

Effiziente Reduktion großer Sprachmodelle: Methodenvergleich und Anwendungsstrategien für Energieoptimierung und Hardware-Kompatibilität

Efficient reduction of Large Language Models: Comparing methods and develop strategies to optimize energy footprint and hardware compatibility

MASTERARBEIT

Dr. Thomas Schmitt
Fachhochschule Südwestfalen
26. September 2024

Autor: Dr. Thomas Schmitt
Referent: Prof. Dr. Heiner Giefers
Korreferent: Max Kuhmichel
Eingereicht: 26. September 2024

Zusammenfassung

Große Sprachmodelle (Englisch: Large Language Models oder kurz LLMs) wie „Chat-GPT“ oder „Gemini“ haben erstaunliche Fähigkeiten entwickelt und Einzug in unseren Alltag gefunden. Zur Entwicklung und dem Betrieb solcher LLMs benötigt man erhebliche Ressourcen, insbesondere leistungsfähige Grafikkarten (Englisch kurz: GPUs) mit großen Speicherkapazitäten. Daher werden diese Modelle z.Zt. fast ausschließlich von kommerziellen Anbietern in großen Rechenzentren betrieben und setzen somit eine Internet Verbindung zur Nutzung voraus.

In dieser Arbeit wird die Möglichkeit untersucht LLMs zu komprimieren, ohne dabei wesentliche Verluste ihrer Fähigkeiten in Kauf nehmen zu müssen. Existierende Technologien zur Kompression werden untersucht und kombiniert, um möglichst kleine Modelle zu erhalten, die im Idealfall auf normaler Consumer-Hardware kostengünstig betrieben werden können und so neue (Offline) Einsatzszenarien erschließen.

Inhaltsverzeichnis

Zusammenfassung	iii
1 Einleitung	1
1.1 Einführung zu Natural Language Processing	1
1.1.1 Entstehung von Natural Language Processing	1
1.1.2 Transformers als Kerntechnologie moderner Sprachmodelle	2
1.2 Large Language Models (LLMs)	3
1.2.1 Entstehung der LLMs	3
1.2.2 Entwicklung seit 2022	4
1.2.3 Ressourcenbedarf von LLMs	5
1.2.4 Ressourcenbedarf für das Training	6
1.2.5 Ressourcenbedarf bei der Nutzung	7
1.2.6 Verfahren zur Verringerung der Ressourcenanforderungen von LLMs	8
1.2.7 Zielsetzung der Arbeit	8
2 Material und Methoden	11
2.1 Quantisierung von LLMs	11
2.1.1 Zeitpunkt der Quantisierung	12
2.1.2 Verfahren für Post-Training-Quantisierung	12
2.1.3 Verwendetes Verfahren zur Quantisierung	13
2.2 Pruning von LLMs	13
2.2.1 Unstrukturiertes Pruning	14
2.2.2 Strukturiertes Pruning	14
2.2.3 Semi-strukturiertes Pruning	14
2.2.4 Verwendetes Verfahren für das Pruning	14
2.3 Knowledge Distillation	16
2.3.1 Response based Knowledge Distillation (KD)	16
2.3.2 Verwendetes Verfahren für die Knowledge Distillation	17
2.4 Low Rank Adaption (LoRA)	17
2.4.1 Funktionsweise von LoRA	17
2.4.2 QLora	18
2.5 Python Bibliotheken zur Arbeit mit LLMs	19
2.5.1 Die Bibliothek Transformers	19
2.5.2 Das Framework torch (pytorch)	19
2.5.3 Die Bibliothek PEFT	20
2.6 Beurteilung der Leistungsfähigkeit eines LLMs	20
2.6.1 Verwendete Benchmarks	21
2.6.2 Perplexity als Qualitätsmarker	21
2.6.3 Inferenz Performance und Speicherauslastung	22
2.7 Verwendete Modelle	22
2.7.1 Meta Llama 2	22
2.7.2 Meta Llama 3	23

2.7.3 Mistral	23
3 Implementation	25
3.1 Quantisierung	25
3.1.1 Quantisierung: AWQ	25
3.1.2 Quantisierung: GPTQ	28
3.2 Pruning: PruneMe	30
3.2.1 Überblick PruneMe Verfahren	30
3.2.2 Erstellen der Arbeitsumgebung	31
3.2.3 PruneMe Workflow	31
3.3 Pruning und LoRA: Shortened-LLM	34
3.3.1 Workflow zur Modell Optimierung	34
3.3.2 Ergebnisse des Workflows	35
3.4 QLoRA	36
3.4.1 Erstellen und konfigurieren der Arbeitsumgebung	36
3.4.2 Qlora Training Workflow	36
3.4.3 Prepare Training	40
3.4.4 Start Training	41
3.4.5 Speichern des Modells	41
3.5 Dual-Space Knowledge Distillation (DSKD)	42
3.5.1 Erstellen der Arbeitsumgebung	42
3.5.2 Prozess der Knowledge Distillation	42
3.6 Evaluierung der Modelle	44
3.6.1 Evaluierung der Modelle mit verschiedenen Benchmarks	44
3.6.2 Inferenz Messung	46
4 Ergebnisse	51
4.1 Experimente zur Quantisierung von Modellen	51
4.1.1 Benchmark Ergebnisse der AWQ Quantisierung	51
4.1.2 Speicherbedarf der AWQ Modelle im Vergleich zum Ausgangsmodell	52
4.1.3 AWQ Quantisierung mit verschiedenen Werten für den Groupsize Parameter	52
4.1.4 GPTQ Quantisierung und Vergleich mit AWQ	53
4.1.5 Verschiedene Bitbreiten bei der GPTQ Quantisierung	53
4.2 Ergebnisse der Pruning Experimente mit „PruneMe“	54
4.3 Kombination von Verfahren zur Kompression von LLM Modellen	55
4.3.1 Ergebnisse der Experimente mit dem QLoRA Verfahren	55
4.3.2 Tuning eines geprunten Modells mittels LoRA („Shortened LLM“)	56
4.3.3 Kombination von Pruning, LoRA und Knowledge Distillation	57
4.3.4 Kombination von Pruning, LoRA und Quantisierung	57
4.3.5 Vergleichende Darstellung der erzeugten Modelle	57
4.3.6 Perplexity Ergebnisse	59
4.3.7 Inferenz Ergebnisse	59
4.4 Systemressourcen	61
4.4.1 Speicherbedarf der Modelle	61
4.4.2 Inferenz auf einem Desktop Computer	62
5 Diskussion der Ergebnisse	65
5.1 Ergebnisse der Experimente	65
5.1.1 Ergebnisse der Quantisierungs-Experimente	65
5.1.2 Ergebnisse der Pruning-Experimente	66

5.2	Kombinierte Verfahren	67
5.2.1	QloRA	67
5.2.2	Shortened LLM	67
5.2.3	Knowlegde Distillation	68
5.2.4	„Shortened LLM“ und AWQ	68
5.2.5	Fazit kombinierte Verfahren	68
5.2.6	Fazit der Experimente	69
5.3	Umfeld der Optimierungsverfahren	69
5.3.1	Fazit zum Umfeld der Optimierungsverfahren	70
5.4	Andere Verfahren zur Modellanpassung	70
5.5	Zusammenfassung und Ausblick	71
5.5.1	Zusammenfassung	71
5.5.2	Ausblick	72
Literatur		73
Abbildungsverzeichnis		77
Tabellenverzeichnis		79

1 Einleitung

1.1 Einführung zu Natural Language Processing

1.1.1 Entstehung von Natural Language Processing

Schon seit den 1950er Jahren gibt es Bestrebungen, natürliche Sprache für Maschinen verständlich zu machen. Das zugehörige Wissensfeld bezeichnet man dabei als „Natural Language Processing (kurz NLP)“ [1]. In dieser Disziplin wurden Techniken aus Linguistik und Informatik kombiniert, um die Struktur der Sprache programmatisch abbilden zu können. Ein erstes eindrucksvolles Beispiel erstellte Joseph Weizenbaum mit der Simulation ELIZA, die einen Psychotherapeuten nachahmte. Immerhin jeder fünfte Proband, glaubte mit einem echten Therapeuten zu interagieren, obwohl nur recht simple Grammatikregeln zur Simulation verwendet wurden [2]. ELIZA wird rückblickend auch gerne als der erste Chatbot bezeichnet.

```
Welcome to
      EEEEEE  LL      IIII    ZZZZZZ   AAAAAA
      EE     LL      II      ZZ      AA     AA
      EEEEEE  LL      II      ZZZ     AAAAAAAA
      EE     LL      II      ZZ      AA     AA
      EEEEEE  LLLLLL  IIII    ZZZZZZ   AA     AA

Eliza is a mock Rogerian psychotherapist.
The original program was described by Joseph Weizenbaum in 1966.
This implementation by Norbert Landsteiner 2005.

ELIZA: Is something troubling you ?
YOU: Men are all alike.
ELIZA: What is the connection, do you suppose ?
YOU: They're always bugging us about something or other.
ELIZA: Can you think of a specific example ?
YOU: Well, my boyfriend made me come here.
ELIZA: Is it important to you that your boyfriend made you come here ?
YOU: He says I'm depressed much of the time.
ELIZA: I am sorry to hear that you are depressed.
YOU: It's true. I am unhappy.
ELIZA: Can you explain what made you unhappy ?
YOU:
```

Abbildung 1.1: Beispiel einer ELIZA Konversation [1].

Einige Jahrzehnte waren die Fortschritte im Bereich NLP gering. Erst in den letzten Jahren wurden durch die rasante Entwicklung des Machine Learnings und speziell des Deep Learnings, auch im Bereich der Sprachadaption für Computer erhebliche Verbesserungen erzielt. Das Aufkommen des Deep Learnings markierte einen Paradigmenwechsel in der Entwicklung von Sprachmodellen. Neuronale Netze, insbesondere rekurrente neuronale Netze (RNN) und Netze mit langem Kurzzeitgedächtnis (LSTMs), brachten bemerkenswerte Verbesserungen bei

der Verarbeitung sequenzieller Daten.

Diese frühen Deep-Learning-Modelle ermöglichten ein besseres Sprachverständnis, ihre Skalierbarkeit war jedoch begrenzt, da die Eingangssequenz bei diesen Modellen sequenziell, also Wort für Wort, abgearbeitet wurde, was ihr Verständnis komplexerer Texte erheblich einschränkte.

1.1.2 Transformers als Kerntechnologie moderner Sprachmodelle

Ein technologisch bedeutender Schritt zu leistungsfähigeren Sprachmodellen war die Entwicklung der sogenannten Transformer Modelle, die in [3] erstmalig 2017 beschrieben wurden. Da alle aktuellen LLMs auf dieser Technologie oder Varianten davon basieren, erfolgt hier eine kurze Vorstellung des Konzepts.

Im Gegensatz zu den zuvor genannten Verfahren heben sich Transformer dadurch ab, dass sie in der Lage sind ganze Absätze bzw. Abschnitte parallel zu verarbeiten und somit Beziehungen zwischen weiter entfernten Textteilen erfassen können. Dieser Unterschied ist im Wesentlichen für die stark verbesserten Fähigkeiten, die auf dieser Technologie basierenden Transformer-Modelle verantwortlich.

Grundlage für die Erfassung weiter entfernter textlicher Abhängigkeiten ist der Aufmerksamkeitsmechanismus (engl. Attention), welcher in der Lage ist, textliche Beziehungen zu bewerten und mathematisch abzubilden. 1.2 zeigt den Aufbau einer Transformer Architektur, wie er in [3] diskutiert wird.

Ein Transformer besteht in diesem Beispiel aus einer Encoder und einer Decoder Struktur die über einen zweiten Aufmerksamkeitsmechanismus gekoppelt sind.

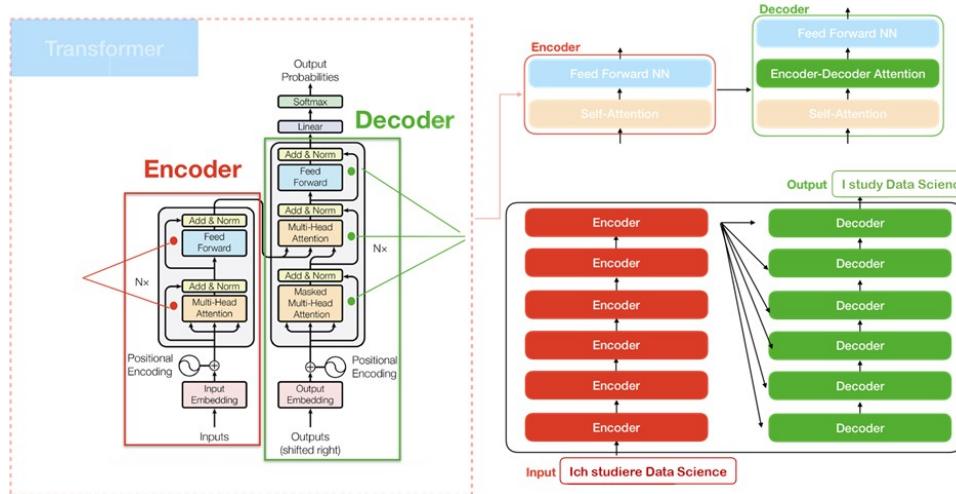


Abbildung 1.2: Schematische Darstellung der Transformer Architektur mit einem Beispiel [4]

Zunächst werden im Embedding-Layer des Encoders die einzelnen Wörter der Eingabe in einen Vektor abgebildet. Diese Vektoren fließen dann durch die Self-Attention und Feed-Forward-Schicht des Encoders. Entscheidend ist dabei, dass jedes Wort in jeder Position durch verschiedene Pfade des Encoders fließt.

Die Self-Attention Schicht ist dadurch in der Lage, die relative Bedeutung eines Wortes, im Verhältnis zu den anderen Wörtern des Textes, zu erfassen und bildet diese relativen

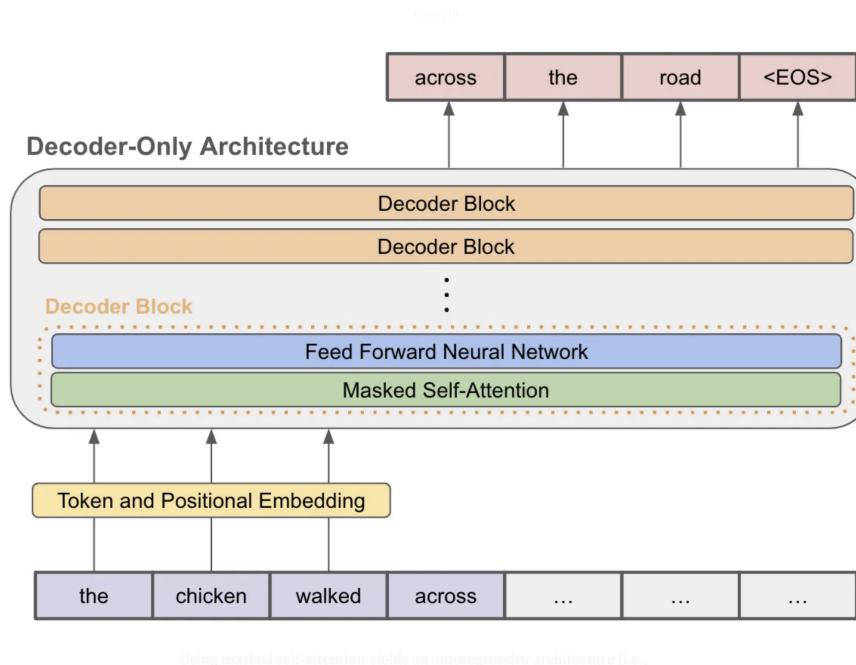


Abbildung 1.3: Schematische Darstellung einer Decoder Architektur [7].

Positionsinformationen in weiteren Vektoren ab.

Auf Seite des Decoders wird derselbe Prozess durchgeführt, wobei der einzige Unterschied dabei die Cross-Attention-Schicht ist, welche die erlernten Attention-Signale des Encoders auf den Decoder überträgt. Diese, hier als Multi-Head-Attention bezeichnete Schicht, unterstützt den Decoder bei der Konzentration auf die relevanten Wörter des Eingabesatzes. Letztlich bestimmt der lineare Layer eines neuronalen Netzes, dann das wahrscheinlichste nächste Wort in der Ausgabesequenz.

Eines der ersten Modelle, welches die Fähigkeiten dieser Technologie demonstrierte, war BERT [5], dessen Transformer Architektur als „encoder only transformer“ Struktur bezeichnet wird. Bei den GPT-Modellen von OpenAI [6] wird hingegen eine „decoder only structure“ verwendet. Welche Struktur gewählt wird, ergibt sich vor allem aus dem Anforderungsprofil für das Modell. Bei einem bidirektionalen Übersetzungsmodell verspricht eine „Encoder only Struktur“ deutlich bessere Ergebnisse im Sprachverständnis und damit in der Übersetzungsqualität. Für das einfache Generieren von Text durch einen Chatbot, überwiegen die Vorteile des „Decoder only“ Ansatzes.

1.2 Large Language Models (LLMs)

1.2.1 Entstehung der LLMs

Die neueren Arbeiten im Bereich Maschine Learning führten zu Sprachmodellen, die auf eine Frage oder Aufforderung mit einer Antwort reagieren, wie sie in ähnlicher Weise auch durch einen Menschen gegeben würde.

Solche Sprachmodelle entstehen i.d.R. durch das Training neuronaler Netze, mit großen Textmengen aus verschiedenen Disziplinen. Dies geschieht durch die Nutzung leicht zugänglicher

textlicher Informationen, wie Beispielsweise von Wikipedia Inhalten.

Man unterscheidet grundsätzlich die Trainingsphase von der Nutzungsphase der Modelle. Die Nutzungsphase wird dabei meist als „Inference“, was im Deutschen mit Schlussfolgerung übersetzt werden könnte, bezeichnet.

Eines der auf diese Weise generierten Sprachmodelle, ist das von OpenAI erstellte Chat GPT, das in der Version 3.5 am 30 November 2022 einer größeren Öffentlichkeit zugänglich gemacht wurde. Das Kürzel GPT steht dabei für „generative pretrained transformer“. Durch die einfache Interaktion, in Form eines Chatbots, konnten auch Laien leicht mit dem Sprachmodell interagieren und sich von den Möglichkeiten der KI einen Eindruck verschaffen.

Die Entwicklung von GPT-3.5 stellte auch vom Aufwand eine neue Dimension des Trainings für ein Sprachmodell dar, welcher in einem Modell mit 175 Milliarden Parametern resultierte. Spätestens seit diesem Zeitpunkt werden solche Modelle als Large Language Models (LLMs) bezeichnet [8].

Getrieben durch die eindrucksvollen Resultate, erfolgte in den nächsten Monaten eine intensive Auseinandersetzung von Medien, Gesellschaft und Politik mit unterschiedlichsten Aspekten, die der Einsatz von weit entwickelten KIs und insbesondere den Sprachmodellen mit sich bringt. Der Produktnname Chat GPT ist in der Allgemeinheit zu einem Synonym für Large Language Models und den breiten Einsatz solcher Modelle in verschiedene Lebensbereiche geworden. Chat GPT wurde seitdem mehrfach aktualisiert und liegt z. Zt (1.8.2024) in der Version 4.0 vor. [9]

1.2.2 Entwicklung seit 2022

Die Entwicklung neuer Sprachmodelle wurde in den letzten Jahren stark vorangetrieben. Insbesondere die amerikanischen Technologie Konzerne wie OpenAI, Google und Anthropic, aber auch europäische Firmen wie Mistral, erstellen neue Modelle mit einer immer größeren Parameteranzahl.

Besteht Chat GPT 3.5 noch aus 175 Milliarden Parametern, vermutet man, dass bereits GPT-4.0 (OpenAI) und Gemini 1.5 (Google) die Billionen Grenze überschritten haben. In einigen Fällen, wie bei Google Gemini, handelt es sich dabei nicht mehr um reine Sprach, sondern um multimodale Modelle, die verschiedene Arten von Daten parallel verarbeiten können. Dazu gehören neben Text, auch Sprache, Videos, Bilder und Programmcode. [10].

Daneben haben einige der großen genannten Technologie Konzerne auch Modelle veröffentlicht, die von der Allgemeinheit kostenfrei genutzt werden können. Hier seien insbesondere der Meta Konzern mit seinen Llama Modellen und das relativ neue französische Unternehmen Mistral AI mit seinem gleichnamigen Modell Mistral 7B [11] genannt. Modelle beider Unternehmen werden in dieser Arbeit genutzt und im Teil 2.7 noch genauer beschrieben.

Natürlich unterliegen auch diese Modelle durch ihre Lizenzbedingungen gewissen Beschränkungen, die auch im Bereich der Lehre beachtet werden müssen. Z. Bsp. muss bei der Weitergabe von den auf Llama basierenden Modellen, deren Herkunft angegeben werden. Betrachtet man alle Aspekte, bilden solche Modelle allerdings eine gute Möglichkeit für Forschung, Lehre und sogar die kommerzielle Nutzung im kleinen Rahmen.

Die in diesem Kapitel beschriebenen Modelle werden auch als „Foundation Models“ (Deutsch in etwa: Grundlagenmodell) bezeichnet, da sie auf einer sehr breiten Themenpalette trainiert

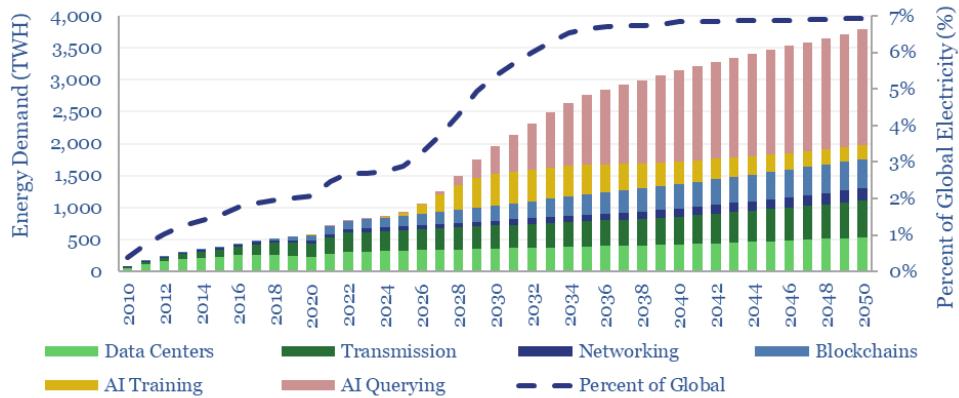


Abbildung 1.4: Internet energy consumption. [12].

werden und daher in vielen Bereichen einsetzbar sind. Darüber hinaus besitzen sie ein gutes Sprachverständnis, allgemeine Fähigkeit zu logischen Schlussfolgerungen und eine gute Lernkompetenz für weitere Inhalte.

In Ergänzung dazu werden viele speziellere Modelle von Unternehmen, Hochschulen und Enthusiasten erstellt, die häufig auf diesen „Foundation Modellen“ basieren, diese aber für spezifische Zwecke abwandeln.

Eine wichtige Richtung dieser Spezialisierung ist auch die Anpassung solcher Modelle auf einen geringeren Bedarf an GPU und Speicherkapazität, so dass sie in Umfeldern genutzt werden können, die über limitierte Ressourcen verfügen.

1.2.3 Ressourcenbedarf von LLMs

Das Training großer Sprachmodelle, aber auch deren Nutzung, geht mit einem erheblichen Ressourcenbedarf einher. Insbesondere sind die Grafikkarten (GPUs) des Herstellers Nvidia z.Zt. eine notwendige Voraussetzung für das Training und die Nutzung großer Sprachmodelle. Andere Hersteller von Grafikprozessoren, wie AMD oder Intel, schließen technologisch erst langsam zu Nvidia auf, so dass dieser eine Beinah-Monopolstellung für das Training von LLMs einnimmt. Neben den reinen Hardwarekosten werden, für Training und Nutzung der Modelle, auch erhebliche Mengen an Energie in Form von Strom für die Rechenzentren benötigt. In Abbildung 1.4 ist eine Extrapolation des Energieverbrauchs, durch verschiedene im Internet bereit gestellte IT-Services zu sehen, die von einer Consulting Agentur für Energie Technologie erstellt wurde [12]. Wenn diese Extrapolation eintrifft, würde sich im Jahr 2050 des Gesamtenergiebedarf in diesem Bereich, getrieben durch Nutzung von KI, mehr als verdoppeln und Internet basierte IT-Services fast 7 % des weltweiten Gesamtenergiebedarfs verursachen.

In einer Studie der Carnegie Melon University [13] wird der CO₂ Ausstoß für verschiedene Szenarien der KI Nutzung bestimmt. Hierbei zeigt sich, dass je nach Anwendung und verwendeten Modell erhebliche Unterschiede bestehen. So benötigt die Generierung eines Bildes, durch ein sehr aufwendiges Modell ca. 5000x mehr Energie, als eine Text Inferenz durch ein effizientes Modell.

Diese Informationen zeigen das es sinnvoll ist, eine Optimierung der Modelle auf ihren Arbeitsbereich hin vorzunehmen und deren Nutzung energieeffizient zu gestalten.

