schwach-zu-Stark-Generalisierung ein anspruchsvolles Ziel bleibt, das weiterer Untersuchung bedarf [Aschenbrenner, 2024].

Die Feinabstimmung von LLMs mit schwacher Überwachung ist nur eine Möglichkeit, kleine Modelle zur Verbesserung großer Modelle zu verwenden. Obwohl sich dieser Abschnitt hauptsächlich auf die Feinabstimmung von LLMs konzentriert, erwähnen wir hier auch andere Methoden, um eine vollständigere Diskussion zu ermöglichen (siehe Abbildung 4.5 für Illustrationen dieser Methoden).

- Anstatt kleine Modelle zur Erzeugung synthetischer Daten zu verwenden, ist es auch einfach, einen auf diesen Modellen basierenden Wissensdestillationsverlust zu integrieren. Beispielsweise kann eine einfache Verlustfunktion, die den Unterschied zwischen den kleinen und großen Modellen misst, wie folgt definiert werden:

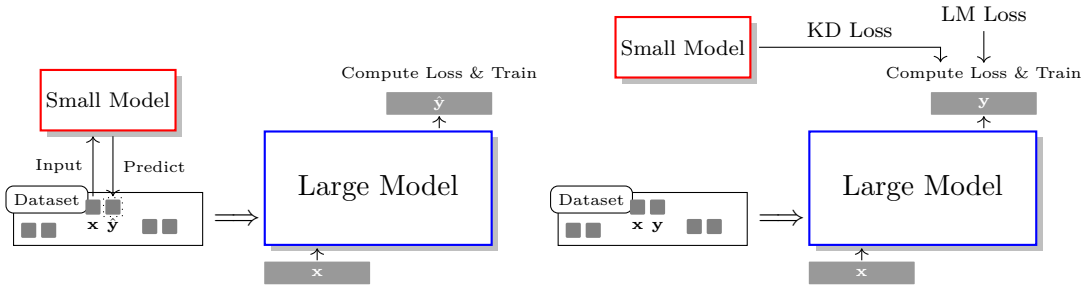
$$\text{Loss}_{\text{kd}} \quad = \quad \text{KL}(\Pr^{w}(\cdot|\mathbf{x}) \;||\; \Pr^{s}_{\theta}(\cdot|\mathbf{x})) \tag{4.12}$$

  Dann können wir diesen Verlust zum ursprünglichen Verlust der Sprachmodellierung hinzufügen und erhalten das folgende Trainingsziel

$$\tilde{\theta} \quad = \quad \arg\max_{\theta} \sum_{(\mathbf{x},\mathbf{y})\in\mathcal{D}} \log \Pr^{s}_{\theta}(\mathbf{y}|\mathbf{x}) - \lambda \cdot \text{Loss}_{\text{kd}} \tag{4.13}$$

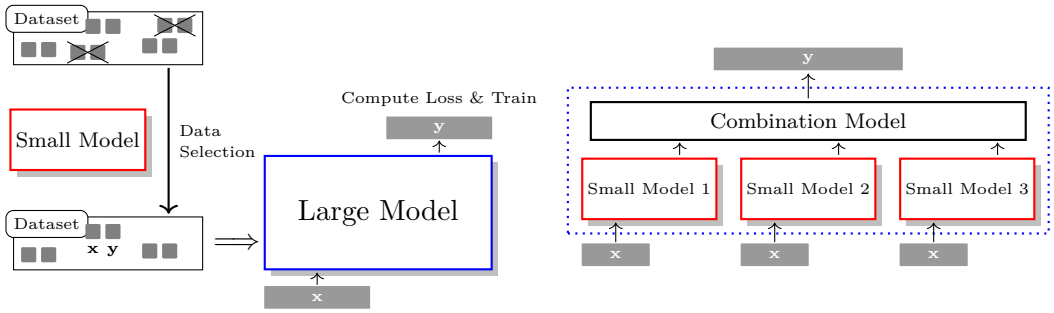
  wobei $\mathcal{D}$ die Menge der Eingabe- und Ausgabepaare ist und λ der Koeffizient der Interpolation ist. Diese Methode kann entweder in der Vortrainings- oder in der Feinabstimmungsphase angewendet werden. Wir können λ anpassen, um zu steuern, wie stark das kleine Modell das Training beeinflusst. Zum Beispiel können wir λ allmählich verringern, damit das Training stärker vom ursprünglichen Sprachmodellierungsverlust abhängt, während das große Modell leistungsfähiger wird.

- Ein weiterer Ansatz, kleine Modelle in das Vortraining und die Feinabstimmung von LLMs einzubeziehen, besteht darin, sie zur Datenauswahl oder -filterung zu verwenden. Für eine gegebene Sequenz können wir die Likelihood oder die Kreuzentropie mit einem kleinen Modell berechnen. Diese Größen können dann als Kriterien für die Auswahl oder Filterung von Daten verwendet werden. Zum Beispiel könnten



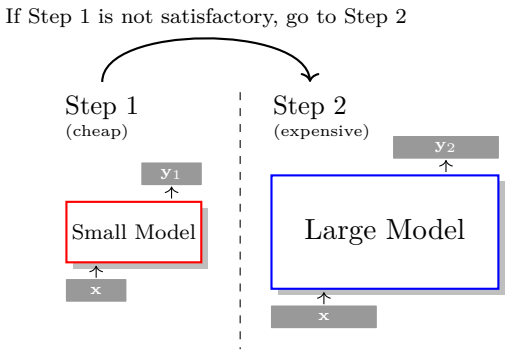
(a) Fine-tuning on data generated by a small model (weak-to-strong generalization)

(b) Fine-tuning with KD Loss from a small model (weak-to-strong generalization)



(c) Data selection with a small model

(d) Ensemble of multiple small models



(e) Cascading (at inference time)

Abbildung 4.5: Illustrationen zur Verwendung kleiner Modelle zur Verbesserung großer Modelle in LLMs. Ein Ansatz besteht darin, kleinere Modelle für das Fine-tuning oder das Vortraining größerer Modelle zu verwenden. Dies umfasst die Generierung synthetischer Daten (a), die Einbeziehung eines auxiliären Verlusts (b) und die Auswahl geeigneter Daten (c). Ein anderer Ansatz besteht in der Kombination von kleinen und großen Modellen. Dies umfasst das Lernen eines starken Modells durch die Aggregation mehrerer kleiner Modelle (d) und die Kaskadierung kleiner Modelle mit großen Modellen (e).

Sequenzen mit geringer Likelihood oder hoher Kreuzentropie aus dem Trainingsdatensatz ausgeschlossen werden, da sie weniger mit der gelernten Verteilung des kleinen Modells übereinstimmen. Umgekehrt können Sequenzen mit hoher Likelihood oder niedriger Kreuzentropie priorisiert werden, um sicherzustellen, dass sich das Training auf relevantere oder qualitativ hochwertigere Daten konzentriert.

- Ensemble-Lernen ist eine einfache und effektive Methode, um durch die Kombination

mehrerer schwacher Modelle ein starkes Modell zu erstellen. Die Anwendung dieser Technik auf LLMs ist unkompliziert. Wir können die von mehreren kleinen Modellen oder spezialisierten Submodellen vorhergesagten Verteilungen aggregieren und die endgültige Vorhersage aus den aggregierten Ergebnissen ableiten. Diese Aggregation kann mit Methoden wie Mehrheitsabstimmung, gewichteter Mittelung oder Stacking erfolgen.

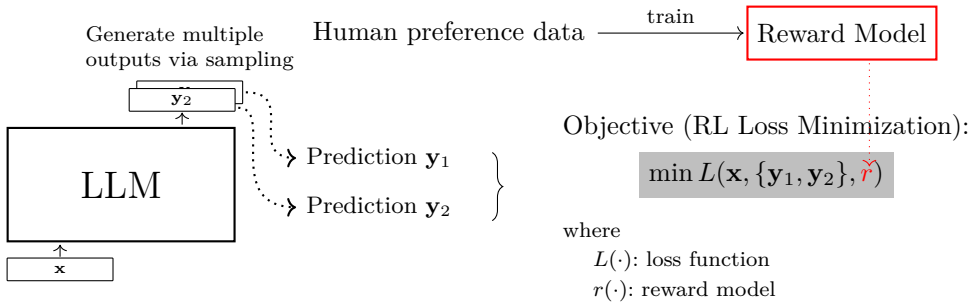
- Kleine Modelle können auch zur Inferenzzeit eingesetzt werden, um die Gesamteffizienz zu verbessern. Angenommen, wir haben ein großes Modell, das langsam, aber genauer ist, und ein kleines Modell, das schnell, aber weniger genau ist. Bei der Modellkaskadierung verarbeitet zunächst das kleine Modell die Eingabedaten und erzeugt schnell vorläufige Ergebnisse. Wenn diese Ergebnisse bestimmte vordefinierte Kriterien erfüllen, können sie direkt verwendet werden. Wenn die anfänglichen Ergebnisse jedoch nicht ausreichend gut sind, wird die Eingabe an das größere, genauere Modell weitergeleitet, um ein besseres Ergebnis zu erzielen. Dieser Ansatz reduziert die Rechenkosten und die Latenz erheblich, da das kleine Modell viele Eingaben effektiv ohne Zugriff auf das große Modell bewältigen kann.

4.3 Anpassung an menschliche Präferenzen: RLHF

Bisher haben wir uns in diesem Kapitel auf das Fine-Tuning von LLMs unter Verwendung von gepaarten Eingabe-Ausgabe-Daten konzentriert. Dieser Ansatz ermöglicht es uns, LLMs durch überwachtes Lernen an das Befolgen von Anweisungen anzupassen. In vielen Anwendungen müssen LLMs jedoch nicht nur Anweisungen befolgen, sondern auch in einer Weise handeln, die besser mit menschlichen Werten und Präferenzen übereinstimmt. Betrachten wir ein Szenario, in dem ein Benutzer ein LLM fragt, wie man in ein Computersystem eindringt. Wenn das LLM nicht entsprechend angepasst ist, könnte es antworten, indem es Details zur Durchführung dieser illegalen Aktivität bereitstellt. Stattdessen wäre eine wünschenswertere Antwort, dem Benutzer von illegalen Aktivitäten abzuraten und einen allgemeinen Überblick über die Konsequenzen solcher Handlungen zu geben. Die Schwierigkeit dabei ist, dass die ethischen Nuancen und kontextuellen Überlegungen, die für eine angemessene Antwort eines LLM in solchen Szenarien erforderlich sind, nicht immer einfach in einen Fine-Tuning-Datensatz zu kodieren sind. Was noch herausfordernder ist, ist, dass Menschen oft selbst ihre eigenen Präferenzen nicht präzise ausdrücken können.

In diesem Abschnitt besprechen wir eine alternative Methode zum Fine-Tuning von LLMs, die als Reinforcement Learning from Human Feedback oder kurz RLHF bezeichnet wird [Christiano et al., 2017; Stiennon et al., 2020]. Die Grundidee hinter RLHF ist, dass LLMs aus Vergleichen von Modellausgaben mithilfe von Belohnungsmodellen lernen können (siehe Abbildung 4.6). Dazu können wir menschliche Experten hinzuziehen, die ihre Präferenzen zwischen Paaren von Ausgaben, die vom LLM generiert wurden, angeben. Diese Präferenzdaten werden verwendet, um ein Belohnungsmodell zu trainieren, das die wahrgenommene Qualität von LLM-Ausgaben vorhersagen kann. Einmal trainiert, gibt das Belohnungsmodell Feedback, indem es den neuen Ausgaben, die das LLM als Reaktion

(a) Supervised fine-tuning (maximizing the prediction probability given the input)



(b) Reinforcement Learning from Human Feedback

Abbildung 4.6: Überwachtes Fine-Tuning vs. verstärkendes Lernen durch menschliches Feedback. Beim überwachten Fine-Tuning optimieren wir das LLM, indem wir die Wahrscheinlichkeit der Vorhersage bei gegebener Eingabe maximieren. Beim verstärkenden Lernen durch menschliches Feedback trainieren wir zunächst ein Belohnungsmodell auf menschlichen Präferenzdaten (für jedes Paar von Vorhersagen werden Evaluatoren gebeten, die von ihnen bevorzugte auszuwählen). Anschließend verwenden wir dieses Belohnungsmodell, um das LLM während des Fine-Tunings zu überwachen.

auf die Eingaben generiert, Bewertungen zuweist. Das LLM verwendet diese Bewertungen, um seine Parameter durch Algorithmen des Verstärkungslernens zu aktualisieren. Im weiteren Verlauf dieses Abschnitts werden wir zunächst die Grundlagen des Verstärkungslernens vorstellen, um die Diskussion zu erleichtern, und dann Methoden zum Trainieren von Belohnungsmodellen und zur Anpassung von LLMs an diese Modelle erörtern.

4.3.1 Grundlagen des Verstärkungslernens

Wir beginnen mit der Betrachtung einiger grundlegender Konzepte des Verstärkungslernens. Beachten Sie, dass die hier verwendete Notation geringfügig von der in den vorherigen Abschnitten und Kapiteln verwendeten abweicht, da wir unsere Beschreibung konsistenter mit der in der Literatur zum Verstärkungslernen gestalten möchten. Dennoch werden wir zeigen, wie diese Notation der Notation der Sprachmodellierung entspricht. Leser, die bereits mit Techniken des Verstärkungslernens vertraut sind, können diesen Unterabschnitt überspringen oder überfliegen.

Ein allgemeines Framework für das Verstärkungslernen beschreibt, wie ein Agent mit einer dynamischen Umgebung interagiert. Diese Interaktion wird als eine Sequenz von Aktionen modelliert, die der Agent als Reaktion auf den Zustand der Umgebung ausführt.

In jedem Zeitschritt beobachtet der Agent den aktuellen Zustand, wählt eine Aktion basierend auf seiner Policy, führt die Aktion aus und erhält dann eine Rückmeldung von der Umgebung in Form einer Belohnung und eines neuen Zustands. Diese Sequenz aus Beobachten, Handeln und Rückmeldung erhalten wird wiederholt, bis der Agent sein Ziel erreicht.

Ein System für Verstärkungslernen umfasst mehrere Komponenten:

- Agent. Dies ist der Lernende oder Entscheidungsträger im Reinforcement Learning. Im Kontext von LLMs kann er als das LLM selbst betrachtet werden.

- Umgebung. Dies umfasst alles, was extern zum Agenten ist und mit dem der Agent interagiert. In LLMs geht es bei der Umgebung jedoch weniger um einen physischen oder virtuellen Raum, sondern mehr um den Rahmen, innerhalb dessen der Agent (z. B. ein LLM) Feedback erhält und lernt.

- Zustand (s). Ein Zustand repräsentiert die aktuelle Situation der Umgebung. Gegeben eine Sequenz von Tokens für das Sprachmodellieren, kann ein Zustand zu einem Zeitpunkt als die bisher beobachteten Tokens betrachtet werden, also die Kontext-Tokens, die wir verwenden, um das nächste Token vorherzusagen. Zum Beispiel können wir $(\mathbf{x}, \mathbf{y}_{<t})$ als Zustand definieren, wenn wir das nächste Token zum Zeitpunkt t vorhersagen.

- Aktion (a). Aktionen repräsentieren mögliche Entscheidungen, die der Agent treffen kann. Wir können sie als mögliche vorhergesagte Tokens im Vokabular sehen.

- Belohnung (R). Die Belohnung ist das Feedback aus der Umgebung, das den Erfolg einer Aktion bewertet. Zum Beispiel bezeichnet $r(s, a, s')$ die Belohnung, die der Agent erhält, wenn er im Zustand s die Aktion a ausführt und in den nächsten Zustand s' übergeht. Wenn die Zustand-Aktions-Sequenz gegeben ist, können wir die Belohnung zum Zeitpunkt t als $r_t = r(s_t, a_t, s_{t+1})$ bezeichnen. Beachte auch, dass, wenn der Entscheidungsprozess deterministisch ist, wir s_{t+1} weglassen können, da es durch s_t und a_t bestimmt werden kann. In solchen Fällen können wir $r(s_t, a_t)$ als Kurzform für $r(s_t, a_t, s_{t+1})$ verwenden.

- Strategie (π). Für ein LLM wird eine Strategie als Wahrscheinlichkeitsverteilung über die Tokens definiert, die das LLM vorhersagt, gegeben die vorhergehenden Kontext-Tokens. Formal kann dies ausgedrückt werden als

$$\pi(a|s) = \Pr(y_t|\mathbf{x}, \mathbf{y}_{<t}) \tag{4.14}$$

  wobei a dem Token y_t entspricht und s dem Kontext $(\mathbf{x}, \mathbf{y}_{<t})$. Abbildung 4.7 veranschaulicht, wie ein LLM im Rahmen des Reinforcement Learning als Strategie betrachtet werden kann.

- Wertfunktion (V und Q). Eine Zustandswertfunktion (oder kurz Wertfunktion) bewertet die erwartete diskontierte Rendite (d. h. akkumulierte Belohnungen) für einen Agenten, der von einem bestimmten Zustand s aus startet und einer spezifischen

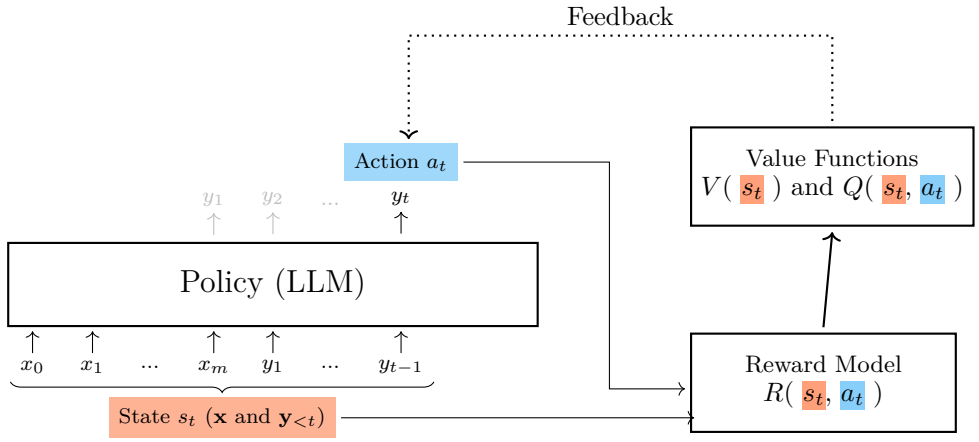


Abbildung 4.7: LLM als Policy im Reinforcement Learning. In jedem Schritt t prognostiziert das LLM ein Token y_t basierend auf der Modelleingabe $\mathbf{x}$ und den zuvor generierten Token $\mathbf{y}_{<t}$. Dieser Prozess kann als ein Problem des Reinforcement Learnings formuliert werden, wobei y_t als die Aktion, $(\mathbf{x}, \mathbf{y}_{<t})$ als der Zustand und die vorhergesagte Verteilung $\Pr(y_t|\mathbf{x}, \mathbf{y}_{<t})$ als die Policy dient. Sobald y_t vorhergesagt ist, gibt das LLM sowohl $(\mathbf{x}, \mathbf{y}_{<t})$ als auch y_t in das Belohnungsmodell ein, welches bewertet, wie effektiv das gewählte Token zum Erreichen des gewünschten Textergebnisses beiträgt. Diese Bewertung erzeugt Belohnungswerte, die verwendet werden, um die Wertefunktionen $V(s_t)$ und $Q(s_t, a_t)$ zu berechnen. Diese Funktionen geben dann eine Rückmeldung an das LLM und leiten das Policy-Training an.

Strategie π folgt. Sie wird definiert als:

$$\begin{aligned} V(s) &= \mathbb{E}\Big[r(s_0, a_0, s_1) + \gamma r(s_1, a_1, s_2) + \gamma^2 r(s_2, a_2, s_3) + \cdots \mid s_0 = s, \pi\Big] \\ &= \mathbb{E}\Big[r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots \mid s_0 = s, \pi\Big] \\ &= \mathbb{E}\Big[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, \pi\Big] \end{aligned} \tag{4.15}$$

wobei $\gamma \in [0, 1]$ der Diskontierungsfaktor ist, der die Bedeutung zukünftiger Belohnungen anpasst, $s_0 = s$ angibt, dass der Agent im Zustand s startet, und die Erwartung $\mathbb{E}$ über alle möglichen Trajektorien (d. h. Zustand-Aktions-Sequenzen) durchgeführt wird. Ebenso misst eine Aktionswertfunktion (oder Q-Wert-Funktion) die erwartete Rendite, ausgehend von einem Zustand s, unter Ausführung einer Aktion a und anschließendem Folgen einer Strategie π, gegeben durch

$$Q(s, a) = \mathbb{E}\Big[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a, \pi\Big] \tag{4.16}$$

wobei $a_0 = a$ angibt, dass die Aktion, die im Anfangszustand ausgeführt wird, a ist.

Das Ziel des Verstärkungslernens ist es, eine Policy zu lernen, die die kumulative Belohnung (oder Return) maximiert, die der Agent auf lange Sicht erhält. Gegeben eine Zustands-Aktions-Sequenz $\tau = \{(s_1, a_1), ..., (s_T, a_T)\}$[1], kann die kumulative Belohnung

[1] Wir nehmen an, dass die Zustands-Aktions-Sequenz mit s_1 und a_1 beginnt, anstatt mit s_0 und a_0, um mit der in diesem Kapitel üblichen Notation übereinzustimmen, bei der die Vorhersage $\mathbf{y}$ typischerweise mit y_1 beginnt. Natürlich ist es in der Literatur auch üblich, eine Zustands-Aktions-Sequenz als

über diese Sequenz wie folgt geschrieben werden

$$R(\tau) = \sum_{t=1}^{T} r_t \tag{4.17}$$

Der Erwartungswert dieser kumulativen Belohnung über einen Raum von Zustands-Aktions-Sequenzen ist in der Form gegeben

$$\begin{aligned} J(\theta) &= \mathop{\mathbb{E}}_{\tau \sim \mathcal{D}} \left[R(\tau) \, | \pi_\theta \right] \\ &= \sum_{\tau \in \mathcal{D}} \Pr_\theta(\tau) R(\tau) \\ &= \sum_{\tau \in \mathcal{D}} \Pr_\theta(\tau) \sum_{t=1}^{T} r_t \end{aligned} \tag{4.18}$$

wobei $\tau \sim \mathcal{D}$ anzeigt, dass τ aus dem Raum der Zustands-Aktions-Sequenzen $\mathcal{D}$ gezogen wird, und der Index θ die Parameter der Policy anzeigt. $J(\theta)$ wird auch als Leistungsfunktion (performance function) bezeichnet.

Das Trainingsziel ist dann die Maximierung von $J(\theta)$:

$$\tilde{\theta} = \arg\max_{\theta} J(\theta) \tag{4.19}$$

Jetzt haben wir einen einfachen Ansatz für das Verstärkungslernen: 1) Wir ziehen eine Anzahl von Zustands-Aktions-Sequenzen; dann 2) bewerten wir jede Sequenz mithilfe der Leistungsfunktion; dann 3) aktualisieren wir das Modell, um diese Leistungsfunktion zu maximieren. Wenn wir Gl. (4.18) nehmen und den Gradientenabstieg verwenden, um die Policy zu optimieren, würde dieser Ansatz eine Form von Policy-Gradient-Methoden darstellen [Williams, 1992].

Beachten Sie, dass in vielen NLP-Problemen, wie der maschinellen Übersetzung, Belohnungen typischerweise spärlich sind. Zum Beispiel wird eine Belohnung nur am Ende eines vollständigen Satzes erhalten. Das bedeutet, dass $r_t = 0$ für alle $t < T$ ist, und r_t nur dann ungleich null ist, wenn $t = T$. Idealerweise würde man eine sofortige und häufige (dichte) Rückmeldung bevorzugen, wodurch das Training der Policy einfacher und effizienter gestaltet werden kann. Obwohl verschiedene Methoden zur Behandlung von spärlichen Belohnungen vorgeschlagen wurden, wie z. B. Reward Shaping, werden wir in unserer Diskussion weiterhin von einer spärlichen Belohnungsumgebung ausgehen, bei der die Belohnung erst nach Abschluss der Vorhersage verfügbar ist.

Das in den Gln. (4.17-4.19) beschriebene Modell stellt eine Grundform des Verstärkungslernens dar, und es wurden viele Varianten und Verbesserungen dieses Modells entwickelt. Bevor wir diese anspruchsvolleren Modelle vorstellen, nehmen wir uns einen Moment Zeit, um die Zielfunktion $J(\theta)$ aus der Perspektive des Policy-Gradienten zu interpretieren. Beim Gradientenabstieg müssen wir den Gradienten von $J(\theta)$ bezüglich θ

$\{(s_0, a_0), ..., (s_T, a_T)\}$ oder $\{(s_0, a_0), ..., (s_{T-1}, a_{T-1})\}$ zu bezeichnen. Diese Variation in der Notation beeinflusst jedoch nicht die Diskussion der hier vorgestellten Modelle.

berechnen:

$$\begin{aligned}\frac{\partial J(\theta)}{\partial \theta} &= \frac{\partial \sum_{\tau \in \mathcal{D}} \Pr_\theta(\tau) R(\tau)}{\partial \theta} \\ &= \sum_{\tau \in \mathcal{D}} \frac{\partial \Pr_\theta(\tau)}{\partial \theta} R(\tau) \\ &= \sum_{\tau \in \mathcal{D}} \Pr_\theta(\tau) \frac{\partial \Pr_\theta(\tau)/\partial \theta}{\Pr_\theta(\tau)} R(\tau) \\ &= \sum_{\tau \in \mathcal{D}} \Pr_\theta(\tau) \frac{\partial \log \Pr_\theta(\tau)}{\partial \theta} R(\tau) \end{aligned} \tag{4.20}$$

In einigen Fällen nehmen wir an, dass jede Sequenz in $\mathcal{D}$ gleich wahrscheinlich ist (d. h. $\Pr_\theta(\tau) = 1/|\mathcal{D}|$). In diesem Fall können wir Gl. (4.20) vereinfachen und müssen nur die Terme $\frac{\partial \log \Pr_\theta(\tau)}{\partial \theta}$ und $R(\tau)$ betrachten:

$$\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{m} \sum_{\tau \in \mathcal{D}} \frac{\partial \log \Pr_\theta(\tau)}{\partial \theta} R(\tau) \tag{4.21}$$

Ein Vorteil dieses Ergebnisses ist, dass $R(\tau)$ nicht differenzierbar sein muss, was bedeutet, dass wir jede Art von Belohnungsfunktion im Verstärkungslernen verwenden können.

Indem wir die Erzeugung der Sequenz τ als einen Markov-Entscheidungsprozess behandeln, können wir $\frac{\partial \log \Pr_\theta(\tau)}{\partial \theta}$ weiter ableiten und erhalten:

$$\begin{aligned}\frac{\partial \log \Pr_\theta(\tau)}{\partial \theta} &= \frac{\partial}{\partial \theta} \log \prod_{t=1}^{T} \pi_\theta(a_t|s_t) \Pr(s_{t+1}|s_t, a_t) \\ &= \frac{\partial}{\partial \theta} \sum_{t=1}^{T} \underbrace{\log \pi_\theta(a_t|s_t)}_{\text{Politik}} + \frac{\partial}{\partial \theta} \sum_{t=1}^{T} \underbrace{\log \Pr(s_{t+1}|s_t, a_t)}_{\text{Dynamik}} \end{aligned} \tag{4.22}$$

wobei der Gradient in zwei Teile zerlegt wird: den Policy-Gradienten und den Dynamik-Gradienten. Die Policy-Komponente, $\log \pi_\theta(a_t|s_t)$, bestimmt die Log-Wahrscheinlichkeit, die Aktion a_t im Zustand s_t auszuführen, und sie wird durch θ parametrisiert. Die Dynamik-Komponente, $\log \Pr(s_{t+1}|s_t, a_t)$, stellt die Log-Wahrscheinlichkeit dar, vom Zustand s_t nach Ausführung der Aktion a_t in den Zustand s_{t+1} überzugehen. In typischen Konfigurationen des Verstärkungslernens wird die Dynamik nicht direkt von den Policy-Parametern θ beeinflusst, und daher sind ihre Ableitungen oft null. In diesem Fall kann daher Gl. (4.22) vereinfacht werden zu:

$$\frac{\partial \log \Pr_\theta(\tau)}{\partial \theta} = \frac{\partial}{\partial \theta} \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) \tag{4.23}$$

Mit anderen Worten, wir konzentrieren uns nur auf die Optimierung der Policy, ohne uns um die zugrunde liegende Dynamik zu kümmern.

Durch Einsetzen von Gl. (4.23) in Gl. (4.21) und Erweitern von $R(\tau)$ erhalten wir

dann

$$\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \frac{\partial}{\partial \theta} \Big(\sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) \sum_{t=1}^{T} r_t \Big) \tag{4.24}$$

Obwohl dieser Policy-Gradient-Ansatz unkompliziert ist, leidet er unter dem Problem, dass die Varianz der geschätzten Gradienten sehr hoch sein kann, was den Lernprozess verrauscht und ineffizient macht. Ein Grund für dieses Problem der hohen Varianz ist, dass die Belohnungen über verschiedene Schritte oder Szenarien hinweg stark variieren können. Stellen Sie sich vor, dass in einer Sequenz von Aktionsentscheidungen das Belohnungsmodell dazu neigt, kleinen Belohnungen für gute Aktionen (z. B. $R_t = 2$) und große Strafen für schlechte Aktionen (z. B. $R_t = -50$) zuzuweisen. Solch variierende Belohnungsskalen für gute und schlechte Aktionen können zu einer sehr niedrigen Gesamtbelohnung für die gesamte Sequenz führen, selbst wenn sie gute Aktionen enthält.

Eine einfache Methode zur Reduzierung der Varianz des Gradienten besteht darin, eine Basislinie b festzulegen und sie von $\sum_{t=1}^{T} r_t$ zu subtrahieren, was zu $\sum_{t=1}^{T} r_t - b$ führt.[2] Hier kann die Basislinie als Referenzpunkt interpretiert werden. Indem wir die Belohnungen um diese Basislinie zentrieren, entfernen wir systematische Verzerrungen im Belohnungssignal, was die Aktualisierungen stabiler und weniger empfindlich gegenüber extremen Schwankungen einzelner Belohnungen macht.

Dieses Policy-Gradient-Modell mit einer Basislinie kann wie folgt angegeben werden

$$\begin{aligned}
\frac{\partial J(\theta)}{\partial \theta} &= \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \frac{\partial}{\partial \theta} \Big(\sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) \Big) \Big(\sum_{t=1}^{T} r_t - b \Big) \\
&= \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \frac{\partial}{\partial \theta} \Big[\sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) \Big(\sum_{k=1}^{T} r_k - b \Big) \Big] \\
&= \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \frac{\partial}{\partial \theta} \Big[\sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) \Big(\sum_{k=1}^{t-1} r_k + \sum_{k=t}^{T} r_k - b \Big) \Big]
\end{aligned} \tag{4.25}$$

Hier schreiben wir $\sum_{k=1}^{T} r_k$ als die Summe von zwei Termen $\sum_{k=1}^{t-1} r_k$ und $\sum_{k=t}^{T} r_k$, um zwischen den Belohnungen zu unterscheiden, die vor und nach der Aktion zum Zeitschritt t anfallen. Beachten Sie, dass in Markov-Entscheidungsprozessen die Zukunft bei gegebenem Gegenwartszustand von der Vergangenheit unabhängig ist. Daher kann die zum Zeitschritt t ausgeführte Aktion die vor t erhaltenen Belohnungen nicht beeinflussen, oder mit anderen Worten, die Belohnungen vor t sind bereits „festgelegt“, wenn die Aktion bei t gewählt wird. Der Term $\sum_{k=1}^{t-1} r_k$ trägt nicht zum Gradienten bei und kann weggelassen werden,

[2] Tatsächlich ändert die Verwendung einer Basislinie b nichts an der Varianz der Gesamtbelohnungen $\sum_{t=1}^{T} r_t$. Es ist jedoch wichtig zu beachten, dass die Einführung einer Basislinie zwar nicht die Gesamtvarianz der Belohnungen verändert, aber dazu beiträgt, die Varianz der Gradientenschätzungen zu reduzieren. Dies liegt daran, dass das Subtrahieren der Basislinie von den Gesamtbelohnungen die Schwankungen um ihren Mittelwert effektiv reduziert, was die Gradientenschätzungen stabiler macht. Im Allgemeinen zentriert die Operation $\sum_{t=1}^{T} r_t - b$ die Belohnungen um null (z. B. wird b als Erwartungswert von $\sum_{t=1}^{T} r_t$ definiert), was zu einer reduzierten Varianz im Produkt $\sum_{t=1}^{T} \log \pi_\theta(a_t|s_t)(\sum_{t=1}^{T} r_t - b)$ führen kann.

was zu einer vereinfachten Version von Gl. (4.25) führt

$$\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \frac{\partial}{\partial \theta} \Big[\sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) \Big(\sum_{k=t}^{T} r_k - b \Big) \Big] \tag{4.26}$$

Beachten Sie auch, dass das Entfernen von $\sum_{k=t}^{T} r_k$ die Varianz des Gradienten weiter reduzieren kann.

Es gibt viele Möglichkeiten, die Basislinie b zu definieren. Hier betrachten wir die Wertefunktion des Zustands s_t, d.h. den geschätzten Wert, sich im Zustand s_t zu befinden: $V(s_t) = \mathbb{E}(r_t + r_{t+1} + \cdots + r_T)$. Daher haben wir

$$\begin{aligned} A(s_t, a_t) &= \sum_{k=t}^{T} r_k - b \\ &= \sum_{k=t}^{T} r_k - V(s_t) \end{aligned} \tag{4.27}$$

wobei $\sum_{k=t}^{T} r_k$ den tatsächlich erhaltenen Return darstellt und $V(s_t)$ den erwarteten Return repräsentiert. $A(s_t, a_t)$ (oder kurz A_t) wird als Vorteil (Advantage) zum Zeitschritt t bezeichnet, der den relativen Nutzen der Aktion a_t im Vergleich zum erwarteten Wert der Befolgung der Policy vom Zustand s_t an quantifiziert.

Durch Verwendung der Vorteilsfunktion $A(s_t, a_t)$ kann der Gradient von $J(\theta)$ in der Form geschrieben werden

$$\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \frac{\partial}{\partial \theta} \Big(\sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) A(s_t, a_t) \Big) \tag{4.28}$$

Dieses Optimierungsziel entspricht der Advantage Actor-Critic (A2C) Methode im Verstärkungslernen [Mnih et al., 2016]. Bei dieser Methode zielt der Actor darauf ab, eine Policy zu lernen. Er aktualisiert die Policy-Parameter mithilfe von Gl. (4.28), um sich stärker auf Aktionen zu konzentrieren, die wahrscheinlich die Leistung verbessern. Der Critic hingegen aktualisiert seine Schätzung der Wertefunktion, die zur Berechnung der Vorteilsfunktion $A(s_t, a_t)$ verwendet wird und somit als Bewerter der vom Actor gelernten Policy dient.

Bei der A2C-Methode wird $A(s_t, a_t)$ typischerweise als die Differenz der Aktions-Werte-Funktion $Q(s_t, a_t)$ und der Zustands-Werte-Funktion $V(s_t)$ ausgedrückt

$$A(s_t, a_t) = Q(s_t, a_t) - V(s_t) \tag{4.29}$$

Auf den ersten Blick mag dieses Modell schwierig zu entwickeln erscheinen, da es zwei separate Untermodelle zur Berechnung von $Q(s_t, a_t)$ bzw. $V(s_t)$ erfordert. Glücklicherweise können wir, da $Q(s_t, a_t)$ als der Return $r_t + V(s_{t+1})$ definiert werden kann, Gl. (4.29) umschreiben als

$$A(s_t, a_t) = r_t + V(s_{t+1}) - V(s_t) \tag{4.30}$$

oder alternativ den Abzinsungsfaktor γ einführen, um eine allgemeinere Form zu erhalten

$$A(s_t, a_t) \quad = \quad r_t + \gamma V(s_{t+1}) - V(s_t) \tag{4.31}$$

$A(s_t, a_t) = r_t + \gamma V(s_{t+1}) - V(s_t)$ wird auch als temporale Differenz (TD) Fehler bezeichnet. Was wir benötigen, ist, ein Critic-Netzwerk für die Wertefunktion $V(s_t)$ zu trainieren und es dann zur Berechnung der Vorteilsfunktion zu verwenden[3].

Bis zu diesem Punkt haben wir beträchtlichen Raum der Diskussion der Grundlagen des Verstärkungslernens gewidmet, insbesondere der Herleitung des Optimierungsziels für die A2C-Methode. Jedoch ist Verstärkungslernen ein weites Feld, und viele technische Details können hier nicht behandelt werden. Der interessierte Leser kann für weitere Details auf Bücher zum Thema Verstärkungslernen zurückgreifen [Sutton and Barto, 2018; Szepesvári, 2010]. Dennoch verfügen wir nun über das notwendige Wissen, um RLHF weiter zu diskutieren. In den folgenden Unterabschnitten werden wir zur Diskussion über die Anpassung von LLMs zurückkehren und zeigen, wie die A2C-Methode zur Anpassung an menschliche Präferenzen verwendet wird.

4.3.2 Training von Belohnungsmodellen

Wir haben gezeigt, dass Belohnungsmodelle eine sehr wichtige Rolle im allgemeinen Rahmen des bestärkenden Lernens spielen und die Grundlage für die Berechnung von Wertfunktionen bilden. Wir betrachten nun das Problem des Trainings dieser Belohnungsmodelle.

Beim RLHF ist ein Belohnungsmodell ein neuronales Netzwerk, das ein Paar aus Eingabe- und Ausgabe-Tokensequenzen auf einen Skalar abbildet. Bei einer Eingabe $\mathbf{x}$ und einer Ausgabe $\mathbf{y}$ kann die Belohnung ausgedrückt werden als

$$r \quad = \quad \text{Reward}(\mathbf{x}, \mathbf{y}) \tag{4.33}$$

wobei Reward$(\cdot)$ das Belohnungsmodell ist. r kann als Maß dafür interpretiert werden, wie gut die Ausgabe $\mathbf{y}$ bei gegebener Eingabe $\mathbf{x}$ mit dem gewünschten Verhalten übereinstimmt. Wie im vorherigen Unterabschnitt besprochen, wird angenommen, dass sowohl $\mathbf{x}$ als auch $\mathbf{y}$ vollständige Texte sind. Dies bedeutet, dass das Belohnungsmodell die Beziehung zwischen Eingaben und Ausgaben bewertet, die einen vollständigen semantischen Inhalt bieten. Wenn beispielsweise das Belohnungsmodell angewendet wird, weist es an jeder Position t in der Ausgabesequenz $\mathbf{y} = y_1 ... y_n$ einen Wert von 0 (oder einen anderen vorbestimmten Wert) zu. Nur an der letzten Position, wenn $t = n$ ist, erzeugt das Belohnungsmodell den tatsächlichen Belohnungswert. Um die Notation übersichtlich zu halten,

[3] Der Trainingsverlust für das Werte-Netzwerk (oder Critic-Netzwerk) in A2C wird im Allgemeinen als der mittlere quadratische Fehler zwischen dem berechneten Return $r_t + \gamma V(s_{t+1})$ und dem vorhergesagten Zustandswert $V(s_t)$ formuliert. Angenommen, das Werte-Netzwerk wird durch ω parametrisiert. Die Verlustfunktion ist gegeben durch

$$\mathcal{L}_v(\omega) \quad = \quad \frac{1}{M} \sum \left(r_t + \gamma V_\omega(s_{t+1}) - V_\omega(s_t) \right)^2 \tag{4.32}$$

wobei M die Anzahl der Trainingsbeispiele ist, zum Beispiel können wir für eine Sequenz von T Token $M = T$ setzen.

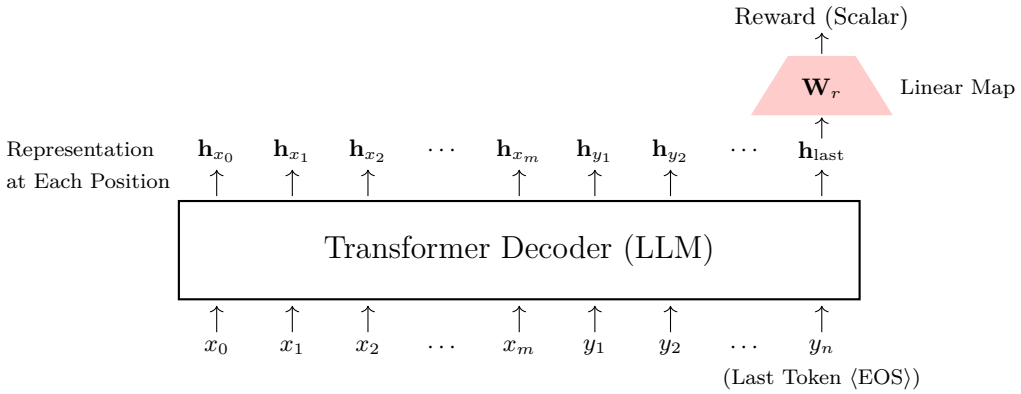


Abbildung 4.8: Architektur des Belohnungsmodells basierend auf dem Transformer. Die Hauptkomponente dieses Modells ist weiterhin ein LLM. Wir verwenden den Transformer-Dekodierer als Modell zur Sequenzrepräsentation. Wir extrahieren die Repräsentation der letzten Position des Dekodierers als Repräsentation der gesamten Sequenz $[\mathbf{x}, \mathbf{y}]$. Anschließend bilden wir diese Repräsentation durch eine lineare Transformation auf einen Skalar ab, der als Belohnungswert für $[\mathbf{x}, \mathbf{y}]$ dient.

verwenden wir von nun an $r(\mathbf{x}, \mathbf{y})$, um das Belohnungsmodell $\text{Reward}(\mathbf{x}, \mathbf{y})$ zu bezeichnen.

Es gibt viele Möglichkeiten, das Belohnungsmodell zu implementieren. Ein einfacher Ansatz besteht darin, das Belohnungsmodell auf der Grundlage eines vortrainierten LLM zu erstellen. Genauer gesagt können wir $\mathbf{x}$ und $\mathbf{y}$ verketten, um eine einzelne Tokensequenz $\text{seq}_{\mathbf{x},\mathbf{y}} = [\mathbf{x}, \mathbf{y}]$ zu bilden. Wir führen wie üblich ein vortrainiertes LLM auf dieser Sequenz aus und erhalten an jeder Position eine Repräsentation aus der obersten Transformer-Schicht. Anschließend nehmen wir die Repräsentation an der letzten Position (bezeichnet mit $\mathbf{h}_{\text{last}}$) und bilden sie über eine lineare Transformation auf einen Skalar ab:

$$r(\mathbf{x}, \mathbf{y}) \quad = \quad \mathbf{h}_{\text{last}} \mathbf{W}_r \tag{4.34}$$

wobei $\mathbf{h}_{\text{last}}$ ein d-dimensionaler Vektor und $\mathbf{W}r$ eine $d \times 1$ lineare Abbildungsmatrix ist. Diese Architektur des Belohnungsmodells ist in Abbildung 4.8 dargestellt.

Um das Belohnungsmodell zu trainieren, besteht der erste Schritt darin, menschliches Feedback zu einer Reihe von generierten Ausgaben zu sammeln. Bei einer gegebenen Eingabe $\mathbf{x}$ verwenden wir das LLM, um mehrere Kandidatenausgaben $\{\mathbf{y}_1, ..., \mathbf{y}_N\}$ zu erzeugen. Menschliches Feedback kann auf verschiedene Weisen eingeholt werden:

- Paarweiser Vergleich (Paarweises Ranking). Bei zwei verschiedenen Ausgaben wählen menschliche Experten aus, welche besser ist.

- Bewertung. Menschliche Experten geben für jede Ausgabe eine Punktzahl oder Bewertung ab. Diese Punktzahl ist oft ein kontinuierlicher oder diskreter numerischer Wert, wie z. B. eine Punktzahl auf einer Skala (z. B. 1-5 Sterne oder 1-10 Punkte). In einigen Fällen kann die Bewertung binär sein und eine „Ja/Nein“- oder „positiv/negativ“-Präferenz anzeigen.

- Listenweises Ranking. Menschliche Experten werden gebeten, die gegebene Menge

möglicher Ausgaben in eine Rangfolge zu bringen.

Hier betrachten wir paarweises Vergleichsfeedback, da es eine der einfachsten und gebräuchlichsten Formen des menschlichen Feedbacks ist, das beim RLHF verwendet wird. In diesem Setting werden jedes Mal zwei Ausgaben $(\mathbf{y}_a, \mathbf{y}_b)$ zufällig aus dem Kandidatenpool $\{\mathbf{y}_1, ..., \mathbf{y}_N\}$ gezogen. Menschlichen Experten werden dann diese Paare vorgelegt und sie werden gebeten, zu entscheiden, welche Ausgabe sie aufgrund spezifischer Kriterien wie Klarheit, Relevanz und Genauigkeit bevorzugen. Das menschliche Feedback kann als binäres Label kodiert werden, $\mathbf{y}_a \succ \mathbf{y}_b$ für eine Präferenz für $\mathbf{y}_a$ und $\mathbf{y}_b \succ \mathbf{y}_a$ für eine Präferenz für $\mathbf{y}_b$.

Ein einfaches und weit verbreitetes Modell zur Beschreibung solcher paarweisen Vergleiche ist das Bradley-Terry-Modell [Bradley and Terry, 1952]. Es ist ein probabilistisches Modell, das die Wahrscheinlichkeit schätzt, dass ein Element einem anderen vorgezogen wird. Wenn wir dieses Modell an die hier verwendete Notation anpassen, können wir die Wahrscheinlichkeit, dass $\mathbf{y}_a$ gegenüber $\mathbf{y}_b$ bevorzugt wird, in der Form schreiben

$$\begin{aligned} \Pr(\mathbf{y}_a \succ \mathbf{y}_b|\mathbf{x}) &= \frac{e^{r(\mathbf{x},\mathbf{y}_a)}}{e^{r(\mathbf{x},\mathbf{y}_a)} + e^{r(\mathbf{x},\mathbf{y}_b)}} \\ &= \frac{e^{r(\mathbf{x},\mathbf{y}_a)-r(\mathbf{x},\mathbf{y}_b)}}{e^{r(\mathbf{x},\mathbf{y}_a)-r(\mathbf{x},\mathbf{y}_b)} + 1} \\ &= \text{Sigmoid}(r(\mathbf{x}, \mathbf{y}_a) - r(\mathbf{x}, \mathbf{y}_b)) \end{aligned} \tag{4.35}$$

Beim Training des Belohnungsmodells möchten wir diese Präferenzwahrscheinlichkeit maximieren. Eine auf dem Bradley-Terry-Modell basierende Verlustfunktion ist gegeben durch

$$\mathcal{L}_r(\phi) = -\mathbb{E}_{(\mathbf{x},\mathbf{y}_a,\mathbf{y}_b)\sim\mathcal{D}_r}\left[\log \Pr_\phi(\mathbf{y}_a \succ \mathbf{y}_b|\mathbf{x})\right] \tag{4.36}$$

wobei $(\mathbf{x}, \mathbf{y}_a, \mathbf{y}_b)$ aus einem von Menschen annotierten Datensatz $\mathcal{D}_r$ gezogen wird, der aus Präferenzpaaren von Ausgaben und ihren entsprechenden Eingaben besteht. ϕ repräsentiert die Parameter des Belohnungsmodells, welche sowohl die Parameter des Transformer-Decoders als auch die lineare Abbildungsmatrix $\mathbf{W}_r$ umfassen. In der Praxis können wir, unter der Annahme, dass $(\mathbf{x}, \mathbf{y}_a, \mathbf{y}_b)$ gleichmäßig aus $\mathcal{D}_r$ abgetastet wird, die Erwartung durch eine Summation ersetzen

$$\mathcal{L}_r(\phi) = -\frac{1}{|\mathcal{D}_r|} \sum_{(\mathbf{x},\mathbf{y}_a,\mathbf{y}_b)\in\mathcal{D}_r} \log \Pr_\phi(\mathbf{y}_a \succ \mathbf{y}_b|\mathbf{x}) \tag{4.37}$$

Das Ziel des Trainings des Belohnungsmodells ist es, die optimalen Parameter $\hat{\phi}$ zu finden, die diese Verlustfunktion minimieren, gegeben durch

$$\hat{\phi} = \arg\min_\phi \mathcal{L}_r(\phi) \tag{4.38}$$

Da das Belohnungsmodell selbst auch ein LLM ist, können wir das Transformer-Trainingsverfahren direkt wiederverwenden, um das Belohnungsmodell zu optimieren. Der Unterschied zum Training eines Standard-LLM besteht darin, dass wir nur den Kreuzentropie-Verlust durch

den paarweisen Vergleichsverlust ersetzen müssen, wie in Gl. (4.37) beschrieben. Nach dem Training des Belohnungsmodells können wir das trainierte Belohnungsmodell $r_{\hat{\phi}}(\cdot)$ anwenden, um das Ziel-LLM zur Ausrichtung zu überwachen.

Es ist erwähnenswert, dass wir, obwohl wir das Belohnungsmodell trainieren, um ein paarweises Ranking durchzuführen, es während des Ausrichtungsprozesses anwenden, um jedes Eingabe-Ausgabe-Paar unabhängig zu bewerten. Das Ziel des paarweisen Rankings stellt sicher, dass das Belohnungsmodell auf subtile Unterschiede zwischen den Ausgaben empfindlich ist, aber wir verlassen uns auf die kontinuierlichen Bewertungen, die vom Belohnungsmodell erzeugt werden, um die Optimierung des LLM zu leiten. Ein Vorteil dieses Ansatzes ist, dass wir aus verschiedenen Ranking-Verlustfunktionen wählen oder diese kombinieren können und die resultierenden Belohnungsmodelle dennoch auf die gleiche Weise anwenden können, wie wir es in diesem Unterabschnitt getan haben. Diese Konsistenz gewährleistet einen einheitlichen Rahmen für die Ausrichtung des LLM, unabhängig von der spezifischen Ranking-Verlustfunktion, die während des Trainings des Belohnungsmodells verwendet wird.

4.3.3 Training von LLMs

Nachdem das Belohnungsmodell erstellt wurde, trainieren wir die Policy (d. h. das LLM) mit der A2C-Methode. Erinnern wir uns an Abschnitt 4.3.1, dass eine Zustands-Aktions-Sequenz oder Trajektorie τ durch die Nutzenfunktion bewertet werden kann

$$U(\tau;\theta) \quad = \quad \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) A(s_t, a_t) \tag{4.39}$$

wobei $A(s_t, a_t)$ der Vorteil der Ausführung der Aktion a_t im Zustand s_t ist. Eine Schätzung von $A(s_t, a_t)$ ist als der TD-Fehler $r_t + \gamma V(s_{t+1}) - V(s_t)$ definiert, wobei die Wertefunktion $V(s_t)$ mit dem Belohnungsmodell trainiert wird.

Gegeben diese Nutzenfunktion, kann die A2C-basierte Verlustfunktion in der Form geschrieben werden

$$\begin{aligned} \mathcal{L}(\theta) &= -\mathbb{E}_{\tau\sim\mathcal{D}}\big[U(\tau;\theta)\big] \\ &= -\mathbb{E}_{\tau\sim\mathcal{D}}\big[\sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) A(s_t, a_t)\big] \end{aligned} \tag{4.40}$$

wobei $\mathcal{D}$ ein Raum von Zustands-Aktions-Sequenzen ist. Wie üblich ist das Ziel des Trainings der Policy, diese Verlustfunktion zu minimieren

$$\tilde{\theta} \quad = \quad \arg\min_\theta \mathcal{L}(\theta) \tag{4.41}$$

Wenn wir das Problem wieder auf das Sprachmodellierungsproblem zurückführen und die Notation von LLMs übernehmen, kann die Verlustfunktion wie folgt geschrieben werden:

$$\mathcal{L}(\theta) \quad = \quad -\mathbb{E}_{(\mathbf{x},\mathbf{y})\sim\mathcal{D}}\big[U(\mathbf{x},\mathbf{y};\theta)\big] \tag{4.42}$$

wobei

$$U(\mathbf{x}, \mathbf{y}; \theta) \quad = \quad \sum_{t=1}^{T} \log \pi_\theta(y_t|\mathbf{x}, \mathbf{y}_{<t}) A(\mathbf{x}, \mathbf{y}_{<t}, y_t) \tag{4.43}$$

Hier ist $\pi_\theta(y_t|\mathbf{x}, \mathbf{y}_{<t}) = \Pr_\theta(y_t|\mathbf{x}, \mathbf{y}_{<t})$ das durch θ parametrisierte LLM.

Im Allgemeinen haben wir bei RLHF keinen von Menschen annotierten Eingabe-Ausgabe-Datensatz $\mathcal{D}$, sondern einen Datensatz, der nur Eingaben enthält. Die Ausgaben sind in diesem Fall typischerweise die vom LLM gemachten Vorhersagen. Die Verlustfunktion wird dann definiert als

$$\mathcal{L}(\theta) \quad = \quad -\mathbb{E}_{\mathbf{x}\sim\mathcal{D}} \mathbb{E}_{\mathbf{y}\sim\pi_\theta(\cdot|\mathbf{x})} \big[U(\mathbf{x}, \mathbf{y}; \theta) \big] \tag{4.44}$$

wobei $\mathcal{D}$ den reinen Eingabedatensatz bezeichnet und $\mathbf{y} \sim \pi_\theta(\cdot|\mathbf{x})$ bedeutet, dass die Ausgabe $\mathbf{y}$ von der Policy $\pi_\theta(\cdot|\mathbf{x})$ abgetastet wird.

Die obige Formulierung stellt eine grundlegende Form der A2C-Methode für LLMs dar. Verbesserte Versionen dieses Modells werden häufiger in RLHF verwendet. In der folgenden Diskussion werden wir weiterhin die Notation des Reinforcement Learning verwenden, um die Darstellung zu vereinfachen, und später auf die Notation der Sprachmodellierung zurückkommen.

Eine übliche Verbesserung von Policy-Gradienten-Methoden ist die Verwendung von Importance Sampling, um die Schätzung von $U(\tau; \theta)$ zu verfeinern. Dies kann wie folgt geschrieben werden

$$U(\tau; \theta) \quad = \quad \sum_{t=1}^{T} \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)} A(s_t, a_t) \tag{4.45}$$

Hier ersetzen wir die Log-Wahrscheinlichkeit $\log \pi_\theta(a_t|s_t)$ durch das Verhältnis $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)}$. θ_{ref} bezeichnet die Parameter der vorherigen Policy (wie z. B. ein initiales Modell, von dem aus wir das Training starten). Daher kann $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)}$, auch als Verhältnisfunktion bezeichnet, als das Log-Wahrscheinlichkeitsverhältnis zwischen der aktuellen Policy π_θ und der vorherigen Policy $\pi_{\theta_{\text{ref}}}$ (nennen wir sie die Referenz-Policy) interpretiert werden. Durch die Verwendung der Verhältnisfunktion gewichten wir die beobachteten Belohnungen neu, basierend auf der Wahrscheinlichkeit der Aktionen unter der aktuellen Policy im Vergleich zur Referenz-Policy. Wenn $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)} > 1$, wird die Aktion a_t von der aktuellen Policy im Vergleich zur Referenz-Policy stärker bevorzugt. Im Gegensatz dazu, wenn $\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)} < 1$, wird die Aktion a_t von der aktuellen Policy weniger bevorzugt[4].

[4]Betrachten wir einen allgemeineren Fall, in dem wir die Policy anhand ihrer erwarteten Belohnung bewerten möchten (siehe auch Gl. (4.18))

$$J(\theta) \quad = \quad \mathbb{E}_{\tau\sim\pi_\theta} \Big[R(\tau) \Big] \tag{4.46}$$

Hier bedeutet $\tau \sim \pi_\theta$, dass die Sequenz τ von der Policy π_θ generiert wird. Alternativ können wir $J(\theta)$ in einer anderen Form schreiben

$$J(\theta) \quad = \quad \mathbb{E}_{\tau\sim\pi_{\theta_{\text{ref}}}} \Big[\frac{\Pr_\theta(\tau)}{\Pr_{\theta_{\text{ref}}}(\tau)} R(\tau) \Big] \tag{4.47}$$

Ein Problem mit dem in Gl. (4.47) (sowie in Gl. (4.39)) vorgestellten Modell ist, dass die Varianz in den Gradientenschätzungen oft hoch ist, was den Lernprozess instabil macht. Um dieses Problem zu mildern, werden oft Techniken wie Clipping eingesetzt, um die Importance-Gewichte zu begrenzen und große Aktualisierungen zu verhindern. Eine abgeschnittene Version der Nutzenfunktion (auch als abgeschnittene Surrogat-Zielfunktion bezeichnet) ist gegeben durch

$$U_{\text{clip}}(\tau;\theta) = \sum_{t=1}^{T} \text{Clip}\Big(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)}\Big) A(s_t,a_t) \tag{4.49}$$

$$\text{Clip}\Big(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)}\Big) = \min\Big(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)}, \text{bound}\big(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon\big)\Big) \tag{4.50}$$

Hier beschränkt die Funktion $\text{bound}(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{ref}}}(a_t|s_t)}, 1-\epsilon, 1+\epsilon)$ die Verhältnisfunktion auf den Bereich $[1-\epsilon, 1+\epsilon]$.

Eine weitere Verbesserung des obigen Modells besteht darin, Trust Regions bei der Optimierung zu berücksichtigen [Schulman et al., 2015]. Im Reinforcement Learning kann eine große Aktualisierung der Policy zu Instabilität führen, bei der der Agent nach einer Aktualisierung schlechter abschneiden kann. Eine vernünftige Idee ist es, das Modell in der Trust Region zu optimieren, was sich auf einen Bereich um die aktuelle Parameterschätzung bezieht, in dem sich das Modell gut verhält. Ein Ansatz zur Einbeziehung von Trust Regions besteht darin, eine Beschränkung für die Größe der Policy-Aktualisierung aufzuerlegen, um sicherzustellen, dass die aktuelle Policy nicht zu stark von der Referenz-Policy abweicht. Dies kann erreicht werden, indem der Zielfunktion ein Strafterm hinzugefügt wird, der auf einer Form der Divergenz zwischen der aktuellen und der Referenz-Policy basiert. Eine einfache Form eines solchen Strafterms ist durch die Differenz der Log-Wahrscheinlichkeit der Sequenz τ unter der aktuellen Policy im Vergleich zur Referenz-Policy gegeben:

$$\text{Penalty} = \log \pi_\theta(\tau) - \log \pi_{\theta_{\text{ref}}}(\tau) \tag{4.51}$$

Es ist nicht schwer zu erkennen, dass die rechten Seiten dieser Gleichungen im Wesentlichen gleich sind, da $\mathbb{E}_{\tau\sim\pi_{\theta_{\text{ref}}}}\Big[\frac{\Pr_\theta(\tau)}{\Pr_{\theta_{\text{ref}}}(\tau)} R(\tau)\Big] = \sum_\tau \Pr_{\theta_{\text{ref}}}(\tau)\frac{\Pr_\theta(\tau)}{\Pr_{\theta_{\text{ref}}}(\tau)} R(\tau) = \sum_\tau \Pr_\theta(\tau) R(\tau) = \mathbb{E}_{\tau\sim\pi_\theta}\Big[R(\tau)\Big]$

Beachten Sie, dass diese Äquivalenz nur gilt, wenn die Erwartung über den gesamten Sequenzraum gebildet wird. In der Praxis tasten wir jedoch beim Policy-Lernen oft nur eine relativ kleine Anzahl von Sequenzen mit einer Policy ab. Daher ist die Abtastmethode selbst von Bedeutung. Gl. (4.47) bietet eine interessante Möglichkeit, die Abtast- und Belohnungsberechnungsprozesse zu trennen: Wir verwenden zuerst eine Basis-Policy (mit θ_{ref}), um eine Anzahl von Sequenzen abzutasten, und verwenden dann die Ziel-Policy (mit θ), um die erwartete Belohnung zu berechnen. Auf diese Weise trennen wir die Policy, die zum Sammeln der Daten verwendet wird, und die Policy, die zur Berechnung des Gradienten verwendet wird. Dieser Ansatz vermeidet die Notwendigkeit, direkt von der Policy abzutasten, die wir bewerten, was in Fällen vorteilhaft sein kann, in denen das Generieren von Sequenzen aus der Ziel-Policy teuer oder schwierig ist. Im Reinforcement Learning wird $\mathbb{E}_{\tau\sim\pi_{\theta_{\text{ref}}}}\Big[\frac{\Pr_\theta(\tau)}{\Pr_{\theta_{\text{ref}}}(\tau)} R(\tau)\Big]$ oft als Surrogat-Ziel bezeichnet.

Gl. (4.47) kann auch aus einer Policy-Gradienten-Perspektive interpretiert werden. Für $\mathbb{E}_{\tau\sim\pi_{\theta_{\text{ref}}}}\Big[\frac{\Pr_\theta(\tau)}{\Pr_{\theta_{\text{ref}}}(\tau)} R(\tau)\Big]$ ist der Gradient bei $\theta = \theta_{\text{ref}}$ gegeben durch

$$\frac{\partial}{\partial\theta}\mathbb{E}_{\tau\sim\pi_{\theta_{\text{ref}}}}\Big[\frac{\Pr_\theta(\tau)}{\Pr_{\theta_{\text{ref}}}(\tau)} R(\tau)\Big]\Big|_{\theta=\theta_{\text{ref}}} = \mathbb{E}_{\tau\sim\pi_{\theta_{\text{ref}}}}\Big[\frac{\partial \Pr_\theta(\tau)|_{\theta=\theta_{\text{ref}}}}{\partial\theta} R(\tau)\Big] \tag{4.48}$$

Die rechte Seite ist eine Standardform, die in Policy-Gradienten-Methoden verwendet wird, was bedeutet, dass wir die Richtung der Parameteraktualisierung am Punkt $\theta = \theta_{\text{ref}}$ auf der Optimierungsoberfläche berechnen.

In der Praxis kann dieser Strafterm angenähert werden, indem nur die Policy-Wahrscheinlichkeiten berücksichtigt und die Dynamik ignoriert wird. Dies ergibt

$$\text{Penalty} \quad = \quad \sum_{t=1}^{T} \log \pi_\theta(a_t|s_t) - \sum_{t=1}^{T} \log \pi_{\theta_{\text{ref}}}(a_t|s_t) \tag{4.52}$$

Indem wir diesen Strafterm in das Optimierungsziel einbeziehen, ermutigen wir die aktuelle Policy, nahe an der Referenz-Policy zu bleiben, und begrenzen so sehr große Aktualisierungen, die das Lernen destabilisieren könnten.

Wir können diesen Strafterm in die abgeschnittene Surrogat-Zielfunktion einbeziehen und erhalten

$$U_{\text{ppo-clip}}(\tau;\theta) \quad = \quad U_{\text{clip}}(\tau;\theta) - \beta\text{Penalty} \tag{4.53}$$

wobei β das Gewicht des Strafterms ist. Diese Trainingsmethode wird als Proximale Policy-Optimierung (PPO) bezeichnet und ist eine der beliebtesten Methoden des Reinforcement Learning, die in LLMs und vielen anderen Bereichen eingesetzt wird [Schulman et al., 2017].

Jetzt können wir das Ziel des Trainings von LLMs in Form von PPO schreiben.

$$U(\mathbf{x},\mathbf{y};\theta) \quad = \quad U_{\text{ppo-clip}}(\mathbf{x},\mathbf{y};\theta) - \beta\text{Penalty} \tag{4.54}$$

wobei

$$\begin{aligned} U_{\text{ppo-clip}}(\mathbf{x},\mathbf{y};\theta) &= \sum_{t=1}^{T} \text{Clip}\Big(\frac{\pi_\theta(y_t|\mathbf{x},\mathbf{y}_{<t})}{\pi_{\theta_{\text{ref}}}(y_t|\mathbf{x},\mathbf{y}_{<t})}\Big) A(\mathbf{x},\mathbf{y}_{<t},y_t) && (4.55)\\ \text{Penalty} &= \log \Pr_\theta(\mathbf{y}|\mathbf{x}) - \log \Pr_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x}) \\ &= \sum_{t=1}^{T} \log \Pr_\theta(y_t|\mathbf{x},\mathbf{y}_{<t}) - \sum_{t=1}^{T} \log \Pr_{\theta_{\text{ref}}}(y_t|\mathbf{x},\mathbf{y}_{<t}) && (4.56) \end{aligned}$$

Obwohl die Notation hier etwas umständlich erscheint, ist die Idee von PPO einfach: Wir entwickeln ein Ziel, indem wir das abgeschnittene Wahrscheinlichkeitsverhältnis der Ziel- und Referenz-Policies mit einer Vorteilsfunktion kombinieren und dann einen Strafterm auferlegen, der sicherstellt, dass die Policy-Aktualisierungen nicht zu groß sind. Das PPO-basierte RLHF wird in Abbildung 4.9 veranschaulicht.

Zusammenfassend lässt sich sagen, dass die Implementierung von RLHF den Aufbau von vier Modellen erfordert, die alle auf der Transformer-Decoder-Architektur basieren.

- Belohnungsmodell ($r_\phi(\cdot)$, wobei ϕ die Parameter bezeichnet). Das Belohnungsmodell lernt aus menschlichen Präferenzdaten, die Belohnung für jedes Paar von Eingabe- und Ausgabe-Token-Sequenzen vorherzusagen. Es handelt sich um einen Transformer-Decoder, gefolgt von einer linearen Schicht, die eine Sequenz (die Verkettung von Eingabe und Ausgabe) auf einen reellwertigen Belohnungswert abbildet.

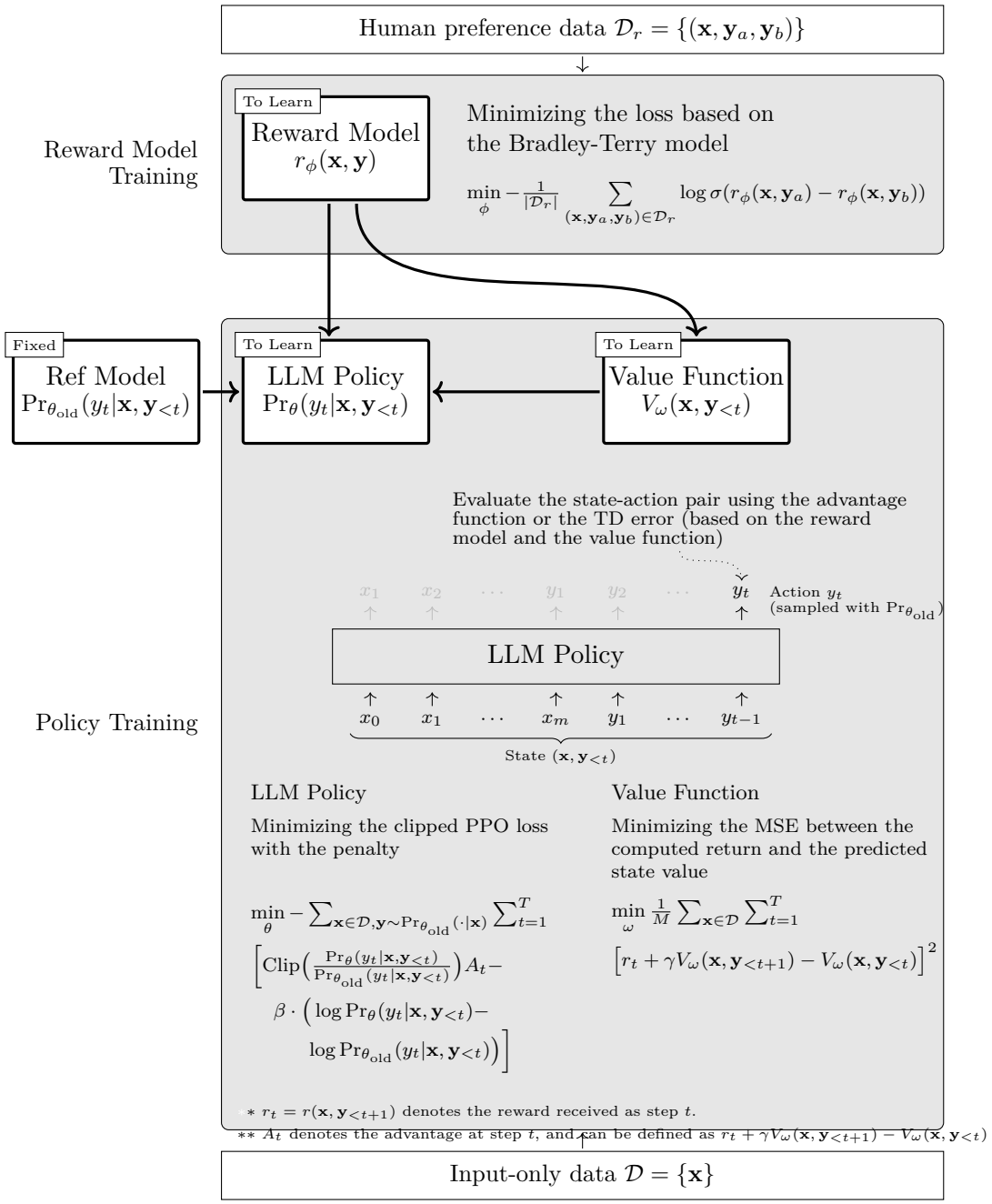


Abbildung 4.9: Darstellung von RLHF. Der erste Schritt besteht darin, menschliche Präferenzdaten zu sammeln und das Belohnungsmodell anhand dieser Daten zu trainieren. Sobald das Belohnungsmodell optimiert ist, trainieren wir zusammen mit dem Referenzmodell sowohl die Policy als auch die Wertefunktion. In jedem Vorhersageschritt berechnen wir die Summe des PPO-basierten Verlusts und aktualisieren die Parameter der Policy. Dies erfordert den Zugriff auf das Belohnungsmodell, das Referenzmodell und die vorhandene Wertefunktion. Gleichzeitig aktualisieren wir die Parameter der Wertefunktion, indem wir den MSE-Verlust minimieren.

- Wertemodell oder Wertefunktion ($V_\omega(\cdot)$, wobei ω die Parameter bezeichnet). Die Wertefunktion erhält Belohnungswerte vom Belohnungsmodell und wird trainiert, um die erwartete Summe der Belohnungen vorherzusagen, die von einem Zustand

ausgehend erzielt werden können. Sie basiert im Allgemeinen auf der gleichen Architektur wie das Belohnungsmodell.

- Referenzmodell ($\pi_{\theta_{\text{ref}}}(\cdot) = \Pr_{\theta_{\text{ref}}}(\cdot)$, wobei θ_{ref} die Parameter bezeichnet). Das Referenzmodell ist das Basis-LLM, das als Ausgangspunkt für das Policy-Training dient. In RLHF stellt es die vorherige Version des Modells oder ein ohne menschliches Feedback trainiertes Modell dar. Es wird verwendet, um ein Sampling über den Raum der Ausgaben durchzuführen und zur Verlustberechnung für das Policy-Training beizutragen.

- Zielmodell oder Policy ($\pi_\theta(\cdot) = \Pr_\theta(\cdot)$, wobei θ die Parameter bezeichnet). Diese Policy steuert, wie das LLM angesichts seines Kontexts das am besten geeignete nächste Token auswählt. Es wird unter der Aufsicht sowohl des Belohnungsmodells als auch des Wertemodells trainiert.

In der Praxis müssen diese Modelle in einer bestimmten Reihenfolge trainiert werden. Zuerst müssen wir sie mit einigen anderen Modellen initialisieren. Zum Beispiel können das Belohnungsmodell und das Wertemodell mit einem vortrainierten LLM initialisiert werden, während das Referenzmodell und das Zielmodell mit einem Modell initialisiert werden können, das durch Anweisungen feinabgestimmt wurde. Beachten Sie, dass das Referenzmodell zu diesem Zeitpunkt einsatzbereit ist und nicht weiter aktualisiert wird. Zweitens müssen wir menschliche Präferenzdaten sammeln und das Belohnungsmodell mit diesen Daten trainieren. Drittens werden sowohl das Wertemodell als auch die Policy gleichzeitig mit dem Belohnungsmodell trainiert. An jeder Position in einer Ausgabe-Token-Sequenz aktualisieren wir das Wertemodell, indem wir den MSE-Fehler der Wertevorhersage minimieren, und die Policy wird durch Minimierung des PPO-Verlusts aktualisiert.

4.4 Verbesserte Anpassung an menschliche Präferenzen

Im vorherigen Abschnitt haben wir die grundlegenden Konzepte des Reinforcement Learning und das allgemeine Framework von RLHF besprochen. In diesem Abschnitt werden wir einige Verfeinerungen von RLHF und alternative Methoden diskutieren, um eine Anpassung an menschliche Präferenzen zu erreichen.

4.4.1 Bessere Belohnungsmodellierung

In Abschnitt 4.3.2 haben wir die Aufgabe des Lernens aus menschlichen Präferenzen sowie die Verwendung des paarweisen Ranking-Verlusts zum Trainieren von Belohnungsmodellen beleuchtet. Hier betrachten wir weitere Methoden für die Belohnungsmodellierung. Unsere Diskussion wird relativ allgemein gehalten sein, und da das Belohnungsmodell in vielen Problemen des bestärkenden Lernens weit verbreitet ist, wird es für uns einfach sein, die hier besprochenen Methoden auf RLHF und verwandte Anwendungen anzuwenden.

4.4.1.1 Überwachungssignale

Das Training von Belohnungsmodellen kann im Großen und Ganzen als ein Ranking-Problem betrachtet werden, bei dem das Modell lernt, Ausgaben Bewertungen zuzuweisen, sodass deren Reihenfolge die von Menschen angegebenen Präferenzen widerspiegelt. Es gibt mehrere Methoden, ein Belohnungsmodell aus der Perspektive des Rankings zu trainieren.

Ein Ansatz besteht darin, das paarweise Ranking auf ein listenweises Ranking zu erweitern. Für jede Stichprobe in einem Datensatz können wir das LLM verwenden, um mehrere Ausgaben zu generieren, und menschliche Experten bitten, diese Ausgaben zu ordnen. Zum Beispiel, bei einem Satz von vier Ausgaben $\{\mathbf{y}_1, \mathbf{y}_2, \mathbf{y}_3, \mathbf{y}_4\}$ kann eine mögliche Reihenfolge von ihnen $\mathbf{y}_2 \succ \mathbf{y}_3 \succ \mathbf{y}_1 \succ \mathbf{y}_4$ sein. Eine sehr einfache Methode, um die Reihenfolge der Liste zu modellieren, besteht darin, den Verlust aus paarweisen Vergleichen zu akkumulieren. Zum Beispiel können wir den listenweisen Verlust definieren, indem wir den Verlust über alle Paare von Ausgaben akkumulieren:

$$\mathcal{L}_{\text{list}} = -\mathbb{E}_{(\mathbf{x},Y)\sim\mathcal{D}_r}\Big[\frac{1}{N(N-1)}\sum_{\substack{\mathbf{y}_a\in Y,\mathbf{y}_b\in Y\\ \mathbf{y}_a\neq\mathbf{y}_b}} \log \Pr(\mathbf{y}_a \succ \mathbf{y}_b|\mathbf{x})\Big] \tag{4.57}$$

wobei Y eine Liste von Ausgaben ist und N die Anzahl der Ausgaben in der Liste ist. $\Pr(\mathbf{y}_a \succ \mathbf{y}_b|\mathbf{x})$ kann mit dem Bradley-Terry-Modell definiert werden, das heißt, $\Pr(\mathbf{y}_a \succ \mathbf{y}_b|\mathbf{x}) = \text{Sigmoid}(r(\mathbf{x}, \mathbf{y}_a) - r(\mathbf{x}, \mathbf{y}_b))$. Hier lassen wir den hochgestellten Index ϕ bei $\Pr(\cdot)$ weg, um die Notation übersichtlich zu halten.

Eine Erweiterung des Bradley-Terry-Modells für das listenweise Ranking könnte einen Ranking-Mechanismus beinhalten, der die gesamte Liste der Ausgaben berücksichtigt, anstatt nur paarweiser Vergleiche. Ein solches Modell ist das Plackett-Luce-Modell, das das Bradley-Terry-Modell verallgemeinert, um mehrere Elemente in einem Ranking zu handhaben [Plackett, 1975]. Im Plackett-Luce-Modell definieren wir für jedes Element in einer Liste einen „Wert“(worth), der seine relative Stärke widerspiegelt, gegenüber anderen Elementen ausgewählt zu werden. Für das Problem der Belohnungsmodellierung hier kann der Wert von $\mathbf{y}$ in der Liste Y definiert werden als

$$\alpha(\mathbf{y}) = \exp(r(\mathbf{x}, \mathbf{y})) \tag{4.58}$$

Dann ist die Wahrscheinlichkeit, $\mathbf{y}$ aus Y auszuwählen, gegeben durch

$$\begin{aligned}\Pr(\mathbf{y} \text{ is selected}|\mathbf{x}, Y) &= \frac{\alpha(\mathbf{y})}{\sum_{\mathbf{y}'\in Y}\alpha(\mathbf{y}')} \\ &= \frac{\exp(r(\mathbf{x}, \mathbf{y}))}{\sum_{\mathbf{y}'\in Y}\exp(r(\mathbf{x}, \mathbf{y}'))}\end{aligned} \tag{4.59}$$

Angenommen, $\mathring{Y}$ ist eine geordnete Liste $\mathbf{y}_{j_1} \succ \mathbf{y}_{j_2} \succ \cdots \succ \mathbf{y}_{j_N}$. Die Gesamt-Log-Wahrscheinlichkeit dieser geordneten Liste kann als die Summe der bedingten Log-Wahrscheinlichkeiten in jeder Auswahlstufe definiert werden, gegeben durch

$$\begin{aligned}
\log \Pr(\mathring{Y}|\mathbf{x}) &= \log \Pr(\mathbf{y}_{j_1} \succ \mathbf{y}_{j_2} \succ \cdots \succ \mathbf{y}_{j_N}|\mathbf{x}) \\
&= \log \Pr(\mathbf{y}_{j_1}|\mathbf{x}, \{\mathbf{y}_{j_1}, \mathbf{y}_{j_2}, ..., \mathbf{y}_{j_N}\}) + \\
&\quad \log \Pr(\mathbf{y}_{j_2}|\mathbf{x}, \{\mathbf{y}_{j_2}, ..., \mathbf{y}_{j_N}\}) + \\
&\quad \cdots + \\
&\quad \log \Pr(\mathbf{y}_{j_N}|\mathbf{x}, \{\mathbf{y}_{j_N}\}) \\
&= \sum_{k=1}^{N} \log \Pr(\mathbf{y}_{j_k}|\mathbf{x}, \mathring{Y}_{\geq k})
\end{aligned} \tag{4.60}$$

wobei $\mathring{Y}_{\geq k}$ die Teilmenge der Liste von Ausgaben darstellt, die in der k-ten Stufe unausgewählt bleiben, d.h., $\mathring{Y}_{\geq k} = \{\mathbf{y}_{j_k}, ..., \mathbf{y}_{j_N}\}$. Gegeben die Log-Wahrscheinlichkeit $\log \Pr(\mathring{Y}|\mathbf{x})$, können wir die Verlustfunktion basierend auf dem Plackett-Luce-Modell definieren durch

$$\mathcal{L}_{\text{pl}} = -\mathbb{E}_{(\mathbf{x},\mathring{Y})\sim\mathcal{D}_r}\big[\log \Pr(\mathring{Y}|\mathbf{x})\big] \tag{4.61}$$

Es gibt auch viele andere paarweise und listenweise Methoden zur Modellierung von Rankings, wie zum Beispiel RankNet [Burges et al., 2005] und ListNet [Cao et al., 2007]. All diese Methoden lassen sich in eine große Familie von Learning-to-Rank-Ansätzen einteilen, und die meisten von ihnen sind auf das Problem der Modellierung menschlicher Präferenzen anwendbar. Die Erörterung dieser Methoden würde jedoch den Rahmen dieses Kapitels sprengen. Interessierte Leser können für weitere Details auf Bücher zu diesem Thema zurückgreifen [Liu, 2009; Li, 2011].

Zusätzlich zum paarweisen und listenweisen Ranking bieten punktweise Methoden zum Trainieren von Belohnungsmodellen eine alternative Möglichkeit, menschliche Präferenzen zu erfassen. Im Gegensatz zu Methoden, die sich auf die relativen Rankings zwischen verschiedenen Ausgaben konzentrieren, behandeln punktweise Methoden jede Ausgabe unabhängig. Zum Beispiel könnten menschliche Experten einer einzelnen Ausgabe eine Bewertung zuweisen, wie etwa eine Bewertung auf einer Fünf-Punkte-Skala. Das Ziel ist es, das Belohnungsmodell so anzupassen, dass seine Ausgaben mit diesen Bewertungen übereinstimmen. Eine einfache Möglichkeit, punktweises Training zu erreichen, sind Regressionsverfahren, bei denen die Belohnung jeder Ausgabe als Zielvariable behandelt wird. Sei $\varphi(\mathbf{x}, \mathbf{y})$ die von Menschen für $\mathbf{y}$ bei gegebenem $\mathbf{x}$ vergebene Bewertung. Punktweise Belohnungsmodelle können durch Minimierung einer Verlustfunktion trainiert werden, die oft auf dem mittleren quadratischen Fehler oder anderen Regressionsverlusten basiert, zwischen der vorhergesagten Belohnung $r(\mathbf{x}, \mathbf{y})$ und dem tatsächlichen menschlichen Feedback $\varphi(\mathbf{x}, \mathbf{y})$. Zum Beispiel könnte die Verlustfunktion sein

$$\mathcal{L}_{\text{point}} = -\mathbb{E}\big[\varphi(\mathbf{x}, \mathbf{y}) - r(\mathbf{x}, \mathbf{y})\big]^2 \tag{4.62}$$

Obwohl punktweise Methoden konzeptionell einfacher sind und das Belohnungsmodell direkt zur Vorhersage von Bewertungen anleiten können, sind sie in RLHF möglicherweise

nicht immer die beste Wahl. Ein Problem ist, dass diese Methoden mit einer hohen Varianz im menschlichen Feedback zu kämpfen haben können, insbesondere wenn verschiedene Experten inkonsistente Bewertungen für ähnliche Ausgaben abgeben. Da sie sich auf die Anpassung an absolute Bewertungen anstatt auf relative Unterschiede konzentrieren, können Inkonsistenzen bei der Bewertung zu einer schlechten Modellleistung führen. Darüber hinaus könnte die Anpassung an spezifische bewertete Ausgaben die Generalisierung beeinträchtigen, insbesondere da die Trainingsdaten in RLHF oft sehr begrenzt sind. Im Gegensatz dazu können Methoden, die relative Präferenzen berücksichtigen, das Erlernen allgemeinerer Erfolgs- und Misserfolgsmuster fördern. Dennoch gibt es Szenarien, in denen punktweise Methoden dennoch geeignet sein könnten. Zum Beispiel können sich punktweise Methoden bei Aufgaben als effektiv erweisen, bei denen Trainingsdaten reichlich vorhanden sind und die Kosten für die Erstellung genauer, konsistenter Annotationen niedrig sind.

Tatsächlich können wir, um das Überwachungssignal für das Training des Belohnungsmodells robuster zu machen, auch zusätzliche Regularisierungsterme in das Training einführen. Wenn wir beispielsweise den ersten Term $U_{\text{ppo-clip}}(\mathbf{x}, \mathbf{y}; \theta)$ in Gl. (4.54) als eine Art verallgemeinerte Belohnung betrachten, kann der zweite Term (d.h. der Strafterm) als eine Form der Regularisierung für das Belohnungsmodell angesehen werden, nur dass hier das Ziel ist, die Policy anstelle des Belohnungsmodells zu trainieren. Ein weiteres Beispiel ist, dass Eisenstein et al. [2023] einen Regularisierungsterm entwickeln, der auf der quadrierten Summe der Belohnungen basiert, und ihn zum Verlust des paarweisen Vergleichs in RLHF hinzufügen:

$$\begin{aligned} \mathcal{L}_{\text{reg}} &= \mathcal{L}_{\text{pair}} + \left(-\mathbb{E}_{(\mathbf{x},\mathbf{y}_a,\mathbf{y}_b)\sim\mathcal{D}_r}\left[r(\mathbf{x},\mathbf{y}_a) + r(\mathbf{x},\mathbf{y}_b)\right]^2\right) \\ &= -\mathbb{E}_{(\mathbf{x},\mathbf{y}_a,\mathbf{y}_b)\sim\mathcal{D}_r}\left[\log \Pr_\phi(\mathbf{y}_a \succ \mathbf{y}_b|\mathbf{x})\right] \\ &\quad -\mathbb{E}_{(\mathbf{x},\mathbf{y}_a,\mathbf{y}_b)\sim\mathcal{D}_r}\left[r(\mathbf{x},\mathbf{y}_a) + r(\mathbf{x},\mathbf{y}_b)\right]^2 \end{aligned} \tag{4.63}$$

Die Optimierung mit diesem Regularisierungsterm kann helfen, die Unterbestimmtheit von Belohnungsmodellen abzuschwächen[5].

4.4.1.2 Spärliche Belohnungen vs. dichte Belohnungen

Wie in Abschnitt 4.3 erörtert, sind die Belohnungen im RLHF sehr spärlich: Sie werden nur am Ende von Sequenzen beobachtet und nicht kontinuierlich während des gesamten Generierungsprozesses. Der Umgang mit spärlichen Belohnungen ist seit langem ein Problem im verstärkenden Lernen und stellt eine der Herausforderungen in vielen praktischen Anwendungen dar. In der Robotik beispielsweise muss oft die Belohnungsfunktion gestaltet werden, um die Optimierung zu erleichtern, anstatt sich ausschließlich auf Belohnungen am Ende der Sequenz zu verlassen. Verschiedene Methoden wurden entwickelt, um dieses Problem anzugehen. Ein gängiger Ansatz ist die Belohnungsformung (Reward Shaping), bei der die ursprüngliche Funktion so modifiziert wird, dass sie Zwischenbelohnungen enthält und somit unmittelbareres Feedback liefert. Außerdem kann man Curriculum Learning einsetzen, um Aufgaben sequenziell so zu strukturieren, dass die Komplexität allmählich zunimmt. Dies kann Modellen helfen, zuerst einfachere Aufgaben zu bewältigen, was sie

[5] Ein Modell wird als unterbestimmt bezeichnet, wenn es mehrere alternative Parametersätze gibt, die dasselbe Ziel erreichen können.

auf komplexere Herausforderungen vorbereitet, während sich ihre Fähigkeiten entwickeln. Es gibt viele solcher Methoden, die die Auswirkungen spärlicher Belohnungen abmildern können, wie z. B. Monte-Carlo-Methoden und intrinsische Motivation. Die meisten dieser Methoden sind allgemein und ihre Erörterung findet sich in der breiteren Literatur zum verstärkenden Lernen, wie z. B. im Buch von Sutton and Barto [2018].

Obwohl wir hier nicht im Detail auf Methoden zur Abmilderung spärlicher Belohnungen eingehen, stellt sich eine interessante Frage: Warum sind spärliche Belohnungen im RLHF so erfolgreich? Erinnern wir uns an Abschnitt 4.3.1, dass das zu jedem Zeitschritt t erhaltene Überwachungssignal nicht die Belohnung für die aktuelle Aktion ist, sondern vielmehr eine Form der akkumulierten Belohnungen von t bis zum letzten Zeitschritt. Solche Überwachungssignale sind über die Sequenz hinweg dicht, da die am Ende der Sequenz erhaltene Belohnung auf diesen Zeitschritt zurückübertragen werden kann, unabhängig davon, um welchen Zeitschritt es sich handelt. Mit anderen Worten, die spärlichen Belohnungen werden in dichte Überwachungssignale umgewandelt. Darüber hinaus zeigen Ng et al. [1999] aus der Perspektive der Belohnungsformung, dass die Belohnung bei t wie folgt definiert werden kann

$$r'(s_t, a_t, s_{t+1}) \quad = \quad r(s_t, a_t, s_{t+1}) + f(s_t, a_t, s_{t+1}) \tag{4.64}$$

wobei $r'(\cdot)$ die transformierte Belohnungsfunktion, $r(\cdot)$ die ursprüngliche Belohnungsfunktion und $f(\cdot)$ die formende Belohnungsfunktion ist. Um die Optimalität der Politik unter der transformierten Belohnungsfunktion zu gewährleisten, kann die formende Belohnungsfunktion in der Form

$$f(s_t, a_t, s_{t+1}) \quad = \quad \gamma\Phi(s_{t+1}) - \Phi(s_t) \tag{4.65}$$

gegeben werden, wobei $\Phi(s)$ als der potentielle Wert des Zustands s bezeichnet wird. Wenn wir $\Phi(s)$ als die übliche Wertfunktion wie in Gl. (4.15) definieren und Gl. (4.65) in Gl. (4.64) einsetzen, erhalten wir

$$r'(s_t, a_t, s_{t+1}) \quad = \quad r(s_t, a_t, s_{t+1}) + \gamma V(s_{t+1}) - V(s_t) \tag{4.66}$$

Es ist interessant zu sehen, dass diese Funktion genau dieselbe ist wie die in PPO verwendete Vorteilsfunktion. Dies stellt eine Verbindung zwischen vorteilsbasierten Methoden und der Belohnungsformung her: Der Vorteil ist im Wesentlichen eine geformte Belohnung.

Andererseits liegt einer der Gründe für die Verwendung von Belohnungen am Ende der Sequenz in der Natur der RLHF-Aufgaben. Im Gegensatz zu traditionellen Umgebungen des verstärkenden Lernens, in denen der Agent mit einer dynamischen Umgebung interagiert, beinhalten RLHF-Aufgaben oft komplexe Entscheidungsfindungen, die auf linguistischen oder anderen übergeordneten kognitiven Prozessen basieren. Diese Prozesse eignen sich nicht leicht für häufige und aussagekräftige Zwischenbelohnungen, da die Qualität und Angemessenheit der Aktionen erst nach Beobachtung ihrer Auswirkungen im größeren Kontext der gesamten Sequenz oder Aufgabe vollständig bewertet werden können. In diesem Fall sind die Belohnungssignale, die auf menschlichem Feedback basieren, obwohl sie sehr spärlich sind, typischerweise sehr informativ und genau. Folglich kann diese Spärlichkeit, zusammen mit der hohen Informativität und Genauigkeit des

menschlichen Feedbacks, das Lernen sowohl robust als auch effizient machen.

4.4.1.3 Feingranulare Belohnungen

Bei vielen Anwendungen ist unser Ziel komplexer als die bloße Bewertung eines gesamten Textes. Beispielsweise bestimmen wir bei der Sentimentanalyse oft nicht nur die Stimmung eines Textes, sondern müssen die Stimmung detaillierter analysieren, indem wir sie mit spezifischen Aspekten eines im Text behandelten Themas in Verbindung bringen. Betrachten Sie den Satz „Die Kamera des Telefons ist ausgezeichnet, aber die Akkulaufzeit ist enttäuschend."In diesem Beispiel müssten wir die über die Kamera und den Akku geäußerten Stimmungen getrennt analysieren. Eine solche Analyse, bekannt als aspektbasierte Sentimentanalyse, trägt zu einem feingranulareren Verständnis der Kundenrezension im Vergleich zur allgemeinen Sentimentanalyse bei.

Für das Problem der Belohnungsmodellierung müssen wir oft auch verschiedene Teile einer Sequenz modellieren. Eine einfache und unkomplizierte Methode hierfür ist, eine Sequenz in verschiedene Segmente zu unterteilen und dann die Belohnung für jedes Segment zu berechnen [Wu et al., 2023b]. Angenommen, eine Ausgabetokensequenz $\mathbf{y}$ kann nach einem bestimmten Kriterium in n_s Segmente $\{\bar{\mathbf{y}}_1, ..., \bar{\mathbf{y}}_{n_s}\}$ unterteilt werden. Wir können das Belohnungsmodell verwenden, um jedes dieser Segmente zu bewerten. Indem wir $\mathbf{x}$, $\mathbf{y}$ und $\bar{\mathbf{y}}_k$ als Eingabe für das Belohnungsmodell verwenden, ergibt sich der Belohnungswert für das k-te Segment wie folgt

$$r^k = r(\mathbf{x}, \mathbf{y}, \bar{\mathbf{y}}_k) \tag{4.67}$$

Dann ergibt sich der Belohnungswert für die gesamte Ausgabesequenz wie folgt

$$r(\mathbf{x}, \mathbf{y}) = \sum_{k=1}^{n_s} r(\mathbf{x}, \mathbf{y}, \bar{\mathbf{y}}_k) \tag{4.68}$$

Hier kann $r(\mathbf{x}, \mathbf{y})$ wie gewohnt zum Trainieren der Policy verwendet werden.

Ein Problem bei diesem Modell ist, dass das Trainieren von Belohnungsmodellen auf Segmentebene nicht so einfach ist wie das Lernen aus menschlichen Präferenzen für ganze Texte, da es schwierig ist, menschliche Präferenzdaten auf Segmentebene zu erhalten. Für bewertungsähnliche Probleme (z. B. bewerten wir ein Segment nach seinem Grad an Fehlinformationen) besteht ein einfacher Ansatz darin, jedem Segment einen Bewertungswert zuzuweisen und das Belohnungsmodell mit punktweisen Methoden zu trainieren. Zum Beispiel können wir ein starkes LLM verwenden, um die Sequenzen $\bar{\mathbf{y}}_1...\bar{\mathbf{y}}_{k-1}$ und $\bar{\mathbf{y}}_1...\bar{\mathbf{y}}_k$ zu bewerten und die Werte $s(\bar{\mathbf{y}}_1...\bar{\mathbf{y}}_{k-1})$ und $s(\bar{\mathbf{y}}_1...\bar{\mathbf{y}}_k)$ zu erhalten. Wir können dann den Wert des Segments $\bar{\mathbf{y}}_k$ als die Differenz zwischen $s(\bar{\mathbf{y}}_1...\bar{\mathbf{y}}_k)$ und $s(\bar{\mathbf{y}}_1...\bar{\mathbf{y}}_{k-1})$ definieren

$$s(\bar{\mathbf{y}}_k) = s(\bar{\mathbf{y}}_1...\bar{\mathbf{y}}_k) - s(\bar{\mathbf{y}}_1...\bar{\mathbf{y}}_{k-1}) \tag{4.69}$$

Mithilfe dieser segmentebenen Werte können wir das Belohnungsmodell mit einer Regressionsverlustfunktion trainieren

$$\mathcal{L}_{\text{rating}} = -\mathbb{E}_{\bar{\mathbf{y}}_k}\left[s(\bar{\mathbf{y}}_k) - r(\mathbf{x}, \mathbf{y}, \bar{\mathbf{y}}_k)\right]^2 \tag{4.70}$$

Manchmal kann die Ausrichtung als ein Klassifikationsproblem behandelt werden, zum Beispiel, wenn wir beurteilen, ob ein Segment ethische Probleme aufweist. In diesem Fall kann das Segment als ethisch oder unethisch gekennzeichnet werden, entweder durch Menschen oder durch zusätzliche Klassifikatoren. Anhand der Kennzeichnung des Segments können wir das Belohnungsmodell mit einer Klassifikationsverlustfunktion trainieren. Angenommen, $r(\mathbf{x}, \mathbf{y}, \bar{\mathbf{y}}_k) = 1$, wenn das Segment als unethisch klassifiziert wird, und andernfalls $r(\mathbf{x}, \mathbf{y}, \bar{\mathbf{y}}_k) = -1$[6]. Der Hinge-Verlust beim Training von binären Klassifikationsmodellen ist gegeben durch

$$\mathcal{L}_{\text{hinge}} \quad = \quad \max(0, 1 - r(\mathbf{x}, \mathbf{y}, \bar{\mathbf{y}}_k) \cdot \hat{r}) \tag{4.71}$$

wobei $\hat{r} \in \{1, -1\}$ die Ground-Truth-Kennzeichnung bezeichnet.

Die verbleibende Frage hier ist, wie $\mathbf{y}$ in Segmente aufgeteilt werden soll. Ein Ansatz besteht darin, eine Segmentierung mit fester Länge zu definieren, bei der $\mathbf{y}$ in gleich lange Abschnitte unterteilt wird. Dies ist jedoch nicht immer ideal, da der Inhalt der Sequenz möglicherweise nicht gut mit festen Grenzen übereinstimmt. Ein alternativer Ansatz besteht darin, $\mathbf{y}$ auf der Grundlage spezifischer linguistischer oder semantischer Hinweise zu segmentieren, wie z. B. Satzgrenzen, Themenwechsel oder andere bedeutungsvolle Strukturen im Text. Eine solche Segmentierung kann durch die Verwendung linguistischer Segmentierungssysteme oder durch das Anweisen von LLMs erreicht werden, natürliche Unterbrechungen in der Sequenz zu identifizieren. Ein weiterer Ansatz ist die Verwendung dynamischer Segmentierungsmethoden, die auf der Komplexität der Sequenz basieren. Zum Beispiel könnten Segmente dort definiert werden, wo sich der Belohnungswert signifikant ändert, was Verschiebungen in der modellierten Aufgabe entsprechen könnte.

4.4.1.4 Kombination von Belohnungsmodellen

Ein Belohnungsmodell kann als Stellvertreter für die Umgebung betrachtet werden. Da die tatsächliche Umgebung oft zu komplex oder unbekannt ist, ist die Entwicklung eines perfekten Stellvertreters für die Umgebung im Allgemeinen nicht möglich. Infolgedessen kann eine übermäßige Anpassung von LLMs an diesen unvollkommenen Stellvertreter zu einer Leistungsminderung führen, was als Überoptimierungsproblem bekannt ist [Stiennon et al., 2020; Gao et al., 2023a][7]. Wir können dies auch durch das Goodhartsche Gesetz erklären, das besagt: Wenn eine Maßnahme zu einem Ziel wird, hört sie auf, eine gute Maßnahme zu sein [Goodhart, 1984].

Die Bewältigung des Überoptimierungsproblems ist nicht einfach, und es gibt noch keine ausgereifte Lösung. Der ideale Ansatz könnte darin bestehen, ein Orakel-Belohnungsmodell zu entwickeln, das die wahren Ziele der Aufgabe perfekt erfasst und den Agenten am

[6] Um dem Belohnungsmodell die Ausgabe von Kategorien zu ermöglichen, können wir die in Abschnitt 4.3.2 beschriebene lineare Schicht durch eine Softmax-Schicht ersetzen.

[7] Dieses Problem wird auch als Reward Hacking oder Reward Gaming bezeichnet [Krakovna et al., 2020; Skalse et al., 2022; Pan et al., 2022], was sich auf das Phänomen bezieht, bei dem der Agent versucht, das Belohnungsmodell auszutricksen, es ihm aber nicht gelingt, seine Handlungen an den eigentlichen Zielen der Aufgabe auszurichten. Stellen Sie sich einen Schüler vor, der Hausaufgaben bekommt und für deren Erledigung mit Punkten oder Lob belohnt wird. Der Schüler könnte dann Wege finden, die Hausaufgaben mit minimalem Aufwand zu erledigen, um die Belohnung zu maximieren, z. B. indem er Lösungen aus dem Internet oder aus früheren Aufgaben kopiert und einfügt, anstatt die Probleme selbst zu lösen.

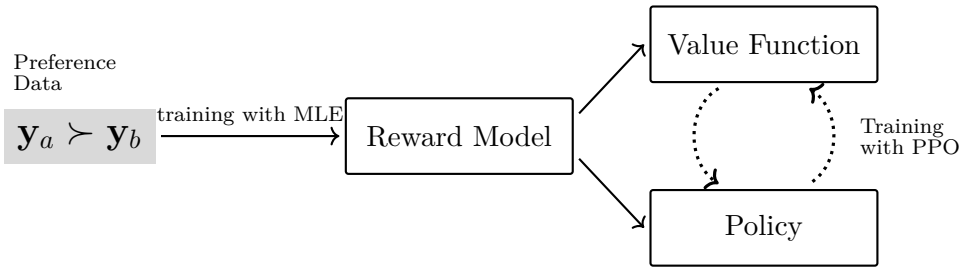
„Austricksen“hindern kann. Die Erstellung eines solchen Modells ist jedoch aufgrund der Komplexität der realen Umgebung sowie der Herausforderung, alle relevanten Faktoren zu definieren, die zum gewünschten Ergebnis beitragen, äußerst schwierig. Stattdessen ist ein praktischerer Ansatz die Kombination mehrerer Belohnungsmodelle, wodurch die Fehlausrichtung zwischen dem Trainingsziel und dem wahren Ziel, die durch die Verwendung eines einzigen, spezifischen Belohnungsmodells entsteht, gemildert wird [Coste et al., 2024].

Bei einer gegebenen Menge von Belohnungsmodellen ist deren Kombination unkompliziert, und in einigen Fällen können wir dieses Problem einfach als ein Ensemble-Lernproblem behandeln. Ein einfacher, aber gängiger Ansatz besteht darin, die Ausgaben dieser Modelle zu mitteln, um eine präzisere Belohnungsschätzung zu erhalten:

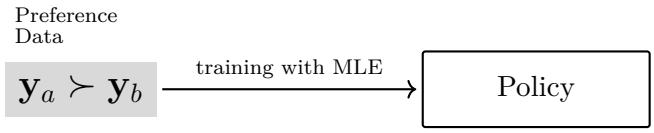
$$r_{\text{combine}} = \frac{1}{K}\sum_{k=1}^{K} w_k \cdot r_k(\mathbf{x}, \mathbf{y}) \tag{4.72}$$

wobei $r_k(\cdot)$ das k-te Belohnungsmodell im Ensemble ist, w_k das Gewicht von $r_k(\cdot)$ ist und K die Anzahl der Belohnungsmodelle ist. Diese kombinierte Belohnung kann dann verwendet werden, um das Training einer Policy zu überwachen. Tatsächlich gibt es viele Möglichkeiten, verschiedene Modelle zu kombinieren. Man kann beispielsweise Vorhersagen mittels Bayes'scher Modellmittelung treffen oder ein Fusionsnetzwerk entwickeln, um zu lernen, wie die Vorhersagen verschiedener Modelle kombiniert werden. Alternativ kann man diese Aufgabe als ein multikriterielles Optimierungsproblem formulieren und mehrere Belohnungsmodelle verwenden, um die Policy gleichzeitig zu trainieren. Diese Methoden wurden in der Literatur zur Optimierung und zum maschinellen Lernen intensiv diskutiert [Miettinen, 1999; Bishop, 2006].

Neben den Methoden zur Modellkombination ist eine weitere wichtige Frage, wie man mehrere unterschiedliche Belohnungsmodelle sammelt oder konstruiert. Einer der einfachsten Ansätze ist die Anwendung von Ensemble-Lerntechniken, wie z. B. die Entwicklung verschiedener Belohnungsmodelle aus unterschiedlichen Teilmengen eines gegebenen Datensatzes oder aus verschiedenen Datenquellen. Für RLHF ist es auch möglich, Belohnungsmodelle auf der Grundlage von Überlegungen zu verschiedenen Aspekten der Anpassung zu konstruieren. Zum Beispiel können wir ein Belohnungsmodell entwickeln, um die faktische Genauigkeit der Ausgabe zu bewerten, und ein weiteres Belohnungsmodell, um die Vollständigkeit der Ausgabe zu bewerten. Diese beiden Modelle ergänzen sich gegenseitig und können kombiniert werden, um die Gesamtbewertung der Ausgabe zu verbessern. Ein anderer Ansatz besteht darin, verschiedene Standard-LLMs (off-the-shelf) als Belohnungsmodelle einzusetzen. Dieser Ansatz ist einfach und praktisch, da es bereits viele gut entwickelte LLMs gibt und wir sie nur mit geringen oder keinen Änderungen verwenden müssen. Eine interessante, wenn auch nicht eng mit der hiesigen Diskussion verbundene Frage stellt sich: Kann ein LLM, das sich an anderen LLMs ausrichtet, diese LLMs übertreffen? Auf den ersten Blick wahrscheinlich nicht. Dies liegt zum Teil daran, dass das Ziel-LLM andere LLMs lediglich auf der Grundlage begrenzter Überwachung imitiert und daher die Nuancen im Verhalten dieser Supervisoren nicht gut erfassen kann. Angesichts der starken Generalisierungsfähigkeit von LLMs kann dieser Ansatz jedoch tatsächlich sehr vorteilhaft sein. Zum Beispiel hat die Verwendung von quelloffenen oder kommerziellen LLMs als Belohnungsmodelle eine starke Leistung bei der Anpassung von



(a) RLHF (PPO)



(b) DPO

Abbildung 4.10: Standard-RLHF (PPO) vs. DPO. Bei RLHF werden die menschlichen Präferenzdaten verwendet, um ein Belohnungsmodell zu trainieren, das dann sowohl beim Training der Policy als auch der Wertfunktion eingesetzt wird. Bei DPO ist die Verwendung der menschlichen Präferenzdaten direkter, und die Policy wird auf diesen Daten trainiert, ohne dass ein Belohnungsmodell trainiert werden muss.

LLMs gezeigt und sogar bei mehreren populären Aufgaben State-of-the-Art-Ergebnisse erzielt [Lambert et al., 2024].

4.4.2 Direkte Präferenzoptimierung

Obwohl das Lernen von Belohnungsmodellen ein Standardschritt im Reinforcement Learning ist, macht es den gesamten Trainingsprozess wesentlich komplexer als überwachtes Training. Das Trainieren eines zuverlässigen Belohnungsmodells ist an sich keine leichte Aufgabe, und ein schlecht trainiertes Belohnungsmodell kann das Ergebnis des Policy-Lernens stark beeinflussen. Wir betrachten nun eine alternative Anpassungsmethode, genannt direkte Präferenzoptimierung (DPO), die das Trainingsframework vereinfacht, indem die Notwendigkeit, Belohnungen explizit zu modellieren, entfällt [Rafailov et al., 2024]. Diese Methode optimiert die Policy direkt auf der Grundlage von Benutzerpräferenzen, anstatt ein separates Belohnungsmodell zu entwickeln. Dadurch können wir eine Anpassung an menschliche Präferenzen auf eine Weise erreichen, die dem überwachten Lernen ähnelt. Abbildung 4.10 zeigt einen Vergleich der Standard-RLHF-Methode und der DPO-Methode.