1.2.4 Ressourcenbedarf für das Training

Wie viele GPUs für das Training eines Modells in der Größe von GPT 3.5 verwendet wurden ist leider nicht veröffentlicht. Es wird aber gemeinhin davon ausgegangen, dass ca. 10.000 GPUs parallel für das Training des Modells benötigt wurden und zur Zeit ca. 30.000 GPUs für die Inference durch Anwender genutzt werden.

Die Firma Meta, mit ihren Llama Modellen, ist etwas offener in der Kommunikation, und gewährt in [14] einen Einblick in ihr Rechenzentrum, inklusive des geplanten weiteren Ausbaus in den nächsten Monaten bzw. Jahren.

Wurde Llama 3 noch auf „nur“ zwei Clustern mit je 24.000 GPUs gerechnet, plant Meta einen Ausbau der Kapazitäten bis Ende 2024 auf 350.000 GPUs der neuesten, leistungsfähigeren Generation.

Es ist daher leicht verständlich, dass alleine die finanziellen Aufwände für die benötigten GPUs, bei der Entwicklung eines neuen Modells, einen erheblichen Kostenblock darstellen, der in [15] mit ca. einem Drittel der Gesamtkosten beziffert wird.

Die dabei benötigte Hardware wird nicht immer, wie z.Bsp. bei Meta, in eigenen Rechenzentren betrieben, sondern zunehmend auch in den Rechenzentren der großen Cloud Computing Anbieter, wie Amazon oder Microsoft, implementiert und dann als Service bezogen. Dies verringert den Anlaufaufwand für die Entwicklung des Modells und begrenzt die Kapitalbindung durch anfängliche Hardwareinvestitionen.

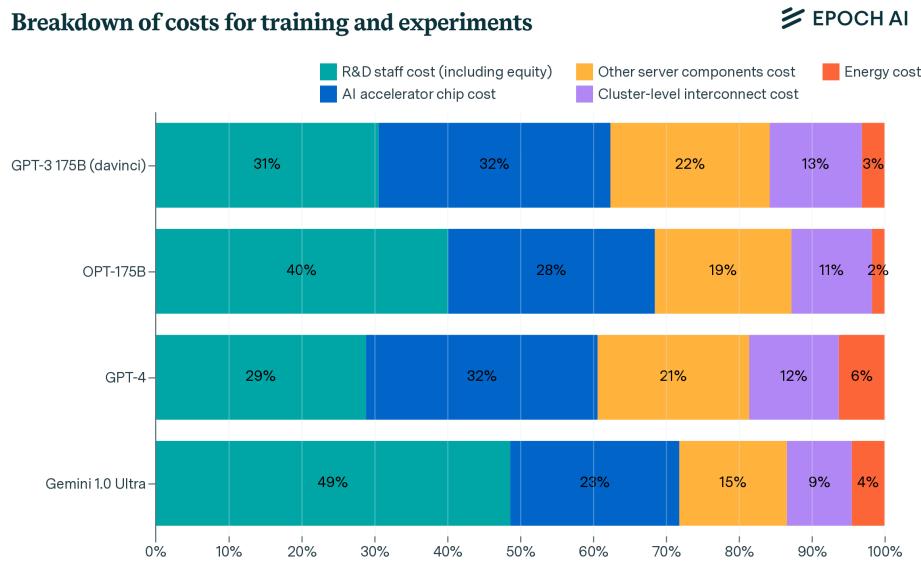


Abbildung 1.5: Kostenanteile bei der Entwicklung ausgewählter KIs [15]

Die Quelle [15] extrapoliert auch die Kostenentwicklung für zukünftige KI Modelle, die wie oben bereits erwähnt, deutlich mehr Parameter, als die Aktuellen enthalten werden [16]. Der Kostenanstieg wird dabei im Durchschnitt mit dem 2,4 fachen pro Jahr für die nächsten Jahre prognostiziert. Siehe auch Abbildung 1.6

Daher ist zu erwarten, dass neue Foundation Modelle in den kommenden Jahren hauptsächlich von sehr finanzkräftigen Konzernen entwickelt werden, welche in der Lage sind, die notwendigen Investitionen zu tätigen. Hochschulen und öffentliche Forschungsinstitute haben i.d.R. kaum die Mittel in ähnlichen Größenordnungen zu agieren.

Um so bemerkenswerter ist die Initiative der EU, den „Exascale Rechner“ Jupiter im Forschungs-

zentrum Jülich zu etablieren, der unter anderem auch für die Entwicklung von KI Foundation Modellen genutzt werden soll [17].

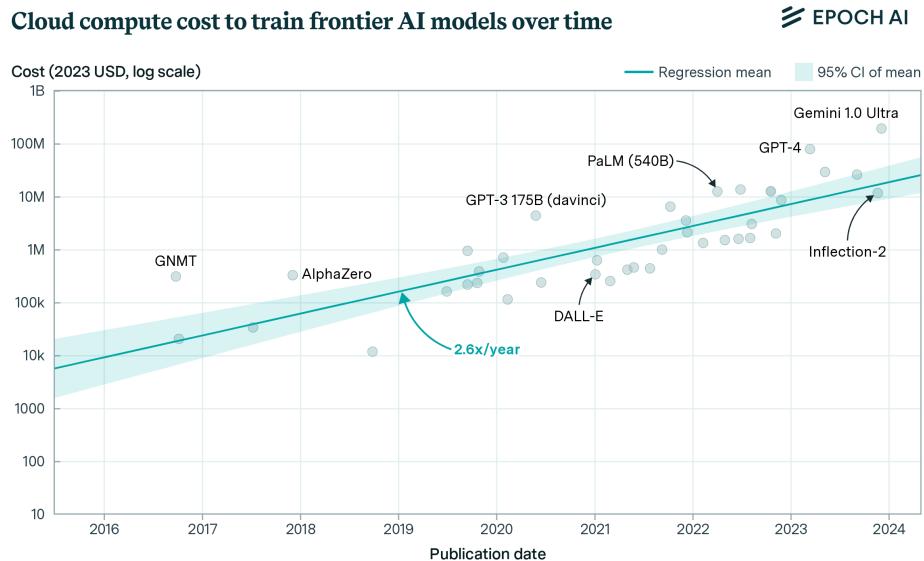


Abbildung 1.6: Kostenentwicklung neue KI Modelle [15]

1.2.5 Ressourcenbedarf bei der Nutzung

Wie weiter oben bereits erwähnt, ist eine Nutzung (Inferenz) der nativen Foundation Modelle, ohne leistungsfähige und entsprechend teure GPUs aus dem Hause Nvidia, kaum sinnvoll möglich. Als kritisches Maß gilt hier vor allem die Latenzzeit bei der Inferenz, d.h. die Zeitspanne bis das Modell eine Antwort liefert. Eine Abschätzung der Firma Nvidia besagt, dass aktuelle Modelle, deren „weights“ in 16 bit Fließkomma Zahlen gespeichert werden, ca. 2 Gigabit GPU Speicher pro einer Milliarde Parameter benötigen. Dazu kommt noch weiterer Speicherbedarf, der für Eingabe, Ausgabe und Inferenz allokiert wird. Dies bedeutet, dass selbst die kleinsten Modelle von Meta und Mistral, die aus 7 bzw. 8 Milliarden Parametern bestehen, schon eine leistungsfähige GPU mit mindestens 16 Gigabyte (bzw. eher mehr) Grafikkartenspeicher, für eine flüssige Nutzung benötigen.

Geeignete GPUs liegen preislich oberhalb von 10.000 € und sind damit für Privatpersonen, Schulen, kleine Firmen usw. kaum erschwinglich. Alternativ können Cloud Ressourcen genutzt werden, welche zumindest bei zeitlich begrenzter Nutzung, eine finanziell tragbare Option darstellen.

Ein weiterer Aspekt, der z.Zt. weniger beachtet wird, ist die Inferenz Nutzung von LLMs auf sogenannten „Edge Devices“. Als „Edge Devices“ bezeichnet man, neben persönlichen IT Geräten wie Laptops, Workstations oder Smartphones, auch Smart TVs, Fahrzeuge, Sensorik und v.a.m.. In der Regel handelt es sich also um Geräte mit begrenzten Rechenressourcen und einer Kostenstruktur, die den Einbau kostenintensiver Zusatzchips nicht erlaubt.

Ein Beispiel, für die Ertüchtigung von Smartphones im Bereich des maschinellen Lernens, ist der von Google seit 2021 in seinen Pixel Modellen verbaute Tensor Chip [18], der aber aktuell nur von Google eigener Software genutzt werden kann.

Wünschenswert wären daher Modelle, die auf weniger performanten Geräten und ggf. ohne Online Verbindung, autonom in der Lage sind eine Inferenz mit qualitativ vernünftigen Ergebnissen zu ermöglichen. Ein Ansatz dies zu erreichen ist die Anpassung der Modelle mittels verschiedener, in den letzten Jahren entwickelter Verfahren, die zur Verringerung des Ressourcenbedarfs führen.

1.2.6 Verfahren zur Verringerung der Ressourcenanforderungen von LLMs

Es existieren unterschiedliche Ansätze, um den Ressourcenbedarf von LLMs zu reduzieren. Eine effektive Größen und Komplexitätsreduzierung kann dabei z.Bsp. durch das Entfernen von Teilen des Modells erfolgen. Dieses Verfahren bezeichnet man als „Pruning“, von dem verschiedene Varianten existieren [19].

Ein anderer Ansatz, welcher in den letzten Jahren kontinuierlich weiterentwickelt wurde, ist die Quantisierung (engl. Quantization) von LLMs [20]. Dabei verringert man die Genauigkeit der Zahlenwerte (Weights) des zugrunde liegenden neuronalen Netzwerks, indem man die verwendeten VariablenTypen durch weniger präzise ersetzt.

Das Knowledge Distillation Verfahren hingegen versucht die Fähigkeiten eines größeren Modells (Teacher) auf ein kleineres Modell (Student) zu übertragen, was im Resultat ebenfalls den Ressourcenbedarf im Verhältnis zu den Fähigkeiten des resultierenden Modells verringern soll [21].

Mit „Low Rank Adaption (LoRA)“ wird eine weitere Technologie verwendet, die helfen kann, solche verkleinerten Modelle qualitativ erneut so zu verbessern, so dass sie bei den Inferenz Ergebnissen wieder möglichst nah an das jeweilige Ursprungsmodell heran reichen [22].

Alle genannten Verfahren werden in dieser Arbeit verwendet und in 2 detaillierter besprochen, so dass der kurze Überblick an dieser Stelle ausreichen soll.

1.2.7 Zielsetzung der Arbeit

Die wesentliche Motivation dieser Arbeit, ist die Verringerung von Ressourcenanforderungen bei Open Source LLMs, unter weitestgehender Beibehaltung ihrer generellen Fähigkeiten. Hierzu sollen, anhand der im letzten Abschnitt genannten Technologien, neue Modelle aus den Open Source Foundation Modellen generiert und ihre Fähigkeiten miteinander verglichen werden. Folgende Kriterien stehen dabei im Mittelpunkt der Untersuchung.

- Verkleinerung des (GPU) Speicherbedarfs
- Verbesserung der Inferenzgeschwindigkeit
- Weitgehende Beibehaltung der Fähigkeiten dieser Modelle
- Nutzung der Modelle auf verschiedenen Plattformen (ggf. auch ohne GPU Unterstützung)

Im Weiteren sollen auch Modelle durch Kombination der genannten Technologien erstellt werden, um ggf. bessere Resultate als bei der Nutzung von nur einer Technologie zu erhalten. Im Ergebnis soll eine Einschätzung möglich sein, welches Potential die aktuellen Technologien, bzw. ihre Kombination, für eine effizientere Nutzung von LLMs besitzen und dies möglichst anhand konkret erstellter Modelle belegt werden.

Diese Arbeit grenzt sich von Ansätzen ab, bei denen ein „Finetuning“ von Modellen vorgenommen wird. Dieses hat i.d.R. das Ziel, Die Fähigkeiten des Modells in bestimmten Bereichen zu erhöhen oder die Interaktion mit dem Enduser zu verbessern. Die Verringerung des Ressourcenbedarfs hat bei solchen Ansätzen oft nur eine sekundäre Bedeutung.

2 Material und Methoden

2.1 Quantisierung von LLMs

Quantisierung ist ein weit verbreitetes Verfahren zur Verkleinerung der Speicherauslastung durch LLMs und zur Beschleunigung der Interferenz, ohne dabei signifikant an Genauigkeit zu verlieren. Damit werden sowohl Kosten reduziert, wie auch der Energieaufwand verringert [20].

Grundlage des Prozesses ist es, die numerische Präzision der Gewichte (Englisch: “glqq weights”) in einem Modell zu verringern. Typischerweise werden LLMs mit 32-Bit-Fließkommazahlen (FP32) trainiert, was eine hohe Präzision gewährleistet. Bei der Quantisierung wird diese Präzision auf 16-Bit-Fließkommazahlen (FP16), 8-Bit-Integer (INT8), 4-Bit-Integer oder 3-Bit-Integer reduziert. In extremen Fällen werden sogar Bitbreiten von 2 oder 1 Bit verwendet, was allerdings zu einer sehr geringen Präzision der Werte führt.

Alleine die Nutzung von Integer Variablen, im Gegensatz zu Fließkomma Variablen (English: Floating Point oder kurz FP) vermindert schon erheblich den Speicherbedarf, da nicht für jeden Wert der zugehörige Exponent gespeichert werden muss (Siehe auch 2.1). Da die meisten Werte in einem Modell sich in einer ähnlichen Größenordnung bewegen, ist der Genauigkeitsverlust durch die Transformation in Integer Variablen sehr gering.

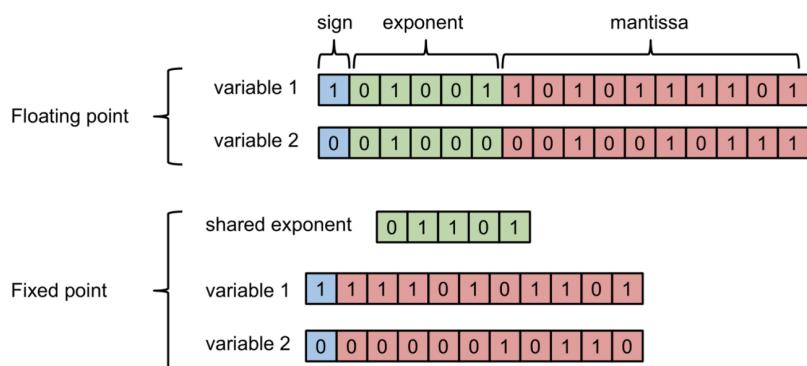


Abbildung 2.1: Speicherbedarf von Floating Point und Integer Variablen [23]

Darüber hinaus lässt sich noch mehr Speicherplatz einsparen, wenn die Zahlen in Integer Variablen mit geringerer Bitbreite abgebildet werden. Der Speicherbedarf einer 4-Bit Integer Variable ist um den Faktor 8 geringer, als der einer 32-Bit-FP Variable. In erster Näherung gilt dieses Verhältnis von 1:8 auch für den Speicherbedarf eines gesamten, auf diese Weise quantisierten Modells.

2.1.1 Zeitpunkt der Quantisierung

Es gibt generell zwei unterschiedliche Zeitpunkte im Lebenszyklus eines Modells, an denen eine Quantisierung sinnvoll ist.

Quantization-Aware Training (QAT)

Bei dieser Methode wird das Modell während des Trainings so angepasst, dass es die Einschränkungen der Quantisierung berücksichtigt. Dadurch kann das Modell lernen, robust gegenüber der niedrigeren Präzision zu sein, was oft zu besseren Ergebnissen führt als das zweite Verfahren PTQ.

Post-Training-Quantisierung (PTQ)

Diese Methode wird nach dem Training des Modells angewendet. Hierbei wird das Modell von einer höheren Präzision (z. B. FP32) auf eine niedrigere Präzision (z. B. INT8) umgewandelt. PTQ ist einfach und erfordert keine Änderung am Trainingsprozess, kann aber in manchen Fällen zu einem gewissen Verlust an Modellgenauigkeit führen.

Da in dieser Arbeit mit bereits existierenden Foundation Modellen gearbeitet wird, kommen hier Verfahren für die Post-Training-Quantisierung (PTQ) zum Einsatz.

2.1.2 Verfahren für Post-Training-Quantisierung

Es gibt verschiedene Implementierungen von PTQ, die sich in der Plattform Unterstützung, der Flexibilität bei den möglichen Bit Raten, der Ergebnisqualität und anderen Kriterien unterscheiden [24]. In dieser Arbeit steht die Ergebnisqualität im Vordergrund, weswegen vor allem zwei neuere Verfahren genutzt werden sollen, die in dieser Hinsicht besonders überzeugen können.

Post-Training Quantization for Generative Pre-Trained Transformers (GPTQ)

Das in [25] beschriebene Verfahren, nutzt eine gradientenbasierte Methode zur Minimierung der Quantisierungsfehler, um ein optimiertes Modell, mit nur geringe Präzisionsverlusten zu generieren.

Die Gewichte des Modells werden dabei schrittweise quantisiert. Dies geschieht in einem „greedy“ (Deutsch: gierig) Verfahren, bei dem in jedem Schritt das Gewicht oder die Gruppe von Gewichten ausgewählt wird, deren Quantisierung den geringsten Einfluss auf die Gesamtleistung des Modells hat. Dabei wird die inverse Hessian-Matrix verwendet, um die Sensitivität der Verlustfunktion gegenüber Änderungen der Gewichte zu bewerten. Nach jedem Schritt wird die verbleibende Menge an zu quantisierenden Gewichten neu bewertet, und der Prozess wird iterativ fortgesetzt.

Activation aware Weight Quantization (AWQ)

AWQ [26] ist ein fortschrittlicher Ansatz zur Quantisierung von neuronalen Netzen, der nicht nur die Gewichte, sondern auch die Aktivierungen des Modells während des Quantisierungsprozesses berücksichtigt.

Die Autoren beschreiben, dass ein relativ geringer Anteil von 0,1 - 1 % der sogenannten „salient weights“ (Deutsch in etwa: bedeutende Gewichte), den größten Einfluss auf die Modell Performance besitzen. AWQ schützt diese Gewichte bei der Quantisierung, indem es die zugehörige Gruppe mit einem Multiplikationsfaktor versieht. Empirisch ließ sich zeigen, dass dieser Ansatz sehr effektiv funktioniert.

Insbesondere hebt er sich ab, von einem sogenannten „Mixed precision“ Ansatz, bei dem die „salient weights“ in einem anderen VariablenTyp (FP16) als die restlichen Gewichte (int 4) gespeichert werden. Dieser führt zwar zu einer vergleichbaren Präzision, kann aber auf normaler Hardware nur ineffizient umgesetzt werden.

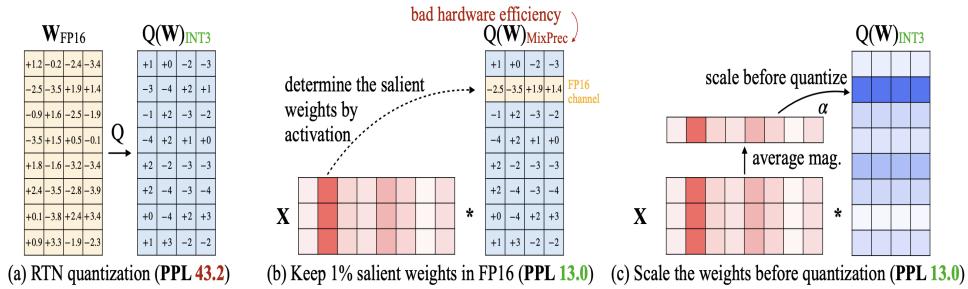


Abbildung 2.2: Vergleich der Quantisierungsmethoden, wie im Text beschrieben [26]

- (a) Zeigt eine einfache Quantisierung mittels eines Rundungsverfahrens (RTN quantization), während (b) „salient weights“ im „Mixed precision“ Ansatz und (c) den AWQ Ansatz darstellt. Der Perplexity Wert (PPL) zeigt die Qualität, der auf diese Weise generierten Modelle.

2.1.3 Verwendetes Verfahren zur Quantisierung

In dieser Arbeit wird primär das AWQ Verfahren zur Quantisierung verwendet [26]. Die Erstellung der quantisierten Modelle orientierte sich an dem in [27] vorgestellten Verfahren und ist im Kapitel Implementation in 3.1 beschrieben.

Zusätzlich wurden einige Versuche mit dem GPTQ Verfahren zu Vergleichszwecken durchgeführt. Dazu wurden das autoGPTQ Verfahren [28] verwendet und auf die Bedürfnisse dieser Arbeit adaptiert. Als zugehöriger Datensatz wird wurde wikitext2 [29] verwendet. Die zugehörige Implementation ist im Kapitel Implementation 3.1.2 beschrieben.

2.2 Pruning von LLMs

Der Begriff „Pruning“ umfasst verschiedene Verfahren, die zum Ziel haben, für die Qualität weniger relevante Anteile eines deep neural networks zu entfernen. Letztlich resultiert dies in einem ausgedünntem Modell von geringerer Größe. Positive Effekte sind im günstigen Fall, ein verringelter Speicherbedarf und eine schnellere Inferenz bei ähnlicher Qualität, im Vergleich

zum Ausgangsmodell. Ein umfangreicher Survey von 2023 [19] klassifiziert drei Kategorien von Pruning, welche im Folgenden erläutert werden.

2.2.1 Unstrukturiertes Pruning

In dieser Kategorie werden Methoden gesammelt, die rein auf die Bedeutung einzelner Gewichte abzielen, unabhängig von der Gesamtstruktur des DNNs.

Zunächst wird die Bedeutung der einzelnen Gewichte im Netzwerk bewertet. Dies geschieht typischerweise durch die Analyse des Betrags der Gewichte, wobei Gewichte mit kleineren Beträgen als weniger wichtig angesehen werden. Es gibt aber auch andere Methoden, welche die Sensitivität des Verlusts gegenüber den Gewichten betrachten. Die als weniger wichtig eingestuften Gewichte werden auf null gesetzt, also „gepruned“.

Im Gegensatz zum strukturierten Pruning, werden beim unstrukturierten Pruning einzelne Verbindungen zwischen Neuronen entfernt, ohne dabei Rücksicht auf die zugrunde liegende Struktur des Netzwerks zu nehmen [30].

Die resultierenden Netzwerkstrukturen können sehr unregelmäßig sein, was die Hardware-Optimierung (wie z.B. auf GPUs) erschweren kann. Außerdem ist oft ein intensives Re-Training notwendig, um eine befriedigende Qualität zu erreichen. Dem gegenüber steht der Vorteil einer feineren Kontrolle und potenziell höhere Kompressionsraten, als bei strukturiertem Pruning.

2.2.2 Strukturiertes Pruning

Beim strukturierten Pruning werden ganze Strukturen innerhalb des Modells entfernt, wie z.B. ganze Neuronen, Heads in einem Attention-Mechanismus oder ganze Netzwerk Schichten [31]. Im Unterschied zum unstrukturierten Pruning, bleibt das Modell dadurch strukturell regelmäßig und die resultierenden Matrizen sind dicht. Die Notwendigkeit zum Re-Training des Modells ist durch die größere Homogenität geringer und es werden keine zusätzlichen Software Komponenten für die Inferenz benötigt.

2.2.3 Semi-strukturiertes Pruning

In den letzten Jahren etablieren sich Verfahren, die versuchen Vorteile beider Ansätze zu kombinieren, d.h. sowohl die Flexibilität des unstrukturierten Prunings als auch die Effizienz des strukturierten Prunings zu nutzen.

Beim semi-strukturierten Pruning werden zusammenhängende Gruppen von Gewichten innerhalb einer bestimmten Struktur entfernt. Diese Gruppen können Teilstrukturen innerhalb einer größeren Struktur sein, wie z.B. bestimmte Reihen oder Spalten einer Gewichtsmatrix, aber nicht die gesamte Struktur (wie im strukturierten Pruning).

2.2.4 Verwendetes Verfahren für das Pruning

In dieser Arbeit werden zwei unterschiedliche Verfahren für das Pruning verwendet, die in den nächsten Kapiteln beschrieben werden. Beide sind dem semi-strukturierten Pruning zuzuordnen.

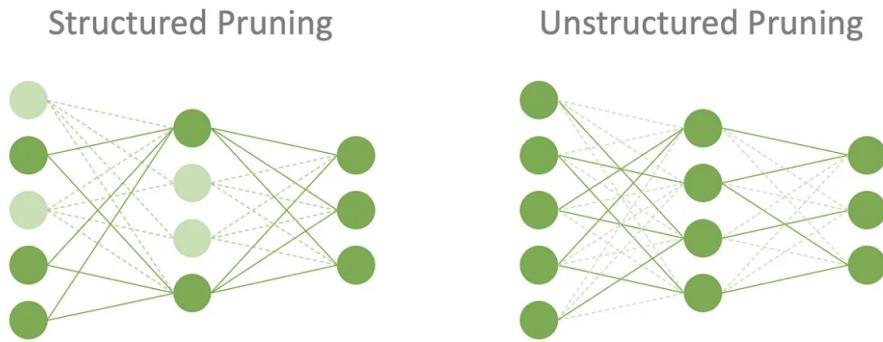


Abbildung 2.3: Strukturiertes und Unstrukturiertes Pruning.