Bevor wir das DPO-Ziel herleiten, wollen wir zunächst das Ziel des Policy-Trainings, das bei RLHF verwendet wird, betrachten. Wie in Abschnitt 4.3.3 erörtert, wird die Policy typischerweise durch die Optimierung einer Verlustfunktion mit einem Strafterm trainiert. Die DPO-Methode geht von einer einfachen Verlustfunktion aus, bei der die Qualität der Ausgabe $\mathbf{y}$ bei gegebener Eingabe $\mathbf{x}$ durch das Belohnungsmodell $r(\mathbf{x}, \mathbf{y})$ bewertet wird.

Das Trainingsziel ist somit gegeben durch

$$\tilde{\theta} = \arg\min_{\theta} \mathbb{E}_{\mathbf{x}\sim\mathcal{D}}\mathbb{E}_{\mathbf{y}\sim\pi_\theta(\cdot|\mathbf{x})}\big[\underbrace{-r(\mathbf{x},\mathbf{y})}_{\text{Verlust}} + \beta\underbrace{\left(\log\pi_\theta(\mathbf{y}|\mathbf{x}) - \log\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x})\right)}_{\text{Strafe}}\big] \tag{4.73}$$

Beachten Sie, dass in diesem Optimierungsproblem nur der Term $\pi_\theta(\mathbf{y}|\mathbf{x})$ von der Ziel-Policy $\pi_\theta(\cdot)$ abhängt. Sowohl das Belohnungsmodell $r(\mathbf{x},\mathbf{y})$ als auch das Referenzmodell $\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x})$ werden bei gegebenem $\mathbf{x}$ und $\mathbf{y}$ als fest angenommen. Dies ist eine starke Annahme im Vergleich zu PPO, aber wie später gezeigt wird, vereinfacht sie das Problem und ist entscheidend für die Herleitung des DPO-Ziels.

Da θ die Variable ist, die wir optimieren möchten, ordnen wir die rechte Seite von Gl. (4.73) neu an, um $\pi_\theta(\mathbf{y}|\mathbf{x})$ als unabhängigen Term zu isolieren:

$$\begin{aligned}
\tilde{\theta} &= \arg\min_{\theta} \mathbb{E}_{\mathbf{x}\sim\mathcal{D}}\mathbb{E}_{\mathbf{y}\sim\pi_\theta(\cdot|\mathbf{x})}\big[\beta\log\pi_\theta(\mathbf{y}|\mathbf{x}) - \beta\log\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x}) - r(\mathbf{x},\mathbf{y})\big] \\
&= \arg\min_{\theta} \mathbb{E}_{\mathbf{x}\sim\mathcal{D}}\mathbb{E}_{\mathbf{y}\sim\pi_\theta(\cdot|\mathbf{x})}\big[\log\pi_\theta(\mathbf{y}|\mathbf{x}) - \big(\log\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x}) + \frac{1}{\beta}r(\mathbf{x},\mathbf{y})\big)\big] \\
&= \arg\min_{\theta} \mathbb{E}_{\mathbf{x}\sim\mathcal{D}}\mathbb{E}_{\mathbf{y}\sim\pi_\theta(\cdot|\mathbf{x})}\big[\underbrace{\log\pi_\theta(\mathbf{y}|\mathbf{x})}_{\text{abhängig von }\theta} - \underbrace{\log\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x})\exp\big(\frac{1}{\beta}r(\mathbf{x},\mathbf{y})\big)}_{\text{nicht abhängig von }\theta}\big]
\end{aligned} \tag{4.74}$$

Diese Gleichung definiert die Zielfunktion als die Differenz zwischen der Log-Wahrscheinlichkeitsverteilung von y und einer anderen Funktion von y. Diese Form der Zielfunktion scheint nicht „ideal "zu sein, da wir normalerweise lieber die Differenz zwischen zwei Verteilungen sehen, damit wir diese Differenz als eine Art Divergenz zwischen den Verteilungen interpretieren können. Eine einfache Idee ist es, den zweiten Term (d.h. $\log\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x})\exp(\frac{1}{\beta}r(\mathbf{x},\mathbf{y}))$) in eine Log-Wahrscheinlichkeitsverteilung über den Definitionsbereich von $\mathbf{y}$ umzuwandeln. Wenn wir $\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x})\exp(\frac{1}{\beta}r(\mathbf{x},\mathbf{y}))$ als eine unnormalisierte Wahrscheinlichkeit von y behandeln, können wir sie in eine normalisierte Wahrscheinlichkeit umwandeln, indem wir sie durch einen Normalisierungsfaktor teilen:

$$Z(\mathbf{x}) = \sum_{\mathbf{y}} \pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x})\exp\big(\frac{1}{\beta}r(\mathbf{x},\mathbf{y})\big) \tag{4.75}$$

Daher können wir eine Wahrscheinlichkeitsverteilung definieren durch

$$\pi^*(\mathbf{y}|\mathbf{x}) = \frac{\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x})\exp\big(\frac{1}{\beta}r(\mathbf{x},\mathbf{y})\big)}{Z(\mathbf{x})} \tag{4.76}$$

Wir schreiben dann Gl. (4.74) um als

$$\begin{aligned}
\tilde{\theta} &= \arg\min_{\theta} \mathbb{E}_{\mathbf{x}\sim\mathcal{D}}\mathbb{E}_{\mathbf{y}\sim\pi_\theta(\cdot|\mathbf{x})}\Big[\log\pi_\theta(\mathbf{y}|\mathbf{x}) - \log\frac{\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x})\exp\left(\frac{1}{\beta}r(\mathbf{x},\mathbf{y})\right)}{Z(\mathbf{x})} \\
&\qquad\qquad - \log Z(\mathbf{x})\Big] \\
&= \arg\min_{\theta} \mathbb{E}_{\mathbf{x}\sim\mathcal{D}}\mathbb{E}_{\mathbf{y}\sim\pi_\theta(\cdot|\mathbf{x})}\Big[\log\pi_\theta(\mathbf{y}|\mathbf{x}) - \log\pi^*(\mathbf{y}|\mathbf{x}) - \log Z(\mathbf{x})\Big] \\
&= \arg\min_{\theta} \mathbb{E}_{\mathbf{x}\sim\mathcal{D}}\Big[\mathbb{E}_{\mathbf{y}\sim\pi_\theta(\cdot|\mathbf{x})}\Big[\log\pi_\theta(\mathbf{y}|\mathbf{x}) - \log\pi^*(\mathbf{y}|\mathbf{x})\Big] \\
&\qquad\qquad -\mathbb{E}_{\mathbf{y}\sim\pi_\theta(\cdot|\mathbf{x})}\big[\log Z(\mathbf{x})\big]\Big] \\
&= \arg\min_{\theta} \mathbb{E}_{\mathbf{x}\sim\mathcal{D}}\Big[\underbrace{\mathrm{KL}\big(\pi_\theta(\cdot|\mathbf{x}) \;||\; \pi^*(\cdot|\mathbf{x})\big)}_{\text{KL-Divergenz}} - \underbrace{\log Z(\mathbf{x})}_{\text{konstant bezüglich }\theta}\Big]
\end{aligned} \tag{4.77}$$

Da $\log Z(\mathbf{x})$ unabhängig von θ ist, beeinflusst es das Ergebnis der $\arg\min_\theta$-Operation nicht und kann aus dem Ziel entfernt werden. Nun erhalten wir ein neues Trainingsziel, das die optimale Policy π_θ findet, indem die KL-Divergenz zwischen $\pi_\theta(\cdot|\mathbf{x})$ und $\pi^*(\cdot|\mathbf{x})$ minimiert wird

$$\tilde{\theta} = \arg\min_{\theta} \mathbb{E}_{\mathbf{x}\sim\mathcal{D}}\Big[\mathrm{KL}\big(\pi_\theta(\cdot|\mathbf{x}) \;||\; \pi^*(\cdot|\mathbf{x})\big)\Big] \tag{4.78}$$

Die Lösung dieses Optimierungsproblems ist offensichtlich gegeben durch

$$\begin{aligned}
\pi_\theta(\mathbf{y}|\mathbf{x}) &= \pi^*(\mathbf{y}|\mathbf{x}) \\
&= \frac{\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x})\exp\left(\frac{1}{\beta}r(\mathbf{x},\mathbf{y})\right)}{Z(\mathbf{x})}
\end{aligned} \tag{4.79}$$

Mit dieser Gleichung können wir die Belohnung $r(\mathbf{x},\mathbf{y})$ unter Verwendung des Zielmodells $\pi_\theta(\mathbf{y}|\mathbf{x})$, des Referenzmodells $\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x})$ und des Normalisierungsfaktors $Z(\mathbf{x})$ ausdrücken:

$$r(\mathbf{x},\mathbf{y}) = \beta\left(\log\frac{\pi_\theta(\mathbf{y}|\mathbf{x})}{\pi_{\theta_{\text{ref}}}(\mathbf{y}|\mathbf{x})} + \log Z(\mathbf{x})\right) \tag{4.80}$$

Dies ist interessant, da wir ursprünglich versuchen, die Policy $\pi_\theta(\cdot)$ mithilfe des Belohnungsmodells $r(\mathbf{x},\mathbf{y})$ zu lernen, aber letztendlich eine Darstellung des Belohnungsmodells basierend auf der Policy erhalten. Gegeben das in Gl. (4.80) definierte Belohnungsmodell,

können wir es auf das Bradley-Terry-Modell anwenden, um die Präferenzwahrscheinlichkeit zu berechnen (siehe auch Abschnitt 4.3.2):

$$\begin{aligned}\Pr_\theta(\mathbf{y}_a \succ \mathbf{y}_b|\mathbf{x}) &= \text{Sigmoid}(r(\mathbf{x},\mathbf{y}_a) - r(\mathbf{x},\mathbf{y}_b)) \\ &= \text{Sigmoid}\Big(\beta\Big(\log\frac{\pi_\theta(\mathbf{y}_a|\mathbf{x})}{\pi_{\theta_{\text{ref}}}(\mathbf{y}_a|\mathbf{x})} + \log Z(\mathbf{x})\Big) - \\ &\qquad \beta\Big(\log\frac{\pi_\theta(\mathbf{y}_b|\mathbf{x})}{\pi_{\theta_{\text{ref}}}(\mathbf{y}_b|\mathbf{x})} + \log Z(\mathbf{x})\Big)\Big) \\ &= \text{Sigmoid}\Big(\beta\log\frac{\pi_\theta(\mathbf{y}_a|\mathbf{x})}{\pi_{\theta_{\text{ref}}}(\mathbf{y}_a|\mathbf{x})} - \beta\log\frac{\pi_\theta(\mathbf{y}_b|\mathbf{x})}{\pi_{\theta_{\text{ref}}}(\mathbf{y}_b|\mathbf{x})}\Big) \end{aligned} \tag{4.81}$$

Diese Formel ist elegant, da sie die Differenz der Belohnungen in die Differenz von Verhältnis-Funktionen umwandelt und wir den Wert von $Z(\mathbf{x})$ nicht berechnen müssen. Ein direktes Ergebnis ist, dass wir kein Belohnungsmodell mehr benötigen, sondern nur die Ziel-Policy und das Referenzmodell, um die Wahrscheinlichkeit der Präferenzen zu berechnen. Schließlich können wir die Ziel-Policy trainieren, indem wir die folgende DPO-Verlustfunktion minimieren

$$\mathcal{L}_{\text{dpo}}(\theta) = -\mathbb{E}_{(\mathbf{x},\mathbf{y}_a,\mathbf{y}_b)\sim\mathcal{D}_r}\big[\log\Pr_\theta(\mathbf{y}_a \succ \mathbf{y}_b|\mathbf{x})\big] \tag{4.82}$$

Die Form dieser Verlustfunktion ist derjenigen sehr ähnlich, die beim Training von Belohnungsmodellen in RLHF verwendet wird (siehe Gl. (4.36)). Es sollte jedoch beachtet werden, dass die Verlustfunktion hier von den Parametern der Policy (d.h. θ) abhängt und nicht von den Parametern des Belohnungsmodells (d.h. ϕ).

Der Hauptvorteil von DPO liegt in seiner Einfachheit und Effizienz. Das DPO-Ziel ist sehr unkompliziert –es optimiert direkt für präferenzbasiertes Feedback, anstatt sich auf separat entwickelte Belohnungsmodelle zu verlassen. Darüber hinaus ist DPO im Allgemeinen dateneffizienter, da es aus einem festen Datensatz lernt, ohne den rechenintensiven Abtastprozess zu benötigen, der bei PPO verwendet wird. Dies macht DPO zu einer beliebten Methode zur Anpassung an menschliche Präferenzen, insbesondere wenn die Entwicklung und Anwendung von Belohnungsmodellen mittels Reinforcement Learning eine Herausforderung darstellt.

DPO kann im Großen und Ganzen als eine Methode des Offline-Reinforcement-Learning betrachtet werden, bei der die Trainingsdaten vorab gesammelt und fest sind und keine Exploration stattfindet. Im Gegensatz dazu haben Online-Reinforcement-Learning-Methoden wie PPO, die das Erkunden neuer Zustände durch Interaktion mit der Umgebung erfordern (wobei das Belohnungsmodell als Proxy dient), ebenfalls ihre einzigartigen Vorteile. Einer der Vorteile des Online-Reinforcement-Learning besteht darin, dass es dem Agenten ermöglicht, sich kontinuierlich an Veränderungen in der Umgebung anzupassen, indem er aus Echtzeit-Feedback lernt. Das bedeutet, dass Online-Methoden im Gegensatz zu Offline-Methoden nicht durch die statische Natur vorab gesammelter Daten eingeschränkt sind und neue Problemlösungsstrategien entdecken können. Darüber hinaus kann die Exploration dem Agenten helfen, eine breitere Palette von Zustands-Aktions-Paaren abzudecken und so die Generalisierung zu verbessern. Dies könnte ein wichtiger Vorteil für LLMs sein, da die Generalisierung als ein kritischer Aspekt bei der Anwendung solch großer Modelle angesehen wird.

4.4.3 Automatische Generierung von Präferenzdaten

Obwohl das Lernen aus menschlichen Präferenzen eine effektive und beliebte Methode zur Ausrichtung von LLMs ist, ist die Annotation von Präferenzdaten kostspielig. Die Verwendung von menschlichem Feedback steht nicht nur vor dem Problem der begrenzten Skalierbarkeit, sondern kann auch zu Bias führen, da menschliches Feedback von Natur aus subjektiv ist. Daher kann man sich KI-Feedback-Methoden zuwenden, um diese Skalierbarkeits- und Konsistenzprobleme ohne die mit menschlichen Annotatoren verbundenen Einschränkungen zu lösen.

Wie bei der Datengenerierung für die Anweisungs-Feinabstimmung ist die Erzeugung von Präferenzdaten mit LLMs unkompliziert. Bei einem Satz von Eingaben verwenden wir zunächst ein LLM, um Paare von Ausgaben zu generieren. Anschließend fordern wir das LLM auf, die Präferenz zwischen jedem Ausgabepaar zusammen mit der entsprechenden Eingabe zu kennzeichnen. Nachfolgend finden Sie ein Beispiel dafür, wie ein LLM aufgefordert wird, ein Präferenzlabel für ein Paar von Kundendienst-Antworten zu generieren.

> Consider a customer service scenario where a customer poses a request. You will review two responses to this request. Please indicate which response is preferred. Note that a good response should be courteous, clear, and concise. It should address the customer's concern directly, provide helpful information or a solution, and maintain a positive tone.
>
> Request:
>
> Hello, I noticed that my order hasn't arrived yet, though it was scheduled to arrive several days ago. Could you please update me on its status? Thank you!
>
> Response A:
>
> I'm very sorry for the delay and understand how disappointing this can be. We're doing our best to sort this out quickly for you.
>
> Response B:
>
> Hey, stuff happens! Your package will get there when it gets there, no need to stress.
>
> Response A is preferred.

Sobald wir solche Präferenzlabels gesammelt haben, können wir sie zusammen mit dem Ausgabepaar und der Eingabe verwenden, um das Belohnungsmodell zu trainieren. Selbstverständlich können wir die Demonstration einiger Beispiele oder die Verwendung fortgeschrittener Prompting-Techniken, wie z.B. CoT, in Betracht ziehen, um die Kennzeichnungsleistung zu verbessern. Zum Beispiel können wir in den Prompt ein Beispiel aufnehmen, das zeigt, wie und warum eine der beiden Antworten auf der Grundlage einer CoT-Begründung bevorzugt wird.

Zusätzlich zu den Präferenzlabels können wir auch die mit jedem Label verbundene Wahrscheinlichkeit erhalten [Lee et al., 2023]. Eine einfache Methode besteht darin, die

Wahrscheinlichkeiten für die Label-Token, wie „A“und „B“, aus den vom LLM ausgegebenen Wahrscheinlichkeiten zu extrahieren. Wir können dann die Softmax-Funktion oder andere Normalisierungstechniken verwenden, um diese Wahrscheinlichkeiten in eine Verteilung über die Labels zu renormalisieren. Diese Wahrscheinlichkeiten der bevorzugten Labels können als punktweise Überwachungssignale für das Training des Belohnungsmodells dienen, wie in Abschnitt 4.4.1 erörtert.

Bei der Datengenerierung ist es, obwohl sie leicht zu skalieren ist, oft notwendig, die Genauigkeit und Vielfalt der Daten sicherzustellen. Hier betreffen die Probleme der Datenqualität und -vielfalt nicht nur die Kennzeichnung von Präferenzen, sondern auch die Ein- und Ausgaben des Modells. Daher müssen wir oft eine Vielzahl von Techniken anwenden, um umfangreiche, qualitativ hochwertige Daten zu erhalten. Zum Beispiel kann man vielfältige Modellausgaben und Annotationen generieren, indem man verschiedene LLMs, Prompts, In-Kontext-Demonstrationen usw. verwendet [Cui et al., 2024]. Dubois et al. [2024] berichten, dass die Variabilität in paarweisen Präferenzdaten wichtig für das Training von LLMs ist, sei es durch menschliches oder KI-Feedback.

Während das Lernen aus KI-Feedback hoch skalierbar und im Allgemeinen objektiv ist, eignet sich diese Methode besser für gut definierte Aufgaben, bei denen objektive Leistungsmetriken verfügbar sind. Im Gegensatz dazu ist das Lernen aus menschlichem Feedback vorteilhafter, wenn es darum geht, KI-Systeme an menschliche Werte, Präferenzen und komplexe reale Aufgaben anzupassen, die das Verständnis eines subtilen oder subjektiven Kontexts erfordern. Diese Methoden können kombiniert werden, um LLMs zu trainieren, die sowohl von menschlichen Erkenntnissen als auch von der Skalierbarkeit des KI-Feedbacks profitieren.

4.4.4 Schrittweise Anpassung

Bisher konzentrierte sich unsere Diskussion über die Anpassung hauptsächlich auf die Verwendung von Belohnungsmodellen zur Bewertung ganzer Eingabe-Ausgabe-Sequenzpaare. Diese Methoden können leicht an Szenarien angepasst werden, in denen die Korrektheit einer Ausgabe durch Überprüfung, ob das gewünschte Ergebnis enthalten ist, untersucht werden kann. Zum Beispiel kann bei der Aufgabe, einen mathematischen Ausdruck zu berechnen, ein Belohnungsmodell positives Feedback geben, wenn die Antwort richtig ist, und negatives Feedback, wenn die Antwort falsch ist. Bei vielen Problemen, die komplexes Schlussfolgern erfordern, reicht die alleinige Überprüfung der Korrektheit des Endergebnisses jedoch nicht zum Lernen aus. Stellen Sie sich einen Schüler vor, dem nur die endgültige Antwort auf eine anspruchsvolle Matheaufgabe gegeben wird. Zu wissen, ob die endgültige Antwort richtig oder falsch ist, hilft dem Schüler nicht herauszufinden, wo er einen Fehler gemacht hat und wie er die richtige Antwort berechnen kann. Ein besserer Ansatz wäre, den Schüler mit einer schrittweisen Aufschlüsselung des Problemlösungsprozesses anzuleiten und das Verständnis der zugrunde liegenden Konzepte und der Logik hinter diesen Schritten zu fördern.

In Kapitel 3haben wir CoT-Methoden untersucht, um LLMs dazu zu veranlassen, Zwischenschritte oder den Denkprozess, der zum Erreichen einer Schlussfolgerung oder zur Lösung eines Problems erforderlich ist, explizit aufzuschreiben. Wir haben gesehen, dass

die Zerlegung eines Problems in kleinere Teile das Verständnis des Lösungsweges erleichtern und die Genauigkeit der Ausgabe erhöhen kann. Diese Methoden können natürlich auf die Anpassung von LLMs erweitert werden, das heißt, wir überwachen das Modell während der Zwischenschritte des Schlussfolgerns. Betrachten wir eine Schlussfolgerungsaufgabe, bei der ein LLM eine Sequenz von Schlussfolgerungsschritten $\mathbf{y} = \{\bar{\mathbf{y}}_1, ..., \bar{\mathbf{y}}_{n_s}\}$ für die gegebene Eingabe erzeugt. Es wird angenommen, dass das Ergebnis des Schlussfolgerns im letzten Schritt $\bar{\mathbf{y}}_{n_s}$ enthalten ist und leicht überprüft werden kann. Für dieses Schlussfolgerungsproblem kategorisieren Uesato et al. [2022] die Feinabstimmungsansätze für LLMs in zwei Klassen:

- Ergebnisbasierte Ansätze. Eine Überwachung findet nur statt, wenn das Endergebnis verifiziert wird. Dies ist eine Standardmethode für das Lernen aus menschlichem Feedback, die wir in diesem Kapitel besprochen haben. Zum Beispiel wird das LLM optimiert, um eine Form der Belohnung $r(\mathbf{x}, \mathbf{y})$ zu maximieren.

- Prozessbasierte Ansätze. Die Überwachung ist in alle Zwischenschritte zusätzlich zum letzten Schritt involviert. Dazu müssen wir ein Modell entwickeln, das bei jedem Schritt ein Überwachungssignal liefert, und Verlustfunktionen entwickeln, die solche Überwachungssignale nutzen können.

Abbildung 4.11 zeigt zwei LLM-Ausgaben für ein Beispiel-Matheproblem. Obwohl das LLM in beiden Fällen die korrekte Endantwort gibt, macht es in der zweiten Ausgabe Fehler während des Problemlösungsprozesses. Ergebnisbasierte Ansätze übersehen diese Fehler und geben positives Feedback für die gesamte Lösung. Im Gegensatz dazu können prozessbasierte Ansätze diese Fehler berücksichtigen und zusätzliche Anleitung zu den detaillierten Schlussfolgerungsschritten geben.

Ein wichtiges Problem bei prozessbasierten Ansätzen ist, dass wir auf einem (potenziell) langen Schlussfolgerungspfad Feedback auf Schrittebene benötigen. Wir können Schlussfolgerungspfade, die Problemen aus bestehenden Datensätzen entsprechen, sammeln oder generieren. Menschliche Experten annotieren dann jeden Schritt in diesen Pfaden auf Korrektheit. Diese Annotationen können verwendet werden, um LLMs direkt zu trainieren oder als Belohnungen im Belohnungsmodellieren. In der Praxis werden jedoch oft reichhaltigere Annotationen eingeführt [Lightman et al., 2024]. Zusätzlich zu den Labels korrekt und inkorrekt kann ein Schritt auch als neutral gekennzeichnet werden, um anzuzeigen, dass der Schritt zwar technisch korrekt sein mag, aber im gesamten Schlussfolgerungsprozess dennoch problematisch sein könnte. Des Weiteren können zur Verbesserung der Effizienz der Datenannotation Techniken wie aktives Lernen eingesetzt werden. Das Identifizieren offensichtlicher Fehler trägt in der Regel nicht wesentlich zum Lernen aus Schlussfolgerungsfehlern bei. Stattdessen ist es oft effektiver, Schritte zu annotieren, die das Modell selbstbewusst für korrekt hält, aber tatsächlich problematisch sind.

Gegeben eine Menge von auf Schrittebene annotierten Schlussfolgerungspfaden und entsprechenden Eingaben, können wir ein Belohnungsmodell trainieren, um Feedback für die Überwachung des Policy-Lernens zu liefern. Das Belohnungsmodell kann als Klassifikationsmodell behandelt werden, und seine Architektur kann daher ein Transformer-Decoder mit einer darüber gestapelten Softmax-Schicht sein. Bei Schritt k nimmt das Belohnungsmodell sowohl die Problembeschreibung (bezeichnet mit $\mathbf{x}$) als auch die bisher generierten

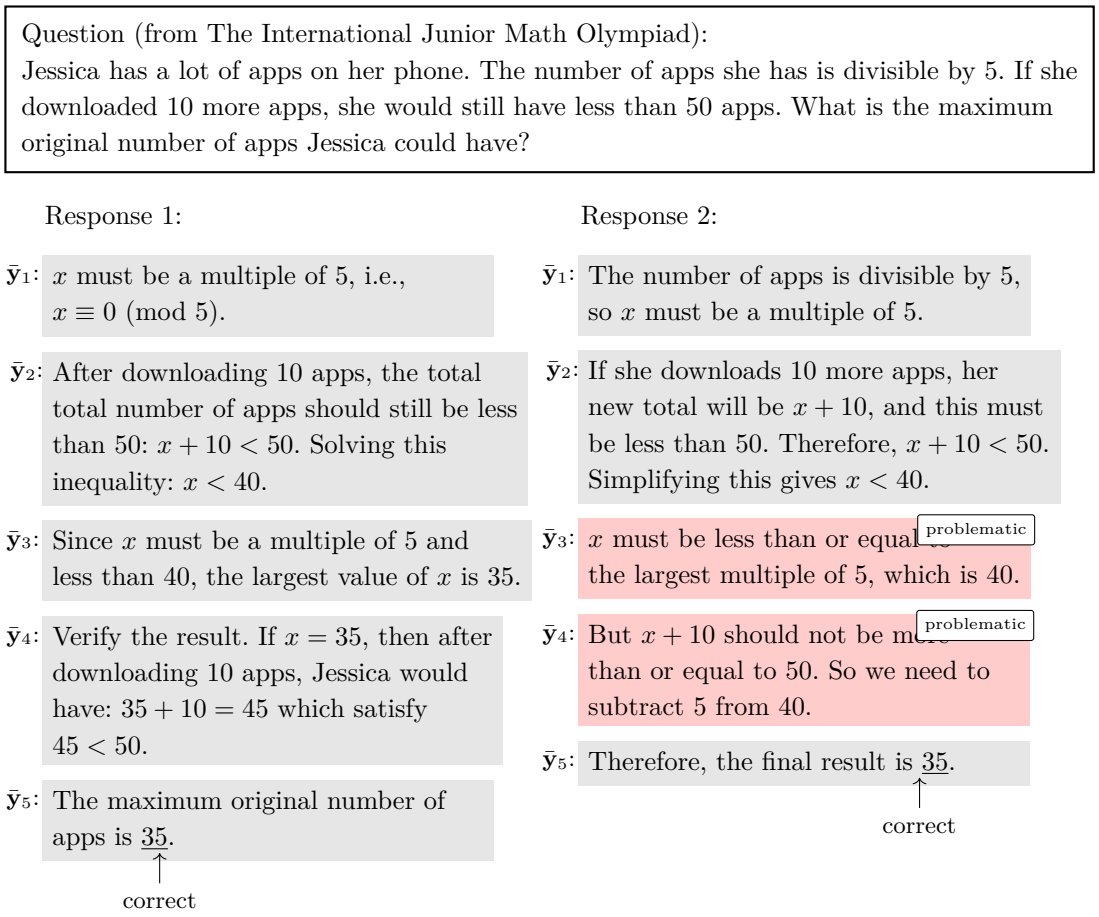


Abbildung 4.11: Zwei LLM-Antworten auf ein mathematisches Problem. In Antwort 1 sind sowohl das Endergebnis als auch alle Begründungsschritte korrekt. In Antwort 2 ist das Endergebnis korrekt, aber es gibt Fehler im Begründungsprozess (rot hervorgehoben). Bei ergebnisbasierten Ansätzen werden beide Antworten als korrekt angesehen. Bei prozessbasierten Ansätzen können die Fehler in Antwort 2 bei der Belohnungsmodellierung berücksichtigt werden.

Schlussfolgerungsschritte (bezeichnet mit $\bar{\mathbf{y}}_{\leq k}$) als Eingabe und gibt eine Wahrscheinlichkeitsverteilung über die Label-Menge {korrekt, inkorrekt} oder {korrekt, inkorrekt, neutral} aus. Das gelernte Belohnungsmodell wird dann verwendet, um Schlussfolgerungspfade zu bewerten, indem die Korrektheit jedes Schrittes beurteilt wird. Eine einfache Methode zur Modellierung der Korrektheit besteht darin, die Anzahl der als korrekt klassifizierten Schritte zu zählen, gegeben durch

$$r(\mathbf{x}, \mathbf{y}) = \sum_{k=1}^{n_s} \delta(\text{korrekt}, C(\mathbf{x}, \bar{\mathbf{y}}_{\leq k})) \tag{4.83}$$

wobei $C(\mathbf{x}, \bar{\mathbf{y}}_{\leq k})$ das Label mit der maximalen Wahrscheinlichkeit bezeichnet. Wir können auch Log-Wahrscheinlichkeiten der Klassifikation verwenden, um die Belohnung des gesamten Pfades zu definieren

$$r(\mathbf{x}, \mathbf{y}) = \sum_{k=1}^{n_s} \log \Pr(\text{korrekt}|\mathbf{x}, \bar{\mathbf{y}}_{\leq k}) \tag{4.84}$$

wobei $\Pr(\text{korrekt}|\mathbf{x}, \bar{\mathbf{y}}_{\leq k})$ die Wahrscheinlichkeit des vom Belohnungsmodell generierten Labels korrekt bezeichnet. Der Belohnungswert $r(\mathbf{x}, \mathbf{y})$ kann dann wie üblich verwendet werden, um die Policy im RLHF zu trainieren.

Obwohl wir unsere Diskussion auf mathematische Probleme beschränken, sind die hier beschriebenen Ansätze allgemein und können auf eine Vielzahl von Aufgaben angewendet werden, die mehrstufiges Schlussfolgern und Entscheiden beinhalten. Darüber hinaus können wir bei der Bewertung der Qualität eines Schrittes verschiedene Aspekte berücksichtigen, nicht nur seine Korrektheit. In Dialogsystemen müssen Antworten beispielsweise nicht nur genau, sondern auch über mehrere Gesprächsrunden hinweg kontextuell angemessen sein. Wenn ein Modell eine korrekte Antwort gibt, aber die Kohärenz im Kontext des laufenden Dialogs nicht aufrechterhält, könnte Feedback auf Schrittebene dem Modell helfen, solche Diskrepanzen zu erkennen und zu korrigieren. Beachten Sie auch, dass die prozessbasierten Ansätze mit den in Abschnitt 4.4.1.3 diskutierten feinkörnigen Belohnungsmodellierungsansätzen verwandt sind. Alle diese Ansätze zielen im Wesentlichen darauf ab, LLMs eine detailliertere Überwachung zu ermöglichen, indem ihre Ausgaben in kleinere, leichter zu handhabende Schritte unterteilt werden. Prozessbasiertes Feedback konzentriert sich jedoch mehr auf die Bewertung der Korrektheit eines Schrittes basierend auf seinen vorangehenden Schritten, während die Ansätze in Abschnitt 4.4.1.3 die unabhängige Bewertung jedes Schrittes betonen.

Die Idee, LLMs schrittweise anzupassen, hat großes Anwendungspotenzial, insbesondere angesichts der jüngsten Verlagerung hin zu komplexeren Schlussfolgerungsaufgaben bei der Nutzung von LLMs. Zum Beispiel sind sowohl die GPT-o1- als auch die GPT-o3-Modelle mit fortschrittlicheren Schlussfolgerungstechniken (wie langem internem CoT) ausgestattet, um anspruchsvolle Probleme wie wissenschaftliches und mathematisches Schlussfolgern zu lösen [OpenAI, 2024]. Diese Aufgaben beruhen oft auf langen und komplexen Schlussfolgerungspfaden, und daher scheint es unerlässlich, detaillierte Überwachungssignale in den Schlussfolgerungsprozess einzuführen. Darüber hinaus verbessert eine effektive Überwachung langer Schlussfolgerungspfade aus praktischer Sicht nicht nur die Schlussfolgerungsleistung, sondern hilft dem Modell auch, redundante oder unnötige Schlussfolgerungsschritte zu eliminieren, wodurch die Komplexität der Schlussfolgerung reduziert und die Effizienz verbessert wird.

4.4.5 Anpassung zur Inferenzzeit

In diesem Abschnitt haben wir eine Vielzahl von Methoden untersucht, um Modelle an menschliche Präferenzen und Annotationen anzupassen. Eine der wesentlichen Einschränkungen vieler dieser Methoden ist jedoch, dass LLMs feinabgestimmt werden müssen. Für RLHF und seine Varianten kann das Training von LLMs mit Belohnungsmodellen rechenintensiv und instabil sein, was zu erhöhter Komplexität und Kosten bei der Anwendung dieser Ansätze führt. In diesem Fall können wir eine Anpassung der Modelle zur Inferenzzeit in Betracht ziehen und so die damit verbundene zusätzliche Komplexität und den Aufwand vermeiden.

Eine einfache Möglichkeit, eine Anpassung zur Inferenzzeit zu erreichen, besteht darin, das Belohnungsmodell zu verwenden, um die beste von N alternativen Ausgaben auszuwählen, die vom LLM generiert wurden. Diese Methode ist als Best-of-N sampling (BoN

sampling) bekannt. Wir können BoN-Sampling als eine Form des Rerankings betrachten. Tatsächlich werden Reranking-Methoden seit langem in NLP-Aufgaben wie der maschinellen Übersetzung weit verbreitet eingesetzt. Sie werden typischerweise in Situationen angewendet, in denen das Training komplexer Modelle kostspielig ist. In solchen Fällen ermöglicht das direkte Reranking der Ausgaben die Einbeziehung dieser komplexen Modelle zu sehr geringen Kosten[8].

Im BoN-Sampling-Prozess nimmt das LLM die Eingabesequenz $\mathbf{x}$ und generiert N verschiedene Ausgabesequenzen $\{\hat{\mathbf{y}}_1, ..., \hat{\mathbf{y}}_N\}$:

$$\{\hat{\mathbf{y}}_1, ..., \hat{\mathbf{y}}_N\} = \underset{\mathbf{y}}{\text{argTopN}} \left[\Pr(\mathbf{y}|\mathbf{x})\right] \tag{4.85}$$

wobei die Operation argTopN die Top-N-Ausgaben zurückgibt, die die Funktion $\Pr(\mathbf{y}|\mathbf{x})$ maximieren. Diese Ausgaben können auf verschiedene Weisen generiert werden, abhängig vom vom Modell verwendeten Suchalgorithmus (z. B. Sampling oder Beam-Search). Sobald die N-besten Ausgabekandidaten generiert sind, wird das Belohnungsmodell verwendet, um den besten zu bewerten und auszuwählen:

$$\hat{\mathbf{y}}_{\text{best}} = \max\{r(\mathbf{x}, \hat{\mathbf{y}}_1), ..., r(\mathbf{x}, \hat{\mathbf{y}}_N)\} \tag{4.86}$$

Es ist erwähnenswert, dass das Ergebnis des BoN-Samplings auch von der Diversität der N-Besten-Liste beeinflusst wird. Dies ist ein häufiges Problem bei den meisten Reranking-Methoden. Typischerweise wünschen wir uns, dass die N-besten Ausgabekandidaten eine relativ hohe Qualität aufweisen, sich aber ausreichend voneinander unterscheiden. In vielen Textgenerierungssystemen sind die N-besten Ausgaben sehr ähnlich und unterscheiden sich oft nur durch ein oder zwei Wörter. Das Diversitätsproblem ist bei LLMs noch herausfordernder, da die von einem LLM generierten N-besten Ausgaben sich in ihrer Formulierung unterscheiden können, ihre semantischen Bedeutungen jedoch oft recht ähnlich sind. In der Praxis kann man die Hyperparameter des Modells anpassen und/oder verschiedene LLMs einsetzen, um vielfältigere Ausgabekandidaten für das Reranking zu generieren. Dennoch müssen wir, wie bei vielen praktischen Systemen, einen Kompromiss zwischen der Auswahl qualitativ hochwertiger Kandidaten und der Gewährleistung einer ausreichenden Variation in den generierten Ausgaben finden.

BoN-Sampling kann auch für das Training von LLMs verwendet werden. Eine eng verwandte Methode ist das rejection sampling. Bei dieser Methode wählen wir zuerst die „besten“Ausgaben aus den N-Besten-Listen mithilfe des Belohnungsmodells aus und verwenden dann diese ausgewählten Ausgaben, um das LLM feinabzustimmen. Auf diese Weise können wir menschliche Präferenzen in das Training von LLMs über einen im Vergleich

[8] Reranking-Methoden können uns auch helfen, sogenannte Modellfehler und Suchfehler zu untersuchen, obwohl diese Probleme im Kontext von LLMs nicht oft diskutiert werden. Angenommen, wir haben ein altes Modell und ein neues, leistungsfähigeres Modell. Wir können das neue Modell verwenden, um die beste Ausgabe aus der N-Besten-Liste des alten Modells als Orakel-Ausgabe auszuwählen. Der Leistungsunterschied zwischen der Orakel-Ausgabe und der Top-1-Ausgabe der ursprünglichen N-Besten-Liste spiegelt den Leistungszuwachs wider, der durch das neue Modell erzielt wird. Wenn der Leistungszuwachs signifikant ist, können wir sagen, dass das alte Modell mehr Modellfehler aufweist. Wenn der Zuwachs gering ist, kann dies darauf hindeuten, dass das Problem in Suchfehlern liegt, da die besten Kandidaten nicht gefunden wurden.

zu RLHF wesentlich einfacheren Ansatz einbringen. Viele LLMs haben das Rejection-Sampling für die Feinabstimmung übernommen [Nakano et al., 2021; Touvron et al., 2023b].

4.5 Zusammenfassung

In diesem Kapitel haben wir eine Reihe von Techniken zur Ausrichtung von LLMs untersucht. Insbesondere haben wir Feinabstimmungsmethoden erörtert, die es LLMs ermöglichen, Anweisungen zu befolgen und sie an menschliche Präferenzen anzupassen. Einer der Vorteile der Feinabstimmung von LLMs ist die Recheneffizienz. Im Gegensatz zum Vortraining, das auf der Optimierung großer neuronaler Netze basiert, ist die Feinabstimmung ein nachgelagerter Schritt und daher rechenintensiver. Darüber hinaus ist sie besser geeignet, um Probleme zu lösen, die im Vortraining nicht einfach zu lösen sind, wie z. B. die Ausrichtung an menschlichen Werten. Die weit verbreitete Aufmerksamkeit für das Thema der Ausrichtung hat auch zu einem Anstieg von Forschungsarbeiten zu diesem Thema geführt, was beim Verfassen dieses Kapitels eine Herausforderung darstellte, da es schwierig ist, alle neuesten Techniken abzudecken. Wir haben jedoch versucht, eine relativ detaillierte Einführung in die grundlegenden Ansätze zur Ausrichtung zu geben, wie z. B. die Anweisungs-Feinabstimmung und RLHF.

Obwohl wir uns in diesem Kapitel auf Techniken zur Ausrichtung von LLMs konzentriert haben, ist der Begriff KI-Ausrichtung ein weitreichendes Konzept. Er bezieht sich im Allgemeinen auf den Prozess, sicherzustellen, dass das Verhalten eines KI-Systems mit menschlichen Werten, Zielen und Erwartungen übereinstimmt. Die Idee der KI-Ausrichtung lässt sich bis in die Anfänge der KI zurückverfolgen. Eine viel zitierte Beschreibung der KI-Ausrichtung stammt aus einem Artikel des Mathematikers und Informatikers Norbert Wiener [Wiener, 1960]. Das Zitat lautet wie folgt

> Wenn wir, um unsere Zwecke zu erreichen, ein mechanisches Mittel verwenden, in dessen Betrieb wir nicht effizient eingreifen können ... sollten wir uns besser ganz sicher sein, dass der in die Maschine eingegebene Zweck der Zweck ist, den wir wirklich wünschen.

Damals war die KI-Ausrichtung für Forscher ein fernes Anliegen. Heute jedoch beeinflusst sie maßgeblich das Design verschiedener KI-Systeme. In der Robotik beispielsweise ist die Ausrichtung entscheidend, um sicherzustellen, dass autonome Roboter sicher mit Menschen und ihrer Umgebung interagieren. Beim autonomen Fahren müssen Autos nicht nur die Verkehrsregeln befolgen, sondern auch komplexe Echtzeitentscheidungen treffen, die die menschliche Sicherheit priorisieren, Unfälle vermeiden und ethische Dilemmata bewältigen.

In der aktuellen KI-Forschung wird die Ausrichtung in der Regel durch die Entwicklung eines Ersatzziels erreicht, das dem eigentlichen Ziel analog ist, und indem das KI-System auf dieses Ziel ausgerichtet wird. Die Gestaltung des Ziels der KI-Ausrichtung ist jedoch sehr schwierig. Ein Grund dafür ist, dass menschliche Werte vielfältig und oft kontextabhängig sind, was es schwierig macht, sie in einer einzigen, universell anwendbaren Zielfunktion zusammenzufassen. Auch die Komplexität realer Umgebungen, in denen

Werte und Ziele oft in Konflikt stehen oder sich im Laufe der Zeit entwickeln, erschwert die Ausrichtungsbemühungen zusätzlich. Selbst wenn wir ein angemessenes Ziel definieren könnten, könnten KI-Systeme unbeabsichtigte Wege finden, es zu erreichen, was zu „fehlausgerichteten“Ergebnissen führt, die das Ziel zwar technisch erfüllen, aber auf schädliche oder kontraproduktive Weise.

Diese Herausforderungen haben die KI-Forschung motiviert und motivieren sie weiterhin, auf besser ausgerichtete Systeme hinzuarbeiten, sei es durch die Entwicklung neuer Mechanismen zur Wahrnehmung der Welt oder durch effizientere und generalisierbarere Methoden zur Anpassung dieser Systeme an gegebene Aufgaben. Noch wichtiger ist, dass mit der zunehmenden Leistungsfähigkeit und Intelligenz von KI-Systemen, insbesondere angesichts der Tatsache, dass jüngste Fortschritte bei LLMs bemerkenswerte Fähigkeiten im Umgang mit vielen anspruchsvollen Problemen gezeigt haben, die Notwendigkeit der KI-Ausrichtung dringlicher geworden ist. Forscher haben begonnen, sich mit der KI-Sicherheit zu befassen und die Gemeinschaft zu warnen, dass sie KI-Systeme mit großer Vorsicht entwickeln und veröffentlichen müssen, um zu verhindern, dass diese Systeme fehlausgerichtet werden [Russell, 2019; Bengio et al., 2024].

Kapitel 5

Inferenz

Sobald wir ein LLM vortrainiert und feinabgestimmt haben, können wir es anwenden, um Vorhersagen für neue Daten zu treffen. Dieser Prozess wird als Inferenz bezeichnet, bei dem das LLM die Wahrscheinlichkeiten verschiedener möglicher Ausgaben für eine gegebene Eingabe berechnet und die Ausgabe auswählt, die die Wahrscheinlichkeit maximiert. Das Inferenzproblem wird im Allgemeinen in der folgenden Form ausgedrückt:

$$\hat{\mathbf{y}} = \arg\max_{\mathbf{y}} \Pr(\mathbf{y}|\mathbf{x}) \tag{5.1}$$

wobei $\mathbf{x}$ die Eingabesequenz ist, $\mathbf{y}$ eine mögliche Ausgabesequenz und $\hat{\mathbf{y}}$ die beste Ausgabesequenz ist.

Dies ist vielleicht eine der am weitesten verbreiteten Formeln im NLP und geht auf die Anfänge der Spracherkennung und der maschinellen Übersetzungssysteme zurück, die auf probabilistischen Modellen basieren. Obwohl das Lösen dieses Optimierungsproblems für einige Anwendungen, wie die Vorhersage eines Tokens mit einem sehr kleinen Sprachmodell, trivial erscheint, ergeben sich in den meisten Situationen die rechnerischen Herausforderungen sowohl aus der Berechnung von $\Pr(\mathbf{y}|\mathbf{x})$ als auch aus der Durchführung der $\arg\max$-Operation. Die Probleme, die wir daher in diesem Kapitel behandeln möchten, umfassen: 1) die effiziente Berechnung der Vorhersagewahrscheinlichkeit bei einem trainierten LLM und 2) die Entwicklung einer effizienten (suboptimalen) Suche nach $\hat{\mathbf{y}}$.

Auf einer hohen Ebene handelt es sich hierbei um grundlegende Probleme der künstlichen Intelligenz, die ausgiebig untersucht wurden. Daher können viele etablierte Techniken direkt angewendet werden, zum Beispiel kann man gierige Suchalgorithmen verwenden, um ein effizientes Inferenzsystem zu implementieren. Andererseits können modellspezifische Optimierungen, wie z. B. effiziente Aufmerksamkeitsmodelle für Transformer, in Betracht gezogen werden, um die Effizienz weiter zu verbessern. In vielen praktischen Anwendungen müssen wir jedoch immer noch einen Kompromiss zwischen Genauigkeit und Effizienz eingehen, indem wir verschiedene Techniken sorgfältig kombinieren.

Die Bedeutung des Inferenzproblems bei LLMs liegt auch darin, dass viele Anwendungsszenarien die Verarbeitung extrem langer Sequenzen erfordern. Jüngste Studien haben ergeben, dass die Injektion zusätzlicher Prompts und kontextueller Informationen, wie z. B. langer Chain-of-Thought-Prompts, während der Inferenz die Leistung von LLMs erheblich verbessern kann. Dies bietet einen neuen Ansatz zur Skalierung von LLMs: Bessere Ergebnisse können durch Erhöhung des Rechenaufwands zur Inferenzzeit erzielt werden. Zum Beispiel haben die Systeme o1 von OpenAI [2024] und R1 von Deepseek [2025] durch Skalierung zur Inferenzzeit beeindruckende Leistungen bei komplexen Schlussfolgerungs- und Programmieraufgaben gezeigt. Dies wiederum hat das NLP-Feld ermutigt, sich stärker auf das Thema der effizienten Inferenz zu konzentrieren.

In diesem Kapitel werden wir grundlegende Konzepte und Algorithmen der LLM-Inferenz vorstellen, einschließlich Prefilling-Decoding-Frameworks, Such- (Dekodierungs-) Algorithmen und Evaluierungsmetriken für die Inferenzleistung. Anschließend werden wir

Methoden zur Verbesserung der Effizienz der LLM-Inferenz vorstellen, die eine Reihe von Techniken zur Beschleunigung des Systems und zur Komprimierung des Modells abdecken. Schließlich werden wir die Skalierung zur Inferenzzeit erörtern, die als eine wichtige Anwendung der Inferenzoptimierung gilt.

5.1 Prefilling und Dekodierung

In diesem Abschnitt stellen wir das Prefilling-Decoding-Framework vor, den gängigsten Ansatz zur Interpretation und Implementierung von LLM-Inferenzprozessen. Wir führen zunächst die Notation und das Hintergrundwissen ein und beschreiben dann die Details des Frameworks, wie zum Beispiel die Dekodierungsalgorithmen für die LLM-Inferenz.

5.1.1 Grundlagen

Obwohl wir LLMs in diesem Buch schon oft beschrieben haben, beginnen wir mit einer kurzen Definition der Notation, um die nachfolgende Diskussion zu erleichtern und dieses Kapitel in sich abgeschlossen zu gestalten.

- $\mathbf{x}$: Die Eingabe-Token-Sequenz. Sie ist konzeptionell äquivalent zu einem „Prompt“, der Anweisungen, Benutzereingaben und zusätzlichen Kontext enthält, der als Eingabe für das LLM gedacht ist. $\mathbf{x}$ besteht aus $m+1$ Tokens, bezeichnet als $x_0...x_m$, wobei x_0 das Startsymbol $\langle\text{SOS}\rangle$ ist.
- $\mathbf{y}$: Die Ausgabe-Token-Sequenz, auch als Antwort auf die Eingabe bezeichnet. $\mathbf{y}$ besteht aus n Tokens, bezeichnet als $y_1...y_n$.
- $\mathbf{y}_{<i}$: Die Ausgabe-Tokens, die der Position i vorausgehen, d. h. $\mathbf{y}_{<i} = y_1...y_{i-1}$.
- $\Pr(\mathbf{y}|\mathbf{x})$: Die Wahrscheinlichkeit, $\mathbf{y}$ gegeben $\mathbf{x}$ mit dem LLM zu generieren. Wenn das LLM durch θ parametrisiert ist, können wir es als $\Pr_\theta(\mathbf{y}|\mathbf{x})$ schreiben.
- $[\mathbf{x},\mathbf{y}]$: Die zusammengefügte Token-Sequenz von $\mathbf{x}$ und $\mathbf{y}$. Das heißt, $[\mathbf{x},\mathbf{y}] = x_0...x_m y_1...y_n$. Gelegentlich verwenden wir die Notation $\text{seq}_{\mathbf{x},\mathbf{y}}$, um $[\mathbf{x},\mathbf{y}]$ darzustellen.
- $\Pr([\mathbf{x},\mathbf{y}])$: Die Wahrscheinlichkeit, die Token-Sequenz $[\mathbf{x},\mathbf{y}]$ mit dem LLM zu generieren.

Wie in Gl. (5.1) beschrieben, ist das Ziel der LLM-Inferenz, $\Pr(\mathbf{y}|\mathbf{x})$ zu maximieren. Die Modellierung dieser bedingten Wahrscheinlichkeit ist im NLP üblich. Auf den ersten Blick scheint es sich um ein Sequenz-zu-Sequenz-Problem zu handeln, bei dem wir eine Sequenz mithilfe von Encoder-Decoder-Modellen in eine andere umwandeln. Wir diskutieren hier jedoch keine Sequenz-zu-Sequenz-Probleme oder Encoder-Decoder-Architekturen. Stattdessen kann dieses Modellierungsproblem, wie in früheren Kapiteln erörtert, durch die Verwendung von reinen Decoder-Modellen gelöst werden. Dazu können wir die Wahrscheinlichkeit im logarithmischen Maßstab $\log\Pr(\mathbf{y}|\mathbf{x})$ als die Differenz zwischen $\log\Pr([\mathbf{x},\mathbf{y}])$

und $\log \Pr(\mathbf{x})$ interpretieren

$$\log \Pr(\mathbf{y}|\mathbf{x}) \quad = \quad \log \Pr([\mathbf{x}, \mathbf{y}]) - \log \Pr(\mathbf{x}) \tag{5.2}$$

wobei $\log \Pr([\mathbf{x}, \mathbf{y}])$ und $\log \Pr(\mathbf{x})$ durch Ausführen des LLM auf den Sequenzen $[\mathbf{x}, \mathbf{y}]$ bzw. $\mathbf{x}$ erhalten werden können. Zum Beispiel können wir die Wahrscheinlichkeit der Generierung von $\mathbf{x}$ mit der Kettenregel berechnen

$$\begin{aligned}
\log \Pr(\mathbf{x}) &= \log \Pr(x_0 ... x_m) \\
&= \log \left[\Pr(x_0) \Pr(x_1|x_0) \cdots \Pr(x_m|x_0 ... x_{m-1}) \right] \\
&= \underbrace{\log \Pr(x_0)}_{=0} + \sum_{j=1}^{m} \log \Pr(x_j|\mathbf{x}_{<j}) \\
&= \sum_{j=1}^{m} \log \Pr(x_j|\mathbf{x}_{<j})
\end{aligned} \tag{5.3}$$

Mit anderen Worten, wir berechnen die Log-Wahrscheinlichkeit der Token-Vorhersage an jeder Position von $\mathbf{x}$ und summieren all diese Log-Wahrscheinlichkeiten.

In gängigen Implementierungen von LLMs müssen wir jedoch nicht die Log-Wahrscheinlichkeit der Eingabesequenz berechnen, sondern verwenden das LLM, um die Log-Wahrscheinlichkeit der Ausgabesequenz direkt in der folgenden Form zu berechnen

$$\log \Pr(\mathbf{y}|\mathbf{x}) \quad = \quad \sum_{i=1}^{n} \log \Pr(y_i|\mathbf{x}, \mathbf{y}_{<i}) \tag{5.4}$$

wobei $[\mathbf{x}, \mathbf{y}_{<i}]$ den Kontext für die Vorhersage von y_i darstellt. Wir verwenden $\Pr(y_i|\mathbf{x}, \mathbf{y}_{<i})$, um $\Pr(y_i|[\mathbf{x}, \mathbf{y}_{<i}])$ zu bezeichnen, und folgen damit der in der Literatur üblichen Notation.

Nun haben wir zwei Teilprobleme bei der Behandlung des in Gl. (5.1) beschriebenen Inferenzproblems:

- Modellberechnung: Wir modellieren $\Pr(y_i|\mathbf{x}, \mathbf{y}_{<i})$ und berechnen es auf effiziente Weise.
- Suche: Wir finden die optimale (oder suboptimale) Ausgabesequenz hinsichtlich $\log \Pr(\mathbf{y}|\mathbf{x})$.

Das zweite Teilproblem ist ein klassisches Problem im NLP. Wir werden in Abschnitt 5.1.3 zeigen, dass es mehrere gut untersuchte Algorithmen gibt, die angewendet werden können, um den Raum möglicher Ausgabesequenzen effizient zu durchsuchen. Das erste Teilproblem erfordert, dass ein Sprachmodell eine Verteilung über ein Vokabular V bei einer gegebenen Sequenz von Kontext-Token erzeugt. Dies können wir durch das Training eines Transformer-Decoders erreichen, der die Verteilung ausgibt

$$\Pr(\cdot|\mathbf{x}, \mathbf{y}_{<i}) \quad = \quad \text{Softmax}(\mathbf{H}\mathbf{W}^o)_{m+i} \tag{5.5}$$

$$\mathbf{H} \quad = \quad \text{Dec}([\mathbf{x}, \mathbf{y}_{<i}]) \tag{5.6}$$

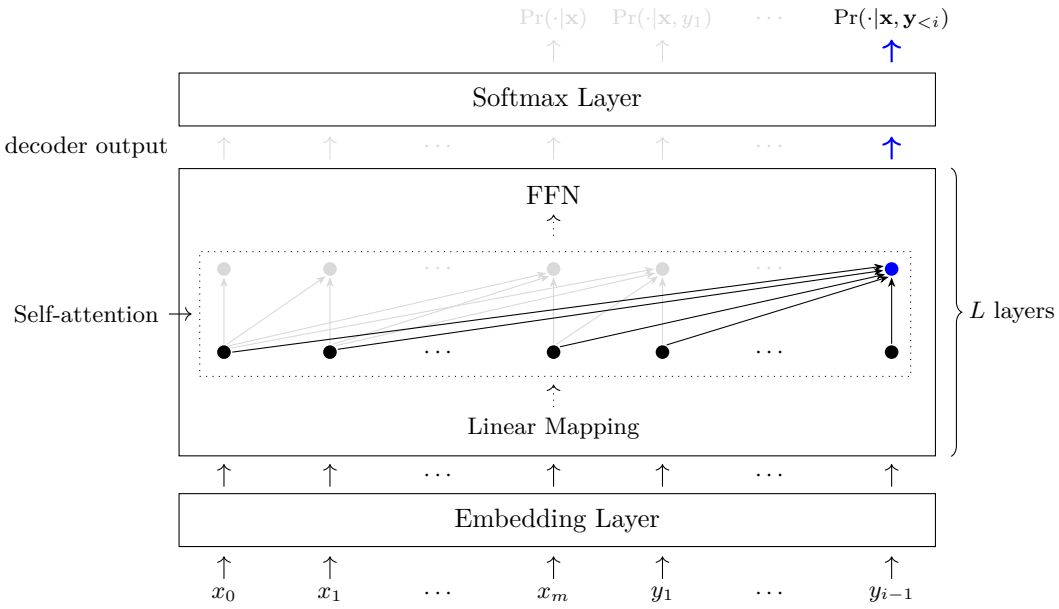


Abbildung 5.1: Die reine Decoder-Architektur für LLMs. Der Decoder besteht aus einer Einbettungsschicht und einem Stapel von Transformer -Schichten. In jeder Transformer -Schicht durchläuft die Eingabe eine lineare Abbildung, ein Self-Attention-Netzwerk und ein FFN. Die Ausgabe des Decoders ist eine Sequenz von Repräsentationen, die als Eingabe an eine Softmax-Schicht übergeben werden, welche eine Verteilung von Tokens für jede Position erzeugt.

Hier erzeugt Dec(·) eine Sequenz von Repräsentationen, von denen jede einer Position der Eingabesequenz entspricht. Wenn wir also $[\mathbf{x}, \mathbf{y}_{<i}]$ in das LLM eingeben, ist $\mathbf{H}$ eine $i' \times d$-Matrix, wobei d die Dimensionalität jeder Repräsentation ist und $i' = m + i$ die Anzahl der Kontext-Token ist. Wir können dann eine Softmax-Schicht verwenden, um diese Repräsentationen in Verteilungen von Token umzuwandeln. $\mathbf{W}^o \in \mathbb{R}^{d \times |V|}$ ist die lineare Abbildungsmatrix der Softmax-Schicht, und $\mathbf{H}\mathbf{W}^o$ transformiert die d-dimensionalen Repräsentationen in $\mathbf{H}$ in die $|V|$-dimensionalen Repräsentationen. Die Verwendung des Subskripts $m + i$ zeigt an, dass die Softmax-Funktion nur auf der Repräsentation an der Position $m + i$ ausgeführt wird. Siehe Abbildung 5.1 für eine Veranschaulichung dieser Architektur.

Dec(·) ist ein Transformer-Decodierungsnetzwerk, das aus einem Einbettungsnetzwerk und einer Anzahl von gestapelten Selbst-Aufmerksamkeits- und FFN-Netzwerken besteht. Wir werden Transformer hier nicht im Detail besprechen, da die Leser diese Modelle leicht aus der Literatur lernen können. Es ist jedoch erwähnenswert, dass die Schwierigkeit der Inferenz teilweise auf die Verwendung des Selbst-Aufmerksamkeitsmechanismus in Transformern zurückzuführen ist. Erinnern Sie sich daran, dass eine allgemeine Form der Single-Head-Selbst-Aufmerksamkeit gegeben ist durch

$$\text{Att}_{\text{qkv}}(\mathbf{q}_{i'}, \mathbf{K}, \mathbf{V}) \quad = \quad \text{Softmax}(\frac{\mathbf{q}_{i'}\mathbf{K}^{\mathrm{T}}}{\sqrt{d}})\mathbf{V} \tag{5.7}$$

wobei $\mathbf{q}_{i'} \in \mathbb{R}^d$ die Query an der Position i' (d.h. die Position von y_i) ist und $\mathbf{K}$ und $\mathbf{V} \in \mathbb{R}^{i' \times d}$ die Keys bzw. Values bis zu i' sind.

Bei jedem Schritt während der Inferenz rufen wir die Selbst-Aufmerksamkeitsfunktion $\text{Att}_{\text{qkv}}(\cdot)$ auf, gefolgt von einem FFN, um eine d-dimensionale Repräsentation zu erzeugen, die Informationen sowohl vom aktuellen Token als auch von seinem linken Kontext integriert. Dieser Prozess wird durch L Schichten von Selbst-Aufmerksamkeit und FFN wiederholt, wodurch ein Stapel von Transformer-Schichten gebildet wird. Die Ausgabe der L-ten Schicht in diesem Stapel ist die endgültige Repräsentation.

Jedes Mal richtet das Modell die Aufmerksamkeit von Position i' auf alle vorhergehenden Positionen, was zu $2i'$ Vektorprodukten führt (i' Mal für $\mathbf{q}_{i'}\mathbf{K}^{\mathrm{T}}$ und i' Mal für das Produkt von Softmax($\frac{\mathbf{q}_{i'}\mathbf{K}^{\mathrm{T}}}{\sqrt{d}}$) und $\mathbf{V}$). Daher hat die Erzeugung einer Sequenz der Länge len eine Zeitkomplexität von $O(L \times len^2)$ für das Selbst-Aufmerksamkeitsnetzwerk. Offensichtlich ist die Inferenz dieses Modells für lange Sequenzen aufgrund seiner quadratischen Zeitkomplexität in Bezug auf die Sequenzlänge langsam. Daher haben sich viele Verbesserungen an Transformern und alternativen Modellen auf effiziente Methoden konzentriert, die schneller als diese quadratische Zeitkomplexität sind, wie z.B. sparse Aufmerksamkeitsmechanismen und lineare Zeitmodelle. Eine detaillierte Diskussion über effiziente Transformer findet sich in den vorherigen Kapiteln, und dieser Abschnitt wird sich auf die Standard-Transformer-Architektur konzentrieren.

Beachten Sie, dass bei der Selbst-Aufmerksamkeit die Queries, Keys und Values einer Schicht lineare Abbildungen derselben Eingabe sind (d.h. der Ausgabe der vorherigen Schicht). Sobald ein neues Key-Value-Paar erzeugt wird, wird es in nachfolgenden Inferenzschritten wiederholt verwendet. Anstatt diese Key-Value-Paare während der Inferenz neu zu generieren, ist es sinnvoller, sie in einer Struktur zu speichern, die als Key-Value-Cache oder KV-Cache bezeichnet wird. Somit kann $(\mathbf{K}, \mathbf{V})$ direkt als KV-Cache betrachtet werden. Dieser Cache wird wie folgt aktualisiert

$$\mathbf{K} = \text{Append}(\mathbf{K}, \mathbf{k}_{i'}) \tag{5.8}$$

$$\mathbf{V} = \text{Append}(\mathbf{V}, \mathbf{v}_{i'}) \tag{5.9}$$

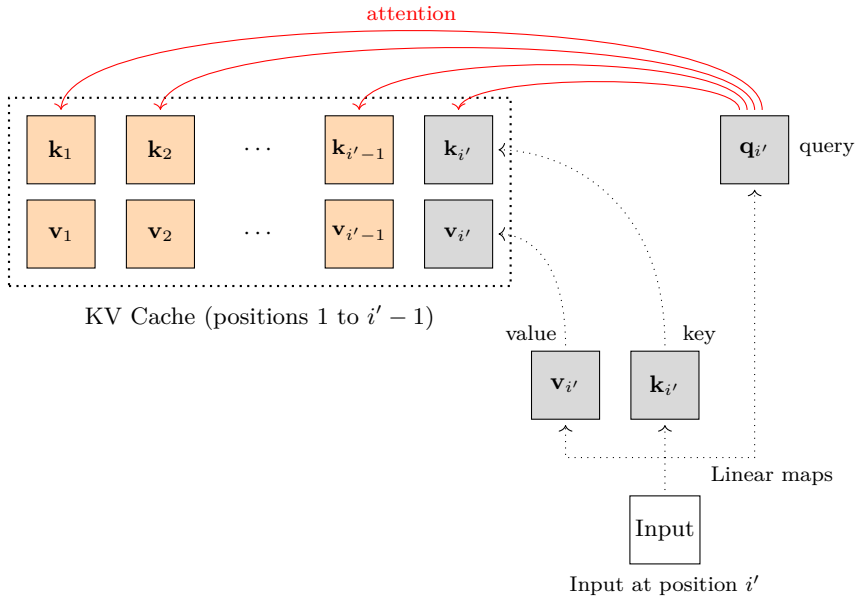
wobei $(\mathbf{k}_{i'}, \mathbf{v}_{i'})$ das neu erzeugte Key-Value-Paar an der Position i' ist und $\text{Append}(\mathbf{a}, \mathbf{b})$ eine Funktion bezeichnet, die einen Zeilenvektor $\mathbf{b}$ an eine Matrix $\mathbf{a}$ anhängt. Abbildung 5.2 zeigt, wie ein Transformer-Decoder mit einem KV-Cache arbeitet.

Schließlich wird der Prozess zur Berechnung von $\log \Pr(\mathbf{y}|\mathbf{x})$ wie folgt zusammengefasst:

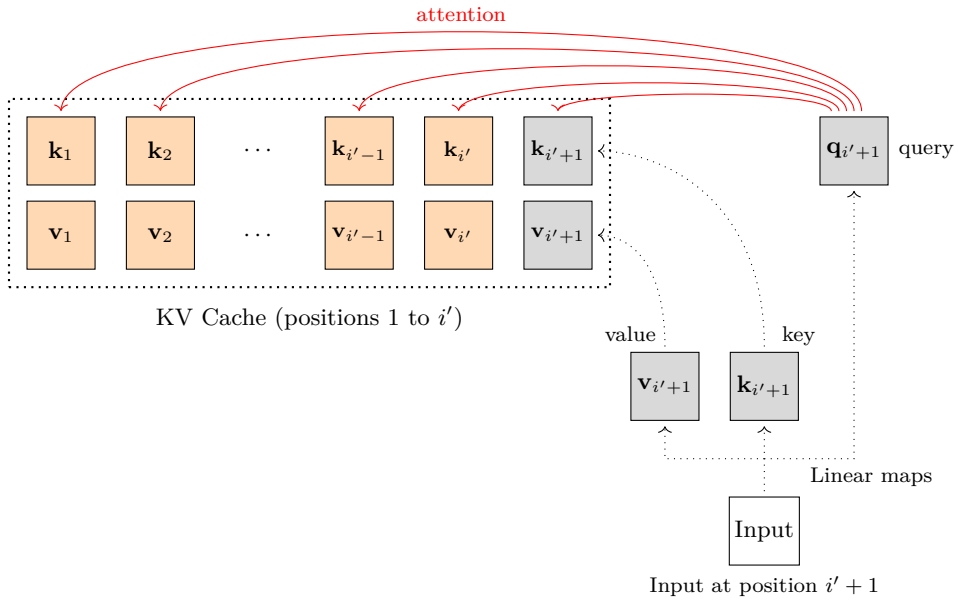
1. Wir fügen $\mathbf{x}$ und $\mathbf{y}$ zu einer Sequenz $[\mathbf{x}, \mathbf{y}]$ zusammen. Für jede Position i' dieser Sequenz führen wir die folgenden Schritte aus.

   (a) Wir berechnen das Embedding des Tokens an Position i' und geben das resultierende Embedding als initiale Repräsentation in den Stapel der Transformer-Schichten ein.

   (b) In jeder Transformer-Schicht leiten wir die Eingaberepräsentation zunächst durch das Self-Attention-Netzwerk und anschließend durch ein FFN. Im Self-Attention-Netzwerk wird die Eingaberepräsentation in $\mathbf{q}_{i'}$, $\mathbf{k}_{i'}$ und $\mathbf{v}_{i'}$ transformiert. Dann aktualisieren wir den KV-Cache $(\mathbf{K}, \mathbf{V})$ unter Verwendung von $\mathbf{k}_{i'}$ und $\mathbf{v}_{i'}$ (siehe Gleichungen (5.8-5.9)). Anschließend berechnen wir die Ausgabe



(a) Updating the KV Cache at Position i'



(b) Updating the KV Cache at Position $i'+1$

Abbildung 5.2: Veranschaulichung des KV-Cache. Wir aktualisieren den KV-Cache an einer Position, führen die Attention-Operation durch und gehen dann zur nächsten Position, um den Vorgang zu wiederholen.

des Attention-Modells, indem wir $\mathbf{q}_{i'}$ auf $(\mathbf{K}, \mathbf{V})$ anwenden (siehe Gleichung (5.7)).

(c) Wenn $i' > m$ (d. h. $i = i' - m \geq 0$), nehmen wir die Ausgabe des Transformer-Stapels und berechnen die Token-Vorhersagewahrscheinlichkeit $\Pr(y_i|\mathbf{x}, \mathbf{y}_{<i})$ über die Softmax-Schicht (siehe Gleichung (5.5)).

2. Am Ende der Sequenz erhalten wir $\log \Pr(\mathbf{y}|\mathbf{x})$, indem wir $\log \Pr(y_i|\mathbf{x}, \mathbf{y}_{<i})$ über

$i \in [1, n]$ summieren (siehe Gleichung (5.4)).

5.1.2 Ein Zwei-Phasen-Framework

Wie wir gesehen haben, ist die Sprachmodellierung ein standardmäßiger autoregressiver Prozess, bei dem jedes Token einzeln generiert wird, bedingt durch die vorhergehenden Tokens. Für Transformer erfordert dies, dass das Modell einen KV-Cache unterhält, der vergangene Repräsentationen speichert, und die neu generierte Repräsentation auf diesen KV-Cache anwendet. Wenn wir das Modell $\Pr(\mathbf{y}|\mathbf{x})$ aus der Perspektive der Berechnung des KV-Caches betrachten, ist es naheliegend, die Inferenz in zwei Phasen zu unterteilen:

- Prefilling. Die Prefilling-Phase berechnet den KV-Cache für die Eingabesequenz $\mathbf{x}$. Sie wird als Prefilling bezeichnet, weil das Modell die Schlüssel-Wert-Paare für jedes Token in der Eingabe vorbereitet und speichert, bevor die eigentliche Inferenz beginnt. Der Prozess des Prefillings in einem LLM kann wie folgt ausgedrückt werden

$$\text{cache} \quad = \quad \text{Dec}_{\text{kv}}(\mathbf{x}) \tag{5.10}$$

  wobei $\text{Dec}_{\text{kv}}(\cdot)$ das Dekodierernetzwerk ist (d. h. dasselbe wie $\text{Dec}(\cdot)$), aber es gibt den KV-Cache in der Selbst-Aufmerksamkeit anstelle der Ausgaberepräsentationen zurück. cache ist eine Liste, gegeben durch

$$\text{cache} \quad = \quad \{\text{cache}^1, ..., \text{cache}^L\} \tag{5.11}$$

  wobei cache^l die Schlüssel-Wert-Paare für die l-te Schicht darstellt.

- Decoding. Die Dekodierphase setzt die Generierung von Tokens auf Basis des KV-Cache fort, wie in Abbildung 5.2 dargestellt. Wenn ein neues Token in den Decoder eingegeben wird, aktualisieren wir den KV-Cache in jeder Schicht, indem wir das neue Schlüssel-Wert-Paar hinzufügen. Der aktualisierte Cache wird dann für die Berechnung der Selbst-Aufmerksamkeit verwendet. Die Token-Generierung stoppt, wenn ein Abbruchkriterium erfüllt ist, z. B. wenn das generierte Token das Endsymbol ist. Das Ziel der Dekodierung ist es, die am besten vorhergesagte Sequenz zu finden, die gegeben ist durch

$$\hat{\mathbf{y}} \quad = \quad \arg\max_{\mathbf{y}} \Pr(\mathbf{y}|\text{cache}) \tag{5.12}$$

  Hier verwenden wir $\Pr(\mathbf{y}|\text{cache})$ anstelle von $\Pr(\mathbf{y}|\mathbf{x})$, um zu betonen, dass der Dekodierungsprozess tatsächlich auf dem KV-Cache und nicht auf $\mathbf{x}$ beruht.

Die Prefilling- und Decoding-Prozesse werden in Abbildung 5.3 veranschaulicht. Beachten Sie, dass beide Prozesse autoregressiv sind. Wie jedoch in Tabelle 5.1 gezeigt, unterscheiden sie sich in mehreren Aspekten, was in der Praxis zu sehr unterschiedlichen Implementierungen führt.

Im Wesentlichen kann das Prefilling, obwohl das zugrunde liegende Modell auf der Token-Vorhersage basiert, als ein Kodierungsprozess betrachtet werden. Dies liegt daran, dass unser Ziel nicht darin besteht, Tokens zu generieren, sondern eine Kontextrepräsentation (d. h. den KV-Cache) für die nachfolgenden Schritte in der Decoding-Phase zu erstellen. In diesem Sinne ähnelt es BERT, wo wir die Eingabesequenz in eine Sequenz von kontextualisierten Token-Repräsentationen kodieren. Andererseits basiert das Prefilling im Gegensatz zu BERT, das bidirektionale Sequenzrepräsentationen erzeugt, auf standardmäßigen Sprachmodellierungsaufgaben und ist daher unidirektional. Beachten Sie, dass, da die gesamte Sequenz $\mathbf{x}$ auf einmal in das Modell eingegeben wird, alle Queries zusammengefasst werden können und die Selbst-Aufmerksamkeits-Operation parallel auf $\mathbf{x}$ durchgeführt wird. Sei $\mathbf{Q}$ die Queries, die in eine Matrix gepackt sind. Das Selbst-Aufmerksamkeits-Modell im Prefilling kann definiert werden als

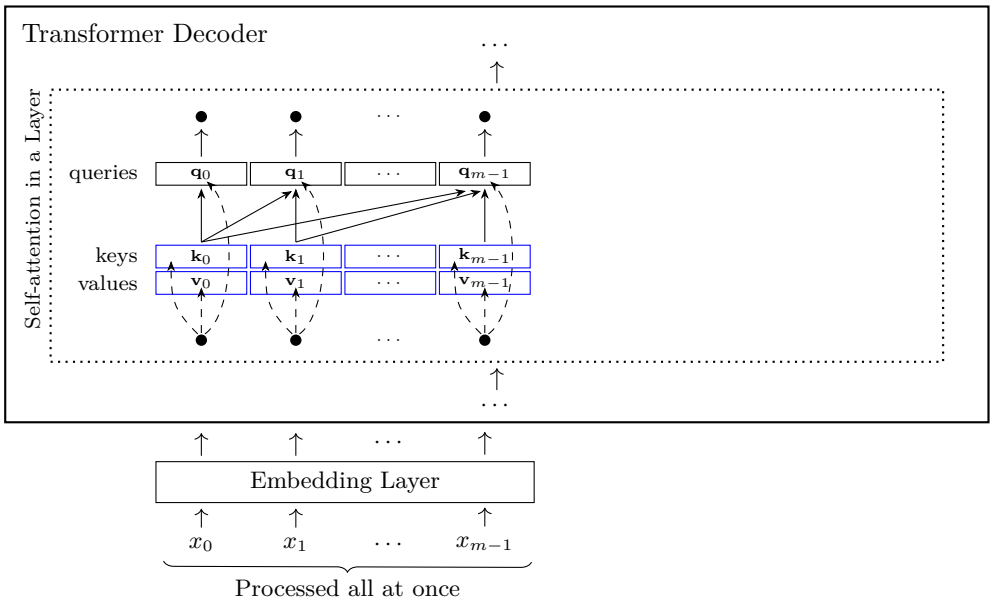
$$\text{Att}_{\text{qkv}}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) \quad = \quad \text{Softmax}(\frac{\mathbf{Q}\mathbf{K}^{\text{T}}}{\sqrt{d}} + \mathbf{Mask})\mathbf{V} \tag{5.13}$$

wobei $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{d\times(m+1)}$. $\mathbf{Mask} \in \mathbb{R}^{(m+1)\times(m+1)}$ ist eine Maske, die sicherstellt, dass jedes Token nur auf sich selbst und die ihm in der Sequenz vorangehenden Tokens achtet. Sie wird dargestellt, indem die Werte in der Maske, die zukünftigen Tokens entsprechen, auf eine große negative Zahl gesetzt werden. Zum Beispiel setzen wir für die Query $\mathbf{q}_i$ und den Key $\mathbf{k}_j$ den Wert des Eintrags (i, j) auf $-\infty$, wenn $i < j$. Ein Vorteil der Verarbeitung der Sequenz mit einer einzigen Selbst-Aufmerksamkeits-Berechnung besteht darin, dass wir die parallelen Rechenkapazitäten moderner GPUs besser nutzen und so das Prefilling beschleunigen können. Im Allgemeinen wird der Prefilling-Prozess als rechenintensiv angesehen. Dies liegt daran, dass das Zusammenführen mehrerer Rechenoperationen in eine einzige Operation die Anzahl der Datenübertragungen reduziert und der Leistungsengpass in der Regel von der Rechenkapazität und nicht von der Speicherbandbreite herrührt.

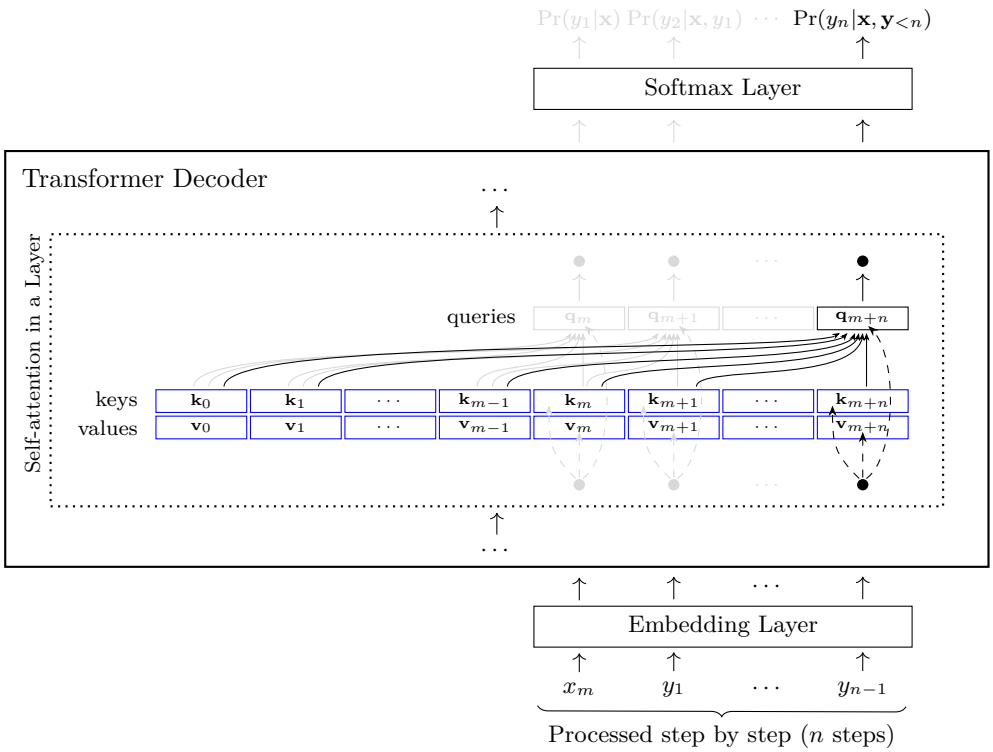
Das Decoding ist ein standardmäßiger Links-nach-Rechts-Textgenerierungsprozess. Die Token-Sequenz wird autoregressiv generiert, indem jeweils ein Token auf Basis des KV-Caches vorhergesagt wird. Jedes Mal, wenn ein neues Token generiert wird, müssen wir seine Aufmerksamkeit gemäß Gl. (5.7) auf die vorherigen Tokens richten. Daher ist der Decoding-Prozess aufgrund des häufigen Zugriffs auf den KV-Cache speicherintensiv. Die Kosten für das Decoding steigen erheblich, je mehr Tokens generiert werden. In den meisten Fällen ist das Decoding rechenintensiver als das Prefilling. Beachten Sie, dass dies nicht nur daran liegt, dass das LLM beim Decoding Tokens einzeln generiert und den KV-Cache wiederholt aktualisiert. Wie wir im folgenden Unterabschnitt sehen werden, müssen wir während des Decodings möglicherweise mehrere verschiedene Token-Sequenzen untersuchen, was das Problem komplexer macht und seine Kosten weiter erhöht.

5.1.3 Dekodierungsalgorithmen

Bisher konzentrierte sich unsere Diskussion der LLM-Inferenz hauptsächlich auf das Problem der Modellberechnung, d. h. wie $\Pr(\mathbf{y}|\mathbf{x})$ berechnet wird. Nun wenden wir uns der Diskussion des Suchproblems zu. Das Problem lässt sich wie folgt formulieren: Gegeben ein LLM $\Pr(\mathbf{y}|\mathbf{x})$, wie suchen wir effizient nach der besten Ausgabesequenz $\hat{\mathbf{y}}$ bei gegebener



(a) Prefilling



(b) Decoding (at the n-th step)

Abbildung 5.3: Veranschaulichung der Vorbefüllungs- und Dekodierungsprozesse. Beim Vorbefüllen wird die gesamte Eingabesequenz zusammen verarbeitet und der KV-Cache gefüllt. Beim Dekodieren generiert das LLM die Ausgabesequenz Schritt für Schritt basierend auf dem vorbefüllten KV-Cache.

Eingabesequenz $\mathbf{x}$ (oder dem generierten KV-Cache)? Naiv könnten wir alle Ausgabesequenzen betrachten, für jede die Vorhersagewahrscheinlichkeit berechnen und dann die Ausgabesequenz mit der höchsten Wahrscheinlichkeit auswählen. Diese Methode kann die global optimale Lösung garantieren, aber eine direkte erschöpfende Suche ist für LLMs

	Prefilling	Decoding
Goal	Set up initial context $\mathbf{x}$.	Continue generating tokens $\mathbf{y}$ after the initial input.
All-at-once Visibility	Tokens in $\mathbf{x}$ are presented all at once.	Tokens in $\mathbf{y}$ are presented sequentially, that is, predicting a token requires waiting for the previous tokens to be predicted first.
Context Use	Build the context or encoded representation of the input.	Use the cached key-value pairs (from prefilling) to generate further tokens.
Resource Limitation	Compute-bound	Memory-bound
Computational Cost	High	Very High

Tabelle 5.1: Vorbefüllung vs. Dekodierung.

unpraktikabel, da die Anzahl der möglichen Ausgabesequenzen exponentiell mit der Länge von $\mathbf{y}$ wächst.

In der Praxis werden verschiedene heuristische Suchalgorithmen, wie die gierige Suche und die stichprobenbasierte Suche, häufig zur Approximation der Lösung eingesetzt. Jede dieser Methoden bietet Kompromisse zwischen Suchqualität und Recheneffizienz. Das Suchproblem wird daher zu einem Balanceakt zwischen Exploration und Exploitation, bei dem das Ziel darin besteht, eine effiziente Strategie zu finden, die qualitativ hochwertige Ausgaben erzeugt, ohne den gesamten Raum zu durchsuchen.

Bevor wir diese Methoden detaillierter erörtern, definieren wir zunächst informell, was ein Suchraum ist und wie er dargestellt wird. Bei der LLM-Inferenz definieren wir eine Hypothese als ein Tupel aus Eingabe- und Ausgabesequenzen. Da $\mathbf{x}$ während der Inferenz fixiert ist, können wir jede Hypothese einfach als eine Ausgabesequenz betrachten. Der Suchraum, bezeichnet mit $\mathcal{Y}$, ist dann die Menge aller möglichen Hypothesen (d. h. Ausgabesequenzen), die das Modell generieren kann. Das Suchproblem für die LLM-Inferenz kann umformuliert werden als

$$\hat{\mathbf{y}} = \arg\max_{\mathbf{y} \in \mathcal{Y}} \Pr(\mathbf{y}|\mathbf{x}) \tag{5.14}$$

Im NLP wird $\mathcal{Y}$ üblicherweise in einer Baumdatenstruktur dargestellt, um die Suche zu erleichtern. Abbildung 5.4 zeigt ein Beispiel für den Suchbaum, der aus einem kleinen Vokabular resultiert. In diesem Beispiel stellt ein Knoten eine Präfix-Teilsequenz dar, die von vielen Sequenzen gemeinsam genutzt werden kann. Die Suche beginnt an der Wurzel des Baumes, die als der Anfang aller generierbaren Sequenzen betrachtet werden kann[1]. Jeder Kindknoten erweitert das Präfix seines Elternknotens, indem er ein Token aus dem Vokabular zur Sequenz hinzufügt, zusammen mit der Wahrscheinlichkeit, das

[1] Da die Vorhersagen in LLMs auf $\mathbf{x}$ basieren, können wir die Wurzel hier als eine Repräsentation von $\mathbf{x}$ betrachten.

Path: node 0 → node 3 → node 9 → node 11 → node 17
Output: cats are playful.

Probability:
node 0 → 0
node 3 → $\log \Pr(\text{“cats”}|\mathbf{x})$
node 9 → $\log \Pr(\text{“are”}|\mathbf{x}, \text{“cats”})$
node 11 → $\log \Pr(\text{“playful”}|\mathbf{x}, \text{“cats are”})$
node 17 → $\log \Pr(\text{“.”}|\mathbf{x}, \text{“cats are playful”})$

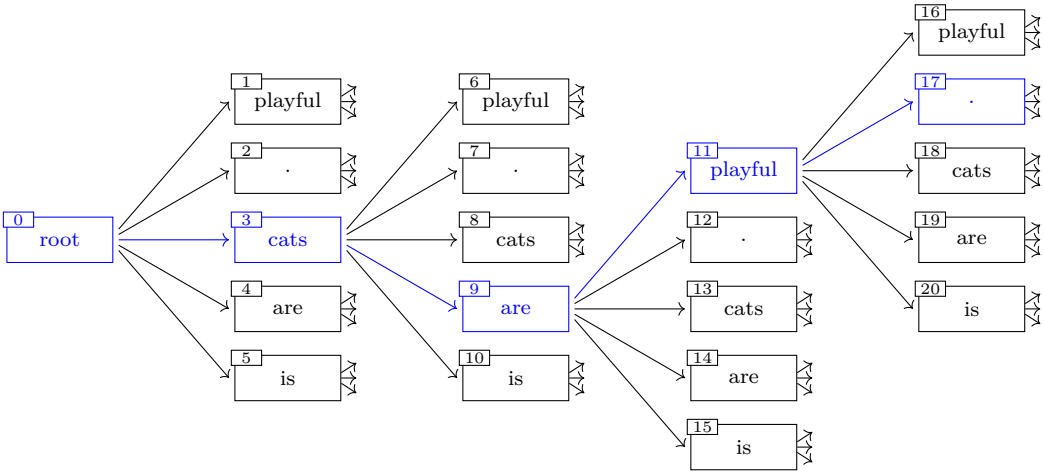


Abbildung 5.4: Ein Suchbaum zur Dekodierung. An jedem Knoten wird der Baum erweitert, indem alle möglichen Tokens berücksichtigt werden, von denen jeder zu einem neuen Knoten führt, der eine potenzielle Fortsetzung des Textes darstellt. Hier heben wir einen Pfad durch die Knoten 0, 3, 9, 11 und 17 hervor. Der Pfad repräsentiert die Ausgabesequenz “cats are playful.”, deren Log-Wahrscheinlichkeit durch Akkumulieren der Log-Wahrscheinlichkeiten dieser Knoten berechnet werden kann.

Token gegeben das Präfix vorherzusagen. Dieser Prozess setzt sich fort, indem sich jeder Knoten weiter in zusätzliche Kindknoten verzweigt, von denen jeder eine neue mögliche Erweiterung der Sequenz um ein weiteres Token darstellt. Der Suchbaum wächst somit in die Tiefe und Breite und repräsentiert eine ständig wachsende Anzahl potenzieller Sequenzen, da mehr Token angehängt werden. Diese Struktur ermöglicht es uns, effizient durch mögliche Sequenzen zu traversieren und jede hinsichtlich der über den Pfad von der Wurzel zu diesem Knoten akkumulierten Log-Wahrscheinlichkeit zu bewerten. Zum Beispiel entspricht in Abbildung 5.4 der Pfad von der Wurzel zum Knoten 17 der Ausgabesequenz „Katzen sind verspielt.“. Die Vorhersage-Log-Wahrscheinlichkeit $\log \Pr(\mathbf{y}|\mathbf{x})$ ist die Summe der Log-Wahrscheinlichkeiten aller Knoten auf diesem Pfad.

Im Allgemeinen ist der Suchbaum in Ebenen organisiert, wobei jede Ebene aus allen Knoten besteht, die den gleichen Abstand zum Wurzelknoten haben. Somit führt eine Breitensuche über den Baum im Wesentlichen eine Links-nach-Rechts-Generierung von Token durch. Knoten auf derselben Ebene entsprechen Sequenzen derselben Länge. Im Verlauf der Suche werden neue Token an diese Sequenzen angehängt, wodurch sie schrittweise erweitert werden.

Sei Y_i die Menge der Sequenzen, die das LLM im Schritt i generiert. Y_i kann durch Erweitern jeder Sequenz in Y_{i-1} mit allen möglichen nächsten Token im Vokabular V erhalten werden, gegeben in der folgenden rekursiven Form

$$Y_i \quad = \quad Y_{i-1} \times V \tag{5.15}$$

wobei $Y_{i-1} \times V$ das kartesische Produkt von Y_{i-1} und V bezeichnet (d. h. jede Sequenz in Y_{i-1} wird mit jedem Token in V verkettet). Beachten Sie, dass, wenn eine Sequenz in Y_{i-1} vollständig ist (z. B. mit dem ⟨EOS⟩-Token endet), sie nicht weiter erweitert wird. Sei $\Psi(Y_i)$ die Menge aller vollständigen Sequenzen in Y_i. Dann kann der Suchraum ausgedrückt werden als

$$\mathcal{Y} = \Psi(Y_1) \bigcup \Psi(Y_2) \bigcup \cdots \bigcup \Psi(Y_{n_{\max}}) \tag{5.16}$$

wobei $n_{\max}$ die maximale Länge einer Sequenz ist.

Die meisten Dekodierungsalgorithmen folgen diesem ebenenweisen Suchprozess. Allerdings besteht $\mathcal{Y}$ aus einer exponentiell großen Anzahl von Sequenzen, und eine direkte Suche in einem so riesigen Raum ist rechentechnisch nicht durchführbar. Daher stützen sich praktische Dekodierungsalgorithmen oft auf Strategien, um den Suchraum zu prunen und die Erkundung von Sequenzen geringer Qualität zu vermeiden. Zum Beispiel kann bei jedem Dekodierungsschritt Y_i auf folgende Weise erhalten werden

$$Y_i = \text{Prune}(Y_{i-1} \times V) \tag{5.17}$$

wobei Prune$(\cdot)$ eine Funktion ist, die selektiv Sequenzen entfernt, die weniger wahrscheinlich zu qualitativ hochwertigen Ergebnissen führen. Im Allgemeinen erwarten wir, dass $|Y_i| << |Y_{i-1}| \cdot |V|$. So können wir die Anzahl der zu berücksichtigenden Sequenzen bei jedem Schritt drastisch reduzieren und sicherstellen, dass die Rechenlast nicht exponentiell mit der Sequenzlänge wächst.

Im Folgenden werden wir diese Dekodierungsalgorithmen vorstellen. Einige von ihnen wurden bereits im Kontext von Sequence-to-Sequence-Modellen behandelt, während andere häufiger in LLMs verwendet werden.

5.1.3.1 Gierige Dekodierung

Die gierige Suche (oder gierige Dekodierung) ist eine der am weitesten verbreiteten Dekodierungsmethoden im NLP, insbesondere bei Textgenerierungsaufgaben wie der maschinellen Übersetzung. Die Idee hinter der gierigen Suche ist einfach: Bei jedem Generierungsschritt wählt sie das nächste Token mit der höchsten Vorhersagewahrscheinlichkeit aus. Für jede Sequenz $\mathbf{y} = y_1 ... y_i \in Y_{i-1} \times V$ können wir sie unter Verwendung von $\log \Pr(\mathbf{y}|\mathbf{x})$ bewerten. Diese Log-Wahrscheinlichkeit kann leicht berechnet werden, indem man beachtet, dass

$$\begin{aligned} \log \Pr(\mathbf{y}|\mathbf{x}) &= \log \Pr(y_1 ... y_i|\mathbf{x}) \\ &= \underbrace{\log \Pr(\mathbf{y}_{<i}|\mathbf{x})}_{\text{bis zum Elternknoten akkumuliert}} + \underbrace{\log \Pr(y_i|\mathbf{x}, \mathbf{y}_{<i})}_{\text{neu berechnet für den aktuellen Knoten}} \end{aligned} \tag{5.18}$$

Hier ist der erste Term die Summe der Log-Wahrscheinlichkeiten des Pfades von der Wurzel zum Elternknoten, die in den vorherigen Dekodierungsschritten berechnet wurde. Im Schritt i müssen wir nur den zweiten Term berechnen, der die Standard-Log-Wahrscheinlichkeit der Token-Vorhersage ist, die vom LLM erzeugt wird.

Das “beste” Token im Schritt i wird dann ausgewählt als

$$\begin{aligned} y_i^{\text{top1}} &= \arg\max_{y_i \in V} \log \Pr(y_1 ... y_i | \mathbf{x}) \\ &= \arg\max_{y_i \in V} \big[\underbrace{\log \Pr(\mathbf{y}_{<i} | \mathbf{x})}_{\text{konstant bzgl. } y_i} + \log \Pr(y_i | \mathbf{x}, \mathbf{y}_{<i}) \big] \\ &= \arg\max_{y_i \in V} \log \Pr(y_i | \mathbf{x}, \mathbf{y}_{<i}) \end{aligned} \tag{5.19}$$

Somit ist die bis zum Schritt i generierte “beste” Sequenz gegeben durch

$$\mathbf{y}^{\text{top1}} = y_1 ... y_{i-1} y_i^{\text{top1}} \tag{5.20}$$

Schließlich enthält Y_i nur diese Sequenz

$$Y_i = \{\mathbf{y}^{\text{top1}}\} \tag{5.21}$$

Die gierige Wahl in einem Dekodierungsschritt wird in Abbildung 5.5 (a) veranschaulicht. Die gierige Suche bietet Recheneffizienz und Einfachheit in der Implementierung für die LLM-Inferenz. Ihr Hauptnachteil liegt jedoch in ihrer suboptimalen Natur — hochwertige Sequenzen werden wahrscheinlich in frühen Phasen der Dekodierung verworfen. Daher ist die gierige Suche für Aufgaben attraktiv, die Geschwindigkeit und Einfachheit erfordern. Für Aufgaben, die bessere Suchergebnisse erfordern, sind alternative Strategien wie die Strahlsuche, die mehrere potenzielle Pfade gleichzeitig untersucht, vorzuziehen.

5.1.3.2 Beam-Dekodierung

Beam-Suche (oder Beam-Dekodierung) ist eine natürliche Erweiterung der gierigen Suche. Anstatt bei jedem Schritt das einzelne wahrscheinlichste Token auszuwählen, behält die Beam-Suche bei jedem Schritt eine feste Anzahl der besten Kandidaten bei, die als „Beam-Breite“bekannt ist. Siehe Abbildung 5.5 (b) für eine Veranschaulichung der Beam-Suche.

Sei K die Beam-Breite. Gegeben ein Elternknoten, der dem Präfix $y_1 ... y_{i-1}$ entspricht, können wir die Top-K nächsten Token auswählen durch

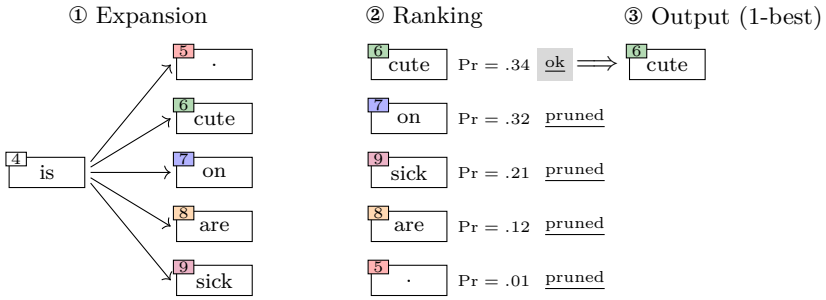
$$\{y_i^{\text{top1}}, ..., y_i^{\text{top}K}\} = \underset{y_i \in V}{\text{argTopK}} \Pr(y_i | \mathbf{x}, \mathbf{y}_{<i}) \tag{5.22}$$

wobei argTopK eine Funktion ist, die die Vorhersagewahrscheinlichkeiten aller möglichen nächsten Token einstuft und die Top-K-Kandidaten auswählt. Gegeben diese Token, sind die Top-K-Sequenzen für Schritt i gegeben durch

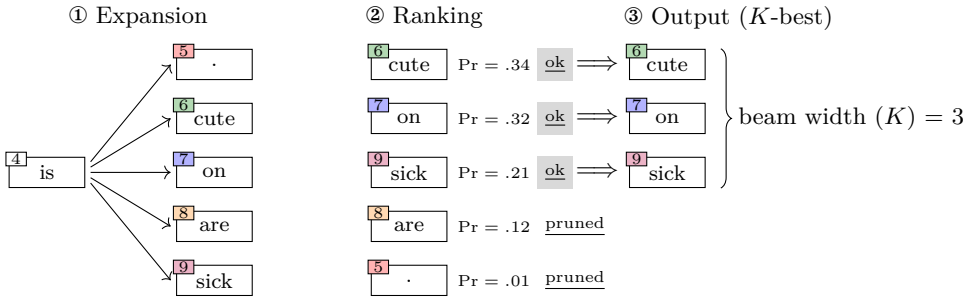
$$\mathbf{y}^{\text{top1}} = y_1 ... y_{i-1} y_i^{\text{top1}} \tag{5.23}$$

$$\vdots$$

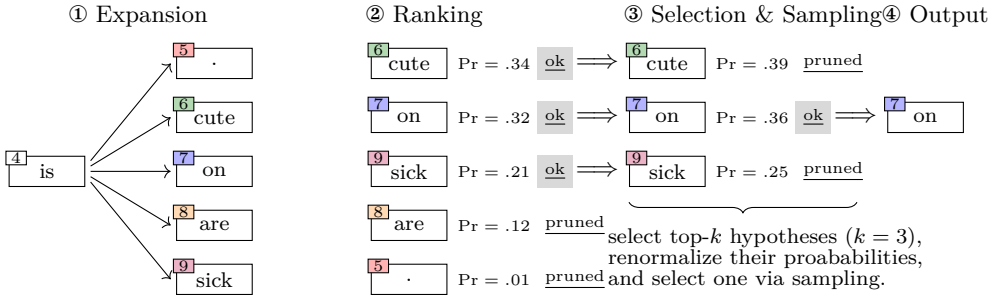
$$\mathbf{y}^{\text{top}K} = y_1 ... y_{i-1} y_i^{\text{top}K} \tag{5.24}$$



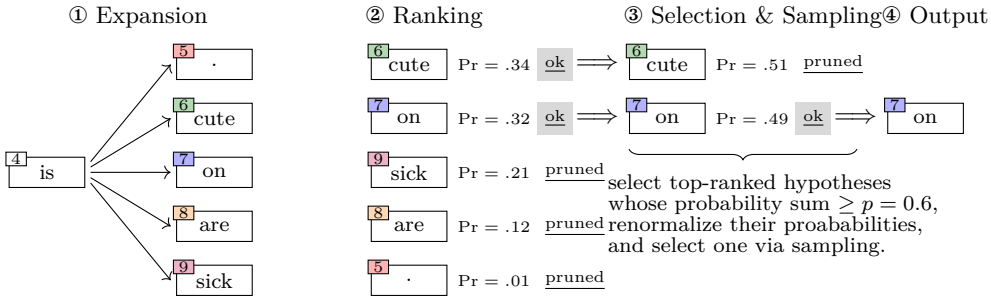
(a) Greedy search



(b) Beam search



(c) Top-k Sampling



(d) Top-p Sampling

Abbildung 5.5: Veranschaulichungen der Dekodierungsverfahren Greedy-Dekodierung, Beam-Dekodierung, Top-k-Dekodierung und Top-p-Dekodierung (in einem Dekodierungsschritt).

Dann können wir Y_i definieren als

$$Y_i \quad = \quad \{\mathbf{y}^{\text{top1}}, ..., \mathbf{y}^{\text{top}K}\} \tag{5.25}$$

Wir können die Beam-Breite K anpassen, um die Sucheffizienz und Genauigkeit auszugleichen. Aber eine sehr große Beam-Breite ist möglicherweise nicht hilfreich. In vielen praktischen Anwendungen ist die Wahl einer relativ kleinen Zahl für K, wie z. B. $K = 2$ oder $K = 4$, oft ausreichend, um eine zufriedenstellende Leistung bei der LLM-Inferenz zu erzielen.

5.1.3.3 Sampling-basierte Dekodierung

Sowohl die gierige Suche als auch die Strahlsuche erzeugen deterministische Ausgaben, das heißt, bei einem gegebenen LLM ist die Ausgabe des Modells bei jeder Verarbeitung derselben Eingabe immer gleich. Die deterministische Natur der gierigen Suche und der Strahlsuche gewährleistet Vorhersagbarkeit und Zuverlässigkeit in Anwendungen, bei denen konsistente Ergebnisse entscheidend sind, wie beispielsweise bei der Erstellung formeller Dokumente, wo unterschiedliche Ausgaben zu Verwirrung oder Fehlern führen könnten. Andererseits ist ein Nachteil dieser Methoden der Mangel an Vielfalt und Flexibilität. Beispielsweise sind bei kreativen Aufgaben wie der Generierung von Geschichten oder bei Konversationsagenten generische oder repetitive Ausgaben, die von deterministischen Systemen erzeugt werden, oft weniger ansprechend.

Um Variation in die Ausgaben von LLMs einzubringen, können wir sampling-basierte Dekodierungsmethoden verwenden. Es gibt zwei häufig verwendete Methoden.

- Top-k-Sampling. Diese Methode wählt bei jedem Schritt des Generierungsprozesses das nächste Token aus den Top-k wahrscheinlichsten Kandidaten aus [Fan et al., 2018]. Sei $\overline{V}_i$ der Auswahlpool für das Top-k-Sampling. Wir können ihn wie folgt definieren:

$$\overline{V}_i = \{y_i^{\text{top1}}, ..., y_i^{\text{top}k}\} \tag{5.26}$$

  wobei $\{y_i^{\text{top1}}, ..., y_i^{\text{top}k}\}$ die Top-k-Token sind, die basierend auf ihren Vorhersagewahrscheinlichkeiten ausgewählt wurden (siehe Gl. (5.22)). Sobald der Auswahlpool bestimmt ist, berechnen wir die Vorhersagewahrscheinlichkeitsverteilung über $\overline{V}_i$ neu. Eine der einfachsten Möglichkeiten hierfür ist die Renormierung ihrer Wahrscheinlichkeiten:

$$\overline{\Pr}(y_i|\mathbf{x}, \mathbf{y}_{<i}) = \frac{\Pr(y_i|\mathbf{x}, \mathbf{y}_{<i})}{\sum_{y_j \in \overline{V}_i} \Pr(y_j|\mathbf{x}, \mathbf{y}_{<i})} \tag{5.27}$$

  Alternativ können wir die Verteilung mithilfe der Softmax-Funktion berechnen:

$$\overline{\Pr}(y_i|\mathbf{x}, \mathbf{y}_{<i}) = \frac{\exp(u_{y_i})}{\sum_{y_j \in \overline{V}_i} \exp(u_{y_j})} \tag{5.28}$$

  wobei u_{y_i} der Logit für das Token y_i ist. Anschließend ziehen wir ein Token $\bar{y}_i$ aus dieser Verteilung:

$$\bar{y}_i \sim \overline{\Pr}(y_i|\mathbf{x}, \mathbf{y}_{<i}) \tag{5.29}$$

Die entsprechende Sequenz ist $\bar{\mathbf{y}} = y_1 ... y_{i-1}\bar{y}_i$, und Y_i ist gegeben durch

$$Y_i = \{\bar{\mathbf{y}}\} \tag{5.30}$$

- Top-p-Sampling. Diese Sampling-Methode, auch bekannt als nucleus sampling, folgt einem ähnlichen Verfahren wie das Top-k-Sampling. Anstatt aus einem Kandidatenpool fester Größe zu ziehen, wählt sie das nächste Token aus der kleinsten Menge von Token aus, die zusammen eine kumulative Wahrscheinlichkeit aufweisen, die höher als ein vordefinierter Schwellenwert p ist [Holtzman et al., 2020]. Auf diese Weise verhindern wir, dass die Vorhersage aus den Token mit geringer Wahrscheinlichkeit im Long Tail auswählt, was zu inkohärenten oder unsinnigen Ausgaben führen könnte. Um den Kandidatenpool bei der Top-p-Sampling-Methode zu erhalten, können wir alle Token nach ihren vorhergesagten Wahrscheinlichkeiten sortieren. Anschließend fügen wir, beginnend mit dem Token mit der höchsten Wahrscheinlichkeit, so lange Token zum Kandidatenpool hinzu, bis die kumulative Wahrscheinlichkeit der Token im Pool p erreicht oder überschreitet (wir bezeichnen die Größe des Kandidatenpools an diesem Punkt als k_p). Der Kandidatenpool kann dann wie folgt ausgedrückt werden:

$$\overline{V}_i = \{y_i^{\text{top1}}, ..., y_i^{\text{top}k_p}\} \tag{5.31}$$

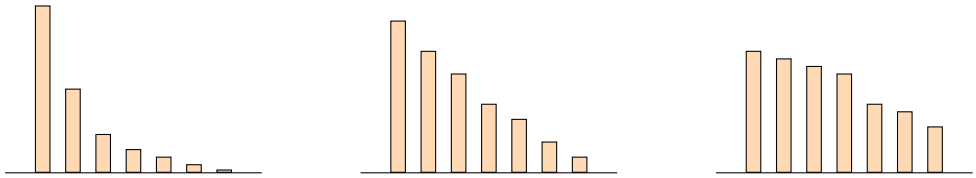
  Die nachfolgenden Schritte, wie die Renormierung der Verteilung und das Sampling, sind die gleichen wie bei der Top-k-Sampling-Methode (siehe Gln.(5.27-5.30)).

Siehe Abbildung 5.5 (c-d) für Illustrationen der Top-k- und Top-p-Sampling-Methoden. Indem sie die Auswahl auf eine kleinere Menge von Tokens mit hoher Wahrscheinlichkeit beschränken, schaffen diese Methoden ein Gleichgewicht zwischen Zufälligkeit und Kohärenz. Sie ermöglichen vielfältigere Ausgaben, während sie gleichzeitig ein angemessenes Maß an Relevanz und Flüssigkeit beibehalten. Der Wert von k oder p muss jedoch sorgfältig gewählt werden: Wenn k oder p zu klein ist, kann die Ausgabe immer noch übermäßig deterministisch sein (ähnlich der gierigen Dekodierung), und wenn k oder p zu groß ist, könnte das LLM degenerierte Ausgaben erzeugen.