Die linke Grafik zeigt wie komplette Strukturbereiche aus dem Netzwerk genommen werden. Die übrigen Strukturen und ihre Beziehungen bleiben unverändert erhalten. Beim unstrukturierten Pruning hingegen wird ein Großteil der Beziehungen verschlankt, während die Struktur vollständig erhalten bleibt.

Extraction von Schichten mittels „PruneMe“

Die Arbeit von Gromov et al. [31] analysiert das Modell auf die Bedeutung der einzelnen Schichten für das Gesamtergebnis. Dabei werden „unwichtige“ transformer blocks aus dem Modell entfernt. Die Auswahl der weniger wichtigen Teile erfolgt dabei durch eine Metric, in der Taylor+ und Perplexity Kriterien als Diskriminatoren genutzt werden. Führt die Entfernung eines Blocks (bzw. von Blöcken) nur zu einer relativ geringen Verschlechterung dieser Werte, werden diese Blöcke als „unwichtig“ angesehen. Die auf dieser Weise identifizierten Teile werden in einem zweiten Schritt aus dem Modell mittels „splitting“ entfernt.

Für die Implementation wurde, dass zur Veröffentlichung zugehörige GitHub Repository [32] verwendet und an die Erfordernisse der Experimente adaptiert.

Kombination von LoRA und Pruning mit dem Verfahren „Shortened-LLM“

Das zweite Verfahren stellt ein Kombination von Pruning und einem anschließenden LoRA Retraining (siehe auch 2.4) dar. Dazu wurde die „depth pruning“ Implementation mit der Bezeichnung „shortened-llm“ verwendet [33].

Während der Pruning Mechanismus sehr ähnlich zu dem im vorhergehenden Kapitel vorgestellten ist, wird hier als anschließender Schritt, das Retraining der so geprunten Modelle mittels der LoRA Technologie vorgenommen. Dies führt zu einer notwendigen Neubalancierung des neuronalen Netzes und erhöht dessen Qualität.

In der Veröffentlichung [33] wurde gezeigt, dass dieses Verfahren anderen Pruning Methoden mindestens ebenbürtig ist, bzw. diese zum Teil auch übertrifft. Bei moderatem Pruning (20 - 45%) können Benchmark Werte erreicht werden, die im Bereich des Ausgangsmodells liegen, aber weniger Speicher benötigen und die Inferenz beschleunigen.

2.3 Knowledge Distillation

Knowledge Distillation ist eine Technik des maschinellen Lernens, die darauf abzielt, das Wissen eines großen, leistungsstarken Modells (genannt Teacher Model) auf ein kleineres, effizienteres Modell (genannt Student Model) zu übertragen. Dieses Verfahren ermöglicht es, ein kleines Modell zu trainieren, das nahezu die gleiche Leistung wie das große Modell erbringt, jedoch einen geringeren Speicherbedarf hat und eine schnellere Inferenz ermöglicht [21].

Unterschieden wird dabei zwischen „white box“ und „black box“ Knowledge Distillation:

- White Box - In der Regel bezeichnet dies die Arbeit mit open source Modellen, bei denen auf die einzelnen Antwortwahrscheinlichkeiten zugegriffen werden kann, die sich in der Form von Logits (s.u.) darstellen lassen.
- Black Box - In der Regel kommerzielle Modelle, bei denen nur die Frage/Antwort Paare für die Knowledge Distillation zur Verfügung stehen.

Im Folgenden wird nur noch der „White Box“ Ansatz betrachtet, da in dieser Arbeit Open Source Modelle verwendet werden.

Für die Distillation wird zunächst das Teacher Modell mit einem Datensatz trainiert. Anstatt nur die harten Klassenvorhersagen (z.B. eine 0 oder 1 bei einer Klassifikationsaufgabe) zu verwenden, nutzt Knowledge Distillation die Wahrscheinlichkeitsverteilungen, die das Teacher Model für jede Klasse ausgibt. Diese Wahrscheinlichkeiten enthalten mehr Informationen als nur die korrekte Klasse, weil sie auch anzeigen, wie sicher das Modell in Bezug auf alle Klassen ist. Diese Ergebnisse bezeichnet man als „Soft Targets“.

Das Training des Student Modells ist so gestaltet, dass die Soft Targets des Teacher Models imitiert werden, anstatt nur die harten Labels aus Trainingsdaten zu verwenden. Dadurch lernt das Student Model nicht nur, welche Klasse korrekt ist, sondern auch, wie sicher das Teacher Model bei seiner Antwort ist.

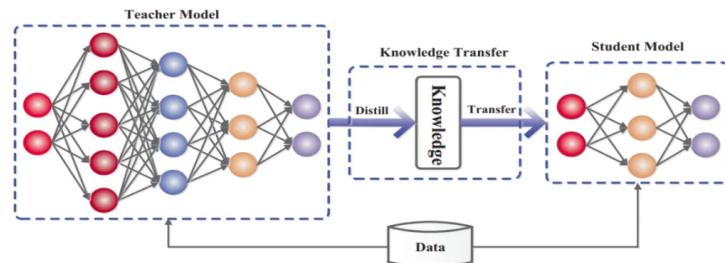


Abbildung 2.4: Schematische Darstellung von Knowledge Distillation. [34].

2.3.1 Response based Knowledge Distillation (KD)

Dies ist das bedeutendste Verfahren, das zur Durchführung von KD verwendet wird. Als Trainingsgrundlage werden dabei die sogenannten „Logits“ des Teacher Models verwendet. „Logits“ sind die Inferenz Ergebnisse des neuronalen Netzes als reelle Zahlen und können damit sowohl positive wie negative Werte annehmen. Durch die Anwendung der Softmax-Funktion auf die Logits entstehen daraus die oben erwähnten „Soft Targets“, welche dann zum Training des Student Models genutzt werden.

Als Maß für den Trainingserfolg vergleicht man die Logits von Teacher und Student Model miteinander, um daraus den „Distillation Loss“ zu ermitteln (siehe auch 2.5). Das Ziel des

Trainings ist dabei, eine möglichst starke Annäherung der Logits von Teacher und Student Model, durch den Knowledge Distillation Prozess.

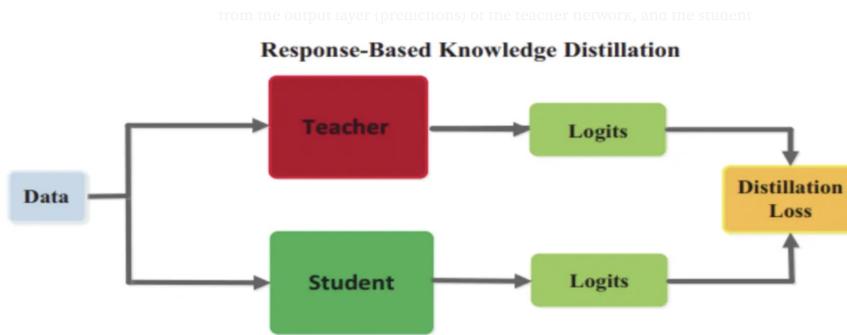


Abbildung 2.5: Schematische Darstellung der response based Knowledge Distillation. [34].

2.3.2 Verwendetes Verfahren für die Knowledge Distillation

Für die Knowledge Distillation wurde das „Dual Space Knowledge Distillation (kurz: DSKD)“ als Verfahren gewählt [35]. Die Wahl fiel auf diesen Ansatz, da es mit diesem relativ leicht möglich ist Student Modelle zu trainieren, die eine abweichende Struktur vom Teacher Modell besitzen (daher Dual Space). Da beide Modelle ihre eigenen unterschiedlichen Vorhersage Mechanismen (engl. „Prediction Heads“) nutzen, wird im DSKD Verfahren eine „cross-model attention (CMA)“ implementiert, welche die verschiedenen Token Sequenzen automatisch miteinander abgleicht. Das Verfahren basiert auf der unter 2.3.1 beschriebenen Variante von Knowledge Distillation.

2.4 Low Rank Adaption (LoRA)

2.4.1 Funktionsweise von LoRA

Diese Technologie hat nicht, im Unterschied zu den vorher gehenden, das primäre Ziel den Ressourcenbedarf von fertigen LLMs zu verringern [22]. Vielmehr geht es hier im Wesentlichen darum, das LLM mit anderen (neuen) Inhalten zu trainieren, bzw. spezielle Fähigkeiten zu verbessern. Ein wichtiger Vorteil ist, dass mit LoRA das bereits fertige Modell nicht verändert werden muss, sondern die neuen Fähigkeiten über einen Adapter dem Modell hinzugefügt werden. Dies ermöglicht eine große Flexibilität bei der Anpassung der oben genannten Foundation Modelle. Darüber hinaus kommt diese Technologie mit wenig (zusätzlichen) Ressourcen aus, so das der Effizienz Gedanke sekundär auch eine Rolle spielt.

LoRA benutzt die Zerlegung von komplexen Matrizen in niederrangigere Matrizen [36]. Anstatt alle Gewichte des Modells zu aktualisieren, wie es bei einem herkömmlichen Fine-Tuning der Fall wäre, führt LoRA eine „Low-Rank“-Dekomposition der Gewichtsmatrizen durch. Konkret werden die ursprünglichen Gewichtsmatrix W in zwei kleinere Matrizen A und B zerlegt, wobei die beiden neuen Matrizen eine niedrigere Rangordnung (daher 'low-rank') haben 2.6. Die Idee ist, dass man W durch $W + A \times B$ ersetzen kann wobei A und B die erlernten Parameter beinhalten und deutlich weniger Parameter besitzen als W 2.7.

Die Anpassung wird dann nicht auf allen Parametern des Modells durchgeführt, sondern nur auf

Rank Decomposition

$$\begin{bmatrix} 2 & 20 & 1 \\ 4 & 40 & 2 \\ 6 & 60 & 3 \end{bmatrix}_{3 \times 3} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}_{3 \times 1} \times \begin{bmatrix} 2 & 20 & 30 \end{bmatrix}_{1 \times 3}$$

Abbildung 2.6: Zeigt die Dekomposition einer höherrangigen 3×3 Matrize in zwei niederrangigere 3×1 bzw. 1×3 Matrizen.

einem Teilraum, der durch „low-rank“-Matrizen beschrieben wird. Man bildet auf diese Weise einen relativ kleinen Adapter, der mit dem ursprünglichen Modell fusioniert wird.

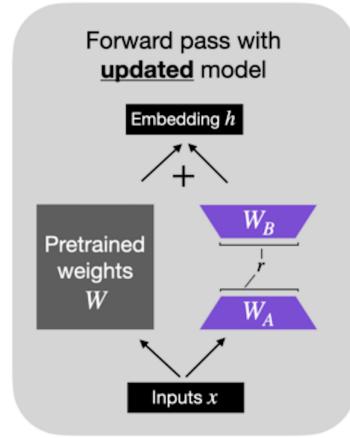


Abbildung 2.7: Zeigt die Kombination des ursprünglichen Modells W mit den niederrangigeren Matrizen W_A und W_B , welche die neu erlernten Gewichte enthalten [37].

LoRA eignet sich nicht nur für Szenarien, in denen große vortrainierte Modelle auf spezifische Aufgaben oder Daten angepasst werden müssen, wie z.B. bei der Verarbeitung von speziellen Textarten, Domänen oder Sprachen. Die Technologie ist auch geeignet um Modelle, die durch Kompression verschlankt wurden zu Re-Trainieren, damit ihre Fähigkeiten auf dem Niveau der Ursprungsmodele bleibt. Diese Technologie wird hier besprochen, da sie ein wesentlicher Bestandteil des im nächsten Abschnitt beschriebenen QLoRA Ansatzes und des bereits vorgestellten „shortened-LLM“ Verfahrens (siehe 2.2.4) ist. Bei beiden wird sie für das Re-Training von Modellen eingesetzt.

2.4.2 QLora

Bei QLoRA handelt es sich um ein kombiniertes Verfahren. Hier erfolgt zunächst eine Quantisierung des Ursprungsmodelels, welches dann einem intensiven Re-Training mittels LoRA unterzogen wird. Die Entwickler des Verfahrens bezeichnen QLoRA auch gerne als Weiterentwicklung des LoRA Verfahrens. [38]. Zusätzlich wird in der Veröffentlichung ein neuer Datentyp beschrieben (4-bit NormalFloat), der die Quantisierung des Ausgangsmodelels besonders effizient durchführen soll.

Für die Implementation wurden Codeblöcke aus einem Github Repository [39] verwendet, welches auf der Veröffentlichung [38] basiert und für die Experimente in dieser Arbeit entsprechend angepasst wurde. Als Datensatz für das Re-Training mittels LoRA wurde „ultrachat_200k“ verwendet [40] und der Umfang des Trainings auf eine Epoche begrenzt, für die etwa sechs Stunden mit einer GPU benötigt wurden.

2.5 Python Bibliotheken zur Arbeit mit LLMs

In vielen der Verfahren werden einige zentrale Python Bibliotheken bzw. Frameworks verwendet, von denen die Wichtigsten hier kurz vorgestellt werden.

2.5.1 Die Bibliothek Transformers

Die aus der Huggingface Community hervorgegangene Transformers Bibliothek [41], ist ein zentrales Werkzeug zu Arbeit mit den gleichnamigen Modellen. Einige der Kernfunktionen sind:

- Einfacher Zugriff auf vortrainierte Modelle
- Feinabstimmung (engl: FineTuning) der Modelle
- Erstellen von Trainings Pipelines
- Integration mit den NVidia CUDA Treibern

Der Funktionsumfang der Bibliothek wächst ständig, getrieben durch die Hugging Face Entwickler Community. Dadurch bietet sie einen umfassenden, aktuellen und benutzerfreundlichen Zugang zu modernen LLMs, und stellt für Forschung, Entwicklung und Produktion ein wichtiges Werkzeug dar.

2.5.2 Das Framework torch (pytorch)

Pytorch ist eine Weiterentwicklung der seit 2002 bestehenden torch bibliothek [42]. Der Name repräsentiert mittlerweile ein Ökosystem von Bibliotheken, mit einer Vielzahl von Funktionen, welche für das Machine Learning und insbesondere die Erstellung und das Training neuronaler Netzwerke genutzt werden.

Ursprünglich entwickelt wurde die Bibliothek von Facebooks (jetzt Meta) AI Research Group (FAIR), die sie im Jahr 2017 als Open Source der Entwicklergemeinde übergab. Seit 2022 steht die weitere Entwicklung unter der Obhut der PyTorch Foundation [42].

Mit Hilfe von Pytorch können insbesondere Tensor Operationen, durch die Nutzung von GPUs stark beschleunigt und somit neuronale Netze effizient trainiert werden.

2.5.3 Die Bibliothek PEFT

Die PEFT-Bibliothek (Parameter-Efficient Fine-Tuning) von Hugging Face ist eine spezialisierte Bibliothek, die sich auf das effiziente Fine-Tuning großer vortrainierter Modelle konzentriert. Ein wesentlicher Aspekt dabei ist der effektive Umgang mit Speicherbedarf und Rechenleistung. Wesentliche Funktionen sind:

- LoRA Fine Tuning mittels der Erstellung geeigneter Adapter
- Optimierung der Eingabe (Prompt) bzw. der Präfixe
- Kompression der Modelle durch Quantisierung

Die PEFT Bibliothek integriert sich nahtlos mit der zuvor erwähnten Transformers Bibliothek, und erlaubt es so den Entwicklern eine effiziente Arbeit mit den Modellen, aus der zugehörigen Huggingface Bibliothek [43].

2.6 Beurteilung der Leistungsfähigkeit eines LLMs

Für die Beurteilung der Leistungsfähigkeit eines LLMs gibt es, aufgrund ihrer Komplexität, eine große Anzahl von Kriterien und Metriken. In der Arbeit von (Guo et. al) [44] wird eine Einteilung der relevanten Metriken in drei hauptsächliche Bereiche vorgenommen (siehe auch Abbildung 2.8). Diese sind:

- Die Bewertung der Kenntnisse und Fähigkeiten (knowledge and capability evaluation) eines Modells. Hierzu zählt u.a. die Fähigkeit zu Schlussfolgerungen, Wissens Vervollständigung und Antwortkompetenz.
- Die Bewertung der Anpassungsfähigkeit (alignment evaluation) konzentriert sich auf die Prüfung der Leistung von LLMs in kritischen Dimensionen. Diese umfassen ethische Erwägungen, moralische Implikationen, Erkennung von Verzerrungen, Vermeidung von Toxizität und die Bewertung der Wahrhaftigkeit von Aussagen.
- Bewertung der Sicherheit (safety evaluation) eines Modells umfasst vor allem die Robustheit. Damit ist sowohl die Konstanz der Leistungen auf hohem Niveau, wie auch die Bewältigung von Störungen durch z.Bsp. böswillige Angriffe oder vorsätzliche Täuschung.

Die vorliegende Arbeit konzentriert sich praktisch ausschließlich auf den erstgenannten Block dieser Einteilung, der Ermittlung von Kenntnissen und Fähigkeiten des LLMs. Ziel ist es, die verschieden komprimierten Modelle miteinander auf diese Kriterien hin zu vergleichen und aus diesen Informationen Rückschlüsse auf die Eignung des Vorgehens zu ziehen. Dazu werden Testverfahren (Benchmarks) verwendet, welche die Nutzung des Modells simulieren. Das Framework „lm_evaluation_harness“ [45] bietet eine einfache und weithin verwendete Möglichkeit, Benchmarks aus einem großen Portfolio auszuwählen und diese für die Evaluierung zu verwenden.

Für diese Arbeit wurden Benchmarks ausgewählt, welche die Ermittlung von Kenntnissen und Fähigkeiten des LLMs exemplarisch abdecken können. Außerdem wurde darauf geachtet, Benchmarks zu verwenden, die in aktuellen Veröffentlichungen eine häufigere Nutzung erfahren, damit ein gewisse Vergleichbarkeit entsteht und die Einordnung der Ergebnisse erleichtert wird. Alle ausgewählten Benchmarks wurden auch bis ca. Mai 2024 im „HuggingFace Leaderboard“ verwendet [46], auf dem viele Vergleiche von Modellen in den letzten Monaten beruhen.

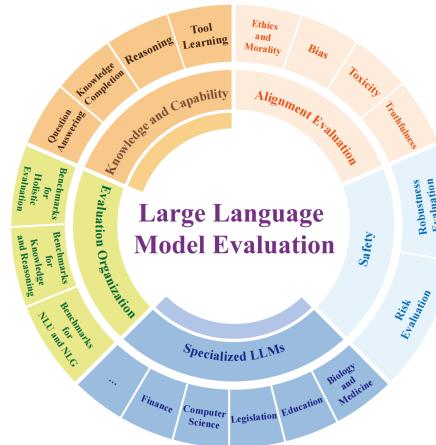


Abbildung 2.8: Darstellung der Kriterien und Metriken nach (Guo et al.)[44]. Neben den im Text behandelten drei Hauptbereichen sind noch Metriken für spezielle LLMs (dunkleres blau) angegeben.

2.6.1 Verwendete Benchmarks

Der nächste Abschnitt beschreibt, die in dieser Arbeit verwendeten Benchmarks für die Evaluierung der Modelle.

- **Winogrande** [47] Eine Auswahl von 273 Problemen, die vor allem auf eine Lösung durch „Gesunden Menschenverstand“ abzielen, dienen als Kriterium für die Lösungsfähigkeiten der LLMs.
- **arc_challenge** [48] Die Autoren der ARC challenges entwickelten eine Definition von Intelligenz, die u.a. Effizienz beim Erwerb von Fähigkeiten und die Fähigkeit zur Generalisierung als Kernelemente enthält. Die zugehörige Benchmark zielt darauf ab, diese Elemente zu quantifizieren.
- **truthfulqa_mc2**, [49] Diese Benchmark besteht aus 817 Fragen die zu 38 Kategorien gehören. Ziel ist es zu quantifizieren, wie oft das Sprachmodell wahrheitsgemäße Antworten auf diese Fragen generiert.
- **hellaswag** [50] Diese Benchmark zielt auf die Fähigkeit der LLMs zur sinnvollen Vervollständigung von Satzfragmenten.

2.6.2 Perplexity als Qualitätsmarker

Neben den oben gezeigten Benchmarks wird zunehmend auch die „Perplexity“ des Modells als Qualitätsmerkmal angegeben [51]. „Perplexity“ drückt dabei die Unsicherheit des Modells aus, das nächste Wort zu bestimmen, wenn es auf neue Inhalte stößt. Es ist gängig diesen Wert mit der „wikitext“ Datensammlung [29] zu ermitteln.

In der Essenz gilt: Um so kleiner der „Perplexity“ Wert, um so besser funktioniert das Modell. Die Perplexity Werte wurden nur für das Llama 3 und das Mistral Modell ermittelt.

2.6.3 Inferenz Performance und Speicherauslastung

Die Inferenz Geschwindigkeit, und damit die Geschwindigkeit der Reaktion eines Modells auf eine Anfrage, ist ein bedeutendes Merkmal für die Nutzbarkeit des Modells. Ist die Reaktionsgeschwindigkeit zu gering, wird das Modell für die direkte Interaktion mit dem Anwender und viele Applikationen nicht geeignet sein. Da der Fokus dieser Arbeit darauf liegt Modelle auch in Umgebungen nutzen zu können, die nur über begrenzte Ressourcen verfügen, kommt diesem Punkt besondere Bedeutung zu.

Gemessen wird hier die Geschwindigkeit mit der die Modelle die Worte (respektive Token) auf eine Anfrage zurück liefern. Diese Daten werden für einige Anfrage Beispiele gemittelt, um einen verlässlicheren Wert zu erhalten. Zusätzlich wird die Belegung des GPU Speichers durch das Modell ermittelt. Die Implementation ist in 3.6.2 beschrieben.

2.7 Verwendete Modelle

Für den Experimentalteil werden die Llama Sprachmodelle der amerikanischen Firma Meta und das Sprachmodell Mistral, der gleichnamigen französischen Firma verwendet. Die Modelle werden in den nächsten Abschnitten charakterisiert. Folgende Argumente führten zur Auswahl dieser Modelle:

- Im Bereich der frei verfügbaren Modelle sehr gute Leistungen im Anwender Dialog
- Liberale Lizenzpolitik, so dass sie weitgehend ohne Einschränkungen in der Forschung genutzt werden können
- Aktualität: Llama 3 und Mistral wurden kurz vor Start dieser Arbeit veröffentlicht
- Aufbau der Modelle und das Trainingsvorgehen sind (teilweise) dokumentiert

Bei allen Modellen wurde immer die kleinstmögliche Variante gewählt. Dies passierte primär, um den Ressourcenbedarf gering und die Laufzeiten der Experimente mit den einzelnen Modelle nicht zu lang zu gestalten. Ein weitere Grund war das Bestreben, möglichst kompakte Modelle zu generieren. Daher war es sinnvoll, schon ein möglichst nicht zu großes Grundmodell als Ausgangspunkt zu benutzen.

2.7.1 Meta Llama 2

Am 18.7.2023 wurde Llama 2 von Meta veröffentlicht [52]. Das Modell liegt in drei verschiedenen Parametergrößen vor: 7 Milliarden, 13 Milliarden und 70 Milliarden Parameter. Alle Modelle liegen darüber hinaus zusätzlich noch in einer finegetunten, für den Anwender Dialog optimierten, „Chat“ Variante vor, so dass sich in Summe sechs verschiedene Versionen ergeben. Für die Experimente in dieser Arbeit wurde das kleinste Modell Llama 7B in der Chat Variante ausgewählt.

Llama 2 folgt einem eher evolutionären Einsatz bei seiner Entstehung. Die Basis Struktur wurde von Llama (1) [53] übernommen, welches wiederum den in 1.1.2 beschriebenen Transformer Ansatz verwendet. Wesentliche Neuerungen gegenüber dem Vorgänger sind, ein neuer größerer Trainingsdatensatz, eine Verdopplung der Prompt-Größe, um mehr Kontext zu ermöglichen und die Nutzung von „grouped-query attention“ für eine schnelle und qualitativ stabile Inferenz [54]. Wie oben erwähnt, war Llama 2 das erste Modell der Firma Meta, dass zusätzlich in einer finegetunten Variante für einen verbesserten Anwenderdialog zur Verfügung gestellt wurde.

Dazu passieren im wesentlichen zwei Schritte. Im ersten Schritt dem „supervised fine-tuning“ werden dem Modell Frage/Antwort Paare vorgelegt, damit es Muster für die Ausgaben besser erfasst. In einem weiteren Schritt, der als „reinforcement learning with human feedback (RLHF)“ bezeichnet wird, nehmen Menschen eine qualitative Bewertung der Antworten des Modells vor. Mittels „rejection sampling“ und „Proximal Policy Optimization“ wird das Modell iterativ mit diesen Antworten verbessert und so die Kompetenz für den Anwender Dialog erhöht (siehe auch Abbildung 2.9).

Da zu erwarten ist, dass in Zukunft die neueren, im Anschluss besprochenen Modellen vermehrt eingesetzt werden, erfolgte eine Konzentration auf Llama 3 und Mistral, während für Llama 2 nicht alle Experimente durchgeführt wurden (siehe auch 4.1).