Um die Zufälligkeit des Token-Auswahlprozesses weiter zu steuern, wird die renormierte Verteilung $\overline{\Pr}(\cdot)$ typischerweise durch die Verwendung der Softmax-Funktion mit dem Temperaturparameter erhalten, gegeben durch

$$\overline{\Pr}(y_i|\mathbf{x}, \mathbf{y}_{<i}) = \frac{\exp(u_{y_i}/\beta)}{\sum_{y_j \in \overline{V}_i} \exp(u_{y_j}/\beta)} \tag{5.32}$$

Hier ist β ein Temperaturparameter, der die Schärfe der aus den Logits abgeleiteten Wahrscheinlichkeitsverteilung steuert. In Abbildung 5.6 zeigen wir einfache Beispiele für Verteilungen, die durch die obige Funktion mit unterschiedlichen Temperaturen erzeugt werden. Wenn die Temperatur auf einen höheren Wert eingestellt wird, wird die resultierende Wahrscheinlichkeitsverteilung gleichmäßiger, da die Unterschiede zwischen den Logits verringert werden. Das bedeutet, dass jedes Token im Kandidatenpool eine gleichmäßigere Chance hat, ausgewählt zu werden, was zu einer größeren Vielfalt in der generierten



(a) $\beta = 0.1$ (b) $\beta = 0.8$ (c) $\beta = 2.0$

Abbildung 5.6: Histogrammschätzungen der Verteilungen, die durch die Softmax-Funktion mit unterschiedlichen Werten des Temperaturparameters β erzeugt werden.

Ausgabe führt. Im Gegensatz dazu wird die Verteilung bei einer niedrigeren Temperatur schärfer, wodurch die Tokens mit hoher Wahrscheinlichkeit noch wahrscheinlicher ausgewählt werden, was oft zu deterministischeren Ausgaben führt. Wenn wir beispielsweise p auf 1 und β auf eine sehr kleine Zahl (nahe null) setzen, wird die Top-p-Sampling-Methode äquivalent zur gierigen Suchmethode.

5.1.3.4 Dekodierung mit Straftermen

Eine übliche Verbesserung von Dekodierungsmethoden bei der Textgenerierung besteht darin, das Suchziel zu modifizieren. Zum Beispiel kann man die Maximum-a-posteriori- (MAP-) Dekodierung durch die Minimum-Bayes-Risk- (MBR-) Dekodierung [Kumar and Byrne, 2004] ersetzen, wobei sich der Fokus von der Auswahl der wahrscheinlichsten einzelnen Ausgabe hin zur Wahl einer Ausgabe verschiebt, die das erwartete Risiko über eine Verteilung möglicher Ausgaben minimiert. Hier untersuchen wir Methoden, die Straftermе in die Dekodierung einbeziehen. Diese Methoden bieten eine einfache, aber effektive Möglichkeit, die Dekodierung steuerbarer zu machen.

Erinnern wir uns aus Gl. (5.14), dass das Ziel der Dekodierung darin besteht, die Likelihood der Ausgabesequenz zu maximieren. Mit Straftermen wird das Ziel um zusätzliche Faktoren erweitert, die bestimmte Verhaltensweisen im generierten Text bestrafen oder belohnen. Eine allgemeine Form des neuen Ziels ist gegeben durch

$$\hat{\mathbf{y}} = \arg\max_{\mathbf{y} \in \mathcal{Y}} \left[\Pr(\mathbf{y}|\mathbf{x}) - \lambda \cdot \text{Penalty}(\mathbf{x}, \mathbf{y}) \right] \tag{5.33}$$

wobei $\text{Penalty}(\mathbf{x}, \mathbf{y})$ eine Funktion ist, die den Grad quantifiziert, in dem die generierte Sequenz $\mathbf{y}$ bei gegebener Eingabe $\mathbf{x}$ bestimmte Einschränkungen verletzt oder unerwünschte Verhaltensweisen aufweist. Das Design von $\text{Penalty}(\cdot)$ ist sehr flexibel, wodurch es uns ermöglicht wird, eine breite Palette von Einschränkungen oder Vorwissen darin zu integrieren. Nachfolgend stellen wir einige gängige Arten von Straffunktionen vor.

- Wiederholungsstrafe. Eine Wiederholungsstrafe hält das Modell davon ab, sich wiederholenden oder redundanten Text zu generieren. Die Strafunktion könnte die Häufigkeit wiederholter Tokens oder Phrasen in der generierten Sequenz messen und eine Strafe proportional zu deren Vorkommen verhängen.

- Längenstrafe. Eine Längenstrafe stellt sicher, dass die generierte Sequenz eine gewünschte Länge einhält. Beispielsweise könnte bei Aufgaben der Textzusammenfassung die Strafunktion Ausgaben bestrafen, die zu kurz oder zu lang sind.

- Diversitätsstrafe. Eine Diversitätsstrafe fördert die Variation im generierten Text. Beispielsweise können wir bei der Beam-Suche die Ähnlichkeit zwischen generierten Hypothesen messen und das Modell ermutigen, verschiedene Hypothesen zu untersuchen.

- Einschränkungsbasierte Strafe. Eine einschränkungsbasierte Strafe erzwingt spezifische Einschränkungen in Bezug auf den Inhalt oder den Stil des generierten Textes. Beispielsweise könnte bei der maschinellen Übersetzung die Strafunktion Ausgaben bestrafen, die von einem gewünschten Ton oder einer gewünschten Terminologie abweichen.

Im Allgemeinen können wir $\text{Penalty}(\mathbf{x}, \mathbf{y})$ als eine Funktion betrachten, die die Kosten für die Generierung der Oberflächenform der Ausgabesequenz $\mathbf{y}$ bei gegebener Eingabesequenz $\mathbf{x}$ definiert. Alternativ kann diese Funktion so definiert werden, dass sie die verborgenen Zustände eines LLM während der Generierung von $\mathbf{y}$ bewertet. Zum Beispiel entwickeln Su et al. [2022] einen Strafterm, der den maximalen Abstand zwischen der Repräsentation des vorhergesagten Tokens und den Repräsentationen der zuvor generierten Tokens berechnet. Daher wird das Suchziel degenerierte Ausgaben bestrafen, wie zum Beispiel Texte mit vielen Wiederholungen.

Die in Gl. (5.33) beschriebene Methode ist allgemein und kann leicht an verschiedene Suchalgorithmen angepasst werden. Bei der gierigen Suche zum Beispiel können wir in jedem Dekodierungsschritt die einzelne Sequenz beibehalten, die $\Pr(\mathbf{y}|\mathbf{x}) - \lambda \cdot \text{Penalty}(\mathbf{x}, \mathbf{y})$ maximiert; bei der stichprobenbasierten Suche können wir die am höchsten bewerteten Sequenzen basierend auf $\Pr(\mathbf{y}|\mathbf{x}) - \lambda \cdot \text{Penalty}(\mathbf{x}, \mathbf{y})$ einstufen und auswählen, um den Kandidatenpool zu bilden.

5.1.3.5 Spekulative Dekodierung

Spekulative Dekodierung stammt vom Konzept der spekulativen Ausführung ab, bei dem ein System fundierte Vermutungen über zukünftige Aktionen anstellt und diese im Voraus ausführt. Wenn die Vermutung korrekt ist, sind die Ergebnisse sofort verfügbar, was die Verarbeitung beschleunigt. Im Fall der LLM-Inferenz nehmen wir an, wir haben zwei Modelle. Eines ist ein kleineres, schnelleres Modell (genannt Entwurfsmodell), und das andere ist das vollständige, genauere Modell (genannt Verifikationsmodell). Diese beiden Modelle stellen zwei Baselines in der LLM-Inferenz dar: Das Entwurfsmodell ist effizient, aber nicht sehr genau; das Verifikationsmodell ist normalerweise das, das wir ausführen möchten, aber es ist sehr langsam. Gegeben ein Präfix, verwenden wir zuerst das Entwurfsmodell, um eine Sequenz wahrscheinlicher zukünftiger Tokens spekulativ vorherzusagen. Dies ist ein standardmäßiger autoregressiver Dekodierungsprozess, der jedoch aufgrund der hohen Effizienz des Entwurfsmodells in der Praxis schnell ist. Anschließend bewertet das Verifikationsmodell die spekulierten Tokens parallel. Es prüft, ob die vorhergesagten Tokens korrekt sind oder angepasst werden müssen. Beachten Sie, dass, da wir diese Tokens alle auf einmal verarbeiten können, die Verifizierung in einem einzigen Schritt für alle

Tokens gleichzeitig durchgeführt werden kann, anstatt auf einer Token-für-Token-Basis. Wenn die spekulierten Tokens korrekt sind, werden sie akzeptiert, und der Prozess wird mit dem nächsten Satz von Tokens fortgesetzt. Wenn sie inkorrekt sind, werden die inkorrekten Spekulationen verworfen, und das Verifikationsmodell wird verwendet, um die korrekten Tokens zu generieren.

Um genauer zu sein, betrachten wir die Methode der spekulativen Dekodierung, die in der Arbeit von Leviathan et al. [2023] vorgestellt wird. Bei dieser Methode ist das Entwurfsmodell ein kleines Sprachmodell, bezeichnet mit $\Pr_q(y_i|\mathbf{x}, \mathbf{y}_{<i})$, während das Verifikationsmodell ein normales LLM ist, bezeichnet mit $\Pr_p(y_i|\mathbf{x}, \mathbf{y}_{<i})$. Das Ziel ist, dass wir bei gegebenem Präfix das Entwurfsmodell verwenden, um autoregressiv bis zu τ Tokens vorherzusagen. Das Verifikationsmodell wird dann eingesetzt, um das letzte Token an dem Punkt zu generieren, an dem Fehler in den spekulativen Vorhersagen auftreten. Abbildung 5.7 illustriert einen Schritt in diesem Dekodierungsprozess.

Der Algorithmus der spekulativen Dekodierung kann wie folgt zusammengefasst werden.

- Gegeben dem Präfix $[\mathbf{x}, \mathbf{y}_{\leq i}]$, verwenden wir das Entwurfsmodell, um die nächsten τ aufeinanderfolgenden Token vorherzusagen, die mit $\{\hat{y}_{i+1}, ..., \hat{y}_{i+\tau}\}$ bezeichnet werden. Dies ist ein Token-für-Token-Generierungsprozess, gegeben durch

$$\hat{y}_{i+t} = \arg\max_{y_{i+t}} \Pr_q(y_{i+t}|\mathbf{x}, \mathbf{y}_{\leq i}, \hat{y}_{i+1}...\hat{y}_{i+t-1}) \tag{5.34}$$

- Wir bewerten $\{\hat{y}_{i+1}, ..., \hat{y}_{i+\tau}\}$ mit dem Verifizierungsmodell, das heißt, wir berechnen $\{\Pr_p(\hat{y}_{i+1}|\mathbf{x}, \mathbf{y}_{\leq i}) \ , ..., \Pr_p(\hat{y}_{i+\tau}|\mathbf{x}, \mathbf{y}_{\leq i}, \hat{y}_{i+1}...\hat{y}_{i+\tau-1})\}$. Beachten Sie, dass wir diese Wahrscheinlichkeiten parallel berechnen können, weshalb dieser Verifizierungsschritt effizient ist.

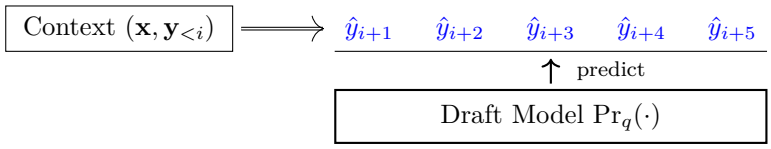
- Wir bestimmen die maximale Anzahl an akzeptierten spekulierten Token. Um die Notation übersichtlich zu halten, bezeichnen wir $\Pr_q(\hat{y}_{i+t}|\mathbf{x}, \mathbf{y}_{\leq i}, \hat{y}_{i+1}...\hat{y}_{i+t-1})$ und $\Pr_p(\hat{y}_{i+t}|\mathbf{x}, \mathbf{y}_{\leq i}, \hat{y}_{i+1}...\hat{y}_{i+t-1})$ einfach als $q(\hat{y}_{i+t})$ bzw. $p(\hat{y}_{i+t})$. Wir definieren dann, dass wir diese Spekulation akzeptieren, wenn $q(\hat{y}_{i+t}) \leq p(\hat{y}_{i+t})$ gilt. Im Gegensatz dazu lehnen wir diese Spekulation mit einer Wahrscheinlichkeit von $1 - \frac{p(\hat{y}_{i+t})}{q(\hat{y}_{i+t})}$ ab, wenn $q(\hat{y}_{i+t}) > p(\hat{y}_{i+t})$ gilt. Beginnend mit $\hat{y}_{i+1}$ wird die maximale Anzahl an akzeptierten aufeinanderfolgenden spekulierten Token wie folgt definiert

$$n_a = \min\left\{t - 1 | 1 \leq t \leq \tau, r_t > \frac{p(\hat{y}_{i+t})}{q(\hat{y}_{i+t})}\right\} \tag{5.35}$$

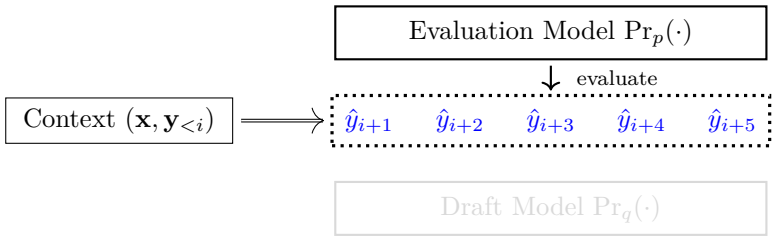
  wobei r_t eine aus der Gleichverteilung $U(0, 1)$ gezogene Variable ist.

- Gegeben n_a behalten wir die spekulierten Token $\{\hat{y}_{i+1}, ..., \hat{y}_{i+n_a}\}$. Anschließend verwenden wir das Verifizierungsmodell, um eine neue Vorhersage bei $i + n_a + 1$ zu treffen

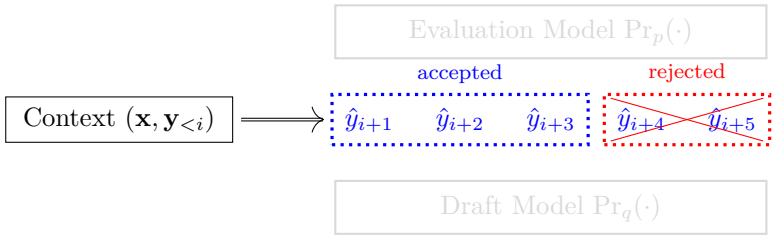
$$\bar{y}_{i+n_a+1} = \arg\max_{y_{i+n_s+1}} \Pr_p(y_{i+n_s+1}|\mathbf{x}, \mathbf{y}_{\leq i}, \hat{y}_{i+1}...\hat{y}_{i+n_s}) \tag{5.36}$$



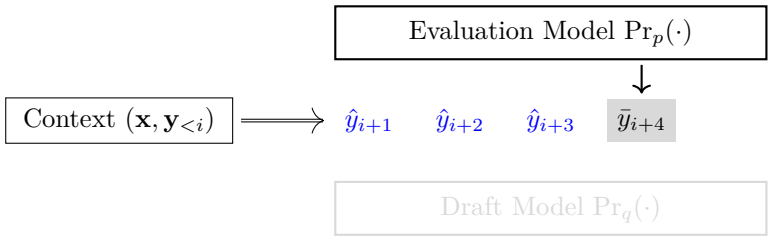
(a) Predict the next τ tokens given the context using the draft model ($\tau = 5$)



(b) Evaluate the predicted tokens using the evaluation model



(c) Determine the number of accepted tokens



(d) Predict a new token following the accepted tokens using the evaluation model

Abbildung 5.7: Veranschaulichung eines Schritts der spekulativen Dekodierung. Das Ziel ist es, mithilfe des Entwurfsmodells möglichst viele nachfolgende Tokens vorherzusagen. Es gibt vier Teilschritte. Ausgehend vom Kontext verwenden wir zunächst das Entwurfsmodell, um die nächsten τ Tokens vorherzusagen. Anschließend evaluieren wir diese Vorhersagen parallel mit dem Evaluierungsmodell. Danach bestimmen wir die maximale Anzahl an vorhergesagten Tokens, die akzeptiert werden können. Schließlich verwenden wir das Evaluierungsmodell, um ein neues Token im Anschluss an die akzeptierten Tokens vorherzusagen.

- Oben haben wir einen Schritt der spekulativen Dekodierung beschrieben. Die Ergebnissequenz (einschließlich des Kontexts und der vorhergesagten Token) wird wie folgt veranschaulicht

$$\underbrace{[\mathbf{x},\ \mathbf{y}_{<i}]}_{\text{Kontext}}\ \underbrace{\hat{y}_{i+1}...\hat{y}_{i+n_a}}_{n_a \text{ Token vorhergesagt mit dem Entwurfsmodell}}\ \underbrace{\bar{y}_{i+n_a+1}}_{\text{Ein Token vorhergesagt mit dem Verifizierungsmodell}}$$

Sobald wir diesen Schritt abgeschlossen haben, fügen wir die vorhergesagten Token $\{\hat{y}_{i+1}, ..., \hat{y}_{i+n_a}, \bar{y}_{i+n_a+1}\}$ dem Kontext hinzu und wiederholen den obigen Vorgang.

In der Praxis möchten wir normalerweise ein kleineres Entwurfsmodell verwenden, sodass die Vorhersage von $\{\hat{y}_{i+1}, ..., \hat{y}_{i+n_a}\}$ rechengünstiger wäre. Aber ein sehr kleines Entwurfsmodell ist weniger genau und kann zu einem kleineren n_a führen. Wir müssen daher das Entwurfsmodell sorgfältig auswählen, um den Kompromiss zwischen Recheneffizienz und Genauigkeit zu finden.

5.1.3.6 Abbruchkriterien

Abbruchkriterien sind eine entscheidende Komponente der LLM-Inferenz. Sie umfassen typischerweise Regeln oder Bedingungen, die festlegen, wann das Modell während der Dekodierung die Textgenerierung beenden soll. Die meisten LLMs sind darauf trainiert, ein Sequenzende-Token (z. B. ⟨EOS⟩ oder $\langle/s\rangle$) zu generieren, um das Ende des generierten Textes zu signalisieren. Eine der einfachsten Strategien besteht also darin, den Generierungsprozess zu beenden, wenn dieses Token erzeugt wird. Bei der Beam-Suche, die mehrere Hypothesen gleichzeitig untersucht, kann der Prozess fortgesetzt werden, bis eine bestimmte Anzahl vollständiger Sequenzen generiert wurde.

In praktischen Anwendungen ist es im Allgemeinen unerwünscht, sehr lange Sequenzen zu generieren, weshalb wir die Dekodierungskosten und unnötige Ausführlichkeit reduzieren müssen. Ein häufig verwendetes Abbruchkriterium ist die maximale Länge der Ausgabe. Das Modell beendet die Textgenerierung, sobald es eine vorbestimmte Anzahl von Tokens erzeugt hat. Alternativ können wir die Dekodierung basierend auf den tatsächlichen Kosten, wie Rechenressourcen oder Zeitbeschränkungen, beenden. Beispielsweise muss bei Echtzeitanwendungen wie Chatbots die Dekodierung möglicherweise nach einer bestimmten Zeitspanne beendet werden, um die Reaktionsfähigkeit sicherzustellen.

Ein weiterer Ansatz besteht darin, Abbruchkriterien basierend auf dem Verhalten von LLMs zu entwerfen. Beispielsweise kann die Dekodierung gestoppt werden, wenn die Wahrscheinlichkeit, das nächste Token vorherzusagen, unter einen bestimmten Schwellenwert fällt. Zusätzlich zum wahrscheinlichkeitsbasierten Anhalten kann ein Wiederholungserkennungsmodul implementiert werden, um das Modell zum Anhalten zu veranlassen, wenn es beginnt, Tokens oder Phrasen über eine vordefinierte Grenze hinaus zu wiederholen. Dies hilft, redundante oder inkohärente Ausgaben zu vermeiden.

5.1.4 Evaluierungsmetriken für die LLM-Inferenz

Die Bewertung der Leistung von LLMs während der Inferenz umfasst eine Vielzahl von Metriken, um zu beurteilen, wie gut diese Modelle gewünschte Standards wie Genauigkeit, Robustheit, Nutzbarkeit und Effizienz erfüllen. Wie bei den meisten NLP-Systemen können wir LLMs mithilfe von genauigkeitsbasierten Metriken wie Perplexität und F1-Score

bewerten. Wir können auch ihre Robustheit untersuchen, indem wir testen, wie gut sie mit mehrdeutigen oder anspruchsvollen Eingaben umgehen, einschließlich adversarieller, gestörter oder Out-of-Distribution-Daten. Zusätzlich kann die Nutzbarkeit bewertet werden, indem gemessen wird, wie gut die generierten Ausgaben den Erwartungen der Benutzer in Bezug auf Flüssigkeit, Kohärenz, Relevanz und Diversität entsprechen. Menschliche Bewerter können die Natürlichkeit des Textes bewerten oder beurteilen, ob die Antworten kontextuell angemessen und logisch konsistent sind. Auch ethische und Fairness-Metriken können einbezogen werden, um sicherzustellen, dass LLMs die Fortführung von Biases vermeiden oder keine schädlichen Inhalte generieren.

Alle oben genannten Evaluierungsmetriken konzentrieren sich im Wesentlichen auf die Bewertung der Qualität der Ausgaben. Angesichts der hohen Kosten für die Bereitstellung und Anwendung von LLMs sind Effizienzmetriken für Praktiker ebenfalls sehr wichtig. Nachfolgend sind einige häufig verwendete Effizienzmetriken aufgeführt [Nvidia, 2025]:

- Anfragelatenz. Diese Metrik misst die Gesamtzeit, die von der Absendung einer Anfrage an das LLM bis zum Empfang der vollständigen Antwort vergeht. Dies schließt die Zeit für die Datenübertragung, die Verarbeitung durch das Modell und die Rückgabe der Ausgabe an den Benutzer ein.

- Durchsatz. Dies bezieht sich auf die Anzahl der Tokens oder Anfragen, die das Modell pro Sekunde verarbeiten kann.

- Zeit bis zum ersten Token (TTFT). Diese Metrik misst die Zeit, die vom Beginn des Sendens einer Anfrage bis zur Generierung des ersten Tokens der Antwort vergeht. Wenn die Datenübertragung nicht zu viel Zeit in Anspruch nimmt, ist die TTFT hauptsächlich die Zeit für das Prefilling und die Vorhersage des ersten Tokens.

- Inter-Token-Latenz (ITL). Diese Metrik bezieht sich auf die Zeit, die benötigt wird, um jedes nachfolgende Token nach dem ersten zu generieren. Sie spiegelt die Effizienz des Dekodierungsprozesses wider.

- Tokens pro Sekunde (TPS). Diese Metrik quantifiziert die Anzahl der Tokens, die das Modell pro Sekunde generieren kann.

- Ressourcennutzung. Dies beinhaltet die Messung der Nutzung von Rechenressourcen (z.B. CPU- und GPU-Auslastung) und des Speicherverbrauchs des Modells während der Inferenz.

Zusätzlich zu diesen Metriken sind Energieeffizienz und Kosteneffizienz praktische Überlegungen für den Einsatz von LLMs im großen Maßstab. Die Energieeffizienz misst die Menge an elektrischer Energie, die das Modell während der Inferenz verbraucht. Die Kosteneffizienz hingegen bewertet die Gesamtkosten, die mit der Bereitstellung und Wartung des Modells verbunden sind.

Im Allgemeinen hängt die Wahl der richtigen Evaluierungsmetriken von der spezifischen Aufgabe und Anwendung ab. Während qualitätsorientierte Metriken für die Bewertung von LLMs unerlässlich sind, sind Effizienzmetriken für ihren effektiven Einsatz in

realen Anwendungen ebenso entscheidend. Ein umfassendes Evaluierungsframework sollte beide Sätze von Metriken umfassen, um die Leistung und Praktikabilität eines LLMs genau einzuschätzen.

5.2 Effiziente Inferenztechniken

In praktischen Anwendungen wünschen wir uns oft, dass ein System so effizient wie möglich ist. Für die LLM-Inferenz beinhaltet dies typischerweise zwei Arten von Verbesserungen: die Reduzierung des Speicherbedarfs und die Beschleunigung des Systems. Beispielsweise können wir die Transformer-Architektur modifizieren, um eine Speicherexplosion bei der Verarbeitung sehr langer Eingabesequenzen zu vermeiden. Ein weiteres Beispiel ist, dass wir Eingabesequenzen komprimieren können, um den Rechenaufwand zu reduzieren, während ihre semantischen Informationen erhalten bleiben. Zusätzlich können Techniken wie Quantisierung und Pruning eingesetzt werden, um die Speichernutzung und die Inferenzgeschwindigkeit weiter zu optimieren.

Effiziente Inferenz ist ein weitreichendes Thema, das sich mit mehreren Teilgebieten von LLMs überschneidet, wie zum Beispiel dem Architekturentwurf und der Modellkomprimierung. Die meisten dieser Themen wurden bereits in früheren Kapiteln behandelt. Zum Beispiel haben wir in Kapitel 2effiziente Transformer-Architekturen und Long-Context-LLMs besprochen; und in Kapitel 3haben wir Methoden zur Prompt-Komprimierung zur Reduzierung der Prompt-Länge diskutiert. In diesem Abschnitt konzentrieren wir uns auf Techniken, die üblicherweise beim Deployment und Serving von LLMs verwendet werden.

5.2.1 Weiteres Caching

In realen Anwendungen ist es üblich, häufige Anfragen und ihre entsprechenden Antworten in einem Cache zu speichern. Wenn eine neue Anfrage auf den Cache trifft, kann das System die Antwort direkt aus dem Cache abrufen, anstatt das Ergebnis neu zu berechnen. Eine einfache Implementierung ist ein Schlüssel-Wert-Datenspeicher (z. B. eine Hashtabelle), der Eingabesequenzen auf ihre von LLMs generierten Ausgabesequenzen abbildet. Im einfachsten Fall können wir häufige Anfragen sammeln, ihre Antworten mit dem LLM generieren und diese Anfrage-Antwort-Paare im Datenspeicher ablegen. Dies schafft einen grundlegenden Caching-Mechanismus auf Sequenzebene, der es dem System ermöglicht, die LLM-Berechnung zu umgehen, wenn die Eingabesequenz genau mit einer zwischengespeicherten Anfrage übereinstimmt.

Eine einfache Erweiterung des Caching-Mechanismus besteht darin, Präfixe und ihre entsprechenden verborgenen Zustände zwischenzuspeichern. Gegeben eine Eingabesequenz $\mathbf{x}$ in einem Datensatz $\mathcal{D}$, können wir sie wie in der Standard-Prefilling-Phase

verarbeiten. So erhalten wir eine Sequenz von Präfixen und ihren entsprechenden KV-Cache-Zuständen:

$$\begin{aligned} x_0 \ (\mathbf{x}_{<1}) &\Rightarrow \text{cache}_{<1} \\ x_0 x_1 \ (\mathbf{x}_{<2}) &\Rightarrow \text{cache}_{<2} \\ &\dots \\ x_0 x_1 \dots x_{m-1} \ (\mathbf{x}_{<m}) &\Rightarrow \text{cache}_{<m} \end{aligned}$$

wobei $\text{cache}_{<i}$ den KV-Cache für das Präfix $\mathbf{x}_{<i}$ bezeichnet (siehe auch Gl. (5.10)). All diese Abbildungen können zur effizienten Wiederverwendung im Präfix-Cache gespeichert werden.

Bei der Verarbeitung einer neuen Sequenz, die ein gemeinsames Präfix mit einer zuvor gesehenen Sequenz in $\mathcal{D}$ teilt, können wir die entsprechenden zwischengespeicherten verborgenen Zustände laden, anstatt sie neu zu berechnen. Konkret, wenn eine neue Eingabe $\mathbf{x}'$ das Präfix $\mathbf{x}_{<k}$ enthält (d.h. $\mathbf{x}'_{<k} = \mathbf{x}_{<k}$ für ein $k \leq m$), können wir den KV-Cache mit $\text{cache}_{<k}$ initialisieren und nur die verborgenen Zustände für die verbleibenden Tokens $\mathbf{x}'_{\geq k}$ berechnen.

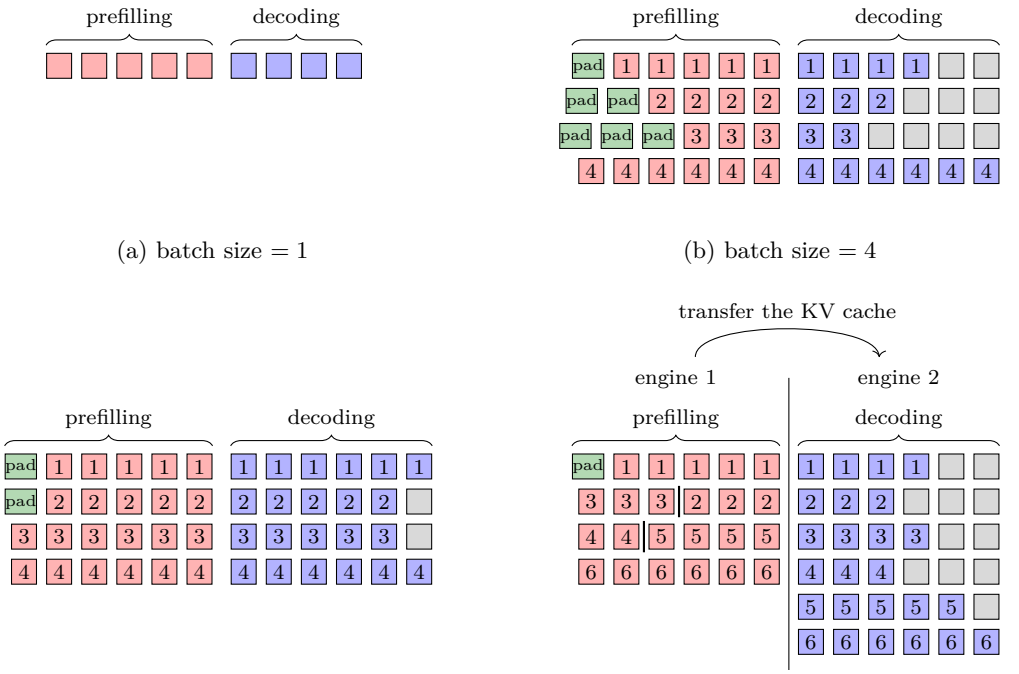
Wie üblich können wir einen Schlüssel-Wert-Datenspeicher pflegen, der häufig auftretende Präfixe auf ihre vorausberechneten KV-Caches abbildet. Die Suche kann mithilfe eines Hashs der Präfix-Tokens durchgeführt werden, was einen Zugriff auf die zwischengespeicherten Zustände in konstanter Zeit ermöglicht. Es muss darauf geachtet werden, die Speichernutzung zu verwalten, da das Speichern aller möglichen Präfixe bei großen Datensätzen möglicherweise nicht durchführbar ist. Praktische Systeme verwenden häufig LRU-Caching-Methoden (Least Recently Used) oder andere Strategien, um ein Gleichgewicht zwischen Rechenersparnis und Speicherbeschränkungen herzustellen.

5.2.2 Batching

Batching bei der LLM-Inferenz bezeichnet den Prozess, bei dem mehrere Eingabesequenzen gleichzeitig als eine Gruppe (ein sogenannter Batch) anstatt einzeln verarbeitet werden. Da moderne GPUs für die parallele Verarbeitung optimiert sind, ermöglicht das Batching die Berechnung mehrerer Sequenzen in einem einzigen Forward-Pass, wodurch die Hardware vollständig ausgelastet wird. Daher ist Batching beim skalierbaren Bereitstellen von LLMs wichtig, um die Recheneffizienz zu verbessern und die Hardware-Auslastung zu maximieren[2].

Um das Konzept des Batchings zu veranschaulichen, zeigen Abbildung 5.8 (a-b) einfache Beispiele mit Batch-Größen von 1 bzw. 4. Bei einer Batch-Größe von 1 (d. h. ohne Batching) verarbeitet die GPU jeweils eine Eingabesequenz. Die Verarbeitung erfolgt also sequenziell: Die nächste Sequenz muss warten, bis die aktuelle Berechnung abgeschlossen ist. Im Gegensatz dazu kann die GPU bei einer Batch-Größe von 4 vier Sequenzen gleichzeitig in einem einzigen Forward-Pass verarbeiten. Da die Eingabesequenzen in ihrer Länge variieren, müssen wir ihre Länge mithilfe von Padding-Techniken standardisieren. Hier verwenden wir Left-Padding, bei dem Dummy-Token am Anfang kurzer Sequenzen

[2] Siehe https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html#understand-perf für eine einfache Auswertung.



(a) batch size = 1

(b) batch size = 4

(c) batch size = 4 (similar sequence lengths)

(d) disaggregation of prefilling and decoding

Abbildung 5.8: Illustrationen von grundlegenden Batching-Methoden. Wir verwenden ein 2D-Layout, um den Batch zu veranschaulichen, wobei jedes Quadrat ein Token darstellt. Rote Quadrate kennzeichnen Tokens in der Prefilling-Phase, blaue Quadrate repräsentieren Tokens in der Dekodierungsphase, grüne Quadrate bezeichnen Padding-Tokens und graue Quadrate entsprechen bedeutungslosen Tokens. Die Teilabbildungen (a) und (b) vergleichen die Fälle, in denen die Batch-Größe 1 bzw. 4 beträgt. Teilabbildung (c) zeigt die Strategie, Sequenzen mit ähnlichen Längen im selben Batch zu gruppieren. Teilabbildung (d) veranschaulicht die Entkopplung von Prefilling und Dekodierung. Bei diesem Ansatz können wir die Parallelität von GPUs besser nutzen, indem wir mehrere kurze Sequenzen zur gemeinsamen Verarbeitung zu einer einzigen langen Sequenz verketten. Dies ermöglicht es uns, die Anzahl der in einem Batch verarbeiteten Tokens zu maximieren und gleichzeitig die Anzahl der Padding-Tokens zu minimieren. Als Kompromiss müssen wir jedoch den KV-Cache in die Dekodierungs-Engine kopieren und ihn nach der Prefilling-Phase neu organisieren, was zusätzlichen Datenübertragungsaufwand verursacht.

hinzugefügt werden, sodass alle Sequenzen im Batch für das Prefilling die gleiche Länge haben. Bei der Dekodierung werden Token für all diese Sequenzen gleichzeitig generiert, und der Generierungsprozess wird fortgesetzt, bis die längste Sequenz abgeschlossen ist.

Die obigen Beispiele implizieren einen Kompromiss zwischen Durchsatz und Latenz, was eine sehr wichtige Überlegung bei der Konzeption und Implementierung von LLM-Inferenzsystemen ist. Wenn wir eine kleinere Batch-Größe wählen, ist die Latenz geringer, da in einem einzigen Inferenzdurchlauf weniger Token verarbeitet werden müssen. Stellen Sie sich vor, wir haben nur eine einzige Sequenz. Das Ergebnis ist sofort nach Abschluss der Generierung verfügbar, ohne zusätzlichen Rechenaufwand. Dieser Vorteil der geringen Latenz geht jedoch mit einer unzureichenden Auslastung der parallelen Rechenressourcen einher, da der Parallelismus von GPUs bei der sequenziellen Verarbeitung weitgehend ungenutzt bleibt. Andererseits können wir bei Verwendung eines größeren Batches den Parallelismus besser nutzen, da GPUs mit umfangreichen Matrixberechnungen beschäftigt werden können. Dadurch können wir in der gleichen Zeit mehr Token verarbeiten, und der Durchsatz wird verbessert. Da das Ergebnis jedoch erst dann vorliegt, wenn das letzte Token im Batch vorhergesagt wurde, ist die Latenz höher.

In der Praxis bevorzugen wir in der Regel einen etwas größeren Batch, versuchen aber, den Batch mit Sequenzen ähnlicher Länge zu füllen, um die Anzahl der Padding-Token zu reduzieren und die Geräteauslastung zu verbessern. Beispielsweise können wir die eingehenden Benutzeranfragen innerhalb eines kurzen Zeitraums in Buckets gruppieren, von denen jeder Sequenzen mit ähnlichen Längen enthält. Anschließend können wir den Batch mit Sequenzen aus demselben Bucket füllen, um die Verschwendung von Rechenressourcen zu minimieren, wie in Abbildung 5.8 (c) dargestellt.

Ein anderer Ansatz zur Implementierung von Batching in LLMs besteht darin, die Prefilling- und Dekodierungsprozesse zu entkoppeln [Wu et al., 2023a; Patel et al., 2024; Zhong et al., 2024]. Beispielsweise können wir das Prefilling auf einer GPU und die Dekodierung auf einer anderen GPU durchführen. Ein Vorteil der Entkopplung besteht darin, dass wir die Eingabesequenzen im Batch neu anordnen können, um ihn besser zu füllen, da es keine Interferenzen zwischen Prefilling und Dekodierung gibt. Zum Beispiel können wir mehrere kurze Sequenzen zu einer längeren verketten und so sicherstellen, dass die Längen der Sequenzen im Batch so konsistent wie möglich sind, wie in Abbildung 5.8 (d) dargestellt. Auf diese Weise können wir den Durchsatz der Prefilling-Phase maximieren. Als Kompromiss müssen wir jedoch den KV-Cache auf die Geräte übertragen, die die Dekodierung durchführen, was zusätzlichen Kommunikationsaufwand verursacht. Typischerweise erfordert diese Methode ein Netzwerk mit hoher Bandbreite und geringer Latenz, um eine optimale Leistung zu erzielen.

In diesem Abschnitt werden wir mehrere Verbesserungen der oben genannten grundlegenden Batching-Strategien erörtern. Die meisten von ihnen basieren auf einer aggregierten Architektur, d. h. Dekodierung und Prefilling können als verschiedene Stufen eines Modells betrachtet werden, das auf demselben Gerät ausgeführt wird.

5.2.2.1 Planung

Ein praktisches LLM-Inferenzsystem besteht typischerweise aus zwei Komponenten:

- Scheduler. Seine Hauptaufgabe besteht darin, Aufgaben (d. h. Eingabesequenzen) basierend auf der aktuellen Systemauslastung und den Aufgabenprioritäten effizient in eine Warteschlange zu stellen und an die Inferenz-Engine zu verteilen. Dies beinhaltet oft eine Vielzahl von Batching-Strategien, die bestimmte Anfragen gruppieren, um die Verarbeitungseffizienz auf irgendeine Weise zu maximieren.

- Inference Engine. Sie ist für die eigentliche Ausführung der LLMs verantwortlich und verarbeitet die in der Warteschlange befindlichen Anfragen, sobald sie eintreffen. Wie bereits besprochen, umfasst diese Engine sowohl Prefilling- als auch Dekodierungsprozesse.

Diese Architektur ist in Abbildung 5.9 dargestellt. Die Einbeziehung der Planung in die Batch-Verarbeitung bietet eine flexible Möglichkeit, sowohl den Durchsatz als auch die Latenz des Systems zu optimieren und so eine bessere Balance zwischen ihnen zu erreichen. Beispielsweise können die in Abbildung 5.8 (a) und (b) gezeigten Batching-Methoden als

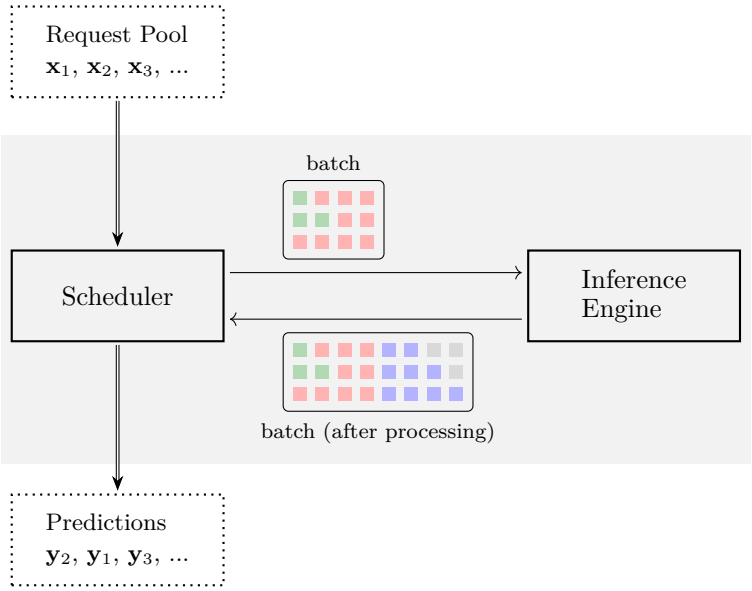


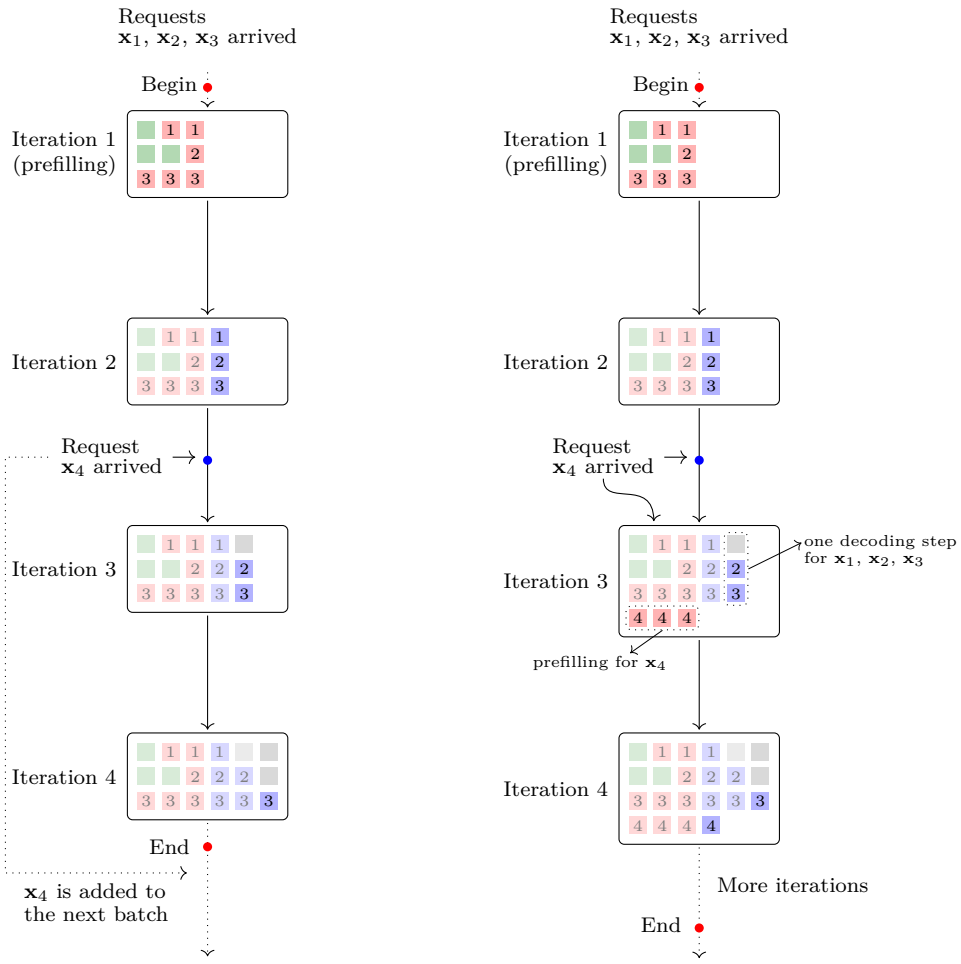
Abbildung 5.9: Veranschaulichung der LLM-Inferenzarchitektur mit einem Scheduler und einer Inferenz-Engine. Der Scheduler wählt jedes Mal eine Anzahl von Benutzeranfragen aus, um einen Batch zu bilden, und sendet diesen an die Inferenz-Engine. Der Scheduler kann mit der Inferenz-Engine interagieren und den Batch an bestimmten Punkten während der Inferenz anpassen, beispielsweise zu Beginn der Batch-Verarbeitung und am Anfang jeder Token-Vorhersage.

eine der einfachsten Planungsstrategien betrachtet werden, die als request-level scheduling bezeichnet wird. Bei dieser Strategie kann die Verarbeitung des gesamten Batches nicht unterbrochen werden, sobald ein Batch gefüllt und an die Engine gesendet wurde. Der Planer wartet, bis dieser Batch verarbeitet ist, bevor er den nächsten Batch bearbeitet [Timonin et al., 2022].

Eine ausgefeiltere Planungsstrategie, die als iteration-based scheduling bezeichnet wird, interagiert mit der Inferenz-Engine bei jedem token-Vorhersageschritt anstatt auf Sequenzebene. Dieser Ansatz ermöglicht eine dynamische Batch-Anpassung während der Inferenz, wie in Abbildung 5.10 dargestellt. Eine solch feingranulare Steuerung ermöglicht es dem System, kritische Tokens oder Sequenzen in Echtzeit zu priorisieren. Wenn beispielsweise bei einem Dekodierungsschritt eine dringende Anfrage eintrifft, kann der Planer diese Anfrage in den Batch aufnehmen, damit sie so früh wie möglich bearbeitet werden kann. In den folgenden Unterabschnitten werden wir Batching-Methoden diskutieren, die auf iterationsbasierter Planung basieren.

5.2.2.2 Kontinuierliches Batching

Continuous batching ist eine iterationsbasierte Planungsmethode, die im Orca-System [Yu et al., 2022] verwendet wird. Bei dieser Methode bezieht sich eine Iteration entweder auf den gesamten Vorbefüllungsprozess oder einen einzelnen Dekodierungsschritt. Beispielsweise gibt es bei einer Eingabesequenz $\mathbf{x} = x_0...x_m$ und einer Ausgabesequenz $\mathbf{y} = y_1...y_n$ insgesamt $n + 1$ Iterationen: eine für die Vorbefüllung und n für die Generierung der Ausgabe-Token (einen pro Token). Während der Planung kann der Batch zwischen den



(a) Request-level Scheduling (b) Iteration-level Scheduling

Abbildung 5.10: Illustrationen des Schedulings auf Anfrageebene und des iterationsbasierten Schedulings. Beim Scheduling auf Anfrageebene kann ein Batch nicht mehr angepasst werden, sobald er erstellt und an die Inferenz-Engine gesendet wurde. Mit anderen Worten, das Scheduling erfolgt erst, nachdem die Verarbeitung eines Batches abgeschlossen ist. Beim iterationsbasierten Scheduling kann das Scheduling während der Batch-Verarbeitung erfolgen. Wenn beispielsweise zu einem beliebigen Zeitpunkt während der Inferenz eine neue Anfrage eintrifft, kann diese dem Batch hinzugefügt und die Verarbeitung fortgesetzt werden.

Iterationen angepasst werden. Beispielsweise können wir entweder eine neue Eingabesequenz zum Batch hinzufügen oder eine vollständige Sequenz aus dem Batch in einer bestimmten Iteration entfernen, auch wenn die Batch-Verarbeitung noch nicht abgeschlossen ist.

Der allgemeine Prozess des kontinuierlichen Batchings umfasst die folgenden Schritte:

- Zunächst wird ein Batch mit einer oder mehreren Eingabesequenzen erstellt, basierend sowohl auf der Verarbeitungskapazität der Inferenz-Engine als auch auf den aktuellen Benutzeranfragen. Der Batch wird dann der Inferenz-Engine zugeführt.
- Die Inferenz-Engine verarbeitet den Batch Iteration für Iteration. Nach jeder Iteration kann der Scheduler den Batch auf eine der folgenden Weisen anpassen:

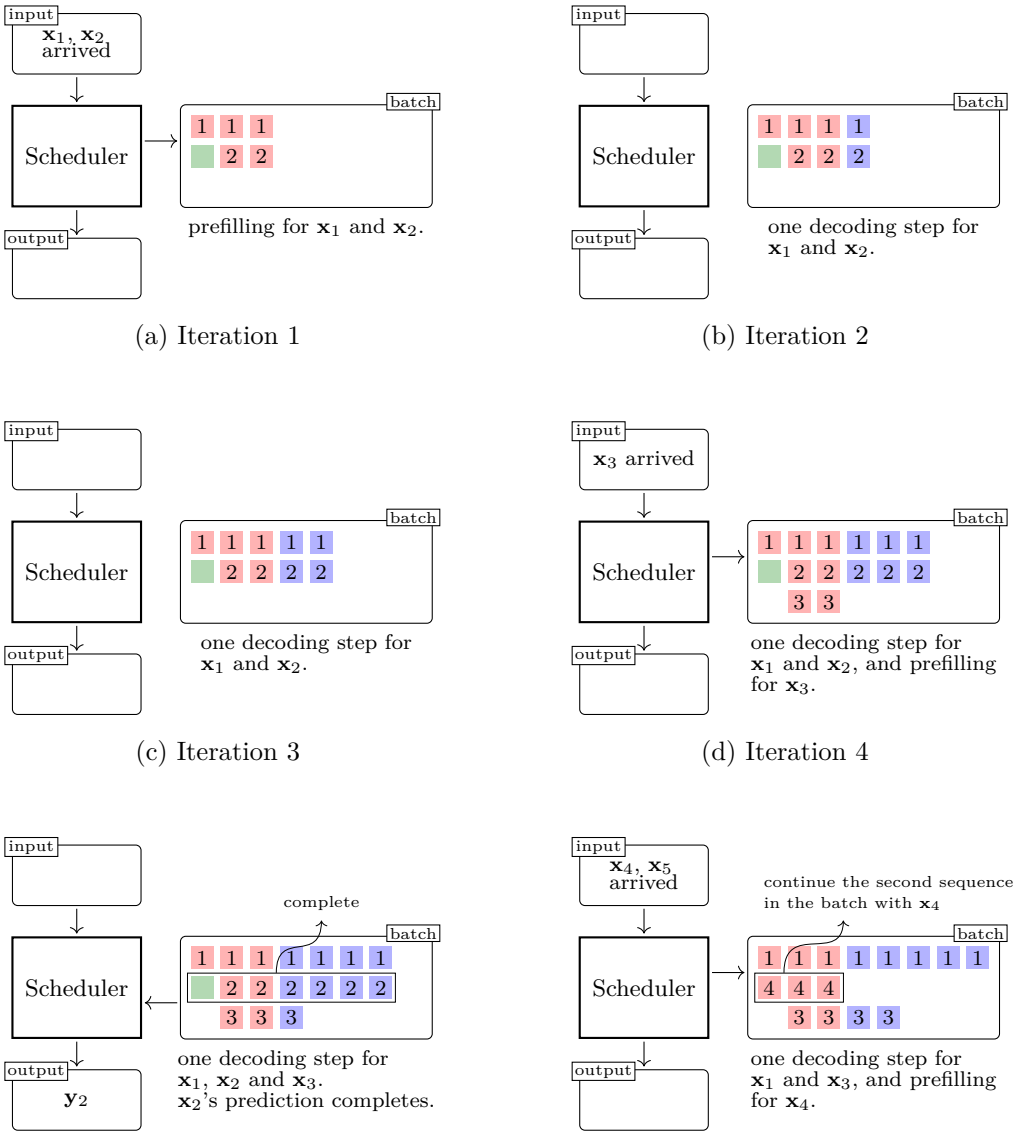
  - Wenn eine Sequenz im Batch die Generierung abschließt (d. h. das Sequenzende-Symbol generiert), wird diese Sequenz aus dem Batch entfernt.
  - Wenn eine neue Benutzeranfrage eintrifft und die Inferenz-Engine über zusätzliche Verarbeitungskapazität verfügt, wird sie dem Batch hinzugefügt.
  - Wenn keine Sequenzen zum Batch hinzugefügt oder daraus entfernt werden, bleibt der Batch unverändert.

- Die Verarbeitung wird erst beendet, wenn alle Sequenzen abgeschlossen sind und keine neuen Benutzeranfragen eingehen.

Siehe Abbildung 5.11 für ein Beispiel für kontinuierliches Batching. In diesem Beispiel beginnen wir mit zwei Benutzeranfragen, $\mathbf{x}_1$ und $\mathbf{x}_2$. Diese beiden Sequenzen werden in einem Batch zusammengefasst und zur Verarbeitung an die Inferenz-Engine gesendet. Nachdem die Engine zwei Iterationen abgeschlossen hat, trifft eine neue Benutzeranfrage, $\mathbf{x}_3$, ein. An diesem Punkt passt der Scheduler den Batch an, indem er $\mathbf{x}_3$ hinzufügt. Die Inferenz-Engine fährt dann mit der Verarbeitung des aktualisierten Batches fort. Beachten Sie, dass die Inferenz-Engine nun verschiedene Sequenzen auf unterschiedliche Weise verarbeitet: $\mathbf{x}_1$ und $\mathbf{x}_2$ fahren mit dem Dekodierungsprozess fort (d. h. der Vorhersage der nächsten Token), während $\mathbf{x}_3$ den Vorbefüllungsprozess durchläuft. Nach einiger Zeit ist die Generierung für $\mathbf{x}_2$ abgeschlossen. Zufälligerweise treffen zwei weitere Benutzeranfragen, $\mathbf{x}_4$ und $\mathbf{x}_5$, ein. Der Scheduler entfernt die abgeschlossene Sequenz $\mathbf{x}_2$ aus dem Batch und fügt unter Berücksichtigung der aktuellen Auslastung der Inferenz-Engine $\mathbf{x}_4$ zum Batch hinzu. $\mathbf{x}_5$ muss jedoch warten, bis eine andere Sequenz im Batch abgeschlossen ist, bevor es hinzugefügt werden kann.

Die Idee hinter dem kontinuierlichen Batching besteht darin, die Inferenz-Engine durch die Verarbeitung so vieler Sequenzen wie möglich vollständig auszulasten und so die Nutzung der Rechenressourcen zu maximieren. Ein wesentlicher Unterschied zwischen kontinuierlichem Batching und Standard-Batching (siehe Abbildung 5.8) liegt darin, dass beim kontinuierlichen Batching die Vorbefüllung und die Dekodierung gleichzeitig über verschiedene Sequenzen hinweg stattfinden können, während beim Standard-Batching diese beiden Phasen für den gesamten Batch sequenziell ausgeführt werden. Wie in Abschnitt 5.1.2 erläutert, wird die Vorbefüllung als rechengebundener Prozess betrachtet, während die Dekodierung als speichergebundener Prozess gilt. Die Intuition hinter der Überlappung von Vorbefüllung und Dekodierung besteht darin, die Leerlaufzeiten sowohl für die Berechnung als auch für die Datenübertragung zu reduzieren. Betrachten wir zwei Mini-Batches: einen für die Vorbefüllung und einen für die Dekodierung. Während der Vorbefüllungs-Mini-Batch die GPUs beschäftigt, kann der Dekodierungs-Mini-Batch gleichzeitig Speicherübertragungen durchführen.

Ein weiterer Unterschied zwischen kontinuierlichem Batching und Standard-Batching besteht darin, dass kontinuierliches Batching die Vorbefüllung priorisiert, während Standard-Batching die Dekodierung priorisiert [Agrawal et al., 2024]. Beim kontinuierlichen Batching fügt der Scheduler neue Anfragen zum Batch hinzu, sobald die Inferenz-Engine über freie Rechenressourcen verfügt. Mit anderen Worten, diese neu hinzugefügten Anfragen werden so früh wie möglich zur Vorbefüllung verarbeitet. Dieser Ansatz verbessert den Systemdurchsatz, geht jedoch zu Lasten einer erhöhten Latenz, da die neu hinzugefügten



(a) Iteration 1

(b) Iteration 2

(c) Iteration 3

(d) Iteration 4

(e) Iteration 5

(f) Iteration 6

Abbildung 5.11: Veranschaulichung der Batch-Anpassung beim kontinuierlichen Batching. Anstatt einen Batch von Eingabesequenzen zu fixieren und bis zur Fertigstellung zu verarbeiten (wie beim anfragebasierten Batching), aktualisiert das kontinuierliche Batching den Batch während der Inferenz dynamisch. Das System akzeptiert und fügt kontinuierlich neue Anfragen (z. B. $\mathbf{x}_3$ und $\mathbf{x}_4$) zum aktuellen Batch hinzu, solange Rechenkapazität verfügbar ist.

Anfragen die Verarbeitungszeit der früheren verlängern. Im Gegensatz dazu müssen wir beim Standard-Batching, sobald der Batch erstellt ist, warten, bis die letzte Sequenz im Batch abgeschlossen ist, bevor neue Anfragen verarbeitet werden können. Dies gewährleistet eine relativ geringe Latenz, führt aber zu einer geringeren Geräteauslastung und einem niedrigeren Systemdurchsatz.

Es ist wichtig zu beachten, dass die Kosten des kontinuierlichen Batchings darin bestehen, dass wir die Batches kontinuierlich reorganisieren müssen, was eine Neuordnung der Daten im Speicher mit sich bringt. Jedes Mal, wenn eine neue Anfrage hinzugefügt

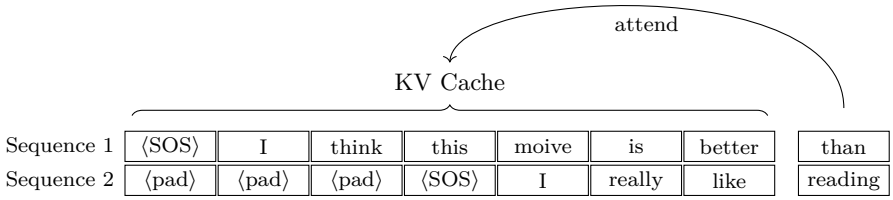
wird, muss der Scheduler die aktuelle Batch-Struktur neu bewerten und optimieren. Diese dynamische Anpassung kann zusätzlichen Speicher- und Rechenaufwand verursachen, insbesondere wenn die Batches häufig angepasst werden. Daher kann diese Methode zwar den Durchsatz verbessern, aber auch zu einer erhöhten Speicherfragmentierung führen und in einigen Fällen zusätzliche Latenz verursachen.

5.2.2.3 PagedAttention

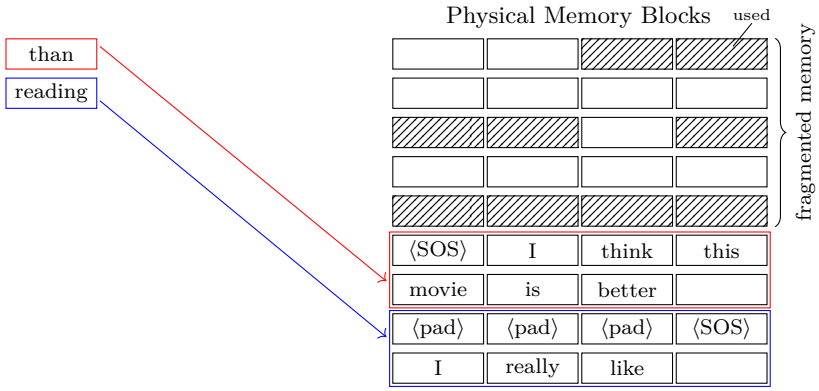
PagedAttention (oder paged KV-Caching) ist eine Technik, die im vLLM-System [Kwon et al., 2023] verwendet wird. Inspiriert vom Paging in Betriebssystemen optimiert es die Speichernutzung während der LLM-Inferenz –insbesondere für den KV-Cache –, indem es die fragmentierte Speicherzuweisung in Szenarien mit dynamischem Batching und variabler Sequenzlänge adressiert. Die Idee hinter PagedAttention ist, den großen Speicherbedarf für das KV-Caching in überschaubarere SSeitenöder Speicherblöcke aufzuteilen. Auf diese Weise müssen wir den KV-Cache der gesamten Sequenz nicht in einem zusammenhängenden Speicher ablegen. Stattdessen wird der KV-Cache in Blöcke fester Größe unterteilt (analog zu Speicherseiten in einem Betriebssystem), die nicht zusammenhängend im physischen Speicher zugewiesen werden können. Ein Vorteil von PagedAttention ist, dass es eine flexible Speicherverwaltung ermöglicht, die ein dynamisches Sequenzwachstum unterstützt, ohne eine aufwändige Neuzuweisung oder das Kopieren großer zusammenhängender Speicherbereiche zu erfordern. Es ist zu beachten, dass PagedAttention nicht speziell für das Batching entwickelt wurde. Es hilft jedoch tatsächlich, die Speichereffizienz in Szenarien der Batch-Inferenz zu verbessern, in denen die Speicherverwaltung anspruchsvoller und komplizierter ist.

Betrachten wir ein einfaches Beispiel für die Speicherzuweisung in Abbildung 5.12, bei dem Selbst-Aufmerksamkeit für einen Batch aus zwei Sequenzen durchgeführt wird. Für jede Sequenz müssen wir, wie von der Selbst-Aufmerksamkeit gefordert, das aktuelle Token auf die Schlüssel-Wert-Paare im KV-Cache dieser Sequenz anwenden. In der Standardimplementierung der Selbst-Aufmerksamkeit wird der KV-Cache in einem zusammenhängenden Speicherblock gespeichert, was uns einen effizienten Zugriff auf diesen kontinuierlichen Speicher ermöglicht. In einem paged KV-Caching-System wird der KV-Cache jedoch in kleinere Speicherblöcke fester Größe unterteilt, die nicht notwendigerweise zusammenhängend sind. Diese kleineren KV-Cache-Blöcke können effektiver fragmentierten Speicherbereichen zugewiesen werden, wodurch die Speicherauslastung verbessert wird. Ein weiterer Vorteil der Verteilung von Teilen des KV-Caches auf verschiedene Speicherblöcke besteht darin, dass dies eine Parallelisierung des Caching-Prozesses ermöglicht. Wenn beispielsweise die Eingabesequenz lang ist und die Speicherbandbreite ausreicht, wäre es vorteilhaft, die Schlüssel- und Wert-Vektoren verschiedener Segmente der Sequenz parallel über mehrere Speicherblöcke hinweg zu schreiben und zu lesen.

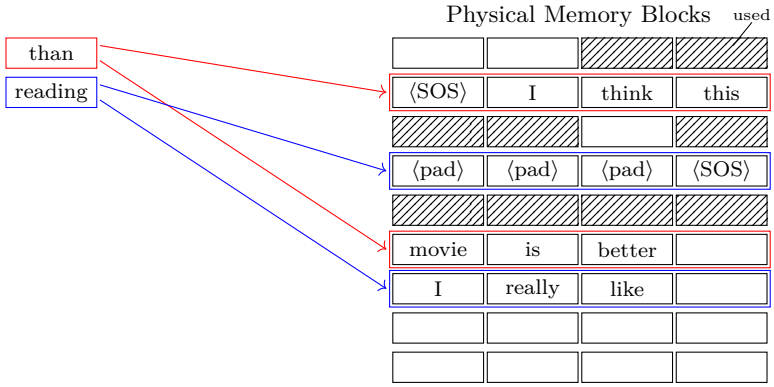
Im Allgemeinen kann die Speicherung zusammenhängender Daten in nicht zusammenhängenden Bereichen zu Problemen führen. Beispielsweise erfordert der Zugriff auf fragmentierte Daten zusätzliche Suchzeit, was die E/A-Effizienz verringert. Bei der Verarbeitung großer Datenmengen (z. B. bei der Multiplikation extrem großer Matrizen) verarbeiten wir jedoch typischerweise nicht alle Daten auf einmal, sondern teilen sie zur blockweisen Berechnung in kleinere Blöcke auf. Aus dieser Perspektive ist es auch sinnvoll,



(a) Two sequences in a batch



(b) Memory allocation for KV caching in standard self-attention



(c) Memory allocation for KV caching in PagedAttention

Abbildung 5.12: Veranschaulichung der Speicherzuweisung in PagedAttention. Im Batch befinden sich zwei Sequenzen, wie in Teilabbildung (a) dargestellt. Da der Speicher fragmentiert ist, wird der KV-Cache bei der Standard-Self-Attention in einem großen, ungenutzten Speicherblock gespeichert (siehe Teilabbildung (b)), der fragmentierte Speicher wird jedoch nicht genutzt. Im Gegensatz dazu wird bei PagedAttention (siehe Teilabbildung (c)) der KV-Cache in kleinere Blöcke unterteilt und passt somit in den fragmentierten Speicher.

die Aufmerksamkeitsberechnung zu partitionieren. Wenn die Paging-Strategie gut konzipiert ist, kann der zusätzliche Overhead beim Speicherzugriff minimal sein, während die Verbesserung der Speicherauslastung erheblich sein kann.

5.2.2.4 Aufgeteiltes Prefilling

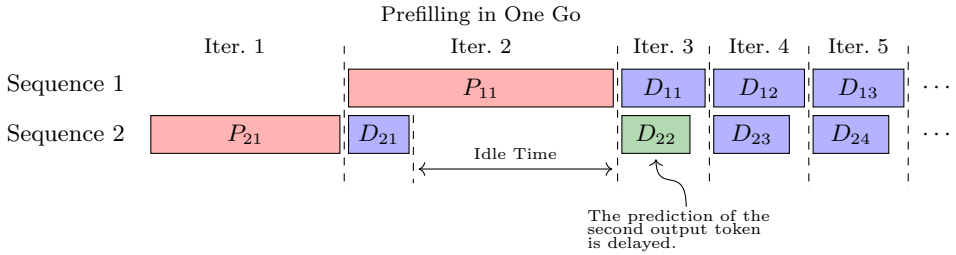
Wir haben gesehen, dass beim Scheduling auf Iterationsebene die Vorbefüllung und Dekodierung für verschiedene Sequenzen gleichzeitig erfolgen können. Dies kann als eine vorbefüllungspriorisierte Strategie angesehen werden, die den Durchsatz maximieren kann. Eine solche Iteration kann jedoch lange dauern, wenn die Eingabesequenz sehr lang ist und der Vorbefüllungsprozess die Berechnung dominiert. In diesem Fall muss die Dekodierung für andere Sequenzen warten, bis die Vorbefüllung abgeschlossen ist, was zu einer erhöhten Latenz bei der Generierung von Ausgabe-Tokens führt. Daher sind vorbefüllungspriorisierte Strategien zwar wirksam zur Maximierung der Hardware-Auslastung, können aber eine erhebliche Variabilität in der Latenz der Token-Generierung mit sich bringen, insbesondere wenn das System eine Mischung aus langen und kurzen Eingabesequenzen verarbeitet.

Eine einfache Möglichkeit, die Dekodierungslatenz zu reduzieren, besteht darin, die Berechnungen für verschiedene Sequenzen im Batch vergleichbar zu machen. Eine solche Methode besteht darin, Sequenzen in Blöcke (Chunks) aufzuteilen und die Vorbefüllung Block für Block durchzuführen. Dieser Ansatz, oft als aufgeteiltes Prefilling (Chunked Prefilling) bezeichnet, verarbeitet jeweils kleinere Teile jeder Sequenz, wodurch das System die Rechenlast besser auf die Sequenzen verteilen kann [Agrawal et al., 2023]. Durch die Wahl einer geeigneten Blockgröße können wir sicherstellen, dass, wenn sich Vorbefüllung und Dekodierung für zwei Sequenzen überschneiden, ihre Verarbeitung innerhalb derselben Iteration tendenziell eine ähnliche Zeit in Anspruch nimmt. Dadurch wird die Leerlaufzeit der Dekodierung reduziert und der Gesamtdurchsatz verbessert.

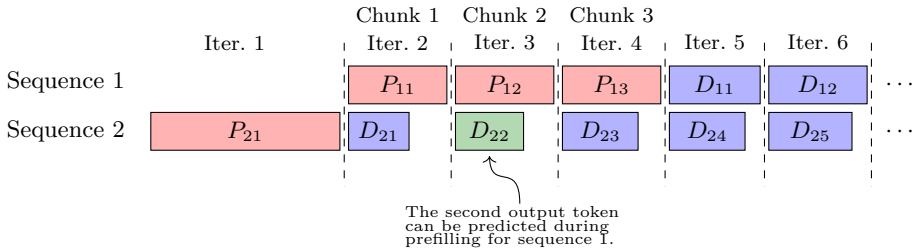
Abbildung 5.13 zeigt eine Veranschaulichung des aufgeteilten Prefillings in einigen Iterationen. In diesem Beispiel enthält der Batch zwei Sequenzen. Der gesamte Vorbefüllungsprozess der ersten Sequenz wird in drei Vorbefüllungsschritte unterteilt, wodurch die mit P_{11}, P_{12} und P_{13} bezeichneten Blöcke entstehen. Jeder Block entspricht einer Iteration und kann sich somit mit einem Dekodierungsschritt überlappen. Auf diese Weise können wir während der Vorbefüllung der ersten Sequenz drei Dekodierungsschritte durchführen, anstatt nur eines einzigen Dekodierungsschritts, wie es beim standardmäßigen Scheduling auf Iterationsebene der Fall ist. Dadurch wird die Leerlaufzeit des Dekodierungsprozesses reduziert, und die Ausgabe-Tokens können früher generiert werden.

Aufgeteiltes Prefilling verbessert die Effizienz der Dekodierung durch die Überlappung von Vorbefüllung und Dekodierung, jedoch auf Kosten von zusätzlichem Speicheraufwand und erhöhter Scheduling-Komplexität. Beim standardmäßigen Prefilling verarbeiten wir die gesamte Eingabesequenz auf einmal und erstellen den KV-Cache in einem Durchgang. Im Gegensatz dazu benötigt beim aufgeteilten Prefilling jeder Block einen separaten Forward-Pass, um seine Attention-Ausgaben zu berechnen und den KV-Cache zu aktualisieren. Daher müssen wir den KV-Cache früherer Blöcke beibehalten, während wir spätere Blöcke verarbeiten. Dies beeinträchtigt auch die Parallelität der Vervollständigung der Vorbefüllung für die gesamte Sequenz in einem einzigen Durchgang. In der Praxis ist es in der Regel möglich, Durchsatz und Latenz durch die Wahl einer geeigneten Blockgröße auszugleichen.

Es ist erwähnenswert, dass die in diesem Unterabschnitt besprochenen Methoden grob als prioritätsbasierte Scheduling-Methoden kategorisiert werden können. Bei diesen Methoden können wir bestimmten Anfragen oder bestimmten Vorbefüllungs- oder Dekodierungsschritten Priorität einräumen, sodass die Systemressourcen so zugewiesen werden,



(a) Simple Iteration-level Scheduling



(b) Chunked Prefilling

Abbildung 5.13: Vergleich von einfachem iterationsbasiertem Scheduling und segmentiertem Prefilling. P_{xy} bezeichnet den y-ten Prefilling-Schritt für die Sequenz x, und D_{xy} bezeichnet den y-ten Dekodierungsschritt für die Sequenz x. Beim einfachen iterationsbasierten Scheduling (oder Prefilling-priorisiertem Scheduling) muss D_{22} auf den Abschluss des Prefillings von Sequenz 1 warten, da das Prefilling als eine einzige Iteration behandelt wird. Beim segmentierten Prefilling kann der Prefilling-Prozess in mehrere Schritte unterteilt werden. Somit kann D_{22} während des Prefillings für Sequenz 1 (d. h. während P_{12}) ausgeführt werden.

dass sie besser mit spezifischen Leistungszielen übereinstimmen. Wie oben dargestellt, können wir beispielsweise die Dekodierung gegenüber der Vorbefüllung priorisieren, um die Latenz der Token-Generierung zu minimieren, oder die Vorbefüllung gegenüber der Dekodierung priorisieren, um den Gesamtdurchsatz in Batch-Verarbeitungsszenarien zu maximieren. Praktiker können benutzerdefinierte Prioritätsrichtlinien für spezifische Bedürfnisse und betriebliche Einschränkungen in realen Anwendungen entwerfen, wie z. B. von Benutzern definierte Fristen für Anfragen und Wichtigkeitsstufen.

5.2.3 Parallelisierung

Parallelisierung ist ein weit verbreiteter Ansatz, um die LLM-Inferenz zu skalieren, insbesondere für große Bereitstellungen. In Kapitel 1haben wir mehrere gängige Parallelisierungsstrategien zur Parallelisierung des LLM-Vortrainings besprochen, wie z. B. Modellparallelität, Tensorparallelität und Pipeline-Parallelität. Wir haben auch effiziente Architekturen erörtert, die sich leicht in verteilten Rechensystemen einsetzen lassen. Beispielsweise weisen wir in MoE-Modellen verschiedenen Geräten unterschiedliche Experten zu[3]. Nur die aktiven Experten für eine gegebene Eingabe werden ausgeführt, was die Recheneffizienz erheblich verbessert und gleichzeitig die Modellqualität beibehält. Viele dieser Methoden können mit minimalen Modifikationen direkt auf die LLM-Inferenz angewendet werden.

[3]In LLMs sind die Experten typischerweise modulare FFNs. Jeder Experte ist also ein Teil der FFN-Komponente in der Transformer -Architektur.

Die Anwendung dieser Parallelisierungstechniken auf die Inferenz stellt jedoch im Vergleich zum Vortraining neue Herausforderungen dar. Diese Probleme werden besonders in Echtzeit- oder Inferenzszenarien mit geringer Latenz deutlich, in denen Lastungleichgewichte über Geräte hinweg und Kommunikations-Overhead die Leistung erheblich beeinträchtigen können. Im Gegensatz zum Vortraining, bei dem Batches im Voraus vorbereitet werden können, muss die Inferenz beispielsweise Sequenzen variabler Länge in Echtzeit verarbeiten. Dies erschwert die Aufrechterhaltung einer optimalen Geräteauslastung und verkompliziert das Scheduling über heterogene Rechenressourcen hinweg. Ein verwandtes Problem ist der Lastausgleich. Wenn eine große Anzahl von Anfragen in kurzer Zeit eintrifft, muss das System die Arbeitslasten effizient auf die verfügbaren Geräte verteilen. Beispielsweise weisen reale Anfragen aufgrund von Unterschieden bei Aufgabentypen und Prompt-Längen typischerweise stark variable Rechenanforderungen auf. Eine solche Variabilität macht einfache statische Lastausgleichsansätze unwirksam, weshalb wir feinkörnigere Strategien verwenden müssen, die sich an Laufzeitbedingungen anpassen können. Das Problem wird noch komplizierter, wenn wir das System auf heterogener Hardware einsetzen und strenge Latenzanforderungen bestehen.

Bei der Entwicklung von LLMs ist die Parallelisierung eng mit dem LLM-Serving verbunden. Im Allgemeinen ist der Aufbau eines hochwertigen LLM-Serving-Systems keine einfache Aufgabe — er erfordert typischerweise die Kombination mehrerer Techniken, wie Architekturentwurf, Arbeitslastverteilung und LLM-spezifische Hardware-/Software-Optimierungen. Daher stellt das LLM-Serving ein außergewöhnlich breites Thema dar, das oft erhebliche Ingenieurkompetenz erfordert. Hier werden wir nicht auf die Details des LLM-Serving eingehen. Für verwandte Konzepte und Techniken können sich die Leser auf relevante Open-Source-Systeme (wie vLLM[4], TensorRT-LLM[5] und TGI[6]) und Veröffentlichungen [Pope et al., 2023; Li et al., 2024a] beziehen.

5.2.4 Anmerkungen

Wir haben in diesem und den vorherigen Kapiteln viele Methoden zur Verbesserung der Effizienz von LLMs betrachtet. Obwohl diese Ansätze unterschiedliche Probleme adressieren, untersuchen die meisten von ihnen im Wesentlichen Kompromisse zwischen verschiedenen Leistungsfaktoren. Ein wichtiger Kompromiss besteht zwischen Inferenzgeschwindigkeit und Genauigkeit. Beispielsweise können Techniken wie Quantisierung, Pruning und Wissensdestillation den Rechenaufwand und die Latenz erheblich reduzieren, können aber zu geringfügigen Verschlechterungen der Modellleistung führen. Umgekehrt erhöht die Beibehaltung der vollen Präzision oder die Verwendung größerer Modelle die Genauigkeit, jedoch auf Kosten einer langsameren Inferenz und eines höheren Ressourcenbedarfs.

Eine weitere wichtige Überlegung bei der LLM-Inferenz ist der Kompromiss zwischen Speicher und Rechenleistung. Wie beim Entwurf von Computersystemen müssen wir das

[4] https://github.com/vllm-project/vllm
[5] https://github.com/NVIDIA/TensorRT-LLM
[6] https://github.com/huggingface/text-generation-inference

Gleichgewicht zwischen Speichernutzung und dem zur Erzeugung der Ausgabe erforderlichen Rechenaufwand berücksichtigen. Insbesondere kann das Speichern von Zwischenergebnissen wie KV-Caches während der Inferenz redundante Berechnungen erheblich reduzieren, jedoch auf Kosten eines erhöhten Speicherverbrauchs. Beim KV-Caching vermeidet das Speichern vergangener Aufmerksamkeitszustände die Neuberechnung der Self-Attention über frühere tokens, wodurch die Rechenzeit pro token reduziert wird. Mit zunehmender Anzahl von tokens wächst jedoch auch der Speicherbedarf des KV-Caches, insbesondere bei der Verarbeitung sehr langer Sequenzen oder mehrerer Sequenzen parallel. Als Reaktion darauf wurden verschiedene Techniken entwickelt, um den Speicherverbrauch durch teilweises Neuberechnen von Zwischenzuständen zu reduzieren. Beispielsweise begrenzt die segmentierte oder gefensterte Aufmerksamkeit (chunked or windowed attention) den Aufmerksamkeitsbereich auf eine aktuelle Teilmenge von tokens, was die Größe des KV-Caches auf Kosten eines reduzierten Kontexts oder zusätzlicher Berechnungen reduziert, falls vergangene Informationen neu verarbeitet werden müssen.

Beachten Sie, dass die Berücksichtigung des Kompromisses zwischen Speicher und Rechenleistung ein sehr allgemeines Prinzip ist. Es kann über Aufmerksamkeitsmechanismen und Transformer hinaus auf andere Komponenten im Systemdesign erweitert werden. Ein Beispiel ist die Wahl der Datenpräzision. Die Verwendung von Formaten mit geringerer Präzision wie FP16 oder INT8 kann sowohl den Speicherverbrauch als auch die Anforderungen an die Speicherbandbreite reduzieren und so den Druck auf das Speichersubsystem wirksam verringern. Eine geringere Präzision kann jedoch zu numerischer Instabilität oder einer leichten Verschlechterung der Genauigkeit führen, was eine sorgfältige Kalibrierung oder ein erneutes Training erfordert. Daher kann dieser Kompromiss auch als ein Speicher-Rechenleistungs-Genauigkeits-Dreieck betrachtet werden, bei dem Verbesserungen in einer Dimension auf Kosten einer anderen gehen können.

Über Geschwindigkeit, Genauigkeit und Speicher hinaus beeinflussen auch mehrere andere Dimensionen die Effizienz der LLM-Inferenz. Einige dieser Dimensionen wurden in diesem Kapitel besprochen, andere nicht. Hier skizzieren wir sie wie folgt.

- Durchsatz vs. Latenz: In groß angelegten Multi-User-LLM-Bereitstellungsszenarien zielen wir oft darauf ab, den Systemdurchsatz zu maximieren. Wie in diesem Abschnitt besprochen, können wir beispielsweise mehrere Anfragen zusammenfassen (batchen), um die Anzahl der gleichzeitig verarbeiteten Tokens zu erhöhen. Jedoch erhöht das Batching die Wartezeit und kann zu einer höheren Latenz pro Anfrage führen, insbesondere bei kurzen oder interaktiven Anfragen. Im Gegensatz dazu erfordert die Optimierung für eine niedrige Latenz oft die Bearbeitung von Anfragen einzeln oder in kleineren Batches, was die Hardwareressourcen nicht voll auslastet und den Durchsatz verringert. Das Erreichen einer guten Balance hängt von den Anforderungen an die Dienstgüte und den Benutzerinteraktionsmustern ab.

- Generalisierung vs. Spezialisierung: Allzweck-LLMs werden darauf trainiert, eine breite Palette von Aufgaben mit einem einzigen Satz von Parametern auszuführen. Obwohl sie flexibel sind, können sie für spezifische Aufgaben weniger effizient oder genau sein. Spezialisierte Modelle können eine bessere Leistung und geringere Inferenzkosten für zielgerichtete Anwendungen erzielen. Jedoch erhöht die Wartung

mehrerer spezialisierter Modelle die Systemkomplexität und die Speicheranforderungen. Der Kompromiss zwischen der Pflege eines einzigen allgemeinen Modells und mehrerer spezialisierter Modelle ist eine wichtige Designentscheidung auf Systemebene.

- Energieeffizienz vs. Leistung: Hochleistungs-Inferenz erfordert oft den Betrieb großer Modelle mit hohem Durchsatz auf leistungsstarken Beschleunigern, was erheblich Energie verbraucht. Dies kann für Edge-Deployments oder energieempfindliche Umgebungen problematisch sein. Techniken wie die Modellkomprimierung können die Energieeffizienz verbessern, jedoch in der Regel mit einer gewissen Verschlechterung der Ausgabequalität oder einer Erhöhung der Latenz. Energiebeschränkungen führen somit eine weitere wichtige Dimension bei der Optimierung der LLM-Inferenz ein.

5.3 Skalierung zur Inferenzzeit

Skalierungsgesetze können als eines der grundlegenden Prinzipien betrachtet werden, die die Entwicklung von LLMs leiten. In früheren Kapiteln haben wir mehrfach erörtert, dass die Skalierung von Trainingsdaten, Modellgröße und Rechenleistung die Leistung des Vortrainings effektiv verbessern kann. Tatsächlich gelten Skalierungsgesetze auch für nachgelagerte Phasen wie Feinabstimmung und Inferenz (siehe Abbildung 5.14). Hier betrachten wir die Skalierung zur Inferenzzeit, die von neueren LLMs weithin eingesetzt wird, um komplexe Probleme, wie zum Beispiel komplexe mathematische Probleme [Snell et al., 2025], zu lösen. Im Gegensatz zur Skalierung beim Vortraining und bei der Feinabstimmung, die sich auf die Verbesserung von LLMs durch Parameteraktualisierungen konzentriert, verbessert die Skalierung zur Inferenzzeit diese Modelle während der Inferenz ohne weiteres Training. Dies umfasst eine große Vielfalt von Methoden, die LLMs in verschiedenen Dimensionen skalieren, wie das Ensembling mehrerer Modellausgaben, die Erhöhung der Kontextlänge, die Anwendung aggressiverer Dekodierungsalgorithmen und die Nutzung externer Werkzeuge zur Erweiterung der Modellfähigkeiten.

Obwohl die Skalierung zur Inferenzzeit weitreichend ist, betrachten wir in diesem Abschnitt jene Methoden, die mehr Rechenleistung in die Inferenz einbeziehen (genannt rechenleistungsbasierte Skalierung zur Inferenzzeit). Hier ist eine Liste von Methoden zur rechenleistungsbasierten Skalierung zur Inferenzzeit (Testzeit), geordnet nach Kategorien:

- Kontextskalierung. Dabei wird die Eingabe oder der Kontext skaliert, um die Generierung zu verbessern (oder potenziell die Ausgabe zu skalieren).
- Suchskalierung. Dies beinhaltet die Erhöhung des Rechenaufwands während der Dekodierung.
- Ausgabe-Ensembling. Dabei werden mehrere Modellausgaben kombiniert.
- Generieren und Verifizieren von Denkwegen. Dies beinhaltet die Anleitung von LLMs, Denkwege zur Lösung komplexer Schlussfolgerungsprobleme zu generieren und zu verifizieren.

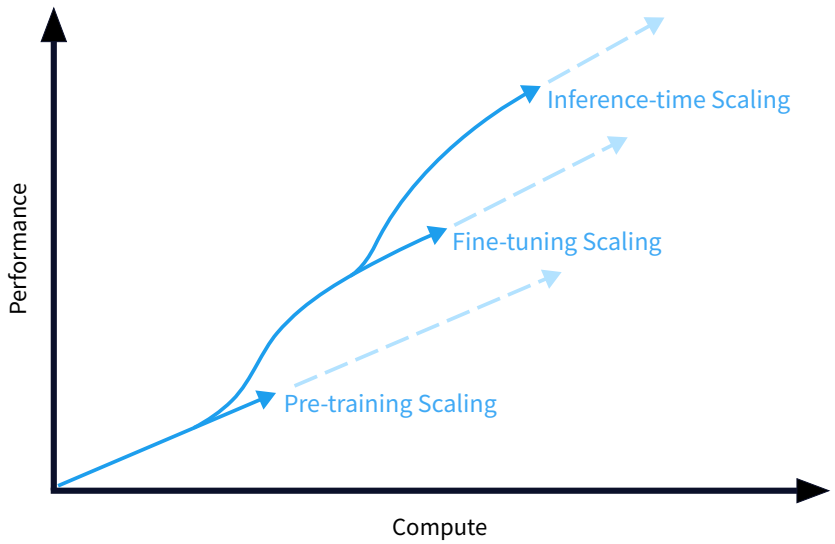


Abbildung 5.14: Skalierung für die Vortrainings-, Feinabstimmungs- und Inferenz-Phasen [Briski, 2025].

Wir werden diese Methoden in den folgenden Unterabschnitten beschreiben.

5.3.1 Kontextskalierung

Die Kontextskalierung verbessert die Leistung von LLMs, indem die Eingabe für das Modell erweitert wird. Ein einfacher Ansatz besteht darin, während der Inferenz mehr hilfreichen Kontext einzubeziehen, wodurch das Modell seine Vorhersagen auf mehr Vorwissen stützen kann. Ein Beispiel dafür ist das Few-Shot-Prompting. Es erweitert den Kontext um mehrere Eingabe-Ausgabe-Beispiele, sodass das Modell das Aufgabenverhalten aus diesen Beispielen implizit und ohne Parameteraktualisierungen lernen kann. Zusätzlich zum Few-Shot-Prompting kann man Chain-of-Thought-Prompting verwenden, um das Modell zu ermutigen, vor den endgültigen Antworten Zwischenschritte der Argumentation zu erzeugen. Es ist anzumerken, dass das Chain-of-Thought-Prompting eine der wichtigsten Methoden ist, um Probleme des logischen Schlussfolgerns anzugehen. Durch die explizite Bereitstellung von Zwischenschritten bei der Problemlösung kann man das Modell dazu anregen, komplexe Aufgaben in einfachere Teilaufgaben zu zerlegen, was sich als sehr vorteilhaft für die Erzeugung genauer und interpretierbarer Ausgaben erwiesen hat.

Über die Erweiterung des Prompts mit Beispielen oder Argumentationsschritten hinaus besteht ein weiterer Ansatz zur Kontextskalierung darin, externes Wissen dynamisch einzubeziehen. Dies wird oft durch RAG erreicht. RAG-Systeme rufen zunächst basierend auf der aktuellen Eingabe relevante Dokumentausschnitte aus einer großen Sammlung von Dokumenten oder einer Datenbank ab. Diese abgerufenen Informationen werden dann dem Kontext hinzugefügt, der dem LLM bereitgestellt wird. Dadurch wird der Kontext im Wesentlichen erweitert, um aktuelles oder spezialisiertes externes Wissen einzubeziehen. Dadurch stützt das Modell seine Antworten auf spezifisches Wissen, das in der externen Quelle gefunden wird. Das LLM kann somit Antworten generieren, die nicht nur für die

Eingabe relevant sind, sondern auch faktisch korrekt und aktuell.

Wenn der Kontext jedoch wächst, leiden diese Methoden oft unter den Beschränkungen der endlichen Kontextfensterlänge. Obwohl sich Modellarchitekturen und -techniken (wie effiziente Aufmerksamkeitsmodelle) kontinuierlich weiterentwickeln, um längere Kontexte zu unterstützen, stellt die Verarbeitung extrem langer Eingaben immer noch eine Herausforderung dar. Erhöhter Rechenaufwand ist ein Faktor. Kritischer ist, dass das Modell bei sehr großen Kontextfenstern Schwierigkeiten haben könnte, die relevantesten Informationen effektiv zu beachten (z. B. das „Lost in the Middle"-Phänomen). Daher geht es bei einer effektiven Kontextskalierung nicht nur darum, mehr Informationen hinzuzufügen, sondern auch darum, die relevantesten Informationen strategisch auszuwählen, zu strukturieren und innerhalb der Verarbeitungskapazitäten des Modells zu präsentieren.

Wir verzichten hier auf eine detaillierte Diskussion dieser Methoden, da sie bereits in früheren Kapiteln behandelt wurden. Siehe Kapitel 2 und 3 für weitere Details, einschließlich Prompting, RAG und Methoden zur Modellierung langer Sequenzen.

5.3.2 Suchskalierung

In LLMs ist die Dekodierung ein Suchprozess, der darauf abzielt, effizient die beste Ausgabesequenz für eine gegebene Eingabesequenz zu finden. Die Suchskalierung (oder Dekodierungsskalierung) umfasst typischerweise zwei Aspekte: die Skalierung der Ausgabelänge und die Skalierung des Suchraums.

Die Skalierung der Ausgabelänge bezieht sich auf die Erhöhung der Anzahl der während der Inferenz generierten Tokens. Dies ist besonders wichtig bei Aufgaben, die eine Langform-Generierung erfordern, wie zum Beispiel das Schreiben von Geschichten. In jüngerer Zeit hat die Generierung von Ausgaben mit langen Denkpfaden eine starke Leistung bei der Lösung mathematischer Probleme und der Codegenerierung gezeigt. Zum Beispiel hat es sich als sehr vorteilhaft für die Durchführung komplexen Schlussfolgerns erwiesen, das Modell zu ermutigen, lange Denkpfade zu generieren, bevor es die endgültigen Antworten produziert. Diese Idee wurde bei der Entwicklung neuerer LLMs für das Schlussfolgern weit verbreitet eingesetzt, wie zum Beispiel bei o1 von OpenAI [2024] und R1 von Deepseek [2025]. Wir werden die Skalierung der Ausgabelänge in Abschnitt 5.3.4 genauer erörtern.

Die Skalierung des Suchraums hingegen bezieht sich auf die Erweiterung der Menge der während der Suche berücksichtigten Kandidaten-Ausgabesequenzen, damit qualitativ hochwertigere Ausgaben gefunden werden können. Wie in Abschnitt 5.1.3 erläutert, ist ein einfaches Beispiel, dass wir bei der Beam-Suche die Strahlbreite erhöhen, um die parallele Erkundung von mehr Kandidatensequenzen bei jedem Dekodierungsschritt zu ermöglichen. Dies erhöht die Chance, bessere Ausgaben zu entdecken, insbesondere bei Aufgaben, bei denen die optimale Lösung nicht sofort aus lokalen Entscheidungen ersichtlich ist.

Zusätzlich zu Anpassungen des Dekodierungsalgorithmus ist es auch möglich, kompakte Strukturen zu untersuchen, um eine große Anzahl von Ausgaben zu kodieren. Zum Beispiel können wir einen Baum oder Graphen von Schlussfolgerungsschritten konstruieren und darin navigieren [Yao et al., 2024]. In diesem Paradigma repräsentiert jeder Knoten eine Teillösung oder einen Zwischenschritt, und Kanten repräsentieren Übergänge

zwischen den Zuständen des Schlussfolgerns. Eine solche strukturierte Suche ermöglicht es dem Modell, mehrere Pfade gleichzeitig zu berücksichtigen. Eine weitere verwandte Richtung ist die von der Monte-Carlo-Baumsuche inspirierte Dekodierung, bei der das Modell stochastisch verschiedene Pfade auf der Grundlage von gelernten Heuristiken oder externen Belohnungsmodellen untersucht und bewertet.

Die Suchskalierung ist eine sehr allgemeine Idee und ist oft implizit in der Gestaltung von Suchverfahren enthalten, die Suchstruktur, Heuristiken und Modellunsicherheit nutzen. Viele der oben genannten Methoden wurden bereits zuvor diskutiert, obwohl sie ursprünglich nicht mit der Skalierung als Hauptziel entwickelt wurden. Allerdings ist die Suchskalierung von Natur aus mit Rechenkosten verbunden. Die Erhöhung der Strahlbreite beispielsweise führt direkt zu einem höheren Speicherverbrauch und längeren Inferenzzeiten. In der Praxis gibt es oft einen Punkt abnehmender Erträge, an dem eine weitere Erweiterung des Suchraums nur geringfügige Verbesserungen der Ausgabequalität bei einem erheblichen Rechenaufwand mit sich bringt. Daher besteht eine effektive Strategie oft darin, ein optimales Gleichgewicht zwischen Skalierung und rechnerischer Machbarkeit zu finden.

5.3.3 Output-Ensembling

Wenn wir mehrere Modellausgaben haben, ist es oft vorteilhaft, diese zu kombinieren, um die Auswirkungen individueller Modellfehler zu mildern und eine überlegene endgültige Ausgabe zu synthetisieren. Jedes Modell könnte unterschiedliche Aspekte der zugrunde liegenden Datenverteilung erfassen oder einzigartige Stärken und Schwächen besitzen. Durch Ensembling können wir das Rauschen oder die zufälligen Fehler, die in einzelnen Vorhersagen vorhanden sind, ausmitteln, was zu einem stabileren und zuverlässigeren Ergebnis führt. Beim LLM-Ensembling besteht einer der einfachsten Ansätze darin, die Wahrscheinlichkeitsverteilungen über das nächste token von jedem Modell zu mitteln und das beste token unter Verwendung dieser gemittelten Verteilung auszuwählen. Oder, wenn wir das Problem als eine diskrete Entscheidungsaufgabe betrachten, kann eine Mehrheitsabstimmung eingesetzt werden. Anspruchsvollere Methoden könnten das Neuranordnen von Kandidatenausgaben, die von verschiedenen Modellen generiert wurden, basierend auf einer separaten Bewertungsfunktion oder sogar die Verwendung eines Meta-Lerners zur intelligenten Kombination der Vorhersagen beinhalten.

Die „Skalierung" durch Output-Ensembling geht mit den Kosten für den Betrieb mehrerer Modelle oder das Sampling mehrerer Ausgaben einher. Dies erhöht nicht nur die Latenz der Inferenz, sondern führt auch zu der zusätzlichen Komplexität der Verwaltung mehrerer Modelle. Aber die Qualität der Ausgaben verbessert sich nicht unbegrenzt, wenn mehr Modelle hinzugefügt werden. In einigen Fällen können die Vorteile des Output-Ensemblings abnehmen, wenn die Anzahl der Komponentenmodelle im Ensemble einen bestimmten Schwellenwert überschreitet. Stattdessen sind die Vorteile des Ensemblings im Allgemeinen größer, wenn die einzelnen Modelle divers sind (d. h., sie machen unterschiedliche Fehler), auch wenn es eine relativ geringe Anzahl von Komponentenmodellen gibt. Daher ist es gängige Praxis, einen Satz diverser LLMs zu verwenden, die sich in ihren Trainingsdaten, Modellarchitekturen oder Feinabstimmungszielen unterscheiden.

In LLMs bedeutet „Skalierung" oft, Dinge für mehr Qualität mit mehr Ressourcen

„größer" zu machen. Jedoch kann Skalierung neben der Steigerung der Qualität noch mehr bedeuten. Sie kann auch die Skalierung der Robustheit (das System weniger fehleranfällig und zuverlässiger zu machen) und der Exploration (das Abdecken eines breiteren Spektrums potenzieller Lösungen) bedeuten. Beim Output-Ensembling werden diese Dimensionen auf natürliche Weise integriert. Zum Beispiel ist der bloße Akt des Mittelns oder Abstimmens über verschiedene Modellausgaben hinweg eine direkte Strategie zur Skalierung der Robustheit gegenüber Ausfällen einzelner Modelle. Darüber hinaus erhöht das Ensembling durch die absichtliche Einbeziehung verschiedener Modelle die Chancen, neuartige oder überlegene Lösungen zu entdecken. In diesem Sinne ist Skalierung nicht darauf beschränkt, Modelle größer zu machen oder sie länger laufen zu lassen — es bedeutet auch Strategien, um die Inferenz robuster, explorativer und anpassungsfähiger zu machen.

5.3.4 Erzeugen und Verifizieren von Denkpfaden

Bisher haben wir die Inferenzzeit-Skalierung als eine allgemeine Klasse von Methoden zur Skalierung verschiedener Aspekte der Inferenz, wie Sequenzlänge, Modellgröße und/oder Suchstrategien, betrachtet. Tatsächlich ist eine erfolgreiche Anwendung die Verwendung der Inferenzzeit-Skalierung zur Verbesserung der Denkfähigkeiten von LLMs. Wie wir gesehen haben, kann die Denkleistung von LLMs durch die Verwendung von Chain-of-Thought-Methoden verbessert werden. Wir können daher die Chain-of-Thought-Prompts nutzen, um zwischengeschaltete Denkschritte zu erzeugen und eine korrekte Antwort zu erreichen. Allerdings sind Denkprobleme oft so kompliziert, dass wir durch die Bereitstellung einfacher Chain-of-Thought-Prompts keine hochwertigen Lösungen erhalten können. Zum Beispiel müssen wir beim Lösen eines mathematischen Problems typischerweise über eine Folge von Schritten schlussfolgern. In jedem Schritt müssen wir ein Zwischenergebnis erarbeiten, es verifizieren und dann bestimmen, was als Nächstes zu tun ist. Der Denkpfad ist kein festes Muster, sondern ein dynamisch erzeugter Denkprozess, der oft Versuch und Irrtum, Backtracking und Selbstkorrektur beinhaltet. Dies erfordert ausgefeiltere Prompting-Strategien oder Suchalgorithmen, um solch komplexes Denken zu navigieren. In diesem Unterabschnitt konzentrieren wir uns auf Methoden zur Inferenz-Skalierung, die über einfache Chain-of-Thought-Ansätze hinausgehen, um komplexe Denkprobleme effektiver zu lösen.

Auf einer übergeordneten Ebene können Methoden zur Skalierung des Denkens von LLMs in zwei Klassen eingeteilt werden:

- Trainingsfreie Methoden. Diese Methoden zielen darauf ab, die Schlussfolgerungsfähigkeiten zu verbessern, ohne dass eine Änderung oder ein erneutes Training der vortrainierten Parameter erforderlich ist. Stattdessen konzentrieren sie sich auf Techniken, die während der Inferenz angewendet werden, wie zum Beispiel ausgefeilte Prompting-Strategien (z. B. Chain-of-Thought) und algorithmische Kontrolle über den Schlussfolgerungsprozess (z. B. Suche).

- Trainingsbasierte Methoden. Diese Methoden beinhalten ein weiteres Training oder eine Feinabstimmung der Modellparameter, um die Schlussfolgerungsfähigkeiten explizit zu verbessern, wie zum Beispiel überwachte Feinabstimmung auf Datensätzen

mit Schlussfolgerungsbeispielen (z. B. mathematische Probleme mit schrittweisen Lösungen).

Im Folgenden diskutieren wir zuerst trainingsfreie Methoden und anschließend trainingsbasierte Methoden.

5.3.4.1 Suche auf Lösungsebene mit Verifizierern

Für eine gegebene Eingabesequenz (z. B. ein mathematisches Problem) gibt es viele mögliche Ausgabesequenzen (z. B. Lösungen für das Problem). Wenn wir ein Modell haben, um jede Lösung zu bewerten oder zu verifizieren, können wir die beste auswählen. Dies ist das grundlegende Prinzip hinter Methoden wie dem Best-of-N-Sampling, bei dem mehrere Ausgaben generiert werden und das optimale Ergebnis anhand eines Auswahlmechanismus ausgewählt wird. Ein solcher Auswahlprozess kann als Suchproblem betrachtet werden, das zwei Komponenten umfasst:

- Suchalgorithmus. Dieser definiert die Strategie, die verwendet wird, um den Raum möglicher Ausgabesequenzen (Lösungen) zu erkunden und einen Satz von Kandidaten zu generieren. Er kann von einfacher unabhängiger Stichprobenziehung bis hin zu komplexeren Suchtechniken reichen, wie in Abschnitt 5.1.3 erläutert.
- Verifizierer. Dies ist ein Modell oder eine Funktion, die für die Bewertung der Qualität, Korrektheit oder Nützlichkeit jeder vom Suchalgorithmus generierten Kandidatenlösung verantwortlich ist. Er liefert eine Bewertung, eine Wahrscheinlichkeit oder ein Urteil, das es dem System ermöglicht, den besten unter den Kandidaten auszuwählen. Der Verifizierer kann ein anderes LLM sein oder sogar ein Satz vordefinierter Regeln oder Heuristiken.

Für ein gegebenes Eingangsproblem $\mathbf{x}$ definieren wir, dass eine Ausgabelösung $\mathbf{y}$ als eine Sequenz von Schlussfolgerungsschritten dargestellt werden kann:

$$\mathbf{y} = (a_1, a_2, ..., a_{n_r}) \tag{5.37}$$

wobei a_i der i-te Schlussfolgerungsschritt ist und a_{n_r} der letzte Schritt ist, der die Antwort auf das Problem enthalten sollte. Siehe Abbildung 5.15 für ein Beispiel eines mehrstufigen Schlussfolgerungspfades.

Der Suchalgorithmus kann effizient einen Satz von Kandidatenlösungen generieren

$$\mathcal{D}_c = \{\mathbf{y}_1, ..., \mathbf{y}_K\} \tag{5.38}$$

Dann können wir einen Verifizierer verwenden, der jede Lösung durch die Funktion $V(\mathbf{y})$ bewertet, um die Kandidaten in $\mathcal{D}_c$ zu bewerten. Die endgültige Ausgabe ist der beste vom Verifizierer ausgewählte Kandidat

$$\hat{\mathbf{y}} = \arg\max_{\mathbf{y} \in \mathcal{D}_c} V(\mathbf{y}) \tag{5.39}$$

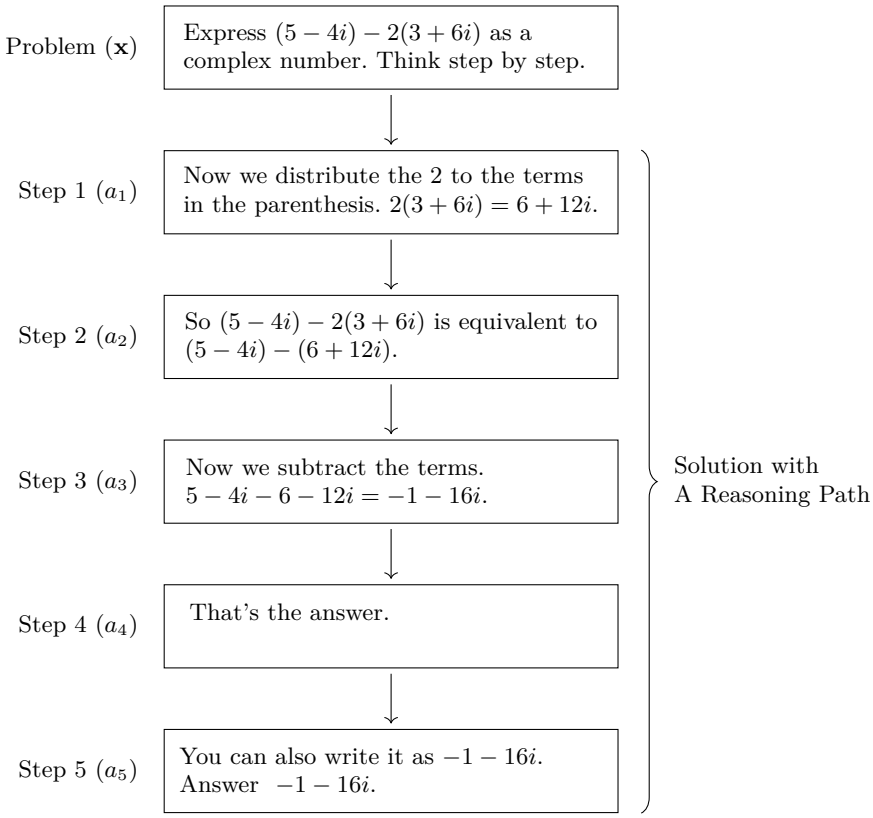


Abbildung 5.15: Illustration des mehrschrittigen Schlussfolgerns. Dieses Beispiel stammt aus dem PRM800K-Datensatz [Lightman et al., 2024]. Bei einem gegebenen mathematischen Problem wird das LLM aufgefordert, einen Denkpfad (oder Schlussfolgerungspfad) zu generieren, der aus mehreren Schlussfolgerungsschritten besteht. Jeder Schritt behandelt ein Teilproblem basierend auf den Ergebnissen der vorherigen Schritte. Die Antwort auf das ursprüngliche Problem ist im letzten Schritt enthalten.

Obwohl die Überprüfung des gesamten Schlussfolgerungspfades möglich ist, besteht eine einfachere Alternative darin, nur den letzten Schlussfolgerungsschritt zu überprüfen. Auf diese Weise wird die Verifiziererfunktion $V(\mathbf{y})$ vereinfacht, sodass sie nur von der in a_{n_r} enthaltenen endgültigen Antwort abhängt. Dies kann auf verschiedene Weisen erreicht werden, abhängig von der Art des Problems und dem erwarteten Antwortformat.

- Für einige Mathematik- und Programmierprobleme können wir handelsübliche Werkzeuge als Verifizierer verwenden. Beispiele hierfür sind Beweisprüfer für mathematische Theoreme, Interpreter oder Compiler zur Codeausführung und Unit-Test-Systeme zur Überprüfung der Programmkorrektheit anhand vordefinierter Testfälle.

- Wenn zur Bewertung der Antwort gelabelte Daten vorhanden sind, wie zum Beispiel menschliche Präferenzdaten, können wir auf solchen Daten ein Belohnungsmodell trainieren. Das gelernte Belohnungsmodell wird dann als Verifizierer verwendet, der jeder Kandidatenantwort einen skalaren Wert zuweist.

- Wenn keine bestehenden Systeme oder geeigneten Belohnungsmodelle vorhanden sind, können wir ein anderes LLM als Verifizierer einsetzen. Dieses LLM wird dazu veranlasst, die Qualität der Kandidatenantwort zu bewerten. Dies könnte potenziell ein leistungsfähigeres Modell sein oder dasselbe LLM, das mit einem spezifischen

„evaluator"-Prompt verwendet wird.

- Alternativ können einfachere, auf Heuristiken basierende Verifizierer entworfen werden. Ein häufig verwendeter Ansatz ist die Anwendung der Mehrheitsabstimmung, bei der die am häufigsten vorkommende Antwort aus einer Reihe von Kandidaten ausgewählt wird.