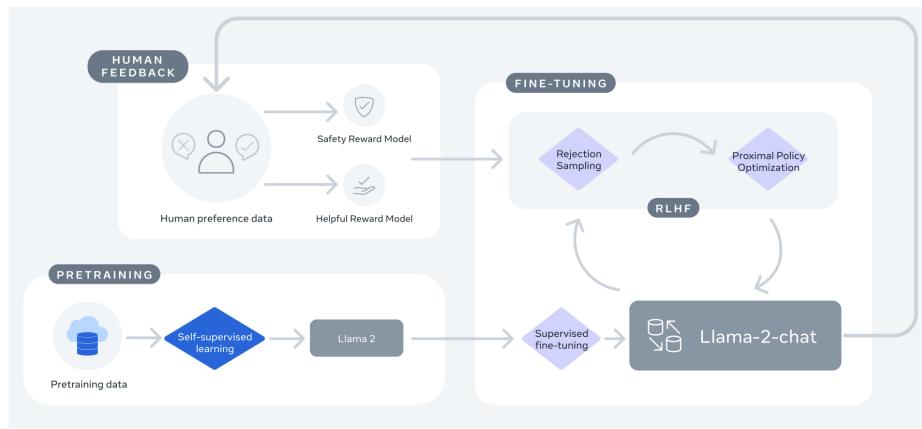


Abbildung 2.9: Erstellung des Chat Modells aus dem Grundmodell [52]. Erklärung siehe Text.

2.7.2 Meta Llama 3

Das Modell Llama 3 wurde am 18.4.2024 von der Firma Meta zwei Versionen veröffentlicht [55]. Im Gegensatz zum Vorgängermodell, wurde die mittlere Version nicht mehr angeboten, so dass nur eine Variante mit 8 Milliarden und eine mit 70 Milliarden Parameter zur Verfügung steht. Wie bei Llama 2, existiert auch hier jeweils eine für den Dialog mit dem Anwender optimierte Chat (Instruct) Variante.

Ähnlich wie Llama 2 verfolgt auch Llama 3 einen eher evolutionären Ansatz, indem es das alte Modell durch neuere Methoden und Erkenntnisse verbessert. Leider ist die Dokumentation zu Llama 3 deutlich rudimentärer, als die umfangreichen Veröffentlichungen zu den beiden Vorgänger Modellen. Bekannt ist, dass ein neuer 128 K Tokenizer etabliert und noch einmal deutlich mehr Trainingsdaten verwendet wurden.

2.7.3 Mistral

Mistral ist ein Transformer Modell mit 7 Milliarden Parametern, welches von der französischen Firma Mistral entwickelt und im September 2023 veröffentlicht wurde [11]. Ähnlich wie bei Metas Llama, gibt es auch hier Schwester Modelle mit deutlich mehr Parametern, die unter den Namen Mixtral 8x7B und 8x22B veröffentlicht wurden. Diese beinhalten 56 bzw. 176 Milliarden Parameter. Ebenfalls gibt es auf Anwender Kommunikation spezialisierte Varianten, die als

„Instruct“ bezeichnet werden.

Obwohl ebenfalls auf einer Transformer Struktur basierend, unterscheiden sich die Mistral Modelle durch den Einsatz anderer Technologien, von den zuvor genannten Meta Modellen. So benutzt Mistral u.a. einen „sliding window attention mechanism“, der die Rechenkomplexität reduziert und damit die Performance verbessert. Obwohl das Modell dabei nur ein begrenztes Fenster auf die Eingabesequenz legt und in diesem die Aufmerksamkeit berechnet, können, durch ein bewegen des Fensters schrittweise entlang der Sequenz, auch längere Abhängigkeiten modelliert werden.

Eine weiteres besonderes Merkmal ist die Nutzung der „flash attention“ Technologie für die Berechnungen. Diese kombiniert sehr geschickt Matrixmultiplikationen und Softmax-Berechnungen in einem speicherfreundlichen Algorithmus, was ebenfalls zu einer Steigerung der Gesamtperformance führt.

3 Implementation

In diesem Kapitel werden die Implementierungen der einzelnen Verfahren dargestellt. Der auszuführende Programmcode ist hellgrau unterlegt und wird an manchen Stellen zum besseren Verständnis zusätzlich kommentiert. Die zugehörigen Jupyter Notebooks zur Ausführung der Verfahren findet man unter [56].

3.1 Quantisierung

Implementation für die Quantisierung der Modelle mit denen in 2.1 beschriebenen Verfahren AWQ und GPTQ.

3.1.1 Quantisierung: AWQ

- Es wird die Bibliothek autoawq 0.1.0. genutzt. Diese setzt eigentlich die transformers >= 4.36 voraus, welche aber Abhängigkeitsprobleme zu anderen Bibliotheken verursachte.
- Lösung: Installation von transformers 4.35 durch ignorieren der Abhängigkeit. Fehler waren nicht feststellbar.
- huggingface_hub Bibliothek muss mindestens die Version 0.21 haben

Erstellen der Umgebung - Nur einmalig notwendig

```
!pip install autoawq nvidia-ml-py3 numpy==1.26.4 sentencepiece torch==2.0.1  
protobuf
```

Login für Hugging Face - Nur notwendig wenn HF Modelle genutzt werden

```
notebook_login()
```

Auswählen der GPU und laden der Bibliotheken

```
import os  
  
os.environ["PYTORCH_CUDA_ALLOC_CONF"] = "max_split_size_mb:256"  
os.environ["CUDA_VISIBLE_DEVICES"] = "1"  
!echo $CUDA_VISIBLE_DEVICES
```

```
from awq import AutoAWQForCausalLM
from transformers import AutoTokenizer
from huggingface_hub import notebook_login
import safetensors
```

Konfiguration der Parameter für die AWQ Quantisierung

```
quant_config = {"zero_point": True, "q_group_size": 128, "w_bit": 4}
```

Pfad zum Modell und Speicherort für die quantisierten Basis Modelle

Llama3-8B-Instruct

```
model_path = "meta-llama/Meta-Llama-3-8B-Instruct"
quant_path = "/home/thsch026/masterarbeit/models/generated/llm-awq/
Meta-Llama-3-8B-Instruct-AWQ"
```

Mistral-7B Instruct v0.2

```
model_path = "mistralai/Mistral-7B-Instruct-v0.2"
quant_path = "/home/thsch026/masterarbeit/models/generated/llm-awq/
Mistral-7B-Instruct-v0.2-AWQ-4bit"
```

Llama2-7B-chat-HF

```
model_path = "meta-llama/Llama-2-7b-chat-hf"
quant_path = "/home/thsch026/masterarbeit/models/generated/llm-awq/
Llama-2-7b-chat-HF-AWQ-4bit"
```

Pfad zum Modell und Speicherort für die vorab veränderten Modelle

Mistral-7B-Instruct-v02-prune-lora

```
model_path = "/home/thsch026/masterarbeit/models/generated/
prune_plus_lora/shortened-llm/Mistral-7B-Instruct-v02_prune_lora"
quant_path = "/home/thsch026/masterarbeit/models/generated/
prune_plus_lora_plus_awq/Mistral-7B-Instruct-v02_prune_lora_awq_256groupsize"
```

Llama 3 8B Instruct prune-lora

```
model_path = "/home/thsch026/masterarbeit/models/generated/prune_plus_lora/shortened-l1m/Meta-Llama-3-8B_prune_lora"
quant_path = "/home/thsch026/masterarbeit/models/generated/prune_plus_lora_plus_awq/Meta-Llama-3-8B_prune_lora_awq"
```

Llama-2-7b-chat-hf_prune_lora_awq

```
model_path = "/home/thsch026/masterarbeit/models/generated/prune_plus_lora/shortened-l1m/Llama-2-7b-chat-hf_prune_lora"
quant_path = "/home/thsch026/masterarbeit/models/generated/prune_plus_lora_plus_awq/Llama-2-7b-chat-hf_prune_lora_awq"
```

Laden des Modells und des zuhgeörigen tokenizers

```
#load the model and tokenizer
model = AutoAWQForCausallM.from_pretrained(model_path, safetensors=True)
tokenizer = AutoTokenizer.from_pretrained(model_path, use_fast=True)
```

Durchführung der Quantisierung

- Benötigt ca. 10 - 20 Minuten

```
model.quantize(tokenizer, quant_config=quant_config)
```

Abspeichern des Modells

```
import os

model.save_quantized(quant_path, safetensors=True)
tokenizer.save_pretrained(quant_path)
```

Freigeben des GPU Speichers

```
import torch
del model
del tokenizer
torch.cuda.empty_cache()
print(torch.cuda.memory_reserved(0))
print(torch.cuda.memory_allocated(0))
```

3.1.2 Quantisierung: GPTQ

Implementation des Verfahrens für die GPTQ Quantisierung 2.1 mittels der AutoGPTQ Technologie [25].

Erstellen der Arbeitsumgebung (einmalig)

```
# Bash Shell
# Auswählen eines Verzeichnisses für die Installation
# hier /home/thsch026/masterarbeit/experiment
cd /home/thsch026/masterarbeit/experiment

# Erstellen des conda environments
conda create --autogptq autogptq python==3.10 -y
conda activate autogptq

# Installieren notwendiger packages
pip install peft gekko pandas

# Kopiern der Sourcen
git clone https://github.com/PanQiWei/AutoGPTQ.git && cd AutoGPTQ

# Komplilieren der Sourcen
pip install -vvv --no-build-isolation -e .
```

AutoGPTQ Workflow (Kernel: autogptq)

```
import os
os.chdir("/home/thsch026/masterarbeit/experiment/AutoGPTQ")
```

Auswahl des zu Quantisierenden Modells

Quantize Llama 3 8B Instruct

```
pretrained_model_dir = "meta-llama/Meta-Llama-3-8B-Instruct"
quantized_model_dir = "Meta-Llama-3-8B-instruct-v2_gptq_2"
```

Quantize Mistral 7B Instruct v0.2

```
pretrained_model_dir = "mistralai/Mistral-7B-Instruct-v0.2"
quantized_model_dir = "Mistral-7B-Instruct-v0.2_gptq_3bit"
```

Funktion für das wikitext dataset

```
def get_wikitext2(nsamples, seed, seqlen, model):
    import numpy as np
    import torch
    from datasets import load_dataset
    traindata = load_dataset('wikitext', 'wikitext-2-raw-v1', split='train')
    testdata = load_dataset('wikitext', 'wikitext-2-raw-v1', split='test')

    from transformers import AutoTokenizer
    try:
        tokenizer = AutoTokenizer.from_pretrained(model, use_fast=False)
    except:
        tokenizer = AutoTokenizer.from_pretrained(model, use_fast=True)
    trainenc = tokenizer("\n\n".join(traindata['text']), return_tensors='pt')
    testenc = tokenizer("\n\n".join(testdata['text']), return_tensors='pt')

    import random
    random.seed(seed)
    np.random.seed(0)
    torch.random.manual_seed(0)

    traindataset = []
    for _ in range(nsamples):
        i = random.randint(0, trainenc.input_ids.shape[1] - seqlen - 1)
        j = i + seqlen
        inp = trainenc.input_ids[:, i:j]
        attention_mask = torch.ones_like(inp)
        traindataset.append({'input_ids':inp, 'attention_mask': attention_mask})
    return traindataset, testenc
```

Durchführen der Quantisierung

Mit der Variable „bits“ wird die gewünschte Bitbreite der Quantisierung bestimmt. Der „Group-size“ Wert wurde in allen Fällen bei 128 belassen.

```
from transformers import AutoTokenizer, TextGenerationPipeline
from auto_gptq import AutoGPTQForCausalLM, BaseQuantizeConfig

tokenizer = AutoTokenizer.from_pretrained(pretrained_model_dir, use_fast=True)
```

```

quantize_config = BaseQuantizeConfig(
    bits=3, # quantize model to 4-bit
    group_size=128, # Default for groupsize 128
    desc_act=False, # False speeds up quantization most timer
)

# Default: Load model to CPU memory
model = AutoGPTQForCausalLM.from_pretrained(pretrained_model_dir,
                                              quantize_config)

# Examples as a list of dict. Keys can only be "input_ids" and "attention_mask"
print ("Quantize Model")
traindataset, testenc = get_wikitext2(128, 0, 4096, pretrained_model_dir)
model.quantize(traindataset, use_triton=False, batch_size=1
, cache_examples_on_gpu=False)

print ("Saving model...")
# save quantized model using safetensors
model.save_quantized(quantized_model_dir, use_safetensors=True)
tokenizer = AutoTokenizer.from_pretrained(pretrained_model_dir, use_fast=True)
tokenizer.save_pretrained(quantized_model_dir)

```

3.2 Pruning: PruneMe

In diesem Abschnitt wird die Implementation des PruneMe Verfahrens aus der Veröffentlichung [31] dargestellt. Zu der Veröffentlichung gehört das folgende GitHub-Repository [32].

3.2.1 Überblick PruneMe Verfahren

Analyse des Ursprungs Modells

- Wechsel in das Verzeichnis compute_block_similarity:
- Ausführen des Tasks layer_similarity generiert eine CSV Datei welche im selben Verzeichnis abgelegt wird
- Aus dieser Datei sucht man die Layer mit den geringsten Unterschied als Basis für den Merge Task heraus

Erstellen des neuen "Pruned"Modells

- Wechsel in das Verzeichnis slice_with_mergekit
- editieren der Datei slice.yaml mit den entsprechenden layern und modellnamen (siehe Beispiel unten)
- ausführen von merge_me.py
- das neue Modell wird im Verzeichnis "merged" abgelegt

3.2.2 Erstellen der Arbeitsumgebung

```
conda activate PruneMe
```

Generelle Pfade

```
export PMODELOUT="/home/thsch026/masterarbeit/models/generated/prune/pruneme/"
export PRERESULTS="/home/thsch026/masterarbeit/Slim/results/PruneMe/"
```

3.2.3 PruneMe Workflow

Auswahl des Modells

Llama-3-8B-Instruct-HF

```
export MODEL="/home/thsch026/masterarbeit/models/llama3/
Meta-Llama-3-8B-Instruct-HF"
export LAYERSPRUNED="6"
export OUT="Meta-Llama-3-8B-Instruct-\"$LAYERSPRUNED\"_Ext"
```

6 layers extracted

```
export MODEL="/home/thsch026/masterarbeit/models/llama3/
Meta-Llama-3-8B-Instruct-HF"
export LAYERSPRUNED="8"
export OUT="Meta-Llama-3-8B-Instruct-\"$LAYERSPRUNED\"_Ext"
```

8 Layers extracted

```
export MODEL="meta-llama/Meta-Llama-3-8B-Instruct"
export LAYERSPRUNED="12"
export OUT="Meta-Llama-3-8B-Instruct-\"$LAYERSPRUNED\"_Ext"
```

```
export MODEL="meta-llama/Meta-Llama-3-8B"
export LAYERSPRUNED="8"
export OUT="Meta-Llama-3-8B-\"$LAYERSPRUNED\"_Ext"
```

12 Layers extracted

```
export MODEL="meta-llama/Meta-Llama-3-8B"
export LAYERSPRUNED="12"
export OUT="Meta-Llama-3-8B-\"$LAYERSPRUNED\"_Ext"
```

12 Layers extracted

Mistral 7B Instruct v0.2

```
export MODEL="mistralai/Mistral-7B-Instruct-v0.2"
export LAYERSPRUNED="8"
export OUT="Mistral-7B-Instruct-v0.2-\"$LAYERSPRUNED\"_Ext"
```

8 Layers extracted

```
export MODEL="mistralai/Mistral-7B-Instruct-v0.2"
export LAYERSPRUNED="12"
export OUT="Mistral-7B-Instruct-v0.2-\"$LAYERSPRUNED\"_Ext"
```

12 Layers extracted

Llama 2 7B chat

```
export MODEL="meta-llama/Llama-2-7b-chat-hf"
export LAYERSPRUNED="8"
export OUT="Llama-2-7b-chat-hf-\"$LAYERSPRUNED\"_Ext"
```

8 Layers extracted

```
export MODEL="meta-llama/Llama-2-7b-chat-hf"
export LAYERSPRUNED="12"
export OUT="Llama-2-7b-chat-hf-\"$LAYERSPRUNED\"_Ext"
```

12 Layers extracted

Beispiel für den Inhalt von slice.yaml

```

slices:
- sources:
  - model: /home/thsch026/masterarbeit/models/llama3/
    Meta-Llama-3-8B-Instruct-HF
    layer_range: [0, 22]
- sources:
  - model: /home/thsch026/masterarbeit/models/llama3/
    Meta-Llama-3-8B-Instruct-HF
    layer_range: [30,32]

merge_method: passthrough
dtype: bfloat16

```

Setzen der Directories

```

export PRUNEMODEL="$PMODELOUT$OUT"
export RESULTS="$PREULTS$OUT"
echo $PRUNEMODEL

```

Festlegen der GPU für die Analyse

```

export CUDA_VISIBLE_DEVICES="0"

```

(/home/thsch026/my-envs/PruneMe)

Analyse für die Extraktion

```

cd '/home/thsch026/masterarbeit/experiment/PruneMe/compute_block_similarity'
python layer_similarity.py --model_path $MODEL \
                        --dataset "arcee-ai/sec-data-mini" \
                        --dataset_column "text" \
                        --batch_size 8 \
                        --max_length 1024 \
                        --layers_to_skip $LAYERSPRUNED \
                        --dataset_size 4000 \
                        --dataset_subset "train"

```

Sichern der Results

```
mkdir $RESULTS
cp layer_distances.csv $RESULTS
```

Ausführen des Merging Tasks

```
export MERGE='/home/thsch026/masterarbeit/experiment/PruneMe/
slice_with_mergekit'
cp $RESULTS/slice.yaml $MERGE
cd $MERGE
python merge_me.py
```

```
mv merged $PRUNEMODEL
```

3.3 Pruning und LoRA: Shortened-LLM

Implementationen zum in 2.2.4 dargestellten Verfahren. Die Implementation basiert auf den in [57] genannten Github Repository.

Download der Dateien und erstellen des conda environments

```
conda create --name shortened python==3.10
```

```
conda create -n shortened-llm python=3.9
conda activate shortened-llm
git clone https://github.com/Nota-NetsPresso/shortened-llm.git
cd shortened-llm
pip install -r requirement.txt
```

Installation zusätzlicher Pakete, die in dem Setup des GitHub Repositories nicht enthalten waren, aber benötigt werden

```
pip install typing_extensions packaging pygments psutil
```

3.3.1 Workflow zur Modell Optimierung

Das Verfahren wird in einer Bash Shell gestartet. Dazu wird die conda Umgebung „shortened-llm“ geladen.

```
conda activate shortened-llm
```

```
cd /home/thsch026/masterarbeit/experiment/shortened-llm
```

Festlegen einer nutzbaren GPU, die manuell festgelegt werden muss

```
export CUDA_VISIBLE_DEVICES="2"
echo $CUDA_VISIBLE_DEVICES
```

Durchführung des Workflows

- In dem Verzeichnis „script“ der Installation findet man vorgefertigte Scripte für die Nutzung des Workflows.
 - In diesen Scripten müssen ggf. Anpassungen vorgenommen werden (z.Bsp. wie viele Schichten des Modells entfernt werden sollen)
- In den Scripten müssen einige Variablen entsprechend der verwendeten Umgebung angepasst werden.
- Das jeweilige Script für das spezifisch zu komprimierende Modell wird ausgeführt.

Llama2-7B-Chat

```
script/prune_llama2-7b_crit-taylor.sh
```

Llama3-8B-Instruct

```
script/prune_llama3-8b-instruct_crit-taylor.sh
```

Mistral-7B-Instruct-v0.2

```
script/prune_Mistral_crit-taylor.sh
```

3.3.2 Ergebnisse des Workflows

- Im Ordner „output_prune“ findet man die geprunten Modelle
- Im zweiten Schritt wird das Model noch mittels der LoRA Verfahrens trainiert. Die fertigen Modelle (pruned + lora) befinden sich im Ordner „output_tune“

3.4 QLoRA

3.4.1 Erstellen und konfigurieren der Arbeitsumgebung

Anlegen der conda Arbeitsumgebung (qlora) und Installation der notwendigen Pakete. Dieser Schritt ist nur einmalig notwendig.

Erstellen der Arbeitsumgebung

```
conda create --name qlora python==3.10
```

```
conda activate qlora
```

Installation der Pakete

```
# Workaround for packaging fehler
pip install setuptools==69.5.1
```

```
pip install torch==2.0.1
pip install bitsandbytes
pip install -U transformers[torch] datasets
pip install -q bitsandbytes trl peft accelerate
pip install flash-attn --no-build-isolation
pip install transformers==4.40.2 # Es geht nur genau diese Version!!
pip install trl
pip install autoawq
pip install huggingface_hub
pip install sentencepiece mistral_inference
```

3.4.2 Qlora Training Workflow

Auswahl der GPU und import der Bibliotheken

```
import os

os.environ["PYTORCH_CUDA_ALLOC_CONF"] = "max_split_size_mb:256"
os.environ["CUDA_VISIBLE_DEVICES"] = "2"
!echo $CUDA_VISIBLE_DEVICES
```

```
from huggingface_hub import notebook_login
from transformers import AutoTokenizer, AutoModelForCausalLM
import torch
```

Auswahl des zu trainierenden Modells

LLama 3 8B Instruct

```
model_id = "meta-llama/Meta-Llama-3-8B-Instruct"

# Storage Location
trained_model_id = "Meta-Llama-3-8B-Instruct_qlora"
output_dir = '/home/thsch026/masterarbeit/models/generated/qlora/'
+ trained_model_id
```

Mistral 7B instruct v0.2

```
model_id = "mistralai/Mistral-7B-Instruct-v0.2"

# Storage Location
trained_model_id = "Mistral-7B-Instruct-v0.2_qlora"
output_dir = '/home/thsch026/masterarbeit/models/generated/qlora/'
+ trained_model_id
```

Llama 2 7b chat hf

```
model_id = "meta-llama/Llama-2-7b-chat-hf"

# Storage Location
trained_model_id = "Llama-2-7b-chat-hf_qlora"
output_dir = '/home/thsch026/masterarbeit/models/generated/qlora/'
+ trained_model_id
```

Laden des Modells

```
model = AutoModelForCausalLM.from_pretrained(model_id)
tokenizer = AutoTokenizer.from_pretrained(model_id)
```

Vorbereiten des Datasets

```

from datasets import load_dataset

# based on config
raw_datasets = load_dataset("HuggingFaceH4/ultrachat_200k")

from datasets import DatasetDict

# remove this when done debugging to include whole dataset
indices = range(0,10000)

dataset_dict = {"train": raw_datasets["train_sft"].select(indices),
                "test": raw_datasets["test_sft"].select(indices)}

raw_datasets = DatasetDict(dataset_dict)

example = raw_datasets["train"][0]
messages = example["messages"]
for message in messages:
    role = message["role"]
    content = message["content"]
    print('{0:20}: {1}'.format(role, content))

tokenizer.eos_token_id

terminators = [
    tokenizer.eos_token_id,
    tokenizer.convert_tokens_to_ids("<|eot_id|>")
]

```

Setzen des Chat Templates

```

import re
import random
from multiprocessing import cpu_count

def apply_chat_template(example, tokenizer):
    messages = example["messages"]
    example["text"] = tokenizer.apply_chat_template(messages, tokenize=False)

return example

```

```

column_names = list(raw_datasets["train"].features)
raw_datasets = raw_datasets.map(apply_chat_template,
                               num_proc=cpu_count(),
                               fn_kwargs={"tokenizer": tokenizer},
                               remove_columns=column_names,
                               desc="Applying chat template",)

# set pad_token_id equal to the eos_token_id if not set
if tokenizer.pad_token_id is None:
    tokenizer.pad_token_id = tokenizer.eos_token_id

# Set reasonable default for models without max_length
if tokenizer.model_max_length > 100_000:
    tokenizer.model_max_length = 2048

# create the splits
train_dataset = raw_datasets["train"]
eval_dataset = raw_datasets["test"]

model = AutoModelForCausalLM.from_pretrained(
    model_id,
    torch_dtype=torch.float16,
    device_map="auto",
)

messages = [
    {"role": "system", "content": "You are a gentle Chatbot!"},
    {"role": "user", "content": "Who are you?"},
]

input_ids = tokenizer.apply_chat_template(
    messages,
    add_generation_prompt=True,
    return_tensors="pt"
).to(model.device)

terminators = [
    tokenizer.eos_token_id,
    tokenizer.convert_tokens_to_ids("<|eot_id|>")
]

torch.backends.cuda.enable_mem_efficient_sdp(False)
torch.backends.cuda.enable_flash_sdp(False)

outputs = model.generate(
    input_ids,
    max_new_tokens=128,
)

```

```

        eos_token_id=terminators,
        do_sample=True,
        temperature=0.6,
        top_p=0.9,
    )

response = outputs[0][input_ids.shape[-1]:]

print(tokenizer.decode(response, skip_special_tokens=True))

```

3.4.3 Prepare Training

```

from trl import SFTTrainer
from peft import LoraConfig
from transformers import TrainingArguments
from transformers import BitsAndBytesConfig

# Konfiguration 8 bit quantization
quantization_config = BitsAndBytesConfig(load_in_8bit=True,
                                           llm_int8_threshold=200.0)

model_kwargs = dict(
    torch_dtype="auto",
    use_cache=False,
    device_map="auto",
    quantization_config=quantization_config,
)

training_args = TrainingArguments(
    fp16=True,
    do_eval=True,
    evaluation_strategy="epoch",
    gradient_accumulation_steps=1,
    gradient_checkpointing=True,
    gradient_checkpointing_kwarg={"use_reentrant": False},
    learning_rate=2.0e-05,
    log_level="info",
    logging_steps=5,
    logging_strategy="steps",
    lr_scheduler_type="cosine",
    max_steps=-1,
    num_train_epochs=1,
    output_dir=output_dir,
)

```

```

overwrite_output_dir=True,
per_device_eval_batch_size=1, # Anpassung an GPU Memory
per_device_train_batch_size=1, # Anpassung an GPU Memory
push_to_hub=True,
hub_model_id=trained_model_id,
report_to="none",
save_strategy="no",
save_total_limit=None,
seed=42,
)

# LoRA Konfiguration mit "best practice" Parametern
peft_config = LoraConfig(
    r=64,
    lora_alpha=16,
    lora_dropout=0.1,
    bias="none",
    task_type="CAUSAL_LM",
    target_modules=["q_proj", "k_proj", "v_proj", "o_proj"],
)
trainer = SFTTrainer(
    model=model_id,
    model_init_kwargs=model_kwargs,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    dataset_text_field="text",
    tokenizer=tokenizer,
    packing=True,
    peft_config=peft_config,
    max_seq_length=tokenizer.model_max_length,
)
# To clear out cache for unsuccessful run
torch.cuda.empty_cache()

```

3.4.4 Start Training

```
train_result = trainer.train()
```

3.4.5 Speichern des Modells

```
tokenizer.save_pretrained(output_dir)
model.save_pretrained(output_dir)
```

3.5 Dual-Space Knowledge Distillation (DSKD)

- Die Implementation verwendet das angegebene GitHub Repository [58]

3.5.1 Erstellen der Arbeitsumgebung

Erstellen der Arbeitsumgebung (kd) mittels conda. Nur einmalig notwendig.

```
conda create --name dskd python==3.10
```

```
pip install deepspeed==0.14.0 torch==2.0.1 transformers==4.40.2 peft==0.8.2
rouge_score==0.1.2 editdistance==0.8.1
```

Aktivieren der Arbeitsumgebung

```
conda activate dskd
# make sure to be in the right working directory
cd /home/thsch026/masterarbeit/experiment/DSKD
```

3.5.2 Prozess der Knowledge Distillation

Beispiel: Vorbereiten des Mistral Modells als teacher

Wichtig: Die Scripte enthalten sehr viele Parameter von denen i.d.R. nur die im Folgenden aufgeführt angepasst werden müssen. Die konkret verwendeten Scripte findet man unter [56], da sie ansonsten den Umfang der Arbeit zu stark erweitern würden.

Folgende Variablen müssen angepasst werden:

- BASE_PATH: Hier z.Bsp. „scripts/tinnyllama/sft_teacher_mistral.sh“
- GPUS: Welche GPU(s) verwendet werden soll(en)
- CKPT_TYPE: Verzeichnis in dem sich die Gruppe der betrachteten Modelle befindet.
- CKPT_NAME: Verzeichnis des Student Modells
- Verschiedene Variablen, die mit TEACHER_ beginnen: Bauen den Pfad zum Teacher Model auf.
- PRECISION: Hier muss ggf. die verwendete Variabletype angepasst werden.

```
scripts/tinnyllama/sft_teacher_mistral.sh
```

Resultate eines Prozess Durchlaufs (Beispiel):

- Das Ausgabe Verzeichnis hängt vom Namen des Modells und dem Prozess ab (Siehe Beispiel unten)
- In diesem Verzeichnis findet man neben den Ergebnissen des Prozess auch die Logdateien des Laufs. Die Subverzeichnisse enthalten im Namen auch die wesentlichen Parameter des spezifischen Tasks.
- Besonders zu Beginn sollte man sehr sorgfältig die Einträge in den Logdateien prüfen, da durch die Komplexität des Prozesses häufig Abbrüche entstehen.

```
cd outputs/mistral/mistral-7b-v0.1/sft/
```

Durchführung der Knowledge Distillation

Es war das Ziel mit diesem Verfahren, die durch Quantisierung und/oder pruning komprimierten Modelle mittels Knowledge Distillation weiter zu verbessern. Daher werden hier die resultierenden Modelle aus den vorhergehenden Experimenten als Ausgangsmodelle verwendet.

- Ein mit AWQ komprimiertes Modell lässt sich mit dem DKSD Verfahren nicht optimieren. Evt. ist dies eine Limitation durch das verwendete Cuda 11.7

Übersicht Finetuning der Teacher Modelle

Finetuning Mixtral

```
cd /home/thsch026/masterarbeit/experiment/DSKD
scripts/toms/sft_teacher_mixtral.sh
```

Finetuning Teacher: Llama 3 8B Instruct

```
cd /home/thsch026/masterarbeit/experiment/DSKD
scripts/toms/sft_tommodel_llama3.sh
```

Finetuning Teacher: Mistral 7B Instruct v0.2

```
cd /home/thsch026/masterarbeit/experiment/DSKD
scripts/toms/sft_tommodel_mistral.sh
```

KD mit den finegetunten Teacher Modellen (sft) für die komprimierten Student Modelle

- In den Scripts wurden, wie weiter oben beschrieben, eine Reihe von Parametern angepasst.

Llama 3 8B prune lora -> Teacher: Llama 3 8B Instruct (sft)

```
cd /home/thsch026/masterarbeit/experiment/DSKD
scripts/toms/dskd_tommodel_llama3.sh
```

Mistral prune Lora -> Teacher: Mistral 7B Instruct v0.2 (sft)

```
cd /home/thsch026/masterarbeit/experiment/DSKD
scripts/toms/dskd_tommodel_mistral.sh
```

3.6 Evaluierung der Modelle

Die Evaluierung der Modelle lässt sich in zwei Bereiche gliedern. Zum einen erfolgt eine Evaluierung mit den unter 2.6.1 besprochenen Verfahren, zum anderen wird die Inferenz Geschwindigkeit und das Resultat auf einige Anfragen an das jeweilige Modell bestimmt.

3.6.1 Evaluierung der Modelle mit verschiedenen Benchmarks

Anlegen des Environments - Einmalig

```
conda create --name eval python==3.10
```

```
# Clone Git Repo and install dependencies
cd "/home/thsch026/masterarbeit/work/"
git clone https://github.com/EleutherAI/lm-evaluation-harness
cd lm-evaluation-harness
```

Initialisieren der Arbeitsumgebung - Muss nach jedem Neustart des Servers erfolgen

```
conda activate eval
cd "/home/thsch026/masterarbeit/work/"
```

```
pip install -e .
pip install torch==2.0.1 sentencepiece
```

```
conda activate eval
```

Festlegen der Definitionen für die Modelle

- Hier wird beispielhaft gezeigt, wie die Variablen für ein Modell gebildet werden. Die vollständige Datei mit den Definitionen für alle evaluierten Modelle findet man unter [56].
- Die WANDB Variablen ermöglichen die einfache Zuordnung der jeweiligen Ergebnisse in der zugehörigen Visualisierungsplattform.

Reference Model Llama-3-8B-Instruct

```
export MODEL="meta-llama/Meta-Llama-3-8B-Instruct"
export WANDB_NAME="Llama 3 8B instruct Referenz"
export WANDB_NOTES="Llama 3 8B instruct Referenz Messung"
```

Durchführung der Evaluierung

Festlegen der GPU

```
export CUDA_VISIBLE_DEVICES="0"
echo $CUDA_VISIBLE_DEVICES
```

Evaluierung mit den Standard Benchmarks

```
lm_eval --model hf --model_args pretrained=$MODEL,dtype=float16 --tasks
arc_challenge,truthfulqa_mc2,winogrande,hellaswag \
--device cuda:$CUDA_VISIBLE_DEVICES --batch_size auto
--wandb_args project=lm-eval-harness-integration --log_samples \
--output_path "/home/thsch026/masterarbeit/Slim/results"
```

Perplexity Evaluierung mit wikitext

- Wird getrennt ausgeführt, da ein recht hoher Speicherbedarf für diese Evaluierung besteht.
- Die WANDB Ergebnisse werden mit dem Suffix "WIKI" versehen, damit sie leichter identifizierbar sind.

```
# Modify entries for the results
export WANDB_NAME=$WANDB_NAME" WIKI"
export WANDB_NOTES=$WANDB_NOTES" WIKI"
echo "Running task for:"
echo $WANDB_NAME
echo $WANDB_NOTES
# Run Task
lm_eval --model hf --model_args pretrained=$MODEL,tokenizer=$MODEL
,dtype=float16,use_safetensors=True,parallelize=True --tasks wikitext2 \
--device cuda --batch_size auto
--wandb_args project=lm-eval-harness-integration --log_samples \
--output_path "/home/thsch026/masterarbeit/Slim/results"
```

3.6.2 Inferenz Messung

Erstellen der Arbeitsumgebung - Einmalig

```
conda create --name inference python==3.10
conda activate inference

pip install transformers optimum sentencepiece
pip install auto-gptq --extra-index-url https://huggingface.github.io/
autogptq-index/wheel/cu117/
```

Laden der Bibliotheken und festlegen der GPU

```
import os

os.environ["PYTORCH_CUDA_ALLOC_CONF"] = "max_split_size_mb:256"
os.environ["CUDA_VISIBLE_DEVICES"] = "3"
!echo $CUDA_VISIBLE_DEVICES

import time
import torch
import transformers
from transformers import AutoTokenizer, AutoModelForCausalLM, BitsAndBytesConfig
import bitsandbytes as bnb
import accelerate
import csv
import packaging
from awq import AutoAWQForCausalLM
```

Konfiguration der Ergebnis Dateien

```
full_csv= "/home/thsch026/masterarbeit/Slim/results/
inference/inference_full.csv"
data_csv= "/home/thsch026/masterarbeit/Slim/results/
inference/inference_memory_mean.csv"
```

Beispiel für die Definition der verwendeten Variablen anhand des Modells Llama 3 8B Instruct

```
model_id ="Meta-Llama-3-8B-Instruct"
model_loc = "meta-llama/Meta-Llama-3-8B-Instruct"
tokenizer = "meta-llama/Meta-Llama-3-8B-Instruct"
```

Durchführung der Inferenz Messung

Laden des konfigurierten Modells

```
print("\nLoading Model: ", model_id)
tokenizer = AutoTokenizer.from_pretrained(tokenizer)

quantization_config = BitsAndBytesConfig()

model = AutoModelForCausalLM.from_pretrained(
    model_loc,
    trust_remote_code=True,
    torch_dtype=torch.float16,
    device_map='cuda'
)
```

Messung der Inferenz mit Bildung der Durchschnittswerte

```
from statistics import mean

# Funktion für das Speichern der Werte
def append_data(result):

    with open(full_csv, mode='a', newline='') as file:
        full_writer = csv.writer(file)
        full_writer.writerow(result)
```

```

def append_avg(result_avg):

    with open(data_csv, mode='a', newline='') as file:
        data_writer = csv.writer(file)
        data_writer.writerow(result_avg)

device = 'cuda'
token_avg_list = []
memory_avg_list = []

question_list = ["Once upon a time",
                 "Why should I eat healthy?",
                 "Is there a second life after death?",
                 "What is the most common injury?",
                 "Who is Konrad Zuse?"]

for question in question_list:
    prompt = f'''You are an helpful assistant. USER: {question} ASSISTANT:'''
    input_ids = tokenizer.encode(question, return_tensors='pt')
    input_ids = tokenizer(question, return_tensors="pt").input_ids.to(device)
    input_ids.to('cuda')
    start_time = time.time()

    # Führe die Inferenz durch
    outputs = model.generate(input_ids, max_length=200)

    # Messe die Zeit nach der Inferenz
    end_time = time.time()

    # Berechne die Inferenz-Zeit
    inference_time = end_time - start_time

    # Dekodiere die Ausgabe
    output_text = tokenizer.decode(outputs[0], skip_special_tokens=True)

    # Berechnung des Ergebnis

    number_of_tokens = len(outputs[0])
    memory_usage = round((torch.cuda.memory_allocated() / (1024 **3)),2)
    token_per_second = number_of_tokens/inference_time
    token_avg_list.append(token_per_second)
    memory_avg_list.append(memory_usage)
    result = [model_id,token_per_second, inference_time,
              memory_usage, question, output_text]
    print(result)
    append_data(result)

token_avg = mean(token_avg_list)
memory_avg = mean(memory_avg_list)

```

```
result_avg = [model_id, token_avg, memory_avg]
append_avg(result_avg)
print(result_avg)
```


4 Ergebnisse

In dem folgenden Kapitel werden die Ergebnisse zu den verschiedenen Verfahren der Modell Optimierung vorgestellt. Eine Übersicht über die durchgeführten Experimente bietet die Tabelle 4.1

Verfahren	Llama 3 8B Instruct	Mistral 7B Instruct v0.2	Llama 2 7B chat
Depth pruning (Layer Extraction mit PruneMe)	X	X	-
Activation-aware Weight Quantization (AWQ)	X	X	X
Generic Pretrained Transformer Quantisation (GPTQ)	X	X	-
Kombinierte Verfahren			
Pruning + Lora (Shortened Llm)	X	X	X
Pruning + Quantization (Qlora)	X	X	X
Pruning + Lora + Knowledge distillation	X	X	-
Pruning + Lora + AWQ	X	X	X

Tabelle 4.1: Übersicht über die im Kapitel Ergebnisse enthaltenen Verfahren und Modelle

4.1 Experimente zur Quantisierung von Modellen

4.1.1 Benchmark Ergebnisse der AWQ Quantisierung

Die Tabelle 4.2 , zeigt die Benchmark Ergebnisse der mit AWQ quantisierten Modelle, jeweils im Vergleich zum Ausgangsmodell. Es ist zu erkennen, dass nur sehr geringe Unterschiede zwischen Ursprungs und den mit dem AWQ Verfahren quantisierten Modellen bestehen.

Benchmarks:	winogrande	truthfulqa_mc2	hellawag	arc_challenge	Durchschnitt
Mistral 7 Referenz	0,7395	0,6682	0,8365	0,5606	0,7012
Standard Fehler	<i>0,0123</i>	<i>0,0152</i>	<i>0,0037</i>	<i>0,0145</i>	
Mistral 7 AWQ	0,7403	0,6752	0,8310	0,5700	0,7041
Standard Fehler	<i>0,0123</i>	<i>0,0151</i>	<i>0,0037</i>	<i>0,0145</i>	
Llama 3 Referenz	0,7206	0,5165	0,7582	0,5674	0,6407
Standard Fehler	<i>0,0126</i>	<i>0,0152</i>	<i>0,0043</i>	<i>0,0145</i>	
Llama 3 AWQ	0,7356	0,5099	0,7532	0,5572	0,6390
Standard Fehler	<i>0,0124</i>	<i>0,0152</i>	<i>0,0043</i>	<i>0,0145</i>	
Llama 2 Referenz	0,6646	0,4532	0,7547	0,4420	0,5786
Standard Fehler	<i>0,0133</i>	<i>0,0156</i>	<i>0,0043</i>	<i>0,0145</i>	
Llama 2 AWQ	0,6456	0,4509	0,7481	0,4462	0,5727
	<i>0,0134</i>	<i>0,0156</i>	<i>0,0043</i>	<i>0,0145</i>	

Tabelle 4.2: Die Tabelle zeigt die quantisierten Modellen zusammen mit den korrespondierenden Referenzen. Für die Quantisierung wurde das Verfahren AutoAWQ (Groupszie 128) verwendet.

4.1.2 Speicherbedarf der AWQ Modelle im Vergleich zum Ausgangsmodell

In diesem Experiment wurde die Auslastung des GPU Speichers durch die Modelle gemessen. Die Ergebnisse sind in 4.1 dargestellt. Man erkennt, dass die mit AWQ quantisierten Modelle, in Vergleich zur jeweiligen Referenz, nur ca. 35 % des GPU Speichers belegen.

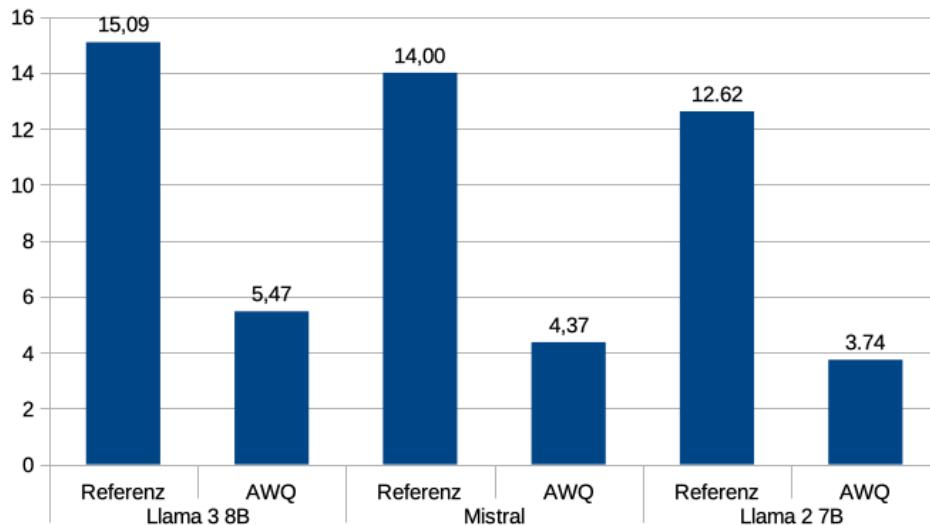


Abbildung 4.1: Auslastung des GPU Speichers durch die jeweiligen Modelle auf einer Nvidia A100 80GB GPU. Y-Achse zeigt die Speicherauslastung in Gigabyte.

4.1.3 AWQ Quantisierung mit verschiedenen Werten für den Groupsize Parameter

In der Tabelle 4.3 wird dargestellt wie sich die Veränderung des „Groupsize“ Parameters bei der AWQ Quantisierung auf die Ergebnisse auswirkt. Man erkennt, dass alle Benchmark Resultate sehr nah bei einander liegen. Die Unterschiede befinden sich im Bereich des Fehlerfensters, daher ist kein Trend erkennbar. Alle folgenden Quantisierungen werden mit dem Wert 128 für den Groupsize Parameter durchgeführt.

Benchmarks:	winogrande	truthfulqa_mc2	hellawag	arc_challenge	Durchschnitt
AWQ 64	0.7048	0.5687	0.7400	0.4804	0.6235
Standard Fehler	<i>0.0128</i>	<i>0.0155</i>	<i>0.0044</i>	<i>0.0146</i>	
AWQ 128	0.7017	0.5666	0.7406	0.4829	0.6229
Standard Fehler	<i>0.0129</i>	<i>0.0155</i>	<i>0.0044</i>	<i>0.0146</i>	
AWQ 256	0.7017	0.5637	0.7379	0.4889	0.6230
Standard Fehler	<i>0.0129</i>	<i>0.0154</i>	<i>0.0044</i>	<i>0.0146</i>	

Tabelle 4.3: Vergleich des Mistral 7B Instruct Modells mit unterschiedlicher Parametrierung des Groupsize Parameters bei der AWQ Quantisierung. Der Groupsize Parameter wurde auf die Werte 64,128 und 256 gesetzt.

4.1.4 GPTQ Quantisierung und Vergleich mit AWQ

In einem weiteren Experiment wurde die GPTQ Quantisierung verwendet und mit dem AWQ Ansatz verglichen. Dieses Experiment wurde nur mit dem Llama 3 8B instruct Basismodell und einer Variante durchgeführt, die in einem späteren Abschnitt beschrieben wird. Die Quantisierung erfolgte mit 4 bit und einer Groupszie von 128.

Llama 3 8B instruct	winogrande	truthfulqa_mc2	hellawag	arc_challenge	Durchschnitt
referenz	0,7206 0,0126	0,5165 0,0152	0,7582 0,0043	0,5674 0,0145	0,6407
GPTQ	0,7230 0,0126	0,5196 0,0152	0,7388 0,0044	0,5341 0,0146	0,6289
AWQ	0,7370 0,0124	0,5099 0,0152	0,7529 0,0043	0,5580 0,0145	0,6395
pruned + lora + GPTQ	0,7222 0,0126	0,5002 0,0151	0,7158 0,0045	0,4940 0,0146	0,6080
pruned + lora + AWQ	0,7230 0,0126	0,5273 0,0150	0,7322 0,0044	0,4872 0,0146	0,6174

Tabelle 4.4: Die Tabelle zeigt einen Vergleich von Modellen die mit zwei unterschiedlichen Verfahren, GPTQ und AWQ, quantisiert wurden

Man erkennt, das beide GPTQ Modelle im Durchschnitt geringfügig schlechtere Benchmark Ergebnisse aufweisen, als die mit den AWQ Verfahren quantisierten Modelle.

4.1.5 Verschiedene Bitbreiten bei der GPTQ Quantisierung

Die Tabellen 4.5 und 4.6 zeigen welchen Einfluss die Genauigkeit der Werte (Weights) auf die Benchmark-Performanz für das Mistral 7 und das Llama 3 Modell hat. Die Verringerung der Bitbreite zeigt bis zu 3-Bit einen relativ geringen Abfall der Werte. Nach Reduzierung der Bandbreite auf 2-Bit fallen die Benchmark Werte dann allerdings deutlich ab.

Mistral 7B Instruct	winogrande	truthfulqa_mc2	hellawag	arc_challenge	Durchschnitt
Referenz	0,7395	0,6682	0,8365	0,5606	0,7012
Standard Fehler	0,0123	0,0152	0,0037	0,0145	
4 Bit	0,7277	0,6600	0,8294	0,5503	0,6919
Standard Fehler	0,0125	0,0153	0,0038	0,0145	
3 Bit	0,7167	0,6193	0,8021	0,5145	0,6632
Standard Fehler	0,0127	0,0158	0,0040	0,0146	
2 Bit	0,4996	0,4656	0,3193	0,2543	0,3847
Standard Fehler	0,0141	0,0161	0,0047	0,0127	

Tabelle 4.5: Die Tabelle zeigt einen Vergleich von quantisierten Modellen, die mit unterschiedlichen Bitbreiten quantisiert wurden. Die Quantisierung erfolgte mit dem GPTQ Verfahren für das Modell Mistral 7B Instruct v0.2.

In einer weiteren Tabelle ist die Inferenz Geschwindigkeit und die Speicherbelegung der GPU durch die GPTQ Modelle für verschiedene Bit Breiten angegeben. Die 3-bit Variante erreicht bei dem Llama 3 Modell die geringste Speicherauslastung, aber die Inferenz Geschwindigkeit ist fast

Llama 3 8B Instruct	winogrande	truthfulqa_mc2	hellawag	arc_challenge	Durchschnitt
Referenz	0,7206	0,5165	0,7582	0,5674	0,6407
<i>Standard Fehler</i>	<i>0,0126</i>	<i>0,0152</i>	<i>0,0043</i>	<i>0,0145</i>	
4 bit	0,7324	0,3897	0,7737	0,4812	0,5943
<i>Standard Fehler</i>	<i>0,0124</i>	<i>0,0146</i>	<i>0,0042</i>	<i>0,0146</i>	
3 bit	0,6543	0,5131	0,6859	0,4249	0,5696
<i>Standard Fehler</i>	<i>0,0134</i>	<i>0,0149</i>	<i>0,0046</i>	<i>0,0144</i>	
2 bit	0,5241	0,4973	0,2653	0,2526	0,3848
<i>Standard Fehler</i>	<i>0,0140</i>	<i>0,0164</i>	<i>0,0044</i>	<i>0,0127</i>	

Tabelle 4.6: Die Tabelle zeigt einen Vergleich von quantisierten Modellen, die mit unterschiedlichen Bitbreiten quantisiert wurden. Die Quantisierung erfolgte mit dem GPTQ Verfahren für das Modell Llama 3 8B Instruct.

halbiert. Unerwartet ist die größere Speicherauslastung durch die 2-bit Modelle und das Mistral 3-bit Modell. Hier war eine geringere Auslastung, als bei den korrespondierenden 4-bit Modellen erwartet worden.

Model	Bit	Token/sec	Speicher GB
Mistral 7B Instruct v0.2	4-bit	6,18	4,38
	3-bit	3,69	8,05
	2-bit	6,61	7,11
Llama-3-8B-Instruct	4-bit	6,12	5,47
	3-bit	3,51	4,76
	2-bit	6,55	8,59

Tabelle 4.7: Inferenz Geschwindigkeit und Speicherallokation von GPTQ Modellen mit verschiedenen Bitbreiten.

4.2 Ergebnisse der Pruning Experimente mit „PruneMe“

Die Tabelle 4.8 zeigt die Benchmark Ergebnisse für Modelle, die mit dem PruneMe Verfahren modifizierte wurden. Die Referenz (0 Schichten extrahiert) wird dabei mit zwei Extraktions Varianten (8 bzw. 12 Schichten extrahiert) verglichen.