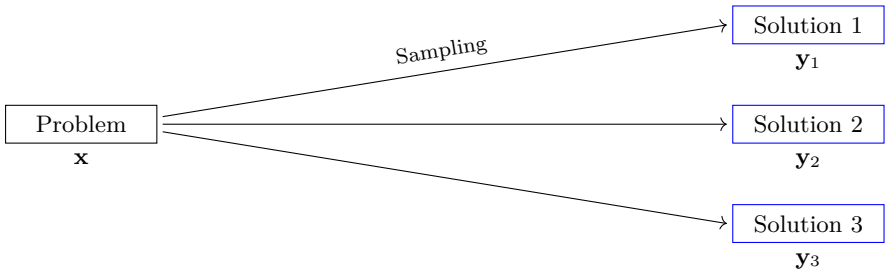
Basierend auf diesen Verifizierern können wir suchen, um einen Satz von Kandidatenlösungen für die Auswahl zu erhalten. Eine einfache Strategie, die oft als parallel scaling [Brown et al., 2024; Snell et al., 2024] bezeichnet wird, besteht darin, K Kandidatenlösungen zu generieren, indem das Basis-LLM K Mal unabhängig voneinander ausgeführt wird. In diesem Prozess können wir die Temperatur beim Sampling anpassen, um die Vielfalt der Ausgaben zu steuern. Der Verifizierer bewertet dann jede dieser K vollständigen Lösungen, und diejenige mit der höchsten Punktzahl wird als endgültige Ausgabe ausgewählt. Dies ist konzeptionell dem Best-of-N-Sampling sehr ähnlich, das wir in früheren Kapiteln hauptsächlich als eine Methode beschrieben haben, um die beste aus einer Reihe von gesampelten Ausgaben mithilfe eines Belohnungsmodells auszuwählen.

Ein anderer Ansatz ist die sequential scaling, bei der eine Sequenz von Lösungen inkrementell aufgebaut wird [Gou et al., 2024; Zhang et al., 2024]. Es beginnt mit einer anfänglichen Lösung, die vom LLM mit Prompting generiert wird. Dann verwenden wir einen Verifizierer (oft dasselbe LLM), um die Lösung zu bewerten. Dies kann als Kritikphase betrachtet werden. Die Ausgabe dieser Phase ist eine Form von Feedback, wie z. B. textuelle Kritiken, die Fehler aufzeigen oder Verbesserungen vorschlagen, numerische Bewertungen, die die Lösungsqualität widerspiegeln, oder sogar ein überarbeiteter Plan oder ein Zwischenschritt, um die nächste Generation anzuleiten. Dieses Feedback wird zusammen mit dem ursprünglichen Problem und der aktuellen Lösung verwendet, um das LLM aufzufordern, eine potenziell verbesserte Lösung zu generieren. Dies kann als Verfeinerungsphase betrachtet werden. Dieser Kritik-Verfeinerungs-Zyklus kann wiederholt werden und bildet eine iterative Schleife:

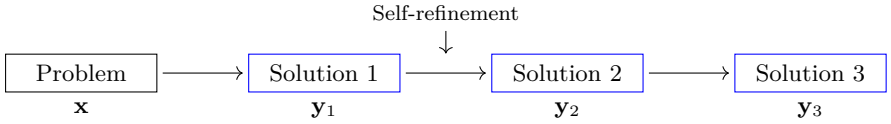
$$\mathbf{y}_{k+1} = \text{Refine}(\mathbf{x}, \mathbf{y}_k, \text{Feedback}(\mathbf{y}_k)) \tag{5.40}$$

wobei $\text{Feedback}(\mathbf{y}_k)$ das Feedback des Verifizierers darstellt. Die Funktion $\text{Refine}(\cdot)$ generiert die verbesserte Lösung $\mathbf{y}_{k+1}$, indem sie das LLM mit dem ursprünglichen Problem $\mathbf{x}$, der vorherigen Lösung $\mathbf{y}_k$ und diesem Feedback ansteuert. Der Prozess kann K Mal wiederholt werden, oder bis die Lösungsqualität, wie vom Verifizierer bewertet, ein zufriedenstellendes Niveau erreicht. Dieses iterative Framework, bei dem eine Lösung durch Zyklen der Generierung, Bewertung (Kritik) und Überarbeitung schrittweise verbessert wird, ist genau das, was Selbstverfeinerung ausmacht [Shinn et al., 2023; Madaan et al., 2024]. In solchen Szenarien besteht die Rolle des Verifizierers nicht nur darin, die beste vollständige Lösung aus einem statischen Satz auszuwählen, sondern den Generierungsprozess selbst aktiv zu steuern.

Siehe Abbildung 5.16 für Illustrationen von paralleler und sequenzieller Skalierung. Beachten Sie, dass es andere Möglichkeiten gibt, eine Suche durchzuführen und verschiedene Sätze von Kandidatenlösungen zu erhalten. Eine alternative Methode besteht darin,



(a) Parallel Scaling



(b) Sequential Scaling

Abbildung 5.16: Illustrationen der parallelen und sequenziellen Skalierung. Bei der parallelen Skalierung werden mehrere Lösungen durch mehrmaliges, unabhängiges Ausführen des LLM erhalten. Bei der sequenziellen Skalierung generiert das LLM eine anfängliche Lösung. Anschließend wird das LLM verwendet, um diese iterativ zu verfeinern, wobei jede Verfeinerung eine neue, möglicherweise bessere Lösung ergibt.

die Suche als Baumstruktur zu organisieren. Dieser Ansatz, oft als Baumsuche bezeichnet, bietet eine strukturiertere Möglichkeit, den Raum möglicher Schlussfolgerungspfade zu erkunden. Bei der Suche auf Lösungsebene repräsentiert jeder Knoten des Baumes eine vollständige Lösung. Während der Suche müssen wir einen Knoten zu einem Satz von Kindknoten erweitern, die neue Lösungen repräsentieren, die bei der Verifizierung berücksichtigt werden können. Der Erweiterungsprozess beinhaltet typischerweise das Nehmen einer bestehenden Lösung (des Elternknotens) und die Verwendung des LLM, um Variationen oder alternative Lösungen zu generieren.

5.3.4.2 Suche auf Schrittebene mit Verifizierern

Während sich die oben diskutierten Methoden hauptsächlich auf die Generierung vollständiger Lösungen vor der endgültigen Auswahl konzentrieren, kann der Suchprozess auch tiefer in die schrittweise Erzeugung des Argumentationspfades selbst integriert werden. Dies führt zu Ansätzen, die eine Suche auf Schrittebene mit Verifizierern durchführen, bei der die Steuerung oder das Pruning bei den Zwischenschritten der Argumentation $\{a_1, ..., a_{n_k}\}$ stattfindet und nicht erst, nachdem eine vollständige Lösung $\mathbf{y}$ gebildet wurde.

Eine solch feingranulare Steuerung ist besonders vorteilhaft bei komplexen Argumentationsproblemen, bei denen ein einziger falscher Zwischenschritt die gesamte nachfolgende Argumentationskette ungültig machen kann. Durch die Bewertung oder Steuerung der Generierung bei jedem Zwischenschritt kann das LLM den Argumentationsraum effektiver erkunden, potenziell wenig aussichtsreiche Pfade frühzeitig beschneiden oder mehr Ressourcen für die Erkundung plausiblerer Pfade zuweisen.

Die Suche auf Schrittebene mit Verifizierern kann auch als ein Baumsuche-Problem

modelliert werden. In diesem Paradigma entspricht jeder Knoten (oder Zustand) einem partiellen Argumentationspfad, $\mathbf{a}_{\leq i} = (a_1, ..., a_i)$, der die Sequenz der bisher durchgeführten i Argumentationsschritte darstellt (d. h. ein Pfad vom Wurzelknoten zum aktuellen Knoten). Das Ziel des Suchprozesses ist es, den zugrunde liegenden Zustandsraum zu erkunden, beginnend mit einem anfänglich leeren Pfad, um einen vollständigen Pfad zu finden, der eine korrekte Lösung darstellt. Beachten Sie, dass wir hier $\mathbf{a}_{\leq i}$ verwenden, um einen partiellen Argumentationspfad anstelle von $\mathbf{y}_{\leq i}$ darzustellen. Obwohl dies die Notation etwas inkonsistent mit der für die Darstellung vollständiger Lösungen ($\mathbf{y}$) oder vollständiger Pfade bei der Suche auf Lösungsebene verwendeten macht, dient es dazu, den Fokus auf einzelne Aktionen oder Schritte zu legen.

Die Kernkomponenten der Suche auf Schrittebene mit Verifizierern sind:

- Knotenrepräsentation. Ein Knoten ist ein partieller Argumentationspfad $\mathbf{a}_{\leq i} = (a_1, ..., a_i)$. Der Wurzelknoten ist ein leerer Pfad, und Endknoten sind vollständige Argumentationspfade.

- Knotenerweiterung. Gegeben ein aktueller partieller Pfad $\mathbf{a}_{\leq i}$, wird das LLM verwendet, um einen oder mehrere Kandidaten für die nächsten Argumentationsschritte $\{a_{i+1}^{(1)}, ..., a_{i+1}^{(M)}\}$ zu generieren. Jeder Kandidatenschritt, wenn an $\mathbf{a}_{\leq i}$ angehängt, bildet einen neuen potenziellen partiellen Pfad $\mathbf{a}_{\leq i+1} = (a_1, ..., a_i, a_{i+1}^{(j)})$.

- Verifizierung. Der Verifizierer $V(\cdot)$ bewertet die Qualität eines neu generierten Schritts im Kontext des aktuellen partiellen Pfads $\mathbf{a}_{\leq i} = (a_1, ..., a_i)$ und des ursprünglichen Problems $\mathbf{x}$. Wie bei der Verifizierung auf Lösungsebene können Verifizierer auf Schrittebene eine numerische Bewertung, eine kategoriale Kennzeichnung und textuelles Feedback ausgeben.

- Suche. Dies steuert, wie der Suchraum erkundet wird. Basierend auf den Bewertungen des Verifizierers entscheidet die Suchstrategie, welche partiellen Pfade weiter ausgebaut, welche verworfen werden und in welcher Reihenfolge die Erkundung stattfindet.

Diese schrittweise Verifizierung ermöglicht dynamische Anpassungen des Argumentationsprozesses. Wenn ein Schritt a_{i+1} von $V(\cdot)$ als falsch oder wenig aussichtsreich eingestuft wird, kann der Suchalgorithmus zurückverfolgen und alternative Schritte von $\mathbf{a}_{\leq i}$ aus erkunden, oder sogar von einem früheren Knoten $\mathbf{a}_{\leq i'}$ (wobei $i' < i$). Umgekehrt können, wenn ein Schritt hoch bewertet wird, Ressourcen darauf konzentriert werden, diesen Pfad zu erweitern. Siehe Abbildung 5.17 für eine Veranschaulichung der Suche auf Schrittebene mit Verifizierern.

Offensichtlich ist dieses Such-Framework dem sehr ähnlich, das bei Dekodierungsmethoden für LLMs verwendet wird, wie in Abschnitt 5.1.3 erläutert. Zum Beispiel hält die Beam-Suche bei jedem Generierungsschritt einen Satz von K der vielversprechendsten partiellen Sequenzen aufrecht. Dies ist eine Form der Suche auf Schrittebene, bei der der „Verifizierer“implizit das Wahrscheinlichkeitsmodell des LLM selbst ist und die „Suche “der Pruning-Mechanismus ist, um die Strahlgröße beizubehalten.

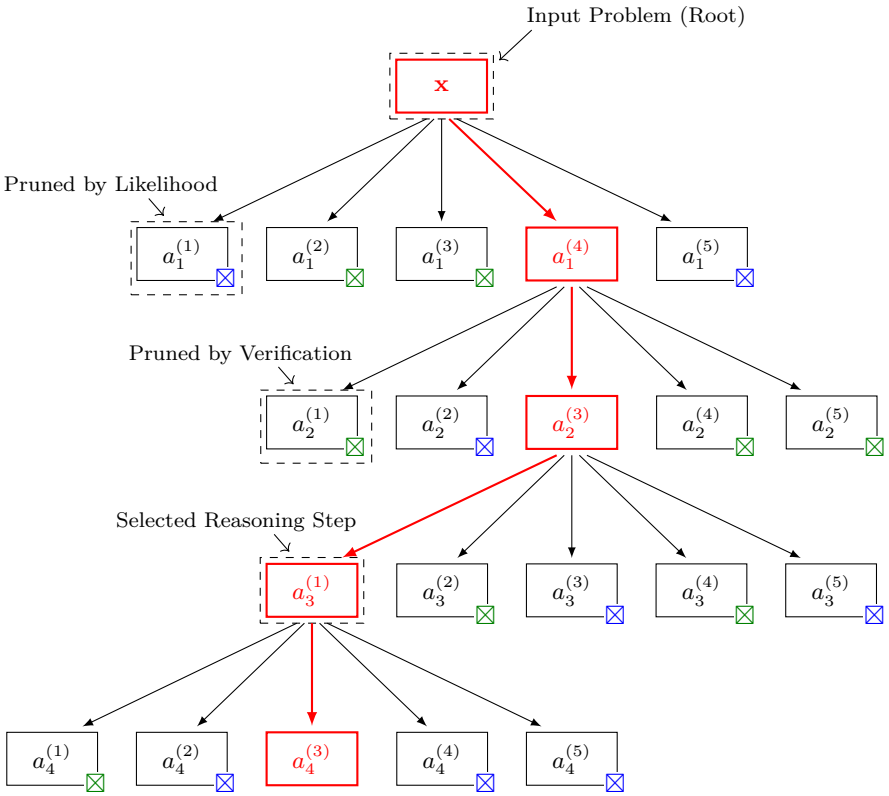


Abbildung 5.17: Darstellung der Suche auf Schrittebene mit Verifizierern. $a_i^{(j)}$ = der j-te Kandidat für den i-ten Schlussfolgerungsschritt, ☒ = Kandidat, der durch die Ausgabewahrscheinlichkeit des LLM verworfen wurde, und ☒ = Kandidat, der vom Verifizierer verworfen wurde. Ausgehend vom Eingabeproblem als Wurzelknoten erweitern wir den Baum, indem wir bei jeder Erweiterung mehrere Schlussfolgerungsschritte generieren. Jeder Kandidat kann entweder durch die Wahrscheinlichkeit (wie bei der Standard-Dekodierung) oder durch die Verifizierung auf Schrittebene verworfen werden. Die nicht verworfenen Kandidaten werden dann erweitert, um weitere Schlussfolgerungsschritte zu generieren. Der Prozess wird wiederholt, bis eine vollständige Schlussfolgerungskette, die zu einer endgültigen Antwort führt, generiert ist oder bis eine vordefinierte Suchgrenze erreicht ist.

Jedoch weist die hier beschriebene Suche auf Schrittebene mit expliziten Verifizierern Unterschiede zur Standard-Dekodierung auf. Einer davon ist, dass der Verifizierer eine weitaus anspruchsvollere Komponente sein kann als nur die rohen Ausgabewahrscheinlichkeiten des generativen LLM. Das Design von Verifizierern auf Schrittebene folgt im Wesentlichen dem der Verifizierung auf Lösungsebene. Ein Verifizierer auf Schrittebene könnte ein Sprachmodell sein, das die Qualität eines einzelnen Argumentationsschrittes im Kontext des vorangehenden Pfades bewertet. Dieses LLM kann sogar einer Feinabstimmung unterzogen werden, um seine Verifizierungsfähigkeit zu verbessern. Alternativ könnte es für Domänen mit gut definierten Regeln eine symbolische Engine oder ein Satz programmatischer Prüfungen sein. Darüber hinaus können Verifizierer so konzipiert werden, dass sie den zukünftigen Nutzen oder die Erfolgswahrscheinlichkeit bei gegebenem partiellem Pfad vorhersagen, wobei sie sich von Wertefunktionen im Reinforcement Learning inspirieren lassen. Menschliche Expertise kann ebenfalls einbezogen werden, um Urteile über kritische Schritte abzugeben, insbesondere in Szenarien mit hohem Einsatz.

Ein Beispiel für einen solchen Verifizierer auf Schrittebene, insbesondere bei der Verwendung von menschlichem Feedback zur Bewertung des Zwischenfortschritts, ist das prozessbasierte Belohnungsmodell (PRM). Ein PRM ist typischerweise ein separates Sprachmodell, das darauf trainiert ist, für jeden Argumentationsschritt $a_{i'}$ innerhalb eines partiellen Pfades $\mathbf{a}_{\leq i}$ eine skalare Belohnung auszugeben. Es liefert ein direkteres und feingranulareres überwachendes Signal im Vergleich zu ergebnisbasierten Belohnungsmodellen (ORMs), die nur die endgültige Lösung bewerten. Die Entwicklung von PRMs stützt sich jedoch auf menschliche Annotationen auf Schrittebene, wie z. B. Präferenzen für verschiedene nächste Schritte. Das Sammeln von Überwachungsdaten für jeden Zwischenschritt ist erheblich arbeitsintensiver und erfordert einen größeren kognitiven Aufwand von menschlichen Annotatoren als das bloße Kennzeichnen von Endergebnissen.

Ein alternativer Ansatz zur Entwicklung von Trainingsdaten für die Verifizierung auf Schrittebene besteht darin, LLMs zu verwenden, um solche Annotationen automatisch zu generieren. Zum Beispiel können wir ein starkes LLM, das als Lehrermodell bezeichnet wird, nehmen und es anweisen, zuerst einen vollständigen Argumentationspfad für ein gegebenes Problem zu generieren. Dann können wir bei jedem Zwischenschritt innerhalb dieses Pfades dasselbe Lehrermodell (oder ein anderes fähiges LLM) anweisen, zusätzlich zu dem ursprünglich gewählten Schritt mehrere alternative Kandidaten für den nächsten Schritt zu generieren. Das Lehrermodell kann dann erneut angewiesen werden, diese Alternativen zu bewerten. Diese Bewertungsergebnisse (z. B. richtig vs. falsch) können dann als Datenannotationen dienen. Alternativ können die Generalisierungsfähigkeiten von PRMs genutzt werden. Wir können ein PRM für Aufgaben trainieren, bei denen die Verifizierung auf Schrittebene einfacher ist, und dieses PRM dann mit wenig oder keinem zusätzlichen Training auf andere Aufgaben generalisieren.

Beachten Sie, dass die Verifizierung auf Schrittebene auch ihre eigenen Probleme mit sich bringt. Häufige Verifizierung, insbesondere bei Verwendung eines LLM als Verifizierer, kann die Berechnungskosten und die Latenz erheblich erhöhen. Das Design effektiver Verifizierer auf Schrittebene ist selbst nicht trivial. Ein ungenauer Verifizierer könnte gute Argumentationspfade vorzeitig verwerfen oder fehlerhafte nicht erkennen und dadurch die Suche in die Irre führen. Dies macht die Entwicklung solcher Systeme komplexer und schwieriger.

5.3.4.3 Förderung des langen Nachdenkens

Bisher basieren die meisten Methoden in diesem Unterabschnitt implizit auf einer einfachen Idee: Das Generieren längerer Argumentationspfade kann helfen. Zusätzlich zu CoT und der Suche mit Verifizierungen können wir alternative Methoden in Betracht ziehen, um dies zu erreichen. Zum Beispiel können wir das LLM auffordern, indem wir explizit um eine ausführliche Erörterung bitten. Über das direkte Prompting hinaus können wir auch Änderungen am Dekodierungsprozess selbst vornehmen, wie z. B. das Anpassen von Token-Grenzen oder das Anwenden von Strafen für kurze Ausgaben. Ein weiterer Ansatz ist die Verwendung von mehrstufigen Generierungsschemata, bei denen das Modell schrittweise auf seiner Argumentation aufbaut.

5.3.4.4 Trainingsbasierte Skalierung

Neben der Betrachtung von Inferenzzeit-Skalierungsmethoden ohne Training möchten wir auch Methoden in Betracht ziehen, die die intrinsischen Schlussfolgerungsfähigkeiten von LLMs durch die Modifikation ihrer Parameter mittels weiterem Training verbessern können. Obwohl solche trainingsbasierten Skalierungsmethoden typischerweise zusätzliche Trainingskosten und Rechenressourcen erfordern, verleihen sie den Modellparametern direkt stärkere Schlussfolgerungsfähigkeiten, was wiederum zu einer effektiveren und effizienteren Schlussfolgerungsleistung führen kann. Wir können sie sogar mit trainingsfreien Methoden kombinieren, um bessere Ergebnisse bei der Inferenzzeit-Skalierung zu erzielen.

Obwohl sich unsere Diskussion hier auf Schlussfolgerungsprobleme beschränkt, sind Methoden zur trainingsbasierten Skalierung weit verbreitet. Die meisten von ihnen wurden in Kapitel 4behandelt. Hier werden wir kurz beschreiben, wie diese Methoden zur Verbesserung der Inferenzzeit-Skalierung für Schlussfolgerungsprobleme angewendet werden können.

- Feinabstimmung auf Schlussfolgerungsdaten. Eine der direktesten Methoden zur Verbesserung von Schlussfolgerungen ist die Feinabstimmung vortrainierter LLMs auf Datensätzen, die speziell für Schlussfolgerungsaufgaben zusammengestellt wurden. Diese Datensätze können von einfachen Eingabe-Ausgabe-Paaren bis hin zu strukturierteren Formaten reichen, die schrittweise Schlussfolgerungsprozesse beinhalten. Typische Beispiele umfassen Datensätze mit mathematischen Textaufgaben, Übungen zur logischen Deduktion oder Codegenerierung mit Erklärungen. Durch das Training mit solchen Daten lernt das Modell aus gängigen Schlussfolgerungsmustern und kann so zur Testzeit detaillierte und kohärente Schlussfolgerungspfade generieren.

- Verstärkungslernen für Schlussfolgerungen. Wenn wir einen Verifizierer als ein Belohnungsmodell betrachten, können wir erkennen, dass die im vorherigen Unterabschnitt besprochenen Methoden eine direkte Anwendung des Belohnungsmodells auf Schlussfolgerungsprobleme sind, obwohl sie trainingsfrei sind. Selbstverständlich können wir dieses Belohnungsmodell auf die Feinabstimmung von LLMs anwenden. Dies folgt einem Standardparadigma des Verstärkungslernens. Gegeben ein Belohnungsmodell wird das LLM, das als Policy agiert, mithilfe von Algorithmen des Verstärkungslernens feingestimmt. Das LLM generiert Schlussfolgerungsschritte oder vollständige Lösungen, erhält Rückmeldung (Belohnungen) vom Belohnungsmodell und aktualisiert seine Parameter, um Ausgaben zu erzeugen, die diese Belohnungen maximieren. Dieser Prozess gleicht die Ausgabe des LLM an Vorstellungen von qualitativ hochwertigen Schlussfolgerungen an und ermutigt das LLM dadurch, zuverlässigere Schlussfolgerungspfade zu generieren. Ein weiteres zentrales Thema ist das Training des Belohnungsmodells. Im Allgemeinen kann dieses Belohnungsmodell ein Ergebnis-Belohnungsmodell sein, das die Korrektheit oder Qualität der endgültigen Antwort bewertet, oder ein Prozess-Belohnungsmodell, das die Qualität jedes zwischengeschalteten Schlussfolgerungsschritts beurteilt, wie im Kontext von schrittweisen Verifizierern diskutiert. In einigen Fällen können wir sogar ein Belohnungsmodell entwickeln, das auf einfachen Regeln basiert, wie zum Beispiel die Vergabe von Boni für längere Ausgaben.

- Wissensdestillation für Schlussfolgerungen. Bei diesem Ansatz wird ein kleineres, effizienteres Schüler-LLM trainiert, um die Schlussfolgerungsausgaben oder internen Repräsentationen eines größeren, leistungsfähigeren Lehrer-LLM nachzuahmen. Das Lehrermodell könnte detaillierte Schlussfolgerungsschritte für eine Vielzahl von Problemen generieren. Das Schülermodell lernt dann, diese qualitativ hochwertigen Schlussfolgerungsdemonstrationen zu reproduzieren. Diese Strategie macht stärkere Schlussfolgerungsfähigkeiten zugänglicher, indem sie in kleineren Modellen eingesetzt werden, die zur Inferenzzeit rechengünstiger sind.

- Iterative Verfeinerung. Trainingsbasiertes Skalieren kann auch eine iterative Verfeinerung beinhalten. Zum Beispiel kann ein LLM Lösungen für eine Reihe von Problemen generieren. Diese Lösungen und ihre Schlussfolgerungspfade werden dann entweder von Menschen oder automatischen Verifizierern überprüft. Die korrekten Schlussfolgerungspfade werden anschließend den Trainingsdaten hinzugefügt, und das LLM wird auf diesem erweiterten Datensatz weiter feingestimmt. Dies schafft einen Zyklus, in dem das LLM seine Schlussfolgerungsfähigkeiten durch wiederholte Generierung, Kritik und Lernen schrittweise verbessert.

Der Hauptvorteil dieser trainingsbasierten Skalierungsmethoden besteht darin, dass sie dem LLM stärkere inhärente Schlussfolgerungsfähigkeiten verleihen. Dies trägt auf verschiedene Weisen direkt zu einer verbesserten Inferenzzeit-Skalierung bei: Es kann zu einer effizienteren Inferenz führen, da das LLM möglicherweise weniger umfangreiche Suchen oder weniger Generierungsstichproben benötigt, um eine korrekte Lösung zu finden. Darüber hinaus ist die grundlegende Qualität der generierten Schritte oder Lösungen höher. Daher kann ein gut trainiertes LLM seine erlernten Schlussfolgerungsfähigkeiten effektiver auf neuartige Probleme verallgemeinern als ein LLM, das sich ausschließlich auf In-Context-Lernen oder trainingsfreie Inferenzschemata stützt.

Andererseits stellen trainingsbasierte Ansätze im Vergleich zu den trainingsfreien Pendants auch Herausforderungen dar. Die Erstellung hochwertiger, umfangreicher Trainingsdatensätze für das Schlussfolgern kann kostspielig und arbeitsintensiv sein. Der Feinabstimmungsprozess selbst, insbesondere bei den größten LLMs oder bei der Verwendung von RL, kann rechenintensiv sein und erheblichen technischen Aufwand erfordern. Es besteht auch das Risiko der Überanpassung (Overfitting) des Modells an die spezifischen Arten von Problemen oder Schlussfolgerungsstilen, die in den Trainingsdaten vorhanden sind, was seine Leistung bei Out-of-Distribution-Aufgaben potenziell einschränken kann.

5.4 Zusammenfassung

In diesem Kapitel haben wir das Inferenzproblem für LLMs diskutiert. Wir haben das Prefilling-Decoding-Framework und verwandte Dekodierungsalgorithmen für die LLM-Inferenz vorgestellt. Anschließend haben wir verschiedene Techniken für eine effiziente Inferenz beschrieben. Wir haben auch die Skalierung zur Inferenzzeit erörtert, die als eine der wichtigsten Methoden zur Verbesserung des logischen Denkens von LLMs angesehen wird.

Die Inferenz über sequentielle Daten ist seit langem ein Thema in der KI [Wozengraft

and Reiffen, 1961; Viterbi, 1967; Forney, 1972]. Im Kontext von NLP reicht dieser Forschungszweig bis in die Anfänge der Spracherkennung und der statistischen maschinellen Übersetzung zurück [Koehn, 2010], wo Forscher vor der Herausforderung standen, riesige Hypothesenräume effizient zu durchsuchen, um die wahrscheinlichste Ausgabesequenz zu finden. Techniken wie die Beam-Search und verschiedene Pruning-Strategien wurden damals entwickelt, um dies rechnerisch handhabbar zu machen. Zu dieser Zeit waren die Modelle relativ schwach, und ein Großteil der Forschung konzentrierte sich auf die Entwicklung leistungsfähiger Suchalgorithmen, um Suchfehler zu reduzieren. Diese grundlegenden Ideen beeinflussen weiterhin moderne Ansätze.

Mit dem Eintritt in die Ära, die von Deep-Learning-Methoden dominiert wird, sind Modelle, die auf tiefen neuronalen Netzen basieren, extrem leistungsfähig geworden. Selbst mit sehr einfachen Suchalgorithmen können diese Modelle hervorragende Ergebnisse erzielen. In diesem Kontext scheint die Inferenz nicht mehr so „wichtig“ wie früher zu sein, und die Forschungsaufmerksamkeit hat sich allmählich auf Modellarchitekturen, Trainingsmethoden und die Skalierung von Modellen verlagert.

Die Geschichte neigt jedoch dazu, sich zu wiederholen. Mit dem Aufkommen von LLMs hat die Inferenz erneut erhebliche Aufmerksamkeit auf sich gezogen. Dieser erneute Fokus manifestiert sich hauptsächlich in zwei Aspekten:

- Die Inferenzkosten für LLMs sind sehr hoch. Beispielsweise bleibt der effiziente Einsatz von LLMs in Szenarien mit hoher Gleichzeitigkeit und geringer Latenz ein herausforderndes Problem, was die Effizienz der Inferenz von entscheidender Bedeutung macht. In diesem Zusammenhang haben effiziente Architekturentwürfe, optimierte Suchalgorithmen und verschiedene Strategien zur Inferenzoptimierung eine erhebliche praktische Bedeutung.

- Die Längen der Eingabe- und Ausgabesequenzen haben bei komplexen Aufgaben erheblich zugenommen. Insbesondere bei Aufgaben wie dem mathematischen Schlussfolgern unterstreicht das Wachstum der Sequenzlängen die Bedeutung der Inferenz-Effizienz zusätzlich. Darüber hinaus hat sich die Skalierung des Inferenzprozesses kürzlich als effektiver Weg zur Verbesserung der Reasoning-Fähigkeiten von Modellen erwiesen. Daher entwickelt sich die Erzielung einer effizienten Inferenzskalierung zu einer besonders vielversprechenden Forschungsrichtung.

Inferenz ist heute ein weitreichendes Thema, das viele Techniken umfasst. Es umfasst nicht nur die Entwicklung von Modellarchitekturen und Dekodierungsalgorithmen, sondern wird zunehmend durch das komplexe Engineering und die anspruchsvollen Optimierungen auf Systemebene geprägt, die für den effektiven und effizienten Einsatz von LLMs erforderlich sind. Viele dieser Techniken gehen über den Rahmen von NLP oder eines bestimmten KI-Bereichs hinaus. Stattdessen erstreckt sich die Grenze der LLM-Inferenzoptimierung nun tief in Bereiche, die traditionell als Kernbereiche der Informatik und des Ingenieurwesens gelten. Diese systemische Perspektive hat viele neue Ideen in die Untersuchung von Inferenzproblemen eingebracht. Leider kann dieses Kapitel nicht alle relevanten Techniken abdecken — das wäre in der Tat eine fast unmögliche Aufgabe für sich. Letztendlich liegt der beste Weg, diese Techniken besser zu verstehen und zu beherrschen, möglicherweise immer noch in der praktischen Anwendung.

Literaturverzeichnis

[Agrawal et al., 2023] Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. Sarathi: Efficient llm inference by piggybacking decodes with chunked prefills. arXiv preprint arXiv:2308.16369, 2023.

[Agrawal et al., 2024] Amey Agrawal, Nitin Kedia, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav Gulavani, Alexey Tumanov, and Ramachandran Ramjee. Taming {Throughput-Latency} tradeoff in {LLM} inference with {Sarathi-Serve}. In 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), pages 117–134, 2024.

[Ainslie et al., 2020] Joshua Ainslie, Santiago Ontanon, Chris Alberti, Vaclav Cvicek, Zachary Fisher, Philip Pham, Anirudh Ravula, Sumit Sanghai, Qifan Wang, and Li Yang. Etc: Encoding long and structured inputs in transformers. In Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP), pages 268–284, 2020.

[Ainslie et al., 2023] Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebron, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. In Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, pages 4895–4901, 2023.

[Akyürek et al., 2023] Ekin Akyürek, Dale Schuurmans, Jacob Andreas, Tengyu Ma, and Denny Zhou. What learning algorithm is in-context learning? investigations with linear models. In Proceedings of The Eleventh International Conference on Learning Representations, 2023.

[Alabdulmohsin et al., 2022] Ibrahim M Alabdulmohsin, Behnam Neyshabur, and Xiaohua Zhai. Revisiting neural scaling laws in language and vision. Advances in Neural Information Processing Systems, 35:22300–22312, 2022.

[Allal et al., 2024] Loubna Ben Allal, Anton Lozhkov, and Daniel van Strien. cosmopedia: how to create large-scale synthetic data for pre-training. https://huggingface.co/blog/cosmopedia, 2024.

[Almazrouei et al., 2023] Ebtesam Almazrouei, Hamza Alobeidli, Abdulaziz Alshamsi, Alessandro Cappelli, Ruxandra Cojocaru, Mérouane Debbah, Étienne Goffinet, Daniel Hesslow, Julien Launay, Quentin Malartic, Daniele Mazzotta, Badreddine Noune, Baptiste Pannier, and Guilherme Penedo. The falcon series of open language models. arXiv preprint arXiv:2311.16867, 2023.

[Andreas et al., 2016] Jacob Andreas, Marcus Rohrbach, Trevor Darrell, and Dan Klein. Neural module networks. In Proceedings of the IEEE conference on computer vision and pattern recognition, pages 39–48, 2016.

[Arjovsky et al., 2016] Martin Arjovsky, Amar Shah, and Yoshua Bengio. Unitary evolution recurrent neural networks. In International conference on machine learning, pages 1120–1128, 2016.

[Aschenbrenner, 2024] Leopold Aschenbrenner. Situational awareness: The decade ahead, 2024. URL https://situational-awareness.ai/.

[Askell et al., 2021] Amanda Askell, Yuntao Bai, Anna Chen, Dawn Drain, Deep Ganguli, Tom Henighan, Andy Jones, Nicholas Joseph, Benjamin Mann, Nova DasSarma, Nelson Elhage, Zac Hatfield-Dodds, Danny Hernandez, Jackson Kernion, Kamal Ndousse, Catherine Olsson, Dario Amodei, Tom B. Brown, Jack Clark, Sam McCandlish, Chris Olah, and Jared Kaplan. A general language assistant as a laboratory for alignment. arXiv preprint arXiv:2112.00861, 2021.

[Bach et al., 2022] Stephen H. Bach, Victor Sanh, Zheng Xin Yong, Albert Webson, Colin Raffel, Nihal V. Nayak, Abheesht Sharma, Taewoon Kim, M. Saiful Bari, Thibault Févry, Zaid Alyafeai, Manan Dey, Andrea Santilli, Zhiqing Sun, Srulik Ben-David, Canwen Xu, Gunjan Chhablani, Han Wang, Jason Alan Fries, Maged Saeed AlShaibani, Shanya Sharma, Urmish Thakker, Khalid

Almubarak, Xiangru Tang, Dragomir R. Radev, Mike Tian-Jian Jiang, and Alexander M. Rush. Promptsource: An integrated development environment and repository for natural language prompts. In Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics: System Demonstrations, pages 93–104, 2022.

[Bengio et al., 2003] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. Journal of Machine Learning Research, 3:1137–1155, 2003.

[Bengio et al., 2006] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. Advances in neural information processing systems, 19, 2006.

[Bengio et al., 2024] Yoshua Bengio, Geoffrey Hinton, Andrew Yao, Dawn Song, Pieter Abbeel, Trevor Darrell, Yuval Noah Harari, Ya-Qin Zhang, Lan Xue, Shai Shalev-Shwartz, Gillian K. Hadfield, Jeff Clune, Tegan Maharaj, Frank Hutter, Atilim Gunes Baydin, Sheila A. McIlraith, Qiqi Gao, Ashwin Acharya, David Krueger, Anca Dragan, Philip Torr, Stuart Russell, Daniel Kahneman, Jan Markus Brauner, and Sören Mindermann. Managing extreme ai risks amid rapid progress. Science, 384(6698):842–845, 2024.

[Bentivogli and Giampiccolo, 2011] Luisa Bentivogli and Danilo Giampiccolo. Pascal recognizing textual entailment challenge (rte-7) at tac 2011. https://tac.nist.gov/2011/RTE/, 2011.

[Besta et al., 2024] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, and Torsten Hoefler. Graph of thoughts: Solving elaborate problems with large language models. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 38, pages 17682–17690, 2024.

[Biderman et al., 2021] Stella Biderman, Sid Black, Charles Foster, Leo Gao, Eric Hallahan, Horace He, Ben Wang, and Phil Wang. Rotary embeddings: A relative revolution. https://blog.eleuther.ai/rotary-embeddings/, 2021.

[Bishop, 2006] Christopher M. Bishop. Pattern Recognition and Machine Learning. Springer, 2006.

[Blum and Mitchell, 1998] Avrim Blum and Tom Mitchell. Combining labeled and unlabeled data with co-training. In Proceedings of the eleventh annual conference on Computational learning theory, pages 92–100, 1998.

[Bradley and Terry, 1952] Ralph Allan Bradley and Milton E. Terry. Rank analysis of incomplete block designs: I. the method of paired comparisons. Biometrika, 39(3/4):324–345, 1952.

[Brandon et al., 2024] William Brandon, Mayank Mishra, Aniruddha Nrusimha, Rameswar Panda, and Jonathan Ragan Kelly. Reducing transformer key-value cache size with cross-layer attention. arXiv preprint arXiv:2405.12981, 2024.

[Brill, 1992] Eric Brill. A simple rule-based part of speech tagger. In Speech and Natural Language: Proceedings of a Workshop Held at Harriman, New York, February 23-26, 1992, 1992.

[Briski, 2025] Kari Briski. How scaling laws drive smarter, more powerful ai, 2025.

[Brown et al., 2024] Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling. arXiv preprint arXiv:2407.21787, 2024.

[Brown et al., 1993] Peter F. Brown, Stephen A. Della Pietra, Vincent J. Della Pietra, and Robert L. Mercer. The mathematics of statistical machine translation: Parameter estimation. Computational Linguistics, 19(2):263–311, 1993.

[Brown et al., 2020] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh,

Daniel Ziegler, Jeffrey Wu, Clemens Winter, Chris Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. Advances in neural information processing systems, 33:1877–1901, 2020.

[Bubeck et al., 2023] Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott M. Lundberg, Harsha Nori, Hamid Palangi, Marco Túlio Ribeiro, and Yi Zhang. Sparks of artificial general intelligence: Early experiments with gpt-4. arXiv preprint arXiv:2303.12712, 2023.

[Bulatov et al., 2022] Aydar Bulatov, Yury Kuratov, and Mikhail Burtsev. Recurrent memory transformer. Advances in Neural Information Processing Systems, 35:11079–11091, 2022.

[Burges et al., 2005] Chris Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to rank using gradient descent. In Proceedings of the 22nd international conference on Machine learning, pages 89–96, 2005.

[Burns et al., 2023] Collin Burns, Pavel Izmailov, Jan Hendrik Kirchner, Bowen Baker, Leo Gao, Leopold Aschenbrenner, Yining Chen, Adrien Ecoffet, Manas Joglekar, Jan Leike, Ilya Sutskever, and Jeff Wu. Weak-to-strong generalization: Eliciting strong capabilities with weak supervision. arXiv preprint arXiv:2312.09390, 2023a.

[Burns et al., 2023] Collin Burns, Jan Leike, Leopold Aschenbrenner, Jeffrey Wu, Pavel Izmailov, Leo Gao, Bowen Baker, and Jan Hendrik Kirchner. Weak-to-strong generalization, 2023b. URL https://https://openai.com/index/weak-to-strong-generalization.

[Caballero et al., 2023] Ethan Caballero, Kshitij Gupta, Irina Rish, and David Krueger. Broken neural scaling laws. In ICLR 2023 Workshop on Mathematical and Empirical Understanding of Foundation Models, 2023.

[Cao et al., 2007] Zhe Cao, Tao Qin, Tie-Yan Liu, Ming-Feng Tsai, and Hang Li. Learning to rank: from pairwise approach to listwise approach. In Proceedings of the 24th international conference on Machine learning, pages 129–136, 2007.

[Chang et al., 2024] Kaiyan Chang, Songcheng Xu, Chenglong Wang, Yingfeng Luo, Tong Xiao, and Jingbo Zhu. Efficient prompting methods for large language models: A survey. arXiv preprint arXiv:2404.01077, 2024.

[Charniak, 1997] Eugene Charniak. Statistical parsing with a context-free grammar and word statistics. AAAI/IAAI, 2005(598-603):18, 1997.

[Chen et al., 2023] Banghao Chen, Zhaofeng Zhang, Nicolas Langrené, and Shengxin Zhu. Unleashing the potential of prompt engineering in large language models: a comprehensive review. arXiv preprint arXiv:2310.14735, 2023a.

[Chen et al., 2023] Lichang Chen, Shiyang Li, Jun Yan, Hai Wang, Kalpa Gunaratna, Vikas Yadav, Zheng Tang, Vijay Srinivasan, Tianyi Zhou, Heng Huang, and Hongxia Jin. Alpagasus: Training a better alpaca with fewer data. arXiv preprint arXiv:2307.08701, 2023b.

[Chen et al., 2024] Lichang Chen, Shiyang Li, Jun Yan, Hai Wang, Kalpa Gunaratna, Vikas Yadav, Zheng Tang, Vijay Srinivasan, Tianyi Zhou, Heng Huang, and Hongxia Jin. Alpagasus: Training a better alpaca with fewer data. In The Twelfth International Conference on Learning Representations, 2024a.

[Chen et al., 2023] Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. Extending context window of large language models via positional interpolation. arXiv preprint arXiv:2306.15595, 2023c.

[Chen et al., 2020] Tianlong Chen, Jonathan Frankle, Shiyu Chang, Sijia Liu, Yang Zhang, Zhangyang Wang, and Michael Carbin. The lottery ticket hypothesis for pre-trained bert networks. Advances in neural information processing systems, 33:15834–15846, 2020.

[Chen et al., 2024] Zixiang Chen, Yihe Deng, Huizhuo Yuan, Kaixuan Ji, and Quanquan Gu. Self-play fine-tuning converts weak language models to strong language models. arXiv preprint arXiv:2401.01335, 2024b.

[Chevalier et al., 2023] Alexis Chevalier, Alexander Wettig, Anirudh Ajith, and Danqi Chen. Adapting language models to compress contexts. In Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, pages 3829–3846, 2023.

[Chi et al., 2022] Ta-Chung Chi, Ting-Han Fan, Peter J Ramadge, and Alexander Rudnicky. Kerple: Kernelized relative positional embedding for length extrapolation. Advances in Neural Information Processing Systems, 35:8386–8399, 2022.

[Chi et al., 2023] Ta-Chung Chi, Ting-Han Fan, Alexander Rudnicky, and Peter Ramadge. Dissecting transformer length extrapolation via the lens of receptive field analysis. In Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 13522–13537, 2023.

[Chiang et al., 2023] Wei-Lin Chiang, Zhuohan Li, Zi Lin, Ying Sheng, Zhanghao Wu, Hao Zhang, Lianmin Zheng, Siyuan Zhuang, Yonghao Zhuang, Joseph E. Gonzalez, Ion Stoica, and Eric P. Xing. Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality, March 2023. URL https://lmsys.org/blog/2023-03-30-vicuna/.

[Chowdhery et al., 2022] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, Parker Schuh, Kensen Shi, Sasha Tsvyashchenko, Joshua Maynez, Abhishek Rao, Parker Barnes, Yi Tay, Noam Shazeer, Vinodkumar Prabhakaran, Emily Reif, Nan Du, Ben Hutchinson, Reiner Pope, James Bradbury, Jacob Austin, Michael Isard, Guy Gur-Ari, Pengcheng Yin, Toju Duke, Anselm Levskaya, Sanjay Ghemawat, Sunipa Dev, Henryk Michalewski, Xavier Garcia, Vedant Misra, Kevin Robinson, Liam Fedus, Denny Zhou, Daphne Ippolito, David Luan, Hyeontaek Lim, Barret Zoph, Alexander Spiridonov, Ryan Sepassi, David Dohan, Shivani Agrawal, Mark Omernick, Andrew M. Dai, Thanumalayan Sankaranarayana Pillai, Marie Pellat, Aitor Lewkowycz, Erica Moreira, Rewon Child, Oleksandr Polozov, Katherine Lee, Zongwei Zhou, Xuezhi Wang, Brennan Saeta, Mark Diaz, Orhan Firat, Michele Catasta, Jason Wei, Kathy Meier-Hellstern, Douglas Eck, Jeff Dean, Slav Petrov, and Noah Fiedel. Palm: Scaling language modeling with pathways. arXiv preprint arXiv:2204.02311, 2022.

[Christiano et al., 2017] Paul F Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei. Deep reinforcement learning from human preferences. Advances in neural information processing systems, 30, 2017.

[Chu et al., 2023] Zheng Chu, Jingchang Chen, Qianglong Chen, Weijiang Yu, Tao He, Haotian Wang, Weihua Peng, Ming Liu, Bing Qin, and Ting Liu. A survey of chain of thought reasoning: Advances, frontiers and future. arXiv preprint arXiv:2309.15402, 2023.

[Chung et al., 2022] Hyung Won Chung, Le Hou, S. Longpre, Barret Zoph, Yi Tay, William Fedus, Eric Li, Xuezhi Wang, Mostafa Dehghani, Siddhartha Brahma, Albert Webson, Shixiang Shane Gu, Zhuyun Dai, Mirac Suzgun, Xinyun Chen, Aakanksha Chowdhery, Dasha Valter, Sharan Narang, Gaurav Mishra, Adams Wei Yu, Vincent Zhao, Yanping Huang, Andrew M. Dai, Hongkun Yu, Slav Petrov, Ed Huai hsin Chi, Jeff Dean, Jacob Devlin, Adam Roberts, Denny Zhou, Quoc V. Le, and Jason Wei. Scaling instruction-finetuned language models. arXiv preprint arXiv:2210.11416, 2022.

[Clark et al., 2019] Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. Electra: Pre-training text encoders as discriminators rather than generators. In Proceedings of International Conference on Learning Representations, 2019.

[Cobbe et al., 2021] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. arXiv preprint

arXiv:2110.14168, 2021.

[Conneau et al., 2020] Alexis Conneau, Kartikay Khandelwal, Naman Goyal, Vishrav Chaudhary, Guillaume Wenzek, Francisco Guzmán, Édouard Grave, Myle Ott, Luke Zettlemoyer, and Veselin Stoyanov. Unsupervised cross-lingual representation learning at scale. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pages 8440–8451, 2020.

[Coste et al., 2024] Thomas Coste, Usman Anwar, Robert Kirk, and David Krueger. Reward model ensembles help mitigate overoptimization. In The Twelfth International Conference on Learning Representations, 2024.

[Cui et al., 2024] Ganqu Cui, Lifan Yuan, Ning Ding, Guanming Yao, Bingxiang He, Wei Zhu, Yuan Ni, Guotong Xie, Ruobing Xie, Yankai Lin, Zhiyuan Liu, and Maosong Sun. ULTRAFEEDBACK: Boosting language models with scaled AI feedback. In Proceedings of the 41st International Conference on Machine Learning, volume 235, pages 9722–9744, 2024.

[Dai et al., 2023] Damai Dai, Yutao Sun, Li Dong, Yaru Hao, Shuming Ma, Zhifang Sui, and Furu Wei. Why can gpt learn in-context? language models secretly perform gradient descent as meta-optimizers. In Findings of the Association for Computational Linguistics: ACL 2023, pages 4005–4019, 2023.

[Dai et al., 2019] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G Carbonell, Quoc Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. In Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, pages 2978–2988, 2019.

[Dao et al., 2022] Tri Dao, Dan Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. Advances in Neural Information Processing Systems, 35:16344–16359, 2022.

[Deepseek, 2025] Deepseek. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. arXiv preprint arXiv:2501.12948, 2025.

[Dehghani et al., 2018] Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Łukasz Kaiser. Universal transformers. arXiv preprint arXiv:1807.03819, 2018.

[Deletang et al., 2024] Gregoire Deletang, Anian Ruoss, Paul-Ambroise Duquenne, Elliot Catt, Tim Genewein, Christopher Mattern, Jordi Grau-Moya, Li Kevin Wenliang, Matthew Aitchison, Laurent Orseau, Marcus Hutter, and Joel Veness. Language modeling is compression. In The Twelfth International Conference on Learning Representations, 2024.

[Deng et al., 2022] Mingkai Deng, Jianyu Wang, Cheng-Ping Hsieh, Yihan Wang, Han Guo, Tianmin Shu, Meng Song, Eric Xing, and Zhiting Hu. Rlprompt: Optimizing discrete text prompts with reinforcement learning. In Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, pages 3369–3391, 2022.

[Devlin et al., 2019] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers), pages 4171–4186, 2019.

[Ding et al., 2024] Yiran Ding, Li Lyna Zhang, Chengruidong Zhang, Yuanyuan Xu, Ning Shang, Jiahang Xu, Fan Yang, and Mao Yang. Longrope: Extending llm context window beyond 2 million tokens. arXiv preprint arXiv:2402.13753, 2024.

[Dolan and Brockett, 2005] Bill Dolan and Chris Brockett. Automatically constructing a corpus of sentential paraphrases. In Proceedings of Third International Workshop on Paraphrasing (IWP2005), 2005.

[Dong et al., 2019] Li Dong, Nan Yang, Wenhui Wang, Furu Wei, Xiaodong Liu, Yu Wang, Jianfeng Gao, Ming Zhou, and Hsiao-Wuen Hon. Unified language model pre-training for natural language

understanding and generation. Advances in neural information processing systems, 32, 2019.

[Dong et al., 2022] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. A survey on in-context learning. arXiv preprint arXiv:2301.00234, 2022.

[Dong et al., 2021] Yihe Dong, Jean-Baptiste Cordonnier, and Andreas Loukas. Attention is not all you need: Pure attention loses rank doubly exponentially with depth. In International Conference on Machine Learning, pages 2793–2803. PMLR, 2021.

[Drozdov et al., 2022] Andrew Drozdov, Nathanael Schärli, Ekin Akyürek, Nathan Scales, Xinying Song, Xinyun Chen, Olivier Bousquet, and Denny Zhou. Compositional semantic parsing with large language models. In Proceedings of The Eleventh International Conference on Learning Representations, 2022.

[Dua et al., 2022] Dheeru Dua, Shivanshu Gupta, Sameer Singh, and Matt Gardner. Successive prompting for decomposing complex questions. In Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, pages 1251–1265, 2022.

[Dubey et al., 2024] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. arXiv preprint arXiv:2407.21783, 2024.

[Dubois et al., 2024] Yann Dubois, Chen Xuechen Li, Rohan Taori, Tianyi Zhang, Ishaan Gulrajani, Jimmy Ba, Carlos Guestrin, Percy S Liang, and Tatsunori B Hashimoto. Alpacafarm: A simulation framework for methods that learn from human feedback. Advances in Neural Information Processing Systems, 36, 2024.

[Eisenstein et al., 2023] Jacob Eisenstein, Chirag Nagpal, Alekh Agarwal, Ahmad Beirami, Alex D'Amour, DJ Dvijotham, Adam Fisch, Katherine Heller, Stephen Pfohl, Deepak Ramachandran, and Peter Shaw. Helping or herding? reward model ensembles mitigate but do not eliminate reward hacking. arXiv preprint arXiv:2312.09244, 2023.

[Elsken et al., 2019] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. Journal of Machine Learning Research, 20(55):1–21, 2019.

[Erhan et al., 2010] Dumitru Erhan, Aaron Courville, Yoshua Bengio, and Pascal Vincent. Why does unsupervised pre-training help deep learning? In Proceedings of the thirteenth international conference on artificial intelligence and statistics, pages 201–208, 2010.

[Fan et al., 2018] Angela Fan, Mike Lewis, and Yann Dauphin. Hierarchical neural story generation. In Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 889–898, 2018.

[Fan et al., 2019] Angela Fan, Edouard Grave, and Armand Joulin. Reducing transformer depth on demand with structured dropout. In Proceedings of International Conference on Learning Representations, 2019.

[Fedus et al., 2022] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. The Journal of Machine Learning Research, 23(1):5232–5270, 2022.

[Fernandes et al., 2023] Patrick Fernandes, Aman Madaan, Emmy Liu, António Farinhas, Pedro Henrique Martins, Amanda Bertsch, José G. C. de Souza, Shuyan Zhou, Tongshuang Wu, Graham Neubig, and André F. T. Martins. Bridging the gap: A survey on integrating (human) feedback for natural language generation. Transactions of the Association for Computational Linguistics, 11:1643–1668, 2023.

[Forney, 1972] GDJR Forney. Maximum-likelihood sequence estimation of digital sequences in the presence of intersymbol interference. IEEE Transactions on Information theory, 18(3):363–378, 1972.

[Franklin and Graesser, 1996] Stan Franklin and Art Graesser. Is it an agent, or just a program?: A taxonomy for autonomous agents. In International workshop on agent theories, architectures, and languages, pages 21–35. Springer, 1996.

[Frensch and Funke, 2014] Peter A Frensch and Joachim Funke. Complex problem solving: The European perspective. Psychology Press, 2014.

[Gale et al., 2019] Trevor Gale, Erich Elsen, and Sara Hooker. The state of sparsity in deep neural networks. arXiv preprint arXiv:1902.09574, 2019.

[Ganguli et al., 2023] Deep Ganguli, Amanda Askell, Nicholas Schiefer, Thomas I. Liao, Kamile Lukosiute, Anna Chen, Anna Goldie, Azalia Mirhoseini, Catherine Olsson, Danny Hernandez, Dawn Drain, Dustin Li, Eli Tran-Johnson, Ethan Perez, Jackson Kernion, Jamie Kerr, Jared Mueller, Joshua Landau, Kamal Ndousse, Karina Nguyen, Liane Lovitt, Michael Sellitto, Nelson Elhage, Noemí Mercado, Nova DasSarma, Oliver Rausch, Robert Lasenby, Robin Larson, Sam Ringer, Sandipan Kundu, Saurav Kadavath, Scott Johnston, Shauna Kravec, Sheer El Showk, Tamera Lanham, Timothy Telleen-Lawton, Tom Henighan, Tristan Hume, Yuntao Bai, Zac Hatfield-Dodds, Ben Mann, Dario Amodei, Nicholas Joseph, Sam McCandlish, Tom Brown, Christopher Olah, Jack Clark, Samuel R. Bowman, and Jared Kaplan. The capacity for moral self-correction in large language models. arXiv preprint arXiv:2302.07459, 2023.

[Gao et al., 2023] Leo Gao, John Schulman, and Jacob Hilton. Scaling laws for reward model overoptimization. In International Conference on Machine Learning, pages 10835–10866. PMLR, 2023a.

[Gao et al., 2023] Luyu Gao, Aman Madaan, Shuyan Zhou, Uri Alon, Pengfei Liu, Yiming Yang, Jamie Callan, and Graham Neubig. Pal: Program-aided language models. In International Conference on Machine Learning, pages 10764–10799. PMLR, 2023b.

[Gao et al., 2023] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang. Retrieval-augmented generation for large language models: A survey. arXiv preprint arXiv:2312.10997, 2023c.

[Garg et al., 2022] Shivam Garg, Dimitris Tsipras, Percy S Liang, and Gregory Valiant. What can transformers learn in-context? a case study of simple function classes. Advances in Neural Information Processing Systems, 35:30583–30598, 2022.

[Ge et al., 2024] Yuan Ge, Yilun Liu, Chi Hu, Weibin Meng, Shimin Tao, Xiaofeng Zhao, Hongxia Ma, Li Zhang, Boxing Chen, Hao Yang, Bei Li, Tong Xiao, and Jingbo Zhu. Clustering and ranking: Diversity-preserved instruction selection through expert-aligned quality estimation. arXiv preprint arXiv:2402.18191, 2024.

[Gemma Team, 2024] Google DeepMind Gemma Team. Gemma: Open Models Based on Gemini Research and Technology, 2024.

[Goodhart, 1984] Charles AE Goodhart. Problems of monetary management: the UK experience. Springer, 1984.

[Gordon et al., 2021] Mitchell A Gordon, Kevin Duh, and Jared Kaplan. Data and parameter scaling laws for neural machine translation. In Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, pages 5915–5922, 2021.

[Gou et al., 2024] Zhibin Gou, Zhihong Shao, Yeyun Gong, Yujiu Yang, Nan Duan, Weizhu Chen, et al. Critic: Large language models can self-correct with tool-interactive critiquing. In The Twelfth International Conference on Learning Representations, 2024.

[Gu and Dao, 2023] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. arXiv preprint arXiv:2312.00752, 2023.

[Gunasekar et al., 2023] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen

Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. Textbooks are all you need. arXiv preprint arXiv:2306.11644, 2023.

[Guo et al., 2024] Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. In The Twelfth International Conference on Learning Representations, 2024.

[Gupta and Berant, 2020] Ankit Gupta and Jonathan Berant. Gmat: Global memory augmentation for transformers. arXiv preprint arXiv:2006.03274, 2020.

[Gupta et al., 2021] Ankit Gupta, Guy Dar, Shaya Goodman, David Ciprut, and Jonathan Berant. Memory-efficient transformers via top-k attention. In Proceedings of the Second Workshop on Simple and Efficient Natural Language Processing, pages 39–52, 2021.

[Han et al., 2021] Xu Han, Zhengyan Zhang, Ning Ding, Yuxian Gu, Xiao Liu, Yuqi Huo, Jiezhong Qiu, Liang Zhang, Wentao Han, Minlie Huang, Qin Jin, Yanyan Lan, Yang Liu, Zhiyuan Liu, Zhiwu Lu, Xipeng Qiu, Ruihua Song, Jie Tang, Ji-Rong Wen, Jinhui Yuan, Wayne Xin Zhao, and Jun Zhu. Pre-trained models: Past, present and future. AI Open, 2:225–250, 2021.

[Han et al., 2024] Zeyu Han, Chao Gao, Jinyang Liu, Jeff Zhang, and Sai Qian Zhang. Parameter-efficient fine-tuning for large models: A comprehensive survey. arXiv preprint arXiv:2403.14608, 2024.

[Harlap et al., 2018] Aaron Harlap, Deepak Narayanan, Amar Phanishayee, Vivek Seshadri, Nikhil Devanur, Greg Ganger, and Phil Gibbons. Pipedream: Fast and efficient pipeline parallel dnn training. arXiv preprint arXiv:1806.03377, 2018.

[He et al., 2019] Kaiming He, Ross Girshick, and Piotr Dollár. Rethinking imagenet pre-training. In Proceedings of the IEEE/CVF International Conference on Computer Vision, pages 4918–4927, 2019.

[He et al., 2021] Pengcheng He, Xiaodong Liu, Jianfeng Gao, and Weizhu Chen. Deberta: Decoding-enhanced bert with disentangled attention. In Proceedings of International Conference on Learning Representations, 2021.

[Hendrycks and Gimpel, 2016] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). arXiv preprint arXiv:1606.08415, 2016.

[Hendrycks et al., 2020] Dan Hendrycks, Xiaoyuan Liu, Eric Wallace, Adam Dziedzic, Rishabh Krishnan, and Dawn Song. Pretrained transformers improve out-of-distribution robustness. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pages 2744–2751, 2020.

[Hendrycks et al., 2021] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. In Proceedings of International Conference on Learning Representations, 2021.

[Hestness et al., 2017] Joel Hestness, Sharan Narang, Newsha Ardalani, Gregory Diamos, Heewoo Jun, Hassan Kianinejad, Md Mostofa Ali Patwary, Yang Yang, and Yanqi Zhou. Deep learning scaling is predictable, empirically. arXiv preprint arXiv:1712.00409, 2017.

[Hewitt, 2024] John Hewitt. Instruction following without instruction tuning, 2024. URL https://nlp.stanford.edu/~johnhew/instruction-following.html.

[Hewitt et al., 2024] John Hewitt, Nelson F Liu, Percy Liang, and Christopher D Manning. Instruction following without instruction tuning. arXiv preprint arXiv:2409.14254, 2024.

[Hochreiter and Schmidhuber, 1997] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. Neural computation, 9(8):1735–1780, 1997.

[Hoffmann et al., 2022] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl,

Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models. arXiv preprint arXiv:2203.15556, 2022.

[Holtzman et al., 2020] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. In International Conference on Learning Representations, 2020.

[Honovich et al., 2023] Or Honovich, Thomas Scialom, Omer Levy, and Timo Schick. Unnatural instructions: Tuning language models with (almost) no human labor. In Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 14409–14428, 2023.

[Houlsby et al., 2019] Neil Houlsby, Andrei Giurgiu, Stanislaw Jastrzebski, Bruna Morrone, Quentin De Laroussilhe, Andrea Gesmundo, Mona Attariyan, and Sylvain Gelly. Parameter-efficient transfer learning for NLP. In Proceedings of the 36th International Conference on Machine Learning, pages 2790–2799. PMLR, 2019.

[Hu et al., 2022] Edward J Hu, yelong shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In International Conference on Learning Representations, 2022.

[Huang, 2009] Liang Huang. Dynamic programming-based search algorithms in NLP. In Proceedings of Human Language Technologies: The 2009 Annual Conference of the North American Chapter of the Association for Computational Linguistics, Companion Volume: Tutorial Abstracts, 2009.

[Huang et al., 2019] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Mia Xu Chen, Dehao Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and Zhifeng Chen. Gpipe: Efficient training of giant neural networks using pipeline parallelism. Advances in neural information processing systems, 32, 2019.

[Hutchins et al., 2022] DeLesley Hutchins, Imanol Schlag, Yuhuai Wu, Ethan Dyer, and Behnam Neyshabur. Block-recurrent transformers. Advances in neural information processing systems, 35:33248–33261, 2022.

[Jelinek, 1998] Frederick Jelinek. Statistical methods for speech recognition. MIT Press, 1998.

[Jiang et al., 2023] Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b. arXiv preprint arXiv:2310.06825, 2023a.

[Jiang et al., 2023] Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. Llmlingua: Compressing prompts for accelerated inference of large language models. In Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, pages 13358–13376, 2023b.

[Jiang et al., 2020] Zhengbao Jiang, Frank F Xu, Jun Araki, and Graham Neubig. How can we know what language models know? Transactions of the Association for Computational Linguistics, 8: 423–438, 2020.

[Jiao et al., 2020] Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. Tinybert: Distilling bert for natural language understanding. In Findings of the Association for Computational Linguistics: EMNLP 2020, pages 4163–4174, 2020.

[Joshi et al., 2017] Mandar Joshi, Eunsol Choi, Daniel S Weld, and Luke Zettlemoyer. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. In Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long

Papers), pages 1601–1611, 2017.

[Joshi et al., 2020] Mandar Joshi, Danqi Chen, Yinhan Liu, Daniel S Weld, Luke Zettlemoyer, and Omer Levy. Spanbert: Improving pre-training by representing and predicting spans. Transactions of the association for computational linguistics, 8:64–77, 2020.

[Jurafsky and Martin, 2008] Dan Jurafsky and James H. Martin. Speech and Language Processing (2nd ed.). Prentice Hall, 2008.

[Kahneman, 2011] Daniel Kahneman. Thinking, fast and slow. macmillan, 2011.

[Kaplan et al., 2020] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. arXiv preprint arXiv:2001.08361, 2020.

[Katharopoulos et al., 2020] Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In International conference on machine learning, pages 5156–5165. PMLR, 2020.

[Khandelwal et al., 2020] Urvashi Khandelwal, Omer Levy, Dan Jurafsky, Luke Zettlemoyer, and Mike Lewis. Generalization through memorization: Nearest neighbor language models. In International Conference on Learning Representations, 2020.

[Khot et al., 2023] Tushar Khot, Harsh Trivedi, Matthew Finlayson, Yao Fu, Kyle Richardson, Peter Clark, and Ashish Sabharwal. Decomposed prompting: A modular approach for solving complex tasks. In Proceedings of The Eleventh International Conference on Learning Representations, 2023.

[Kim et al., 2023] Sehoon Kim, Coleman Hooper, Thanakul Wattanawong, Minwoo Kang, Ruohan Yan, Hasan Genc, Grace Dinh, Qijing Huang, Kurt Keutzer, Michael W. Mahoney, Yakun Sophia Shao, and Amir Gholami. Full stack optimization of transformer inference: a survey. arXiv preprint arXiv:2302.14017, 2023.

[Kirkpatrick et al., 2017] James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A. Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, Demis Hassabis, Claudia Clopath, Dharshan Kumaran, and Raia Hadsell. Overcoming catastrophic forgetting in neural networks. Proceedings of the national academy of sciences, 114(13):3521–3526, 2017.

[Koehn, 2010] Philipp Koehn. Statistical Machine Translation. Cambridge University Press, 2010.

[Kojima et al., 2022] Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. Advances in neural information processing systems, 35:22199–22213, 2022.

[Korthikanti et al., 2023] Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. Proceedings of Machine Learning and Systems, 5, 2023.

[Krakovna et al., 2020] Victoria Krakovna, Jonathan Uesato, Vladimir Mikulik, Matthew Rahtz, Tom Everitt, Ramana Kumar, Zac Kenton, Jan Leike, and Shane Legg. Specification gaming: the flip side of ai ingenuity. https://deepmind.google/discover/blog/specification-gaming-the-flip-side-of-ai-ingenuity, 2020.

[Kumar and Byrne, 2004] Shankar Kumar and William Byrne. Minimum bayes-risk decoding for statistical machine translation. In Proceedings of the Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics: HLT-NAACL 2004, pages 169–176, 2004.

[Kung and Peng, 2023] Po-Nien Kung and Nanyun Peng. Do models really learn to follow instructions? an empirical study of instruction tuning. arXiv preprint arXiv:2305.11383, 2023.

[Kwon et al., 2023] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. arXiv preprint arXiv:2309.06180, 2023.

[Lake and Baroni, 2018] Brenden Lake and Marco Baroni. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In International conference on machine learning, pages 2873–2882. PMLR, 2018.

[Lambert et al., 2024] Nathan Lambert, Valentina Pyatkin, Jacob Morrison, LJ Miranda, Bill Yuchen Lin, Khyathi Chandu, Nouha Dziri, Sachin Kumar, Tom Zick, Yejin Choi, Noah A. Smith, and Hannaneh Hajishirzi. Rewardbench: Evaluating reward models for language modeling. arXiv preprint arXiv:2403.13787, 2024.

[Lample and Conneau, 2019] Guillaume Lample and Alexis Conneau. Cross-lingual language model pretraining. arXiv preprint arXiv:1901.07291, 2019.

[Lan et al., 2020] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. In Proceedings of International Conference on Learning Representations, 2020.

[Lee et al., 2023] Harrison Lee, Samrat Phatale, Hassan Mansoor, Kellie Ren Lu, Thomas Mesnard, Johan Ferret, Colton Bishop, Ethan Hall, Victor Carbune, and Abhinav Rastogi. Rlaif: Scaling reinforcement learning from human feedback with ai feedback. arXiv preprint arXiv:2309.00267, 2023.

[Lester et al., 2021] Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. In Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, pages 3045–3059, 2021.

[Leviathan et al., 2023] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In Proceedings of International Conference on Machine Learning, pages 19274–19286. PMLR, 2023.

[Lewis et al., 2020] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pages 7871–7880, 2020.

[Li et al., 2024] Baolin Li, Yankai Jiang, Vijay Gadepally, and Devesh Tiwari. Llm inference serving: Survey of recent advances and opportunities. arXiv preprint arXiv:2407.12391, 2024a.

[Li et al., 2023] Bei Li, Rui Wang, Junliang Guo, Kaitao Song, Xu Tan, Hany Hassan, Arul Menezes, Tong Xiao, Jiang Bian, and JingBo Zhu. Deliberate then generate: Enhanced prompting framework for text generation. arXiv preprint arXiv:2305.19835, 2023a.

[Li, 2011] Hang Li. Learning to Rank for Information Retrieval and Natural Language Processing. Online access: Morgan & Claypool Synthesis Collection Five. Morgan & Claypool Publishers, 2011. ISBN 9781608457076.

[Li et al., 2022] Huayang Li, Yixuan Su, Deng Cai, Yan Wang, and Lemao Liu. A survey on retrieval-augmented text generation. arXiv preprint arXiv:2202.01110, 2022.

[Li et al., 2024] Shanda Li, Chong You, Guru Guruganesh, Joshua Ainslie, Santiago Ontanon, Manzil Zaheer, Sumit Sanghai, Yiming Yang, Sanjiv Kumar, and Srinadh Bhojanapalli. Functional interpolation for relative positions improves long context transformers. In The Twelfth International Conference on Learning Representations, 2024b.

[Li et al., 2023] Shenggui Li, Fuzhao Xue, Chaitanya Baranwal, Yongbin Li, and Yang You. Sequence parallelism: Long sequence training from system perspective. In Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 2391–2404, 2023b.

[Li and Liang, 2021] Xiang Lisa Li and Percy Liang. Prefix-tuning: Optimizing continuous prompts for generation. In Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers), pages 4582–4597, 2021.

[Li, 2023] Yinheng Li. A practical survey on zero-shot prompt design for in-context learning. In Proceedings of the 14th International Conference on Recent Advances in Natural Language Processing, pages 641–647, 2023.

[Li et al., 2023] Yucheng Li, Bo Dong, Frank Guerin, and Chenghua Lin. Compressing context to enhance inference efficiency of large language models. In Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, pages 6342–6353, 2023c.

[Lialin et al., 2023] Vladislav Lialin, Vijeta Deshpande, and Anna Rumshisky. Scaling down to scale up: A guide to parameter-efficient fine-tuning. arXiv preprint arXiv:2303.15647, 2023.

[Lightman et al., 2024] Hunter Lightman, Vineet Kosaraju, Yuri Burda, Harrison Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. In The Twelfth International Conference on Learning Representations, 2024.

[Liu et al., 2024] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. Deepseek-v3 technical report. arXiv preprint arXiv:2412.19437, 2024a.

[Liu et al., 2022] Jiachang Liu, Dinghan Shen, Yizhe Zhang, William B Dolan, Lawrence Carin, and Weizhu Chen. What makes good in-context examples for gpt-3? In Proceedings of Deep Learning Inside Out (DeeLIO 2022): The 3rd Workshop on Knowledge Extraction and Integration for Deep Learning Architectures, pages 100–114, 2022.

[Liu et al., 2023] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing. ACM Computing Surveys, 55(9):1–35, 2023a.

[Liu et al., 2024] Tianqi Liu, Yao Zhao, Rishabh Joshi, Misha Khalman, Mohammad Saleh, Peter J Liu, and Jialu Liu. Statistical rejection sampling improves preference optimization. In The Twelfth International Conference on Learning Representations, 2024b.

[Liu, 2009] Tie-Yan Liu. Learning to rank for information retrieval. Foundations and Trends® in Information Retrieval, 3(3):225–331, 2009.

[Liu et al., 2023] Xiao Liu, Yanan Zheng, Zhengxiao Du, Ming Ding, Yujie Qian, Zhilin Yang, and Jie Tang. Gpt understands, too. AI Open, 2023b.

[Liu et al., 2023] Xiaoxia Liu, Jingyi Wang, Jun Sun, Xiaohan Yuan, Guoliang Dong, Peng Di, Wenhai Wang, and Dongxia Wang. Prompting frameworks for large language models: A survey. arXiv preprint arXiv:2311.12785, 2023c.

[Liu et al., 2024] Xinyu Liu, Runsong Zhao, Pengcheng Huang, Chunyang Xiao, Bei Li, Jingang Wang, Tong Xiao, and Jingbo Zhu. Forgetting curve: A reliable method for evaluating memorization capability for long-context models. In Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing, pages 4667–4682, 2024c.

[Liu et al., 2019] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. arXiv preprint arXiv:1907.11692, 2019.

[Longpre et al., 2023] Shayne Longpre, Le Hou, Tu Vu, Albert Webson, Hyung Won Chung, Yi Tay, Denny Zhou, Quoc V. Le, Barret Zoph, Jason Wei, and Adam Roberts. The flan collection: Designing data and methods for effective instruction tuning. In International Conference on Machine Learning, pages 22631–22648. PMLR, 2023.

[Ma et al., 2023] Xuezhe Ma, Chunting Zhou, Xiang Kong, Junxian He, Liangke Gui, Graham

Neubig, Jonathan May, and Luke Zettlemoyer. Mega: Moving average equipped gated attention. In The Eleventh International Conference on Learning Representations, 2023.

[Ma et al., 2024] Xuezhe Ma, Xiaomeng Yang, Wenhan Xiong, Beidi Chen, Lili Yu, Hao Zhang, Jonathan May, Luke Zettlemoyer, Omer Levy, and Chunting Zhou. Megalodon: Efficient llm pretraining and inference with unlimited context length. arXiv preprint arXiv:2404.08801, 2024.

[Madaan et al., 2024] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. Advances in Neural Information Processing Systems, 36, 2024.

[Manning, 2022] Christopher D Manning. Human language understanding & reasoning. Daedalus, 151(2):127–138, 2022.

[Marcus, 1993] Gary F Marcus. Negative evidence in language acquisition. Cognition, 46(1):53–85, 1993.

[Martins et al., 2022] Pedro Henrique Martins, Zita Marinho, and André FT Martins. ∞-former: Infinite memory transformer-former: Infinite memory transformer. In Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 5468–5485, 2022.

[Mavi et al., 2024] Vaibhav Mavi, Anubhav Jangra, and Adam Jatowt. Multi-hop question answering. Foundations and Trends® in Information Retrieval, 17(5):457–586, 2024.

[Michel et al., 2019] Paul Michel, Omer Levy, and Graham Neubig. Are sixteen heads really better than one? Advances in neural information processing systems, 32, 2019.

[Micikevicius et al., 2018] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training. In Proceedings of International Conference on Learning Representations, 2018.

[Miettinen, 1999] Kaisa Miettinen. Nonlinear multiobjective optimization, volume 12. Springer Science & Business Media, 1999.

[Mikolov et al., 2013] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In Proceedings of the International Conference on Learning Representations (ICLR 2013), 2013a.

[Mikolov et al., 2013] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2, pages 3111–3119, 2013b.

[Min et al., 2019] Sewon Min, Victor Zhong, Luke Zettlemoyer, and Hannaneh Hajishirzi. Multi-hop reading comprehension through question decomposition and rescoring. In Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, pages 6097–6109, 2019.

[Minaee et al., 2024] Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. Large language models: A survey. arXiv preprint arXiv:2402.06196, 2024.

[Mishra et al., 2022] Swaroop Mishra, Daniel Khashabi, Chitta Baral, and Hannaneh Hajishirzi. Cross-task generalization via natural language crowdsourcing instructions. In Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 3470–3487, 2022.

[Mnih et al., 2016] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves,

Tim Harley, Timothy P Lillicrap, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In Proceedings of the 33rd International Conference on International Conference on Machine Learning, pages 1928–1937, 2016.

[Mohtashami and Jaggi, 2024] Amirkeivan Mohtashami and Martin Jaggi. Random-access infinite context length for transformers. Advances in Neural Information Processing Systems, 36, 2024.

[Mu et al., 2024] Jesse Mu, Xiang Li, and Noah Goodman. Learning to compress prompts with gist tokens. Advances in Neural Information Processing Systems, 36, 2024.

[Munkhdalai et al., 2024] Tsendsuren Munkhdalai, Manaal Faruqui, and Siddharth Gopal. Leave no context behind: Efficient infinite context transformers with infini-attention. arXiv preprint arXiv:2404.07143, 2024.

[Nakano et al., 2021] Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. Webgpt: Browser-assisted question-answering with human feedback. arXiv preprint arXiv:2112.09332, 2021.

[Narayanan et al., 2021] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, Amar Phanishayee, and Matei Zaharia. Efficient large-scale language model training on gpu clusters using megatron-lm. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–15, 2021.

[Ng et al., 1999] Andrew Y Ng, Daishi Harada, and Stuart J Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In Proceedings of the Sixteenth International Conference on Machine Learning, pages 278–287, 1999.

[Nvidia, 2025] Nvidia. Nvidia nim llms benchmarking. https://docs.nvidia.com/nim/benchmarking/llm/latest/metrics.html, 2025. Retrieved 2025-03-17.

[OpenAI, 2024] OpenAI. Learning to reason with llms, September 2024. URL https://openai.com/index/learning-to-reason-with-llms/.

[Ouyang et al., 2022] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. Advances in Neural Information Processing Systems, 35:27730–27744, 2022.

[Pal et al., 2023] Koyena Pal, Jiuding Sun, Andrew Yuan, Byron C Wallace, and David Bau. Future lens: Anticipating subsequent tokens from a single hidden state. In Proceedings of the 27th Conference on Computational Natural Language Learning (CoNLL), pages 548–560, 2023.

[Pan et al., 2022] Alexander Pan, Kush Bhatia, and Jacob Steinhardt. The effects of reward misspecification: Mapping and mitigating misaligned models. In International Conference on Learning Representations, 2022.

[Pan et al., 2024] Liangming Pan, Michael Saxon, Wenda Xu, Deepak Nathani, Xinyi Wang, and William Yang Wang. Automatically correcting large language models: Surveying the landscape of diverse automated correction strategies. Transactions of the Association for Computational Linguistics, 12:484–506, 2024.

[Parisi et al., 2022] Aaron Parisi, Yao Zhao, and Noah Fiedel. Talm: Tool augmented language models. arXiv preprint arXiv:2205.12255, 2022.

[Parisi et al., 2019] German I Parisi, Ronald Kemker, Jose L Part, Christopher Kanan, and Stefan Wermter. Continual lifelong learning with neural networks: A review. Neural networks, 113:

54–71, 2019.

[Parmar et al., 2018] Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Lukasz Kaiser, Noam Shazeer, Alexander Ku, and Dustin Tran. Image transformer. In International conference on machine learning, pages 4055–4064. PMLR, 2018.

[Patel et al., 2024] Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative llm inference using phase splitting. In 2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA), pages 118–132. IEEE, 2024.

[Penedo et al., 2023] Guilherme Penedo, Quentin Malartic, Daniel Hesslow, Ruxandra Cojocaru, Alessandro Cappelli, Hamza Alobeidli, Baptiste Pannier, Ebtesam Almazrouei, and Julien Launay. The refinedweb dataset for falcon llm: outperforming curated corpora with web data, and web data only. arXiv preprint arXiv:2306.01116, 2023.

[Peng et al., 2024] Bowen Peng, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. YaRN: Efficient context window extension of large language models. In The Twelfth International Conference on Learning Representations, 2024.

[Pennington et al., 2014] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In Proceedings of Empirical Methods in Natural Language Processing (EMNLP), pages 1532–1543, 2014.

[Peters et al., 2018] Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers), 2018.

[Plackett, 1975] Robin L Plackett. The analysis of permutations. Journal of the Royal Statistical Society Series C: Applied Statistics, 24(2):193–202, 1975.

[Pope et al., 2023] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. In Proceedings of Machine Learning and Systems, 2023.

[Prasad et al., 2023] Archiki Prasad, Peter Hase, Xiang Zhou, and Mohit Bansal. Grips: Gradient-free, edit-based instruction search for prompting large language models. In Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics, pages 3845–3864, 2023.

[Press et al., 2022] Ofir Press, Noah Smith, and Mike Lewis. Train short, test long: Attention with linear biases enables input length extrapolation. In Proceedings of International Conference on Learning Representations, 2022.

[Press et al., 2023] Ofir Press, Muru Zhang, Sewon Min, Ludwig Schmidt, Noah A Smith, and Mike Lewis. Measuring and narrowing the compositionality gap in language models. In Findings of the Association for Computational Linguistics: EMNLP 2023, pages 5687–5711, 2023.

[Pryzant et al., 2023] Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descentänd beam search. In The 2023 Conference on Empirical Methods in Natural Language Processing, 2023.

[Qiu et al., 2020] Xipeng Qiu, Tianxiang Sun, Yige Xu, Yunfan Shao, Ning Dai, and Xuanjing Huang. Pre-trained models for natural language processing: A survey. Science China Technological Sciences, 63(10):1872–1897, 2020.

[Radford et al., 2018] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. OpenAI, 2018.

[Radford et al., 2019] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. OpenAI blog, 1(8), 2019.

[Radford et al., 2021] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. In International conference on machine learning, pages 8748–8763. PMLR, 2021.

[Rae et al., 2019] Jack W Rae, Anna Potapenko, Siddhant M Jayakumar, Chloe Hillier, and Timothy P Lillicrap. Compressive transformers for long-range sequence modelling. In International Conference on Learning Representations, 2019.

[Rafailov et al., 2024] Rafael Rafailov, Archit Sharma, Eric Mitchell, Christopher D Manning, Stefano Ermon, and Chelsea Finn. Direct preference optimization: Your language model is secretly a reward model. Advances in Neural Information Processing Systems, 36, 2024.

[Raffel et al., 2020] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. Journal of Machine Learning Research, 21(140):1–67, 2020.

[Ramachandran et al., 2017] Prajit Ramachandran, Barret Zoph, and Quoc V Le. Searching for activation functions. arXiv preprint arXiv:1710.05941, 2017.

[Rolnick et al., 2019] David Rolnick, Arun Ahuja, Jonathan Schwarz, Timothy Lillicrap, and Gregory Wayne. Experience replay for continual learning. Advances in Neural Information Processing Systems, 32, 2019.

[Rosenfeld et al., 2020] Jonathan S Rosenfeld, Amir Rosenfeld, Yonatan Belinkov, and Nir Shavit. A constructive prediction of the generalization error across scales. In Proceedings of International Conference on Learning Representations, 2020.

[Ruan et al., 2024] Junhao Ruan, Long Meng, Weiqiao Shan, Tong Xiao, and Jingbo Zhu. A survey of llm surveys. https://github.com/NiuTrans/ABigSurveyOfLLMs, 2024.

[Rubin et al., 2022] Ohad Rubin, Jonathan Herzig, and Jonathan Berant. Learning to retrieve prompts for in-context learning. In Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pages 2655–2671, 2022.

[Russell, 2019] Stuart Russell. Human Compatible: Artificial Intelligence and the Problem of Controls. Viking, 2019.

[Sanh et al., 2020] Victor Sanh, Thomas Wolf, and Alexander Rush. Movement pruning: Adaptive sparsity by fine-tuning. Advances in Neural Information Processing Systems, 33:20378–20389, 2020.

[Sanh et al., 2022] Victor Sanh, Albert Webson, Colin Raffel, Stephen Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Arun Raja, Manan Dey, M Saiful Bari, Canwen Xu, Urmish Thakker, Shanya Sharma Sharma, Eliza Szczechla, Taewoon Kim, Gunjan Chhablani, Nihal Nayak, Debajyoti Datta, Jonathan Chang, Mike Tian-Jian Jiang, Han Wang, Matteo Manica, Sheng Shen, Zheng Xin Yong, Harshit Pandey, Rachel Bawden, Thomas Wang, Trishala Neeraj, Jos Rozen, Abheesht Sharma, Andrea Santilli, Thibault Fevry, Jason Alan Fries, Ryan Teehan, Teven Le Scao, Stella Biderman, Leo Gao, Thomas Wolf, and Alexander M Rush. Multitask prompted training enables zero-shot task generalization. In Proceedings of International Conference on Learning Representations, 2022.

[Schick et al., 2023] Timo Schick, Jane A. Yu, Zhengbao Jiang, Fabio Petroni, Patrick Lewis, Gautier Izacard, Qingfei You, Christoforos Nalmpantis, Edouard Grave, and Sebastian Riedel. PEER: A collaborative language model. In Proceedings of The Eleventh International Conference on Learning Representations, 2023.

[Schick et al., 2024] Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer:

Language models can teach themselves to use tools. Advances in Neural Information Processing Systems, 36, 2024.

[Schmidhuber, 2015] Jürgen Schmidhuber. Deep learning in neural networks: An overview. Neural networks, 61:85–117, 2015.

[Schulman et al., 2015] John Schulman, Sergey Levine, Philipp Moritz, Michael Jordan, and Pieter Abbeel. Trust region policy optimization. In Proceedings of the 32nd International Conference on International Conference on Machine Learning-Volume 37, pages 1889–1897, 2015.

[Schulman et al., 2017] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. arXiv preprint arXiv:1707.06347, 2017.

[Sennrich et al., 2016] Rico Sennrich, Barry Haddow, and Alexandra Birch. Improving neural machine translation models with monolingual data. In Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 86–96, 2016.

[Seo et al., 2017] Minjoon Seo, Aniruddha Kembhavi, Ali Farhadi, and Hannaneh Hajishirzi. Bidirectional attention flow for machine comprehension. In Proceedings of International Conference on Learning Representations, 2017.

[Shannon, 1951] Claude E Shannon. Prediction and entropy of printed english. Bell system technical journal, 30(1):50–64, 1951.

[Shaw et al., 2018] Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 2 (Short Papers), pages 464–468, 2018.

[Shazeer, 2019] Noam Shazeer. Fast transformer decoding: One write-head is all you need. arXiv preprint arXiv:1911.02150, 2019.

[Shazeer, 2020] Noam Shazeer. Glu variants improve transformer. arXiv preprint arXiv:2002.05202, 2020.

[Shen et al., 2020] Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Q-bert: Hessian based ultra low precision quantization of bert. In Proceedings of the AAAI Conference on Artificial Intelligence, volume 34, pages 8815–8821, 2020.

[Shinn et al., 2023] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. Advances in Neural Information Processing Systems, 36:8634–8652, 2023.

[Shoeybi et al., 2019] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. arXiv preprint arXiv:1909.08053, 2019.

[Skalse et al., 2022] Joar Skalse, Nikolaus Howe, Dmitrii Krasheninnikov, and David Krueger. Defining and characterizing reward gaming. Advances in Neural Information Processing Systems, 35:9460–9471, 2022.

[Snell et al., 2022] Charlie Snell, Dan Klein, and Ruiqi Zhong. Learning by distilling context. arXiv preprint arXiv:2209.15189, 2022.

[Snell et al., 2024] Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters. arXiv preprint arXiv:2408.03314, 2024.

[Snell et al., 2025] Charlie Victor Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning. In The Thirteenth International Conference on Learning Representations, 2025.

[Socher et al., 2013] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In Proceedings of the 2013 conference on empirical methods in natural language processing, pages 1631–1642, 2013.

[Song et al., 2019] Kaitao Song, Xu Tan, Tao Qin, Jianfeng Lu, and Tie-Yan Liu. Mass: Masked sequence to sequence pre-training for language generation. In International Conference on Machine Learning, pages 5926–5936. PMLR, 2019.

[Stiennon et al., 2020] Nisan Stiennon, Long Ouyang, Jeffrey Wu, Daniel Ziegler, Ryan Lowe, Chelsea Voss, Alec Radford, Dario Amodei, and Paul F Christiano. Learning to summarize with human feedback. Advances in Neural Information Processing Systems, 33:3008–3021, 2020.

[Su et al., 2024] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. Neurocomputing, 568: 127063, 2024.

[Su et al., 2022] Yixuan Su, Tian Lan, Yan Wang, Dani Yogatama, Lingpeng Kong, and Nigel Collier. A contrastive framework for neural text generation. Advances in Neural Information Processing Systems, 35:21548–21561, 2022.

[Sun et al., 2020] Zhiqing Sun, Hongkun Yu, Xiaodan Song, Renjie Liu, Yiming Yang, and Denny Zhou. Mobilebert: a compact task-agnostic bert for resource-limited devices. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pages 2158–2170, 2020.

[Sutskever et al., 2014] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. Advances in neural information processing systems, 27, 2014.

[Sutton and Barto, 2018] Richard S. Sutton and Andrew G. Barto. Reinforcement Learning: An Introduction (2nd ed.). The MIT Press, 2018.

[Szepesvári, 2010] Csaba Szepesvári. Algorithms for reinforcement learning. Synthesis Lectures on Artificial Intelligence and Machine Learning, 4(1):1–103, 2010.

[Talmor and Berant, 2018] Alon Talmor and Jonathan Berant. The web as a knowledge-base for answering complex questions. arXiv preprint arXiv:1803.06643, 2018.

[Taori et al., 2023] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca, 2023.

[Tay et al., 2020] Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. CoRR, abs/2009.06732, 2020.

[Team et al., 2024] Gemma Team, Morgane Riviere, Shreya Pathak, Pier Giuseppe Sessa, Cassidy Hardin, Surya Bhupatiraju, Léonard Hussenot, Thomas Mesnard, Bobak Shahriari, Alexandre Ramé, et al. Gemma 2: Improving open language models at a practical size. arXiv preprint arXiv:2408.00118, 2024.

[Teknium, 2023] Teknium. Openhermes 2.5: An open dataset of synthetic data for generalist llm assistants, 2023. URL https://huggingface.co/datasets/teknium/OpenHermes-2.5.

[Timonin et al., 2022] Denis Timonin, BoYang Hsueh, and Vinh Nguyen. Accelerated inference for large transformer models using nvidia triton inference server. https://developer.nvidia.com/blog/accelerated-inference-for-large-transformer-models-using-nvidia-fastertransformer-and-nvidia-tr 2022.

[Touvron et al., 2023] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and

efficient foundation language models. arXiv preprint arXiv:2302.13971, 2023a.

[Touvron et al., 2023] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models. arXiv preprint arXiv:2307.09288, 2023b.

[Uesato et al., 2022] Jonathan Uesato, Nate Kushman, Ramana Kumar, Francis Song, Noah Siegel, Lisa Wang, Antonia Creswell, Geoffrey Irving, and Irina Higgins. Solving math word problems with process-and outcome-based feedback. arXiv preprint arXiv:2211.14275, 2022.

[Vaswani et al., 2017] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In Proceedings of Advances in Neural Information Processing Systems, volume 30, 2017.

[Viterbi, 1967] Andrew J Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. IEEE Transactions on Information Theory, 1967.

[Von Oswald et al., 2023] Johannes Von Oswald, Eyvind Niklasson, Ettore Randazzo, João Sacramento, Alexander Mordvintsev, Andrey Zhmoginov, and Max Vladymyrov. Transformers learn in-context by gradient descent. In Proceedings of International Conference on Machine Learning, pages 35151–35174. PMLR, 2023.

[Wang et al., 2024] Chenglong Wang, Hang Zhou, Yimin Hu, Yifu Huo, Bei Li, Tongran Liu, Tong Xiao, and Jingbo Zhu. Esrl: Efficient sampling-based reinforcement learning for sequence generation. In Proceedings of the AAAI Conference on Artificial Intelligence, pages 19107–19115, 2024.

[Wang et al., 2023] Liyuan Wang, Xingxing Zhang, Hang Su, and Jun Zhu. A comprehensive survey of continual learning: Theory, method and application. arXiv preprint arXiv:2302.00487, 2023a.

[Wang et al., 2019] Qiang Wang, Bei Li, Tong Xiao, Jingbo Zhu, Changliang Li, Derek F Wong, and Lidia S Chao. Learning deep transformer models for machine translation. In Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics, pages 1810–1822, 2019.

[Wang et al., 2022] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, and Denny Zhou. Rationale-augmented ensembles in language models. arXiv preprint arXiv:2207.00747, 2022a.

[Wang et al., 2023] Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In Proceedings of The Eleventh International Conference on Learning Representations, 2023b.

[Wang et al., 2022] Yizhong Wang, Swaroop Mishra, Pegah Alipoormolabashi, Yeganeh Kordi, Amirreza Mirzaei, Atharva Naik, Arjun Ashok, Arut Selvan Dhanasekaran, Anjana Arunkumar, David Stap, Eshaan Pathak, Giannis Karamanolakis, Haizhi Gary Lai, Ishan Purohit, Ishani Mondal, Jacob Anderson, Kirby Kuznia, Krima Doshi, Kuntal Kumar Pal, Maitreya

Patel, Mehrad Moradshahi, Mihir Parmar, Mirali Purohit, Neeraj Varshney, Phani Rohitha Kaza, Pulkit Verma, Ravsehaj Singh Puri, Rushang Karia, Savan Doshi, Shailaja Keyur Sampat, Siddhartha Mishra, Sujan Reddy A, Sumanta Patro, Tanay Dixit, and Xudong Shen. Super-naturalinstructions: Generalization via declarative instructions on 1600+ nlp tasks. In Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing, pages 5085–5109, 2022b.

[Wang et al., 2023] Yizhong Wang, Hamish Ivison, Pradeep Dasigi, Jack Hessel, Tushar Khot, Khyathi Raghavi Chandu, David Wadden, Kelsey MacMillan, Noah A. Smith, Iz Beltagy, and Hannaneh Hajishirzi. How far can camels go? exploring the state of instruction tuning on open resources. Advances in Neural Information Processing Systems, 36:74764–74786, 2023c.

[Wang et al., 2023] Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. In Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 13484–13508, 2023d.

[Wang et al., 2023] Zhenyi Wang, Enneng Yang, Li Shen, and Heng Huang. A comprehensive survey of forgetting in deep learning beyond continual learning. arXiv preprint arXiv:2307.09218, 2023e.

[Warstadt et al., 2019] Alex Warstadt, Amanpreet Singh, and Samuel R Bowman. Neural network acceptability judgments. Transactions of the Association for Computational Linguistics, 7:625–641, 2019.

[Wei et al., 2022] Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. Finetuned language models are zero-shot learners. In Proceedings of International Conference on Learning Representations, 2022a.

[Wei et al., 2022] Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, Ed H. Chi, Tatsunori Hashimoto, Oriol Vinyals, Percy Liang, Jeff Dean, and William Fedus. Emergent abilities of large language models. arXiv preprint arXiv:2206.07682, 2022b.

[Wei et al., 2022] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. Advances in Neural Information Processing Systems, 35:24824–24837, 2022c.

[Welleck et al., 2023] Sean Welleck, Ximing Lu, Peter West, Faeze Brahman, Tianxiao Shen, Daniel Khashabi, and Yejin Choi. Generating sequences by learning to self-correct. In Proceedings of The Eleventh International Conference on Learning Representations, 2023.

[Weng, 2021] Lilian Weng. How to train really large models on many gpus? lilianweng.github.io, Sep 2021. URL https://lilianweng.github.io/posts/2021-09-25-train-large/.

[Wiener, 1960] Norbert Wiener. Some moral and technical consequences of automation: As machines learn they may develop unforeseen strategies at rates that baffle their programmers. Science, 131(3410):1355–1358, 1960.

[Williams et al., 2018] Adina Williams, Nikita Nangia, and Samuel Bowman. A broad-coverage challenge corpus for sentence understanding through inference. In Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers), pages 1112–1122, 2018.

[Williams, 1992] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. Machine learning, 8:229–256, 1992.

[Wingate et al., 2022] David Wingate, Mohammad Shoeybi, and Taylor Sorensen. Prompt compression and contrastive conditioning for controllability and toxicity reduction in language models. In Findings of the Association for Computational Linguistics: EMNLP 2022, pages 5621–5634, 2022.

[Wozengraft and Reiffen, 1961] John M. Wozengraft and Barney Reiffen. Sequential Decoding. The MIT Press, 1961.

[Wu et al., 2023] Bingyang Wu, Yinmin Zhong, Zili Zhang, Shengyu Liu, Fangyue Liu, Yuanhang Sun, Gang Huang, Xuanzhe Liu, and Xin Jin. Fast distributed inference serving for large language models. arXiv preprint arXiv:2305.05920, 2023a.

[Wu et al., 2024] Wilson Wu, John X Morris, and Lionel Levine. Do language models plan for future tokens? arXiv preprint arXiv:2404.00859, 2024.

[Wu et al., 2021] Yuhuai Wu, Markus Norman Rabe, DeLesley Hutchins, and Christian Szegedy. Memorizing transformers. In Proceedings of International Conference on Learning Representations, 2021.

[Wu et al., 2023] Zeqiu Wu, Yushi Hu, Weijia Shi, Nouha Dziri, Alane Suhr, Prithviraj Ammanabrolu, Noah A. Smith, Mari Ostendorf, and Hannaneh Hajishirzi. Fine-grained human feedback gives better rewards for language model training. In Thirty-seventh Conference on Neural Information Processing Systems, 2023b.

[Xia et al., 2024] Mengzhou Xia, Sadhika Malladi, Suchin Gururangan, Sanjeev Arora, and Danqi Chen. Less: Selecting influential data for targeted instruction tuning. arXiv preprint arXiv:2402.04333, 2024.

[Xiao et al., 2024] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. Efficient streaming language models with attention sinks. In Proceedings of The Twelfth International Conference on Learning Representations, 2024.

[Xiao and Zhu, 2023] Tong Xiao and Jingbo Zhu. Introduction to transformers: an nlp perspective. arXiv preprint arXiv:2311.17633, 2023.

[Xiao et al., 2013] Tong Xiao, Jingbo Zhu, and Tongran Liu. Bagging and boosting statistical machine translation systems. Artificial Intelligence, 195:496–527, 2013.

[Xiao et al., 2019] Tong Xiao, Yinqiao Li, Jingbo Zhu, Zhengtao Yu, and Tongran Liu. Sharing attention weights for fast transformer. In Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence (IJCAI-19), pages 5292–5298, 2019.

[Xie et al., 2022] Sang Michael Xie, Aditi Raghunathan, Percy Liang, and Tengyu Ma. An explanation of in-context learning as implicit bayesian inference. In Proceedings of International Conference on Learning Representations, 2022.

[Xin et al., 2020] Ji Xin, Raphael Tang, Jaejun Lee, Yaoliang Yu, and Jimmy Lin. Deebert: Dynamic early exiting for accelerating bert inference. In Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, pages 2246–2251, 2020.

[Xu et al., 2024] Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, Qingwei Lin, and Daxin Jiang. Wizardlm: Empowering large pre-trained language models to follow complex instructions. In The Twelfth International Conference on Learning Representations, 2024.

[Yang et al., 2024] An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, et al. Qwen2. 5 technical report. arXiv preprint arXiv:2412.15115, 2024.

[Yang et al., 2019] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Russ R Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. Advances in neural information processing systems, 32, 2019.

[Yao et al., 2024] Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. Advances in Neural Information Processing Systems, 36, 2024.

[Yarowsky, 1995] David Yarowsky. Unsupervised word sense disambiguation rivaling supervised methods. In Proceedings of the 33rd annual meeting of the association for computational linguistics, pages 189–196, 1995.

[Yu et al., 2022] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for {Transformer-Based} generative models. In 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), pages 521–538, 2022.

[Yu et al., 2023] Zihan Yu, Liang He, Zhen Wu, Xinyu Dai, and Jiajun Chen. Towards better chain-of-thought prompting strategies: A survey. arXiv preprint arXiv:2310.04959, 2023.

[Zaheer et al., 2020] Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, C. Alberti, S. Ontañón, Philip Pham, Anirudh Ravula, Qifan Wang, L. Yang, and A. Ahmed. Big bird: Transformers for longer sequences. Advances in neural information processing systems, 33:17283–17297, 2020.

[Zellers et al., 2018] Rowan Zellers, Yonatan Bisk, Roy Schwartz, and Yejin Choi. Swag: A large-scale adversarial dataset for grounded commonsense inference. In Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, pages 93–104, 2018.

[Zhang and Sennrich, 2019] Biao Zhang and Rico Sennrich. Root mean square layer normalization. Advances in Neural Information Processing Systems, 32, 2019.

[Zhang et al., 2024] Yunxiang Zhang, Muhammad Khalifa, Lajanugen Logeswaran, Jaekyeom Kim, Moontae Lee, Honglak Lee, and Lu Wang. Small language models need strong verifiers to self-correct reasoning. In ACL (Findings), 2024.

[Zhang et al., 2023] Zhuosheng Zhang, Yao Yao, Aston Zhang, Xiangru Tang, Xinbei Ma, Zhiwei He, Yiming Wang, Mark Gerstein, Rui Wang, Gongshen Liu, and Hai Zhao. Igniting language intelligence: The hitchhiker's guide from chain-of-thought reasoning to language agents. arXiv preprint arXiv:2311.11797, 2023a.

[Zhang et al., 2023] Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. Automatic chain of thought prompting in large language models. In The Eleventh International Conference on Learning Representations, 2023b.

[Zhao et al., 2024] Hao Zhao, Maksym Andriushchenko, Francesco Croce, and Nicolas Flammarion. Long is more for alignment: A simple but tough-to-beat baseline for instruction fine-tuning. arXiv preprint arXiv:2402.04833, 2024.

[Zhao et al., 2023] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Z. Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jianyun Nie, and Ji rong Wen. A survey of large language models. arXiv preprint arXiv:2303.18223, 2023.

[Zhong et al., 2024] Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. {DistServe}: Disaggregating prefill and decoding for goodput-optimized large language model serving. In 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), pages 193–210, 2024.

[Zhou et al., 2023] Chunting Zhou, Pengfei Liu, Puxin Xu, Srini Iyer, Jiao Sun, Yuning Mao, Xuezhe Ma, Avia Efrat, Ping Yu, Lili Yu, Susan Zhang, Gargi Ghosh, Mike Lewis, Luke Zettlemoyer, and Omer Levy. Lima: Less is more for alignment. arXiv preprint arXiv:2305.11206, 2023a.

[Zhou et al., 2023] Denny Zhou, Nathanael Schärli, Le Hou, Jason Wei, Nathan Scales, Xuezhi Wang, Dale Schuurmans, Claire Cui, Olivier Bousquet, Quoc V. Le, and Ed H. Chi. Least-to-most prompting enables complex reasoning in large language models. In Proceedings of The Eleventh International Conference on Learning Representations, 2023b.

[Zhou et al., 2020] Wangchunshu Zhou, Canwen Xu, Tao Ge, Julian McAuley, Ke Xu, and Furu Wei. Bert loses patience: Fast and robust inference with early exit. Advances in Neural

Information Processing Systems, 33:18330–18341, 2020.

[Zhou et al., 2023] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. In The Eleventh International Conference on Learning Representations, 2023c.

[Zoph and Le, 2016] Barret Zoph and Quoc Le. Neural architecture search with reinforcement learning. In Proceedings of International Conference on Learning Representations, 2016.

[Zoph et al., 2020] Barret Zoph, Golnaz Ghiasi, Tsung-Yi Lin, Yin Cui, Hanxiao Liu, Ekin Dogus Cubuk, and Quoc Le. Rethinking pre-training and self-training. Advances in neural information processing systems, 33:3833–3845, 2020.

Index

k-NN, 88
k-NN LM, 89
k-NN-Sprachmodellierung, 89
k-nächste Nachbarn, 88

A2C, 207
abruf-erweiterte Generierung, 90
Advantage Actor-Critic, 207
Agent, 55
Aktionswertfunktion, 203
ALiBi, 99
Alignment, 54
Anweisungs-Feinabstimmung, 182
attention with linear biases, 99
automatisches Prompt-Design, 161
automatisiertes maschinelles Lernen, 161
AutoML, 161
autonome Agenten, 158

BART, 22
Berechnungsanmerkung, 131
BERT, 1
Best-of-N sampling, 232
bestärkendes Lernen durch menschliches Feedback, 55
BoN sampling, 233
Bradley-Terry-Modell, 210

catastrophic forgetting, 40
chain-of-thought prompting, 62
completion, 7
Continuous batching, 262
CoT, 132
COT prompting, 62

Decoding, 242
deliberate-then-generate, 149
demonstrations, 8
direkte Präferenzoptimierung, 224
Dokument-Rotation, 23
DPO, 224
DTG, 149

einrundige Vorhersage, 183
emergente Fähigkeiten, 74
Ergebnisbasierte Ansätze, 230
ergebnisbasierten Belohnungsmodellen, 283
externe Speicher, 88
Extrapolation, 95

few-shot COT prompting, 63

Gaußsche Fehler-Lineareinheit, 69
Gedankenkette, 132
GeLU, 69
Generierung von Teilproblemen, 138
gesteuerten linearen Einheit, 69
gleichgerichtete lineare Einheit, 68
GLU, 69
GPT, 1
GQA, 93
Grouped query attention, 93

harte Prompts, 165
human preference alignment, 179

ICL, 62
ICT, 8
Importance Sampling, 212
in-context learning, 8, 62
In-Kontext-Lernen, 111
Inference Engine, 261
input inversion, 192
instruction alignment, 179
Instruktions-Feinabstimmung, 51
Interferenz, 34
interne Speicher, 88
Interpolation, 96
iteration-based scheduling, 262

kausale Sprachmodellierung, 11
Key-Value-Cache, 80, 240
kompositionellen Generalisierung, 143
kreuzlingualen Sprachmodellen, 33
kumulative Belohnung, 203
KV-Cache, 80, 240

Label-Mapping, 122
Least-to-Most-Prompting, 139
Leistungsfunktion, 204
Leistungsschätzung, 161
Lernen aus menschlichem Feedback, 55
LLMs mit langem Kontext, 77
Lösung von Teilproblemen, 138

maskierte Sprachmodellierung, 1, 11
mBERT, 32
mehrsprachige BERT, 32
MQA, 93
multi-query attention, 93

NAS, 161
nicht reduzierbaren Fehler, 75
NSP, 15
nucleus sampling, 251
Nächster-Satz-Vorhersage, 15

Offline-Reinforcement-Learning, 227
one-shot COT prompting, 63
ORMs, 283

parallel scaling, 279
permutierten Sprachmodellierung, 12
PGR, 198
Plackett-Luce-Modell, 217
PPO, 59, 214
Prefilling, 242
prefix fine-tuning, 169
PRM, 283
Problemdekomposition, 136
Prompt-Einbettungen, 174
Prompt-Engineering, 111
Prompt-Optimierung, 161
Prompt-Suchraum, 161
prompting engineering, 60
Proximale Policy-Optimierung, 214
proximale Policy-Optimierung, 59
Prozessbasierte Ansätze, 230
prozessbasierte Belohnungsmodell, 283
Präfix-Sprachmodellierung, 18

Q-Wert-Funktion, 203

RAG, 90
reinforcement learning from human feedback, 180
rejection sampling, 233
relation extraction, 125
ReLU, 68
request-level scheduling, 262
Return, 203
Reward Gaming, 222
Reward Hacking, 222
Reward Model, 56
RLHF, 55, 180
RoBERTa, 31

Satz-Neuordnung, 22
Scheduler, 261
Schwach-zu-Stark-Generalisierung, 197
Selbsttraining, 4
selbstüberwachtes Lernen, 4
self-consistency, 152
self-instruct, 190
Sequence Encoding Models, 4
Sequence Generation Models, 4
sequential scaling, 279
SFT, 55, 180
Skalierung zur Inferenzzeit, 272
Skalierungsgesetze, 74
Span-Maskierung, 22
speicherbasierten Methoden, 88
Spekulative Dekodierung, 253
spekulativen Ausführung, 253
stichprobeneffizient, 194
Strong Ceiling Performance, 197
Suche nach neuronalen Architekturen, 161
superficial alignment hypothesis, 193
supervised fine-tuning, 180
Surrogat-Ziel, 213

T5, 17
TD, 208
temporale Differenz, 208
Texttransformation, 127
Textvervollständigung, 127
Token-Löschung, 22
Token-Maskierung, 22
Transformers, 2
Trust Regions, 213

unüberwachtes Lernen, 3

Verhältnisfunktion, 212
Vorteil, 207

Weak Performance, 197
Weak-to-strong Performance, 197
weichen Prompts, 165
wiederhergestellte Leistungslücke, 198

XLMs, 33

zero-shot COT, 63

Zero-Shot-Lernen, 53
Zustandswertfunktion, 202

Überoptimierungsproblem, 222
Übersetzungs-Sprachmodellierung, 33
Überwachtes Fine-Tuning, 55
überwachtes Lernen, 3