Benchmarks:	Layers Pruned	winogrande	truthfulqa_mc2	hellawag	arc_challenge	Durchschnitt
Llama 3 8B	0	0,7206	0,5165	0,7582	0,5674	0,6407
	<i>Standard Fehler</i>	<i>0,0126</i>	<i>0,0152</i>	<i>0,0043</i>	<i>0,0145</i>	
	8	0,6369	0,5211	0,5897	0,4283	0,5440
	<i>Standard Fehler</i>	<i>0,0135</i>	<i>0,0159</i>	<i>0,0049</i>	<i>0,0145</i>	
	12	0,5706	0,5201	0,4204	0,3208	0,4580
Mistral 7 B	0	0,7395	0,6682	0,8365	0,5606	0,7012
	<i>Standard Fehler</i>	<i>0,0123</i>	<i>0,0152</i>	<i>0,0037</i>	<i>0,0145</i>	
	8	0,6811	0,6313	0,6646	0,4044	0,5954
	<i>Standard Fehler</i>	<i>0,0131</i>	<i>0,0158</i>	<i>0,0047</i>	<i>0,0143</i>	
	12	0,5943	0,5543	0,3239	0,3677	0,4601
	<i>Standard Fehler</i>	<i>0,0138</i>	<i>0,0167</i>	<i>0,0047</i>	<i>0,0141</i>	

Tabelle 4.8: In der Tabelle sind die Benchmark Ergebnisse für Modelle aufgeführt, bei denen 8 oder 12 Schichten mittels Pruning entfernt wurden.

Der Abstand bei den Resultaten zwischen 0 und 8 extrahierten Schichten ist dabei in etwa so groß, wie der Abstand zwischen 8 und 12 extrahierten Schichten. In der Diskussion wird dieses Ergebnis näher besprochen. Dieses Experiment wurde nicht mit dem Modell Llama 2 7B durchgeführt.

Die Abbildung 4.2 zeigt, zu welcher Veränderung der Speicherauslastung die Extraktion der Schichten führt.

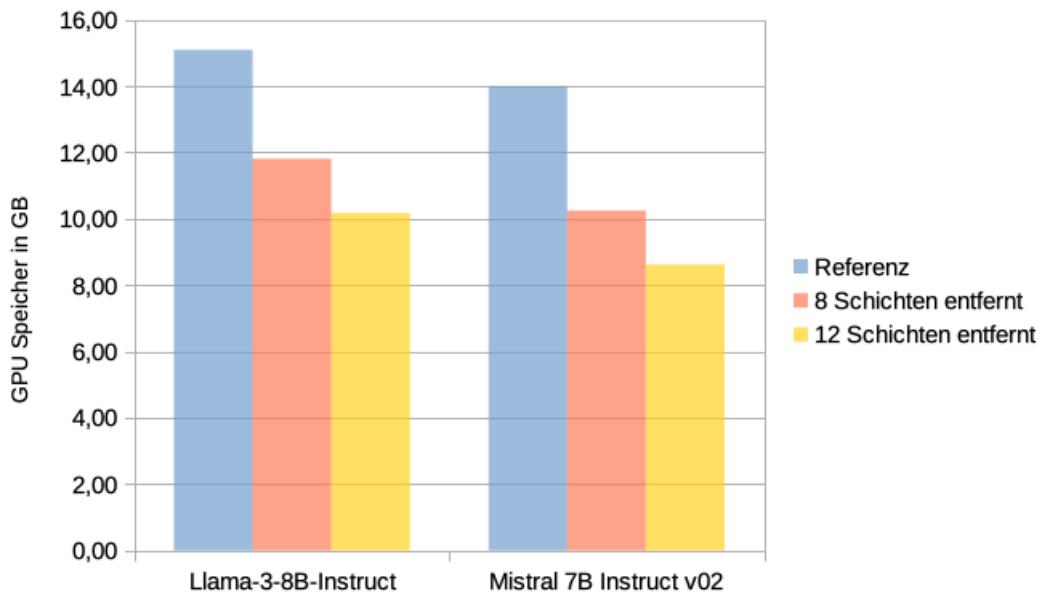


Abbildung 4.2: Auslastung des GPU Speichers einer Nvidia A100 80GB GPU durch Modelle, die mit dem PruneMe Verfahren verkleinert wurden.

4.3 Kombination von Verfahren zur Kompression von LLM Modellen

In diesem Abschnitt werden die Ergebnisse vorgestellt, welche durch die Anwendung einer Kombination von verschiedenen Technologien auf die Ursprungsmodelle erhalten wurden. Dabei werden, neben Quantisierung und Pruning, auch LoRA , QLoRA und Knowledge Distillation verwendet.

4.3.1 Ergebnisse der Experimente mit dem QLoRA Verfahren

Tabelle 4.9 zeigt die Ergebnisse der Experimente mit dem QLoRA Verfahren. Die Benchmark Ergebnisse liegen für das Llama 3 und das Mistral Modell sehr nah bei denen, des zugehörigen Referenz Modells. Das mit QLora modifizierte Llama 2 Modell fällt gegenüber dem Referenz Modell deutlich ab. Die Messung mit der „truthfulqa_mc2“ Benchmark schlug aus unbekannten Gründen bei dem Llama 2 Modell fehl.

	winogrande	truthfulqa_mc2	hellawag	arc_challenge	Durchschnitt
Llama 3 8B instruct Referenz	0,7206 0,0126	0,5165 0,0152	0,7582 0,0043	0,5674 0,0145	0,6407
Llama 3 8B Instruct QLoRA	0,7190 0,0126	0,5169 0,0153	0,7539 0,0043	0,5478 0,0145	0,6344
Mistral 7B Instruct v0.2 Referenz	0,7395 0,0123	0,6682 0,0152	0,8365 0,0037	0,5606 0,0145	0,7012
Mistral 7B Instruct v0.2 QLoRA	0,7230 0,0126	0,6479 0,0154	0,8294 0,0038	0,5674 0,0145	0,6919
Llama 2 7B chat Referenz	0,6646 0,0133	0,4532 0,0156	0,7547 0,0043	0,4420 0,0145	0,5786
Llama 2 7B chat QLoRA	0,4957 0,0141	na	0,2504 0,0043	0,2270 0,0122	0,3244

Tabelle 4.9: In der Tabelle sind die Benchmark Ergebnisse für die QLoRA Modelle, im Vergleich zur Referenz aufgeführt.

4.3.2 Tuning eines geprunten Modells mittels LoRA („Shortened LLM“)

Die Tabelle 4.10 zeigt die Benchmark Ergebnisse für die Anwendung des „Shortened LLM“ Verfahrens. Das Modell wird dabei zunächst geprunt und in einem zweiten Schritt mit dem LoRA Ansatz retrainiert.

In diesem Experiment wurden, im Unterschied zum vorhergehenden, nur sechs Schichten aus dem jeweiligen Grundmodell extrahiert. Dies führt erwartungsgemäß zu etwas besseren Benchmark Ergebnissen nach der pruning Phase, als bei den vorhergehenden Experimenten, bei dem 8 bzw. 12 Schichten extrahiert wurden. Die Benchmark Ergebnisse der Ursprungsmodelle sind zum einfacheren Vergleich ebenfalls aufgeführt.

Man erkennt, dass bei den beiden Llama Modellen durch das LoRA Retraining eine signifikante Verbesserung nach dem Pruning erreicht wird. Für das Mistral Modell ist dieser Effekt nicht erkennbar. Diese Beobachtung wird in einem späteren Kapitel der Arbeit diskutiert.

Modell	Verfahren	winogrande	truthfulqa_mc2	hellawag	arc_challenge	Durchschnitt
Llama 3 8B Instruct	Referenz	0,7206 0,0126	0,5165 0,0152	0,7582 0,0043	0,5674 0,0145	0,6407
	pruned	0,7111 0,0127	0,5000 0,0157	0,6255 0,0048	0,4625 0,0146	0,5748
	pruned + lora	0,7316 0,0125	0,5189 0,0151	0,7269 0,0044	0,4974 0,0146	0,6187
Mistral 7B Instruct v0.2	Referenz	0,7395 0,0123	0,6682 0,0152	0,8365 0,0037	0,5606 0,0145	0,7012
	pruned	0,7072 0,0128	0,6370 0,0158	0,7298 0,0044	0,4599 0,0146	0,6335
	pruned + lora	0,7174 0,0127	0,5774 0,0154	0,7457 0,0043	0,4855 0,0146	0,6315
Llama 2 7B chat	Referenz	0,6646 0,0133	0,4532 0,0156	0,7547 0,0043	0,4420 0,0145	0,5786
	pruned	0,6409 0,0135	0,4955 0,0159	0,6384 0,0048	0,3968 0,0143	0,5429
	pruned + lora	0,6732 0,0132	0,4886 0,0154	0,7082 0,0045	0,4206 0,0144	0,5727

Tabelle 4.10: Die Tabelle zeigt die Ergebnisse der Experimente mit dem Shortened-LLM Verfahren.

4.3.3 Kombination von Pruning, LoRA und Knowledge Distillation

In diesem Experiment wurde versucht, ein mit „Shortened LLM“ bereits gepruntes und retrainiertes Modell, mittels Knowledge Distillation weiter zu verbessern. Als Teacher Modell wurde dabei jeweils das korrespondierende, unbearbeitete Grundmodell verwendet.

Es ist unerwartet, dass die Qualität der Modelle nach dem Knowledge Distillation Prozess abnimmt. Vielmehr wurde der gegenteilige Effekt erwartet. Um einen systematischen Fehler auszuschließen, wurden die Versuche noch einmal unabhängig wiederholt. Die Ergebnisse der zweiten Versuchsrunde entsprachen exakt denen der ersten Versuchsreihe, was einen systematischen Fehler unwahrscheinlich erscheinen lässt.

Benchmarks:	winogrande	truthfulqa_mc2	hellaswag	arc_challenge	Durchschnitt
LLama 3 8B pruned lora	0,7316	0,5189	0,7269	0,4974	0,6187
Fehler	0,0125	0,0151	0,0044	0,0146	
LLama 3 8B pruned lora dist	0,6535	0,4425	0,6645	0,4386	0,5498
Fehler	0,0134	0,0166	0,0047	0,0145	
Mistral pruned lora	0,7174	0,5774	0,7457	0,4855	0,6315
Fehler	0,0127	0,0154	0,0043	0,0146	
Mistral pruned lora dist	0,6330	0,4654	0,6694	0,4437	0,5529
Fehler	0,0135	0,0162	0,0047	0,0145	

Tabelle 4.11: Optimierungsversuch mit dem Knowledge Distillation Verfahren, bei Modellen die zuvor mit pruning verkleinert und mit LoRA retrainiert wurden.

4.3.4 Kombination von Pruning, LoRA und Quantisierung

In einem weiteren Experiment wurden, die durch das „Shortened LLM“ Verfahren erhaltenen Modelle, mit Hilfe des AWQ Verfahrens einer 4-bit Quantisierung unterzogen. Die Ergebnisse werden im Vergleich mit den anderen Modellvarianten im nächsten Abschnitt dargestellt. Es handelt sich dabei jeweils um das letzte Modell (pruned + lora +awq) in den Tabellen 4.12, 4.14 und 4.13.

4.3.5 Vergleichende Darstellung der erzeugten Modelle

Die drei zugehörigen Tabellen 4.12, 4.14 und 4.13 zeigen eine Zusammenfassung der Benchmark Ergebnisse, für die zuvor beschriebenen Experimente. Man erkennt bei den Llama Modellen, dass nur die jeweils geprunte Variante, gegenüber den anderen Modellen einen signifikanten Abfall in den Benchmark Werten hat. Alle übrigen Varianten der beiden Llama Modelle bewegen sich in der Nähe des Ursprungsmodeells. Wie im Abschnitt 4.3.2 bereits angesprochen, bleibt das Mistral Modell nach dem Pruning auf einem niedrigeren Niveau und zeigt keine Verbesserung durch die Anwendung des LoRA Verfahrens.

Llama 3 8B instruct	winogrande	truthfulqa_mc2	hellawag	arc_challenge	Durchschnitt
Referenz	0,7206 Standard Fehler 0,0126	0,5165 0,0152	0,7582 0,0043	0,5674 0,0145	0,6407
pruned (shortened-LLM)	0,7111 Standard Fehler 0,0127	0,5000 0,0157	0,6255 0,0048	0,4625 0,0146	0,5748
AWQ	0,7370 Standard Fehler 0,0124	0,5099 0,0152	0,7529 0,0043	0,5580 0,0145	0,6395
GPTQ	0,7230 Standard Fehler 0,0126	0,5196 0,0152	0,7388 0,0044	0,5341 0,0146	0,6289
pruned + lora	0,7316 Standard Fehler 0,0125	0,5189 0,0151	0,7269 0,0044	0,4974 0,0146	0,6187
pruned + lora + awq	0,7230 Standard Fehler 0,0126	0,5273 0,0150	0,7322 0,0044	0,4872 0,0146	0,6174
pruned + lora + gptq	0,7222 Standard Fehler 0,0126	0,5002 0,0151	0,7158 0,0045	0,4940 0,0146	0,6080
QLoRA	0,7190 Standard Fehler 0,0126	0,5169 0,0153	0,7539 0,0043	0,5478 0,0145	0,6344

Tabelle 4.12: Die Tabelle zeigt die Benchmark Ergebnisse der verschiedenen Llama 3 8B Instruct Modelle im Vergleich.

Mistral 7 B Instruct v0.2	winogrande	truthfulqa_mc2	hellawag	arc_challenge	Durchschnitt
Referenz	0,7395 Standard Fehler 0,0123	0,6682 0,0152	0,8365 0,0037	0,5606 0,0145	0,7012
pruned (shortened-LLM)	0,7072 Standard Fehler 0,0128	0,6370 0,0158	0,7298 0,0044	0,4599 0,0146	0,6335
AWQ 4bit	0,7403 Standard Fehler 0,0123	0,6752 0,0151	0,8310 0,0037	0,5700 0,0145	0,7041
GPTQ 4-bit	0,7277 Standard Fehler 0,0125	0,6600 0,0153	0,8294 0,0038	0,5503 0,0145	0,6919
Prune + lora	0,7174 Standard Fehler 0,0127	0,5774 0,0154	0,7457 0,0043	0,4855 0,0146	0,6315
prune + lora + awq	0,7017 Standard Fehler 0,0129	0,5666 0,0155	0,7406 0,0044	0,4829 0,0146	0,6229
Qlora	0,7230 Standard Fehler 0,0126	0,6479 0,0154	0,8294 0,0038	0,5674 0,0145	0,6919

Tabelle 4.13: Die Tabelle zeigt die Benchmark Ergebnisse der verschiedenen Mistral 7B Instruct Modelle im Vergleich.

Llama 2 7B chat	winogrande	truthfulqa_mc2	hellawag	arc_challenge	Durchschnitt
Referenz	0,6646 Standard Fehler 0,0133	0,4532 0,0156	0,7547 0,0043	0,4420 0,0145	0,5786
pruned (shortened-LLM)	0,6409 Standard Fehler 0,0135	0,4955 0,0159	0,6384 0,0048	0,3968 0,0143	0,5429
awq 4-Bit	0,6456 Standard Fehler 0,0134	0,4509 0,0156	0,7481 0,0043	0,4462 0,0145	0,5727
pruned + lora	0,6732 Standard Fehler 0,0132	0,4886 0,0154	0,7082 0,0045	0,4206 0,0144	0,5727
pruned + lora + awq	0,6614 Standard Fehler 0,0133	0,4873 0,0154	0,6986 0,0046	0,4224 0,0144	0,5674
QLoRA	0,4957 Standard Fehler 0,0141	na	0,2504 0,0043	0,2270 0,0122	0,3244

Tabelle 4.14: Die Tabelle zeigt die Benchmark Ergebnisse der verschiedenen Llama 2 7B chat Modelle im Vergleich.

4.3.6 Perplexity Ergebnisse

Die Tabelle 4.15 zeigt die Perplexity Ergebnisse für verschiedene vom Referenz Modell abgeleitet Varianten des Grundmodells Llama 3 8B instruct. Man erkennt den deutlich höheren(schlechteren) Wert, der ausschließlich geprunten Modell Variante. Alle anderen Varianten liegen auf einem deutlich besseren Niveau.

Ein ähnliches Ergebnis zeigen die korrespondierenden Ergebnisse 4.16 für da Mistral Modell. Der Wert des geprunten Modells ist auch hier deutlich schlechter als alle anderen, wenn auch nicht ganz so extrem, wie bei dem Llama 3 Modell.

Für diese Messung wurde jeweils das geprunte Modell aus dem „Shortened LLM“ Verfahren verwendet, bei dem sechs Schichten extrahiert wurden.

Modell: Llama 3 8B instruct	bits_per_byte	byte_perplexity	word_perplexity
Referenz	0,62	1,54	9,94
AWQ	0,63	1,55	10,30
GPTQ	0,65	1,57	10,53
pruned	1,08	2,11	53,86
prune + lora	0,71	1,64	13,96
prune + lora + awq	0,70	1,62	13,24
QLoRA	0,64	1,56	10,19

Tabelle 4.15: Perplexity Messungen für die verschiedenen Varianten des Llama 3 8B Instruct Modells. Bei der Pruned Variante wurden sechs Schichten entfernt. Bei GPTQ und AWQ handelt es sich um 4-bit Quantisierungen.

Modell: Mistral 7B Instruct v0.2	bits_per_byte	byte_perplexity	word_perplexity
Referenz	0,64	1,56	10,15
AWQ	0,65	1,57	10,35
GPTQ	0,65	1,57	10,26
QLoRA	0,65	1,57	10,25
pruned (shortened-LLM)	0,99	1,96	32,71
Pruned + LoRA	0,74	1,66	14,15
Pruned + lora + awq	0,75	1,68	14,64

Tabelle 4.16: Perplexity Messungen für die verschiedenen Varianten des Mistral 7B Instruct v0.2 Modells. Bei der Pruned Variante wurden sechs Schichten entfernt. Bei GPTQ und AWQ handelt es sich um 4-bit Quantisierungen.

4.3.7 Inferenz Ergebnisse

In diesem Abschnitt werden die Inferenz Ergebnisse für die verschiedenen Modelle dargestellt. Die Ergebnisse sind dabei in zwei Tabellen aufgeteilt, da sie in verschiedenen Umgebungen erstellt wurden. Hintergrund ist, dass die CUDA 11.7 Umgebung des Fachhochschul-Clusters, die Quantisierung mit AWQ nicht unterstützt. Die in Tabelle 4.17 mit einem * gekennzeichneten Inferenz Werte stellen somit nicht die optimal erreichbaren Werte da.

In Tabelle 4.18 wurde daher das Referenz Modell noch einmal mit der Variante verglichen, die

nach pruning und LoRA Optimierung (Siehe auch: 2.2.4), mit AWQ quantisiert wurde. In diesem Experiment wurde eine Google Colab Umgebung verwendet, die mit einer Nvidia T4 GPU und der Cuda 12.5 Version ausgestattet war. Hier sollten die AWQ Modelle ihre vollständige Leistungsfähigkeit erreichen können.

Aufgrund der unterschiedlichen Leistungsstärke von T4 und A100 GPU können die Werte beider Tabellen nicht direkt miteinander verglichen werden. Man erkennt aber anhand von Abbildung 4.18, dass die getestete AWQ Modellvarianten eine ca. 3,2 - 3,5 X höhere Inferenz Geschwindigkeit, als die zugehörigen Grundmodelle erreichen.

Modell	Referenz	awq*	gptq	pruned	qlora	Pruned + lora	Pruned + lora +awq*
Llama-3-8B-Instruct	27,35	2,40	6,12	34,42	29,39	36,08	3,03
Mistral-7B-Instruct-v0.2	35,03	2,47	6,18	41,35	31,60	38,50	3,01
Llama-2-7b-chat	31,42	1,40	na	38,22	28,41	40,03	1,84

Tabelle 4.17: Inferenz Performance von nativen zu bearbeiteten Modellen. Die mit einem * gekennzeichneten Modelle können aufgrund der fehlenden AWQ Unterstützung der Cuda 11.7 Umgebung, nicht ihre volle Leistungsfähigkeit entfalten. Im Falle von Llama 2 wurde für das GPTQ Modell kein Wert bestimmt. Verwendet wurde hier eine Nvidia A100 GPU mit 80GB Speicher.

Modell Name	Referenz	prune+lora+awq
Llama 3 8B Instruct	10,22	33,34
Mistral 7B Instruct v02	11,10	38,99
Llama 2 7B chat	11,75	35,69

Tabelle 4.18: Inferenz Performance von nativen zu AWQ quantisierten Modellen. Die Messungen wurden in einer Google Colab Umgebung mit einer Nvidia T4 15 GB durchgeführt. Hier konnten Cuda 12.x Treiber genutzt werden, die eine Verwendung des AutoAWQ python Moduls ermöglichten.

Antwortqualität der Modelle

In den Tabellen 4.19 und 4.20 sind beispielhaft Antworten jeweils drei verschiedener Modell Varianten aufgeführt. Man erkennt, dass die Qualität der Reaktionen bei den ausschließlich geprunten Varianten deutlich schlechter ausfällt, als bei den Original Modellen. Hier gibt es Wiederholungen und Abbrüche, welche die schlechten Perplexity Ergebnisse aus den vorherigen Experimenten deutlich belegen.

Die Antworten der „pruned + lora + awq“ Modelle liegen auf dem Niveau des Original Models, auch wenn im Falle von Llama 3 die Antwort des Originals etwas besser zu Erwartung passt.

Tabelle 4.19: Antworten verschiedener Varianten des LLaMA 3 8B Instruct Modells auf die Frage:
 „Why should I eat healthy?“

Reaktionen verschiedener Varianten des Modells Mistral 7B Instruct auf den Ausspruch „Once upon a time“	
Referenz	Once upon a time, in a land far, far away, there was a beautiful princess named Isabella. She lived in a magnificent castle nestled atop a hill, surrounded by lush gardens and rolling hills. The princess was known for her kindness, her grace, and her beauty. But there was a problem. The kingdom was in grave danger. A terrible dragon had taken up residence in the nearby forest, and it demanded a daily tribute of livestock and gold from the people. If they failed to pay, the dragon would burn their homes and villages to the ground. The king, who was a wise and just ruler, was determined to put an end to the dragon's reign of terror. He called upon his bravest knights to slay the beast, but none returned. The dragon was too powerful, too fierce, and too cunning. The princess, who was deeply troubled by the plight of
pruned (sechs Schichten)	Once upon a time, the world was a much simpler and safer environment. Unterscheidung. 100% Free Online Date & Personals Classifiedsads.
pruned + lora +awq	The first time I heard the story of Jesus Christ dying for our sin and coming back to save the world, it was a story that resonated deeply with me. The story of Jesus Christ dying for our sin and saving the world sounds like a story that is far removed from the world of the average Christian. The story of Jesus Christ dying for our sin sounds far removed from the world of the average Christian. The story of Jesus Christ dying for our sin sounds far removed from the world of the average Christian. The story sounds far removed from the world of the average Christian. The story sounds far removed from the world of the average Christian. The story sounds far removed from the world of the average Christian. The story sounds far removed from the world of the average Christian. The story sounds far removed from the world of the average Christian.
pruned + lora +awq	Once upon a time there were two friends Tom and Nick. Both were very intelligent and had a great potential to do something good in the society, but what made them different was the approach of their thinking and actions. Tom was always looking for ways to do something good for his friends, neighbors and his society. Nick on the other hand always had a different take on things. He always liked to look for any loopholes in any good deed, just to find an excuse to avoid helping others while he got back to his comfort zone. One day, Tom told Nick about a charity drive that he thought that both would like to join, it would involve selling donuts and giving the profit towards charity. "This is an excellent idea", said Nick, it is something that benefits everyone. So the day of the charity drive arrived, and Tom and Nick were standing on a street corner selling donuts. A whole lot of people were stopped and had their attention their

Tabelle 4.20: Reaktionen verschiedener Varianten des Mistral 7B Instruct Modells auf die Aussage:
 „Once upon a time“

4.4 Systemressourcen

4.4.1 Speicherbedarf der Modelle

Die Grafik zeigt die GPU Speicherauslastung aller Varianten der drei Grundmodelle. Im Vergleich zu den Grundmodellen zeigen praktisch alle Varianten Einsparungen beim Speicherverbrauch.

Besonders ausgeprägt ist dieser Effekt bei den mittels des AWQ Verfahrens quantisierten Modellen.

Modell	Referenz	pruned	AWQ	GPTQ	Pruned + LoRA	Pruned + LoRA + AWQ	QLoRA
Llama 3 8B Instruct	15,09	12,63	5,47	5,47	12,63	4,81	5,44
Mistral 7B Instruct v0.2	14	11,47	4,37	4,38	11,47	3,65	9,79
Llama 2 7B chat	12,62	10,35	3,74	na	10,35	3,13	na

Tabelle 4.21: Speicherbedarf der meisten Modellvarianten im Vergleich. Angegeben ist jeweils die Belegung des GPU Speichers in Gigabyte (Nvidia A100 80GB). In zwei Fällen konnte für das Llama 2 Modell kein Wert ermittelt werden.

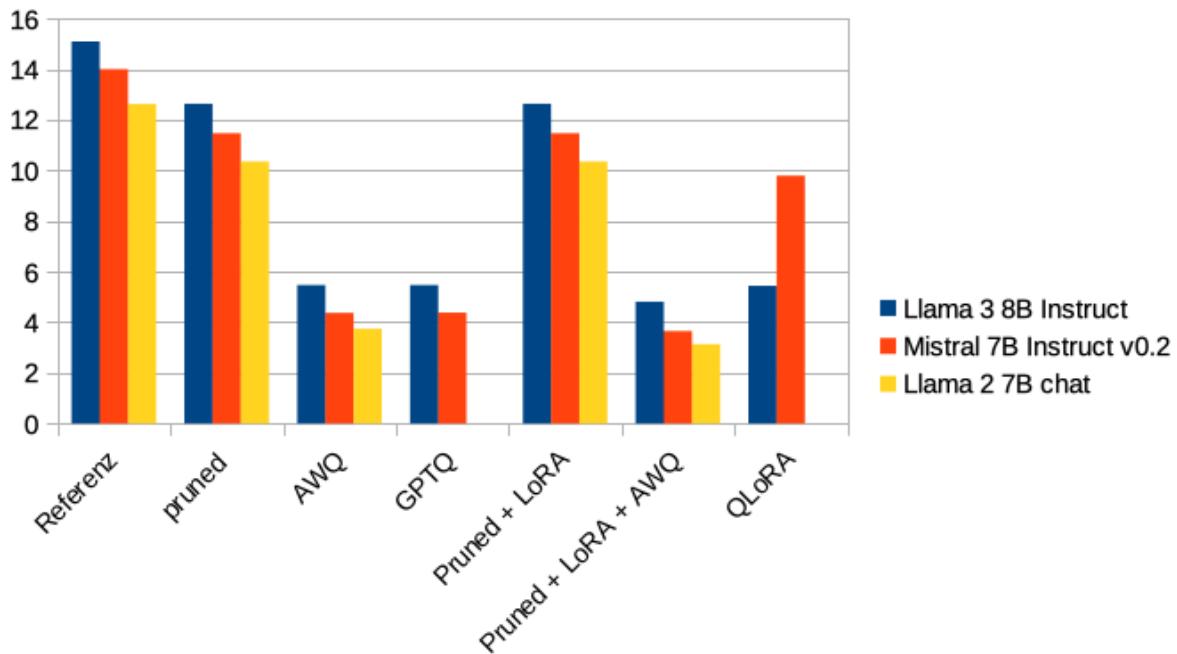


Abbildung 4.3: Speicherbedarf der Modelle auf einer Nvidia A100 80GB GPU. Aus dem jeweiligen Referenz Modell wurden alle weiteren Modelle abgeleitet. Die X-Achse zeigt die verwendeten Verfahren.

4.4.2 Inferenz auf einem Desktop Computer

Die Abbildung 4.4 zeigt beispielhaft die Inferenz mit dem Modell „Llama 3 8B Instruct pruned+lora+awq“ Modell auf einem Desktop Computer (Apple iMac) von 2015. Der Computer verfügt über keine nutzbare GPU Unterstützung, sondern das Modell wird auf der CPU ausgeführt. Es belegte bei der Ausführung ca. 4,8 GB Hauptspeicher und generierte die Antwort in 76 Sekunden. Die Qualität der Inferenz ist auf dem Niveau des nicht modifizierten Ursprungsmodells.

```
[>>> Why does AI need so much resources?  
AI ( Artificial Intelligence) technologies require a significant amount of computing power and memory to process and analyze large amounts  
of data quickly and efficiently. This is because the algorithms used in AI are complex, and they work by making many calculations  
simultaneously to produce results.
```

The exact resources needed for AI depend on the specific task or problem being solved, but generally, high-performance computing systems with multiple processors, specialized graphics processing units (GPUs), and large amounts of memory are required. These can be either dedicated servers or powerful laptops that have been modified for computational tasks.

There are several reasons why AI requires so much resources:

1. Computational power: The algorithms used in AI require a lot of mathematical calculations, which is more efficient with high-performance computing systems that can perform these calculations quickly and accurately.
2. Data storage: Many AI applications use data from external sources such as databases or web crawlers, which can generate large amounts of data. This data needs to be stored on servers or hard drives for easy access and processing.
3. Training: Training an AI system requires a lot of resources because the process involves feeding it large datasets to learn patterns and make decisions. The more data that is used in training, the better the system will become at solving problems and making predictions.
4. Model complexity: Some AI models are complex and require significant processing power to run. These models can take up a considerable amount of memory, and they may need to be trained for extended periods of time, which also requires large amounts of resources.

In summary, AI requires a lot of resources because it needs high-performance computing systems, large data storage capacity, powerful training methods, and complex algorithms that require significant computational power.

```
>>> send a message (/? for help)
```

Abbildung 4.4: Beispielhafte Inferenz eines optimierten Modells auf einem älteren Desktop Computer.

5 Diskussion der Ergebnisse

In diesem Kapitel werden die Ergebnisse der Experimente besprochen und ergänzende Betrachtungen durchgeführt. Im einzelnen werden die folgenden Punkte behandelt:

- Ergebnisse der Experimente mit den Einzelverfahren
- Ergebnisse der kombinierten Verfahren
- Beurteilung zu Größe/Geschwindigkeit/Qualität der Modelle
- Relation zu andern Techniken (Finetuning, RAG)
- Schlussfolgerung und Ausblick

5.1 Ergebnisse der Experimente

5.1.1 Ergebnisse der Quantisierungs-Experimente

Das erste wesentliche Verfahren zur Verkleinerung von Modellen ist die Verringerung der Variablen Präzision. Aus einer Vielzahl von Quantisierungs-Verfahren wurde die „Activation aware Weight Quantization (AWQ)“ und die „Generative Pre-Trained Transformers Quantization (GPTQ)“ ausgewählt (Siehe auch 2.1.2). Verglichen wurden hier nur Quantisierungen mit 4-Bit, da das AutoAWQ Modul keine Unterstützung für andere Bitbreiten liefert.

Beide Verfahren erreichen beeindruckende Ergebnisse bei der Reduktion der GPU Speicherauslastung (ca. 2/3). Dieses Ergebnis geht mit nur geringen Einschränkungen der Modell Qualität einher, wie sich in den 4.2, 4.4 und 4.15 an den nur geringfügig schlechteren Werten im Vergleich zum Ursprungsmodell erkennen lässt. Das GPTQ Verfahren erreicht dabei im Durchschnitt geringfügig schlechtere Benchmark Werte als die AWQ Quantisierung.

Das AWQ Verfahren zeigte darüber hinaus auch eine deutliche Erhöhung der Inferenz Geschwindigkeit (ca. 3-3,5 fach). Für die Ermittlung der AWQ Inferenz Werte musste dabei auf die Google Colab Plattform mit Cuda 12.x ausgewichen werden, da nur dort das notwendige Python Modul AutoAWQ unterstützt wurde (siehe auch 4.18).

Bei den GPTQ Modellen konnte keine Erhöhung der Inferenz Geschwindigkeit festgestellt werden, vielmehr stellte sich sogar eine deutliche Verschlechterung ein (siehe auch 4.17), was u.U. an einer fehlenden Optimierung des Verfahrens im Inferenz Bereich liegen könnte. Dieser Punkt ließ sich leider nicht abschließend klären.

Verschiedene Groupsize Werte

Ein Parameter, der potentiell die Qualität der Quantisierung beeinflussen kann, ist die Verwendung unterschiedlicher Werte für die „Groupsize“. Das zugehörige Experiment ist in der Tabelle 4.3 dargestellt.

Die Unterschiede zwischen den ermittelten Benchmark Werten, für verschiedene „Groupsize“ Varianten bei der AWQ Quantisierung, sind sehr gering und liegen in vielen Fällen im Bereich des Standard Fehlers. Das führt zu dem Fazit, dass die Variation dieses Parameters bei der Optimierung nur einen nachgeordneten Stellenwert hat. Es kann aber grundsätzlich nicht ausgeschlossen werden, dass sich der Einfluss bei anderen Modellen deutlicher auswirkt.

Unterschiedliche Bitbreiten

Wesentlich interessanter sind die Experimente, bei denen unterschiedliche Bitbreiten für die GPTQ Quantisierung verwendet wurden. Leider konnte dieses Experiment, wegen fehlender Unterstützung für andere Bitbreiten nicht mit dem AWQ Verfahren durchgeführt werden.

Sowohl die Quantisierung mit 4-Bit, als auch mit 3-Bit ergibt in den Benchmarks Werte, die sehr nahe an den Werten des Ursprungsmodells liegen. Die mit 2-Bit quantisierten Modelle fallen dagegen bei der Qualität deutlich ab und sind vermutlich nicht mehr sinnvoll für eine Inferenz geeignet. Die 3-Bit Quantisierung verringert die GPU-Speicherbelegung bei dem Llama 3 Modell noch einmal um ca. 15 %, im Vergleich zum 4-Bit Modell, führt aber beinahe zu einer Halbierung der Inferenz Geschwindigkeit.

Fazit Quantisierung

Quantisierung stellt ein sehr mächtiges Werkzeug für die Ressourcenoptimierung von Modellen dar und sollte daher als primäre Maßnahme für die Optimierung genutzt werden. Die Verminderung der notwendigen Ressourcen ist signifikant, bei nur leichten Verlusten der Modell Qualität. Darüber hinaus ist die Quantisierung relativ leicht anzuwenden und robust in ihren Ergebnissen. Wünschenswert wäre, wenn das sehr effiziente AWQ Verfahren auch für eine 3-Bit Quantisierung zur Verfügung stände.

In der Summe der Eigenschaften (Benchmark Werte, GPU- Speicherbelegung und Inferenz Geschwindigkeit), fiel die Wahl auf die AWQ Quantisierung für die nachfolgenden Experimente. Je nach Priorität der Optimierung und verwendeten Modellen, kann in ähnlichen Ansätzen aber auch die Nutzung von GPTQ statt AWQ sinnvoll sein.

5.1.2 Ergebnisse der Pruning-Experimente

Der zweite Block der Experimente befasste sich mit der Verkleinerung der Modelle durch Pruning. In 2.2 wurde erläutert, dass aktuell das „Semi strukturierte Pruning“ besonders gute Ergebnisse verspricht. Daher wurde dieser Ansatz für diese Arbeit ausgewählt und in zwei Ausprägungen untersucht.

Mit dem PruneMe Verfahren wurden Modelle erstellt, die 8 bzw. 12 Schichten weniger besaßen, als die Grundmodelle mit 32 Schichten. Erwartungsgemäß entstanden Modelle, die einen entsprechend geringeren Speicherbedarf aufwiesen (-25% bzw. -32% siehe 4.2) und deren Inferenz Geschwindigkeit sich moderat verbesserte (Tabelle 4.17).

Die zugehörigen Benchmark Ergebnisse zeigen aber einen deutlichen Abfall der Qualität (siehe

4.8) dieser Modelle, gegenüber der Referenz. Insbesondere die Werte für die Modelle mit 12 extrahierten Schichten, liegen auf einem niedrigen Niveau. Unterstützt wird dieses Bild von dem schlechten Perplexity Ergebnis in 4.15, für das mit sechs extrahierten Schichten geprunte Modell aus dem „Shortened LLM“ Verfahren.

In Summe ist festzuhalten, dass Pruning alleine kaum Mehrwert für eine Optimierungs-Strategie zu Verringerung des Ressourcenbedarfs ergibt. Die deutlichen Nachteile in der Kompetenz der Modelle wiegen die moderaten Vorteile bei Speicherauslastung und Inferenz Geschwindigkeit nicht auf. Wir werden aber bei den kombinierten Verfahren noch sehen, dass Pruning durchaus einen Mehrwert bieten kann.

5.2 Kombinierte Verfahren

Zur Erstellung Ressourcen optimierter Modelle wurden auch Kombinationen von Verfahren auf die Grundmodelle angewandt und die zugehörigen Messungen durchgeführt. Neben den oben besprochenen Verfahren, wurde dabei auch das LoRA Verfahren und die zugehörige Variante QLoRA verwendet. Darüber hinaus wurden auch Kombinationen mit dem Verfahren Knowledge Distillation untersucht.

5.2.1 QLoRA

Wie in 2.4.2 bereits beschrieben, wird bei QLoRA ein quantisiertes Modell mittels des LoRA Verfahrens erneut trainiert. Im Gegensatz zum dem weiter unten beschriebenen Modell (pruned+lora+awq) wird bei dem QLoRA Verfahren das Modell zuerst quantisiert und erst im zweiten Schritt mit dem LoRA Verfahren retrainiert.

Die in 4.9 dargestellten Benchmarks und die in 4.16 und 4.15 angegebenen Perplexity Ergebnisse, sind für das Llama 3 und das Mistral Modell sehr nah bei den Referenz Modellen. Die Speicherauslastung ist dabei für das Llama 3 Modell um fast 2/3 gegenüber dem Referenz Modell reduziert (Tabelle 4.21). Eine solch starke Reduktion der Speicherauslastung, konnte für Mistral 7 nicht beobachtet werden. Hier liegt die Einsparung bei verhältnismäßig geringen 25 %.

Bei den Messungen der Inferenz Geschwindigkeit liegen die Werte auf Höhe des Ausgangsmodells (Tabelle 4.17), somit gibt es bei diesem Aspekt keinen Vorteil für die QLoRA Modelle.

5.2.2 Shortened LLM

Das „Shortened LLM“ Verfahren kombiniert das Pruning eines Modells mit dem LoRA Verfahren. Im ersten Schritt werden aus dem Modell mittels Pruning sechs Schichten entfernt, was als moderates Pruning gelten kann. Dies schlägt sich in einer signifikanten Verschlechterung der Benchmark und Inferenz Ergebnissen nieder, wie man in Tabelle 4.10 gut erkennen kann.

Daher wendet „Shortened LLM“ nach dem Pruning im zweiten Schritt das LoRA Verfahren an, um die Modellqualität durch Re-Training wieder zu erhöhen. Tabelle 4.10 zeigt, dass dieser Ansatz, zumindest bei den beiden Llama Modellen einen positiven Effekt hat. Im Falle von Llama 2 liegt der Benchmark-Durchschnittswert, nach dem LoRA Re-Training, sogar auf Höhe des Originalmodells .

Bei dem Mistral Modell ist ein positiver Effekt durch den zweiten Schritt des LoRA Re-Trainings nicht zu erkennen. Das Modell verbleibt auf dem Niveau des geprunten Modells. Ein Grund für die ausbleibende Verbesserung, könnte eine fehlende Optimierung des Verfahrens für das Mistral

Modell sein. Dies ließ sich, trotz eines persönlichen Kontakts mit den Entwicklern des Verfahrens, nicht abschließend klären.

5.2.3 Knowlegde Distillation

Aufbauend auf den mit „Shortened LLM“ erstellten Modellen, wurde das Knowledge Distillation Verfahren (siehe auch 2.3.2) verwendet, um ggf. eine weitere Verbesserung der Benchmark Werte zu erhalten. Die Ergebnisse in Tabelle 4.11 zeigen allerdings, ein zu den Erwartungen gegensätzliches Bild. Die Modelle, welche dem Knowledge Distillation Verfahren unterzogen wurden, weisen durchweg signifikant schlechtere Ergebnisse als die Ausgangsmodelle auf. Folgende Gründe könnten für das unerwartete Ergebnis verantwortlich sein:

- Obwohl die Versuche wiederholt wurden, um mögliche Verwechslungen zu erkennen, kann die Wahrscheinlichkeit eines systematischen Fehlers nicht vollständig ausgeschlossen werden.
- Das Teacher Modell war in diesem Fall das Ursprungsmodell. Ggf. sollten die Versuche mit einem Teacher Modell wiederholt werden, dass einen größeren Abstand zu dem Student Modell hat. Hier würde sich die Llama 3 70B Variante bzw. das Mistral Mixtral Modell anbieten.
- Mit dem DSKD Ansatz [35] wurde ein relativ neues Verfahren verwendet, mit dem wenig Erfahrungen vorliegen. Evt. gibt es in dem Verfahren selbst noch Fehler, die zu unerwarteten Ergebnissen führen.

In weiteren Experimenten sollten diese Punkte genauer überprüft, bzw. ein anderes unabhängiges KD Verfahren zu Verifizierung der Ergebnisse eingesetzt werden.

5.2.4 „Shortened LLM“ und AWQ

In einem weiteren Experiment wurden die mit „Shortened LLM“ erstellten Modelle mittels der AWQ Verfahrens quantisiert. Hier wurde also im Grunde der umgekehrte Ansatz zu 5.2.1 gewählt, bei dem ein quantisiertes Modell mit LoRA erneut trainiert wurde.

Die Ergebnisse dieses Experiments sind in den Tabellen 4.14, 4.12 und 4.13 in jeweils dem letzten Zeilenpaar dargestellt und werden mit den anderen Varianten verglichen.

Man erkennt, dass die zusätzliche Quantisierung (pruned + lora + awq) der Modelle nur einen geringen Einfluss auf die Benchmark Ergebnisse, im Vergleich zu den Ausgangsmodellen (pruned + lora) hat. Gleichzeitig wirkt der Speicherbedarf der Modelle aber erheblich verringert wie man in Abbildung 4.3 erkennen kann und die Inferenz Geschwindigkeit deutlich erhöht (Tabelle 4.18).

5.2.5 Fazit kombinierte Verfahren

Vergleicht man in den Tabellen 4.12, 4.14 und 4.13 die Ergebnisse der „pruned+lora+awq“ Modelle mit den Ursprungsmodellen (Referenz), erkennt man, dass im Falle der Llama Modelle nur eine sehr geringer Unterschied bei den Durchschnittswerten der Benchmarks besteht. Die Perplexity Werte für diese Modell Varianten sind etwas schlechter, befinden sich aber durchaus noch in einem akzeptablen Bereich.

Im Falle des Mistral Modells sind die Unterschiede in den Benchmarks etwas größer, was vor allem auf den ausbleibenden positiven Effekts des LoRA Retrainings (siehe oben) zurückzuführen ist. Dem gegenüber steht in allen drei Fällen eine um 70-85 % geringere GPU Speicher Auslastung

(siehe 4.3) und eine ca. 3,5 x höhere Inferenz Geschwindigkeit (siehe 4.18) der kombinierten Modelle.

In der Relation von Ressourcenersparnis zu Qualitätsverlust, ergibt sich damit ein durchaus positives Bild, für die mit kombinierten Verfahren erstellten Modelle. Der AWQ Quantisierung kommt dabei der größte Anteil der Ressourcenersparnis zu.

Für die beiden Llama Modelle zeigt die Kombination der Quantisierung mit einer geprunten und mittels LoRA erneut trainierten Variante, dass eine erhebliche Verringerungen der Speicherauslastung bei gleichzeitig ansteigender Inferenz Geschwindigkeit möglich sind, ohne die Modell Fähigkeiten wesentlich zu beschneiden.

Alternativ erhält man mit dem QLoRA Verfahren ebenfalls sehr gute Benchmark Ergebnisse, allerdings sind im Bereich der Inferenz Geschwindigkeit keine Vorteile gegenüber der Referenz zu erkennen. (siehe 4.17). Die Speicherauslastung des QLoRA Modells liegt bei dem Llama 3 Modell nur ca. 10% über dem des pruned+lora+awq Modells. Bei dem Mistral Modell ist sie ca. doppelt so hoch. Dieser Unterschied lässt sich nicht leicht erklären, vermutet wird aber, dass die Implementation des QLoRA Algorithmus nicht optimal auf die Mistral Modelle abgestimmt ist.

5.2.6 Fazit der Experimente

Schon mit den modernen Quantisierungsverfahren alleine erhält man sehr gute Ergebnisse, bei relativ geringem Aufwand. Durch eine Kombination von Verfahren, kann die Effizienz noch gesteigert werden, wobei aber der Aufwand für Durchführung und Evaluation der Ergebnisse erheblich ansteigt.

Im konkreten Anwendungsfall wird man anhand der eigenen Prioritäten entscheiden müssen, welchem Ansatz man den Vorrang einräumt.

5.3 Umfeld der Optimierungsverfahren

Im Vorfeld des Experimentalteils wurden eine Reihe von Ansätzen zu den einzelnen Technologien untersucht, um geeignete Verfahren für diese Arbeit auszuwählen. Dabei wurden einige Erfahrungen und Erkenntnisse gesammelt, welche für die Beurteilung der weiteren Entwicklung in diesem Bereich ggf. Relevanz besitzen und daher hier kurz erläutert werden sollen.

Entwicklungsgeschwindigkeit Im Bereich der verwendeten Technologien besteht ein sehr hohes Entwicklungstempo, welches die allgemeine Dynamik im Bereich der künstlichen Intelligenz reflektiert. Veröffentlichungen (und der zugehörige Programmcode), die vor mehr als ein bis zwei Jahren getätigten wurden, sind in vielen Fällen schon von neueren Arbeiten und Technologien überholt. Die Lernkurve im Bereich der Optimierung steigt steil an, was grundsätzlich begrüßenswert ist, aber auch dazu führt das es wenig standardisierte, gut etablierte Verfahren gibt.

So sind die meisten, die in dieser Arbeit verwendeten Verfahren nur wenige Monate alt und müssen, zumindest teilweise, als experimentell betrachtet werden. Spürbar wird dies häufig an notwendigen Nacharbeiten für eine sinnvolle Nutzung der Verfahren.

Dependency Hell Dieser noch aus der Zeit älterer Microsoft Windows Versionen stammende Begriff, beschreibt die Schwierigkeiten mit Versionen und Inkompatibilitäten zwischen Programmbibliotheken, Hardware und Treibern umzugehen. Obwohl durch die Anlage von isolierten Arbeitsumgebungen, mittels Werkzeugen wie Conda, die Möglichkeit besteht viele Abhängigkeiten zu entzerren, können Fälle entstehen die sich kaum auflösen lassen. Insbesondere wird dies durch die hohe Entwicklungsgeschwindigkeit bei den in 2.5 genannten Python Bibliotheken, in Abhängigkeit zu den Versionen der Hardware Treiber für die Nvidia Grafikkarten verursacht. Alle Verfahren sind sehr sensibel für suboptimale Konfigurationen der Evaluierungs bzw. Nutzungsumgebung. Schon minimale Abweichungen von der optimalen Kombination aus Bibliotheken/Treibern/Hardware, können zu drastischen Verschlechterungen der Ergebnisse führen.

Veröffentlichung von Technologien und Community-Support Die Veröffentlichung neuer oder verbesserter Technologien, erfolgt häufig zusammen mit dem Programmcode in einem GitHub Repository. Zum einen ist es damit möglich, die Ergebnisse der Veröffentlichung zu reproduzieren, zum andern aber auch die Technologie weiterzuentwickeln.

Häufig hängt das „Schicksal“ eines Verfahrens dabei an der Akzeptanz durch die Entwickler Community. Meist aufgrund von besonderem allgemeinen Interesse, entwickelt sich bei manchen Technologien eine Community, welche die Technologie fördert, aktualisiert und Unterstützung bietet. Ein gutes Beispiel hierfür ist, das auch in dieser Arbeit verwendete Benchmark-Framework `llm_evaluation_harness`, welches z.Zt. von ca. 190 Entwicklern betreut wird.

Bei sehr speziellen Technologien und daraus abgeleiteten Verfahren, bildet sich diese selbstorganisierende Struktur leider nicht aus, was oft auch für den Bereich der Ressourcen Optimierung gilt. Das Ergebnis sind schnell veraltende, nicht aktualisierte Verfahren, die sich kaum mehr sinnvoll nutzen lassen, bzw. einen zu hohen Aufwand für die Implementation erfordern. Insbesondere ist eine Kompatibilität mit neueren Modellen oft nicht gegeben.

In dieser Arbeit wurden daher im wesentlichen nur durch die Community gut unterstützte oder sehr neue Verfahren ausgewählt, damit die Implementationsaufwände handhabbar blieben.

5.3.1 Fazit zum Umfeld der Optimierungsverfahren

Die Dynamik im Bereich künstliche Intelligenz und die Verfügbarkeit offener Technologien, bietet Entwicklern und Forschern außergewöhnlich gute Möglichkeiten, an deren Weiterentwicklung teilzuhaben und diese mit zu gestalten. In zugehörigen Projekten sollten allerdings signifikante Aufwände für Verständnis, Recherche und Implementation eingeplant werden. Aufgrund der schnelllebigen Weiterentwicklung kommt, mehr noch als in anderen etablierteren IT-Disziplinen, diesen Aufgaben eine besondere Bedeutung zu, welche die Projektplanung reflektieren sollte.

5.4 Andere Verfahren zur Modellanpassung

Optimierungen von LLMs haben unterschiedliche Motivationen. Die hier vorgestellte Optimierungsexperimente in Richtung eines geringeren Ressourcenverbrauchs, sind aktuell nur ein kleinerer Bereich der Gesamtbelastungen zur Anpassung von LLMs.

Häufiger geht es darum, eine Verbesserung der LLMs für spezielle Themengebiete bzw. Aufgaben zu erreichen. Beispiele dafür wären, das zusammen fassen von Email Konversationen oder auch Preise, abhängig von der aktuellen Nachfrage, dynamisch zu verändern. Dazu werden i.d.R. Foundation Modelle durch „Finetuning“, „Retrival-Augmented Generation“ oder verwandte

Technologien so angepasst, dass sie diese Aufgaben besser erfüllen können. Die beiden genannten Verfahren werden hier kurz beschrieben, da sie z.Zt. besonders prominent eingesetzt werden.

Finetuning Für das „Finetuning“ von LLMs auf spezifische Inhalte und spezifisches Verhalten, gibt es eine Reihe von Verfahren [59]. Dabei gibt es auf der Technologie Ebene durchaus Überschneidungen zur Ressourcenoptimierung, da auch hier - neben anderen - LoRA und Knowledge Distillation als Verfahren eingesetzt werden. In Bereich des Finetunings erfolgt eine Anwendung dieser Technologien zusammen mit spezialisierten Datensammlungen, um die Kompetenz der LLMs für einen spezifischen Einsatzbereich zu optimieren.

Retrieval-Augmented Generation Neben diesen statischen Verfahren zur Modelloptimierung, wird mit „Retrieval-Augmented Generation (RAG)“ angestrebt, Modelle mit aktuellen Informationen zu versorgen, ohne ein Veränderung an den „weights“ des Modells vorzunehmen [60]. Grob gesagt verwendet RAG dazu Informationen aus einer externen Quelle, im Regelfall einer Datenbank, und assoziert diese mit der Query über die Nutzung von Embedding Ähnlichkeiten. Abbildung 5.1 ist der Veröffentlichung [60] entnommen und illustriert den Effekt dieser Technologie.

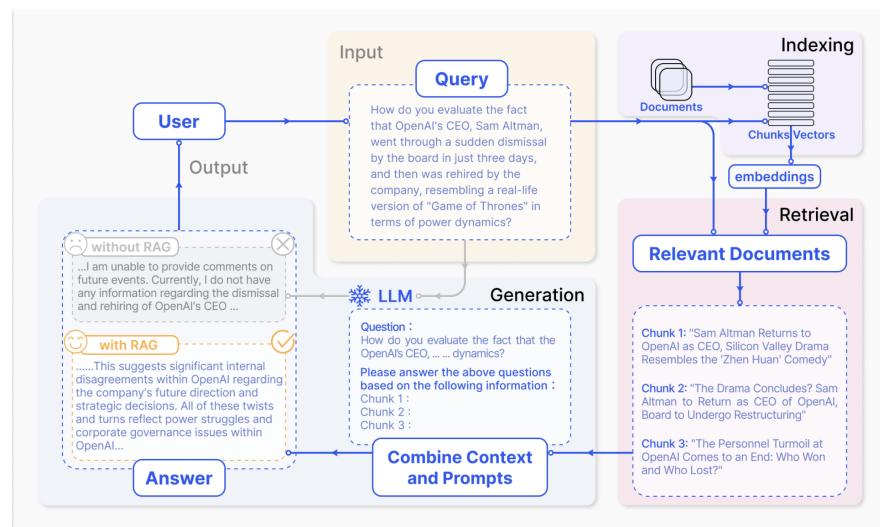


Abbildung 5.1: Darstellung des RAG Prozesses (siehe 5.4). Abbildung aus [60]

5.5 Zusammenfassung und Ausblick

5.5.1 Zusammenfassung

Die vorliegende Arbeit konnte zeigen, dass anhand aktueller Optimierungsverfahren zur Verschlankung von LLMs, ein erheblich verminderter Ressourcenbedarf, bei weitgehender Erhaltung der Fähigkeiten des Modells realisierbar ist. So konnte die GPU Speicherallokation in optimalen Konstellationen fast um 80 % verringert und die Inferenz Geschwindigkeit um den Faktor 3,5 erhöht werden. Am effektivsten zeigten sich Modelle, bei denen mehrere Verfahren kombiniert wurden, wobei der Quantisierung ein besonders hoher Anteil am Gesamteffekt zukommt. Wenig

zufriedenstellend war die Tatsache, dass sich mittels Knowledge Distillation keine positiven Effekte erreichen ließen.

5.5.2 Ausblick

Ressourcenschonende LLMs bieten die Möglichkeit neue Einsatzszenarien zu erschließen und die Abhängigkeit zu wenigen, großen Konzernen zu verringern. Ähnlich wie kleine, spezialisierte KIs schon seit längerem auf Smartphones für Fotooptimierung und Spracherkennung genutzt werden, könnten LLMs Einzug in die Welt von PCs, Smartphones, Fahrzeugen und Automaten nehmen, ohne eine permanente Online Verbindung zu benötigen.

Dabei könnte die Minimierung des Ressourcenverbrauchs in Kombination mit weiteren Optimierungen, z.Bsp. zur Spezialisierung auf ein Themengebiet, erfolgen. Der LoRA Schritt nach dem Pruning könnte mit einem geeigneten Datensatz durchgeführt und so eine LLM erzeugt werden, welches sowohl ressourcenschonend, als auch für den spezifischen Einsatzzweck optimal geeignet ist.

Eine weitere Möglichkeit könnte die Nutzung hybrider Architekturen sein, bei denen schlanke, lokale, offline LLMs primär genutzt werden und bei unbefriedigenden Ergebnissen ein kompetenteres online LLM hinzugenommen wird. So ein „Umschaltvorgang“ könnte für den Nutzer weitgehend transparent sein, in dem z.Bsp. die Perplexity der ersten Antwort Token bestimmt, und je nach Qualität, ein online LLM einbezogen wird.

Die Technologieentwicklung im Bereich der Optimierung ist z.Zt. sehr dynamisch, daher ist zu erwarten, dass sich die Methoden in den nächsten Jahren weiter verbessern werden und auch ihre Anwendung größeren Raum einnimmt. Auf diese Weise können zunehmend kostengünstige, ressourcenschonende, offlinefähige und differenzierte LLMs entstehen, die neben den großen, universellen Systemen einen wichtigen Mehrwert liefern.

Literatur

- [1] Wikipedia contributors. *History of natural language processing — Wikipedia, The Free Encyclopedia*. [Online; accessed 5-August-2024]. 2024.
- [2] Cameron R. Jones und Benjamin K. Bergen. „People cannot distinguish GPT-4 from a human in a Turing test“. In: (2024). arXiv: 2405.08007 [cs.HC]. URL: <https://arxiv.org/abs/2405.08007>.
- [3] Ashish Vaswani u. a. „Attention Is All You Need“. In: (2023). arXiv: 1706.03762 [cs.CL]. URL: <https://arxiv.org/abs/1706.03762>.
- [4] Matin Emrich. *Transfer Learning & BERT Transformer Funktionsweise*. 2020. URL: <https://www.alexanderthamm.com/de/blog/transfer-learning-bert-funktionsweise/>.
- [5] Jacob Devlin u. a. „BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding“. In: (2019). arXiv: 1810.04805 [cs.CL]. URL: <https://arxiv.org/abs/1810.04805>.
- [6] Xu Han u. a. „Pre-Trained Models: Past, Present and Future“. In: (2021). arXiv: 2106.07139 [cs.AI]. URL: <https://arxiv.org/abs/2106.07139>.
- [7] Cameron R. Wolfe. *Towards Data Science "Language Models: GPT and GPT-2"*. URL: <https://towardsdatascience.com/language-models-gpt-and-gpt-2-8bdb9867c50a>.
- [8] OpenAI. *Introducing ChatGPT*. 2022. URL: <https://openai.com/index/chatgpt/>.
- [9] OpenAI u. a. „GPT-4 Technical Report“. In: (2024). arXiv: 2303.08774 [cs.CL]. URL: <https://arxiv.org/abs/2303.08774>.
- [10] Wikipedia contributors. *Gemini (language model) — Wikipedia, The Free Encyclopedia*. [Online; accessed 5-August-2024]. 2024.
- [11] Albert Q. Jiang u. a. „Mistral 7B“. In: (2023). arXiv: 2310.06825 [cs.CL]. URL: <https://arxiv.org/abs/2310.06825>.
- [12] Thunder Said Energy. *Internet energy consumption*. <https://thundersaidenergy.com/downloads/internet-energy-consumption-data-models-forecasts/>. Accessed: 2024-09-04. 2023.
- [13] Sasha Luccioni, Yacine Jernite und Emma Strubell. *Power Hungry Processing: Watts Driving the Cost of AI Deployment?* Juni 2024. doi: 10.1145/3630106.3658542. URL: <http://dx.doi.org/10.1145/3630106.3658542>.
- [14] Mathew Oldham Kevin Lee Adi Gangidi. *Building Meta's GenAI Infrastructure*. 2024. URL: <https://engineering.fb.com/2024/03/12/data-center-engineering/building-metas-genai-infrastructure/>.
- [15] Ben Cottier u. a. „The rising costs of training frontier AI models“. In: (2024). arXiv: 2405.21015 [cs.CY].
- [16] Heise (emw). *Zuckerberg: Llama 4 braucht zehnmal mehr Rechenleistung als Vorgänger*. 2024. URL: <https://www.heise.de/news/Zuckerberg-Llama-4-braucht-zehnmal-mehr-Rechenleistung-als-Vorgaenger-9831779.html>.

- [17] Forschungszentrum Jülich. *JUPITER / Der Start von Exascale in Europa*. 2024. URL: <https://www.fz-juelich.de/de/ias/jsc/jupiter>.
- [18] Monika Gupta. *Google Tensor - der erste Prozessor von Google*. <https://blog.google/intl/de-de/produkte/hardware/google-tensor-details/>. Accessed: 2024-07-23. 2021.
- [19] Hongrong Cheng, Miao Zhang und Javen Qinfeng Shi. „A Survey on Deep Neural Network Pruning-Taxonomy, Comparison, Analysis, and Recommendations“. In: (2023). arXiv: 2308.06767 [cs.LG]. URL: <https://arxiv.org/abs/2308.06767>.
- [20] Amir Gholami u.a. *A Survey of Quantization Methods for Efficient Neural Network Inference*. 2021. arXiv: 2103.13630 [cs.CV]. URL: <https://arxiv.org/abs/2103.13630>.
- [21] Xiaohan Xu u.a. „A Survey on Knowledge Distillation of Large Language Models“. In: (2024). arXiv: 2402.13116 [cs.CL].
- [22] Edward J. Hu u.a. „LoRA: Low-Rank Adaptation of Large Language Models“. In: (2021). arXiv: 2106.09685 [cs.CL].
- [23] Matthieu Courbariaux, Yoshua Bengio und Jean-Pierre David. „Training deep neural networks with low precision multiplications“. In: (2015). arXiv: 1412.7024 [cs.LG]. URL: <https://arxiv.org/abs/1412.7024>.
- [24] Huggingface. *Huggingface - Quantization Overview*. 2024. URL: <https://huggingface.co/docs/transformers/main/quantization/overview#>.
- [25] Elias Frantar u.a. „GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers“. In: (2023). arXiv: 2210.17323 [cs.LG].
- [26] Ji Lin u.a. „AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration“. In: (2024). arXiv: 2306.00978 [cs.CL]. URL: <https://arxiv.org/abs/2306.00978>.
- [27] Ji Lin u.a. „AWQ: Activation-aware Weight Quantization for LLM Compression and Acceleration“. In: *MLSys*. 2024.
- [28] GitHub Contributors. *erg:shortened*. <https://github.com/erg:shortened/erg:shortened>. Accessed: 2024-06-14. 2023.
- [29] Stephen Merity u.a. *Pointer Sentinel Mixture Models*. 2016. arXiv: 1609.07843 [cs.CL].
- [30] Song Han, Huizi Mao und William J. Dally. „Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding“. In: (2016). arXiv: 1510.00149 [cs.CV]. URL: <https://arxiv.org/abs/1510.00149>.
- [31] Andrey Gromov u.a. *The Unreasonable Ineffectiveness of the Deeper Layers*. 2024. arXiv: 2403.17887 [cs.CL]. URL: <https://arxiv.org/abs/2403.17887>.
- [32] Andrey Gromov. *PruneME*. <https://github.com/arcee-ai/PruneMe>. Accessed: 2024-06-18. 2023.
- [33] Bo-Kyeong Kim u.a. „Shortened LLaMA: Depth Pruning for Large Language Models with Comparison of Retraining Methods“. In: (2024). arXiv: 2402.02834 [cs.LG]. URL: <https://arxiv.org/abs/2402.02834>.
- [34] Amit S. *Everything You Need To Know About Knowledge Distillation, aka Teacher-Student Model*. 2023. URL: <https://amit-s.medium.com/everything-you-need-to-know-about-knowledge-distillation-aka-teacher-student-model-d6ee10fe7276>.
- [35] Songming Zhang u.a. „Dual-Space Knowledge Distillation for Large Language Models“. In: *arXiv preprint arXiv:2406.17328* (2024).

- [36] Yangyang Xu u. a. „Parallel matrix factorization for low-rank tensor completion“. In: *Inverse Problems Imaging* 9.2 (2015), S. 601–624. ISSN: 1930-8345. DOI: 10.3934/ipi.2015.9.601. URL: <http://dx.doi.org/10.3934/ipi.2015.9.601>.
- [37] Sebastian Raschka. *Parameter-Efficient LLM Finetuning With Low-Rank Adaptation*. 2023. URL: <https://lightning.ai/pages/community/tutorial/lora-llm/>.
- [38] Tim Dettmers u. a. „QLoRA: Efficient Finetuning of Quantized LLMs“. In: (2023). arXiv: 2305.14314 [cs.LG]. URL: <https://arxiv.org/abs/2305.14314>.
- [39] AvisP@github.com. *LM-Finetune*. https://github.com/AvisP/LM_Finetune/blob/main/llama-3-finetune-qlora.ipynb. Accessed: 2024-06-22. 2023.
- [40] Ning Ding u. a. *Enhancing Chat Language Models by Scaling High-quality Instructional Conversations*. 2023. arXiv: 2305.14233 [cs.CL].
- [41] Thomas Wolf u. a. *HuggingFace’s Transformers: State-of-the-art Natural Language Processing*. 2020. arXiv: 1910.03771 [cs.CL]. URL: <https://arxiv.org/abs/1910.03771>.
- [42] Wikipedia contributors. *PyTorch — Wikipedia, The Free Encyclopedia*. [Online; accessed 25-August-2024]. 2024.
- [43] Hugging Face. *PEFT Documentation*. <https://huggingface.co/docs/peft>. Accessed: 2024-09-02. 2023.
- [44] Zishan Guo u. a. „Evaluating Large Language Models: A Comprehensive Survey“. In: *arXiv preprint arXiv:2310.19736* (2023).
- [45] Leo Gao u. a. *A framework for few-shot language model evaluation*. Version v0.4.0. Dez. 2023. DOI: 10.5281/zenodo.10256836. URL: <https://zenodo.org/records/10256836>.
- [46] Community. *Open LLM Leaderboard*. 2024. URL: <https://huggingface.co/open-llm-leaderboard>.
- [47] Keisuke Sakaguchi u. a. *WinoGrande: An Adversarial Winograd Schema Challenge at Scale*. 2019. arXiv: 1907.10641 [cs.CL]. URL: <https://arxiv.org/abs/1907.10641>.
- [48] François Chollet. *On the Measure of Intelligence*. 2019. arXiv: 1911.01547 [cs.AI]. URL: <https://arxiv.org/abs/1911.01547>.
- [49] Stephanie Lin, Jacob Hilton und Owain Evans. *TruthfulQA: Measuring How Models Mimic Human Falsehoods*. 2022. arXiv: 2109.07958 [cs.CL]. URL: <https://arxiv.org/abs/2109.07958>.
- [50] Rowan Zellers u. a. *HellaSwag: Can a Machine Really Finish Your Sentence?* 2019. arXiv: 1905.07830 [cs.CL]. URL: <https://arxiv.org/abs/1905.07830>.
- [51] Yutong Hu u. a. *Can Perplexity Reflect Large Language Model’s Ability in Long Text Understanding?* 2024. arXiv: 2405.06105 [cs.CL]. URL: <https://arxiv.org/abs/2405.06105>.
- [52] Hugo Touvron u. a. „Llama 2: Open Foundation and Fine-Tuned Chat Models“. In: (2023). arXiv: 2307.09288 [cs.CL].
- [53] Hugo Touvron u. a. *LLaMA: Open and Efficient Foundation Language Models*. 2023. arXiv: 2302.13971 [cs.CL]. URL: <https://arxiv.org/abs/2302.13971>.
- [54] Joshua Ainslie u. a. *GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints*. 2023. arXiv: 2305.13245 [cs.CL]. URL: <https://arxiv.org/abs/2305.13245>.
- [55] Meta. *Introducing Meta Llama 3: The most capable openly available LLM to date*. 2024. URL: <https://ai.meta.com/blog/meta-llama-3/>.

- [56] Thomas Schmitt. *Jupyter Notebooks Masterarbeit*. <https://github.com/fhswf/Slim/tree/main/notebooks>. Accessed: 2024-09-03. 2024.
- [57] Bo-Kyeong Kim u. a. *Shortened LLM*. 2024. URL: <https://github.com/Nota-NetsPresso/shortened-llm>.
- [58] Songming Zhang u. a. *Dual Space Knowledge Distillation - Github Repository*. 2024. URL: <https://github.com/songmzhang/DSKD>.
- [59] Shengyu Zhang u. a. *Instruction Tuning for Large Language Models: A Survey*. 2024. arXiv: 2308.10792 [cs.CL]. URL: <https://arxiv.org/abs/2308.10792>.
- [60] Yunfan Gao u. a. *Retrieval-Augmented Generation for Large Language Models: A Survey*. 2024. arXiv: 2312.10997 [cs.CL]. URL: <https://arxiv.org/abs/2312.10997>.

Abbildungsverzeichnis

1.1	Beispiel einer ELIZA Konversation [1].	1
1.2	Schematische Darstellung der Transformer Architektur mit einem Beispiel [4]	2
1.3	Schematische Darstellung einer Decoder Architektur [7].	3
1.4	Internet energy consumption. [12].	5
1.5	Kostenanteile bei der Entwicklung ausgewählter KIs [15]	6
1.6	Kostenentwicklung neue KI Modelle [15]	7
2.1	Speicherbedarf von Floating Point und Integer Variablen [23]	11
2.2	Vergleich der Quantisierungsmethoden, wie im Text beschrieben [26] (a) Zeigt eine einfache Quantisierung mittels eines Rundungsverfahrens (RTN quantization), während (b) „salient weights“ im „Mixed precision“ Ansatz und (c) den AWQ Ansatz darstellt. Der Perplexity Wert (PPL) zeigt die Qualität, der auf diese Weise generierten Modelle.	13
2.3	Strukturiertes und Unstrukturiertes Pruning. Die linke Grafik zeigt wie komplette Strukturbereiche aus dem Netzwerk genommen werden. Die übrigen Strukturen und ihre Beziehungen bleiben unverändert erhalten. Beim unstrukturierten Pruning hingegen wird ein Großteil der Beziehungen verschlankt, während die Struktur vollständig erhalten bleibt.	15
2.4	Schematische Darstellung von Knowledge Distillation. [34].	16
2.5	Schematische Darstellung der response based Knowledge Distillation. [34].	17
2.6	Zeigt die Dekomposition einer höherrangigen 3×3 Matrize in zwei niederrangigere 3×1 bzw. 1×3 Matrizen.	18
2.7	Zeigt die Kombination des ursprünglichen Modells W mit den niederrangigeren Matrizen W_A und W_B , welche die neu erlernten Gewichte enthalten [37].	18
2.8	Darstellung der Kriterien und Metriken nach (Guo et al.)[44]. Neben den im Text behandelten drei Hauptbereichen sind noch Metriken für spezielle LLMs (dunkleres blau) angegeben.	21
2.9	Erstellung des Chat Modells aus dem Grundmodell [52]. Erklärung siehe Text.	23
4.1	Auslastung des GPU Speichers durch die jeweiligen Modelle auf einer Nvidia A100 80GB GPU. Y-Achse zeigt die Speicherauslastung in Gigabyte.	52
4.2	Auslastung des GPU Speichers einer Nvidia A100 80GB GPU durch Modelle, die mit dem PruneMe Verfahren verkleinert wurden.	55
4.3	Speicherbedarf der Modelle auf einer Nvidia A100 80GB GPU. Aus dem jeweiligen Referenz Modell wurden alle weiteren Modelle abgeleitet. Die X-Achse zeigt die verwendeten Verfahren.	62
4.4	Beispielhafte Inferenz eines optimierten Modells auf einem älteren Desktop Computer.	63
5.1	Darstellung des RAG Prozesses (siehe 5.4). Abbildung aus [60]	71

Tabellenverzeichnis

4.1	Übersicht über die im Kapitel Ergebnisse enthaltenen Verfahren und Modelle	51
4.2	Die Tabelle zeigt die quantisierten Modellen zusammen mit den korrespondierenden Referenzen. Für die Quantisierung wurde das Verfahren AutoAWQ (Groupsize 128) verwendet.	51
4.3	Vergleich des Mistral 7B Instruct Modells mit unterschiedlicher Parametrierung des Groupsize Parameters bei der AWQ Quantisierung. Der Groupsize Parameter wurde auf die Werte 64,128 und 256 gesetzt.	52
4.4	Die Tabelle zeigt einen Vergleich von Modellen die mit zwei unterschiedlichen Verfahren, GPTQ und AWQ, quantisiert wurden	53
4.5	Die Tabelle zeigt einen Vergleich von quantisierten Modellen, die mit unterschiedlichen Bitbreiten quantisiert wurden. Die Quantisierung erfolgte mit dem GPTQ Verfahren für das Modell Mistral 7B Instruct v0.2.	53
4.6	Die Tabelle zeigt einen Vergleich von quantisierten Modellen, die mit unterschiedlichen Bitbreiten quantisiert wurden. Die Quantisierung erfolgte mit dem GPTQ Verfahren für das Modell Llama 3 8B Instruct.	54
4.7	Inferenz Geschwindigkeit und Speicherallokation von GPTQ Modellen mit verschiedenen Bitbreiten.	54
4.8	In der Tabelle sind die Benchmark Ergebnisse für Modelle aufgeführt, bei denen 8 oder 12 Schichten mittels Pruning entfernt wurden.	54
4.9	In der Tabelle sind die Benchmark Ergebnisse für die QLoRA Modelle, im Vergleich zur Referenz aufgeführt.	56
4.10	Die Tabelle zeigt die Ergebnisse der Experimente mit dem Shortened-LLM Verfahren.	56
4.11	Optimierungsversuch mit dem Knowledge Distillation Verfahren, bei Modellen die zuvor mit pruning verkleinert und mit LoRA retraniert wurden.	57
4.12	Die Tabelle zeigt die Benchmark Ergebnisse der verschiedenen Llama 3 8B Instruct Modelle im Vergleich.	58
4.13	Die Tabelle zeigt die Benchmark Ergebnisse der verschiedenen Mistral 7B Instruct Modelle im Vergleich.	58
4.14	Die Tabelle zeigt die Benchmark Ergebnisse der verschiedenen Llama 2 7B chat Modelle im Vergleich.	58
4.15	Perplexity Messungen für die verschiedenen Varianten des Llama 3 8B Instruct Modells. Bei der Pruned Variante wurden sechs Schichten entfernt. Bei GPTQ und AWQ handelt es sich um 4-bit Quantisierungen.	59
4.16	Perplexity Messungen für die verschiedenen Varianten des Mistral 7B Instruct v0.2 Modells. Bei der Pruned Variante wurden sechs Schichten entfernt. Bei GPTQ und AWQ handelt es sich um 4-bit Quantisierungen.	59
4.17	Inferenz Performance von nativen zu bearbeiteten Modellen. Die mit einem * gekennzeichneten Modell können aufgrund der fehlenden AWQ Unterstützung der Cuda 11.7 Umgebung, nicht ihre volle Leistungsfähigkeit entfalten. Im Falle von Llama 2 wurde für das GPTQ Modell kein Wert bestimmt. Verwendet wurde hier eine Nvidia A100 GPU mit 80GB Speicher.	60

4.18 Inferenz Performance von nativen zu AWQ quantisierten Modellen. Die Messungen wurden in einer Google Colab Umgebung mit einer Nvidia T4 15 GB durchgeführt. Hier konnten Cuda 12.x Treiber genutzt werden, die eine Verwendung des AutoAWQ python Moduls ermöglichen.	60
4.19 Antworten verschiedener Varianten des LLama 3 8B Instruct Modells auf die Frage: „Why should I eat healthy?“	61
4.20 Reaktionen verschiedener Varianten des Mistral 7B Instruct Modells auf die Aussage: „Once upon a time“	61
4.21 Speicherbedarf der meisten Modellvarianten im Vergleich. Angegeben ist jeweils die Belegung des GPU Speichers in Gigabyte (Nvidia A100 80GB). In zwei Fällen konnte für das Llama 2 Modell kein Wert ermittelt werden.	62

Erklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und dabei keine anderen als die angegebenen Hilfsmittel benutzt habe. Sämtliche Stellen der Arbeit, die im Wortlaut oder dem Sinn nach Werken anderer Autoren entnommen sind, habe ich als solche kenntlich gemacht. Die Arbeit wurde bisher weder gesamt noch in Teilen einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

26. September 2024



Dr. Thomas Schmitt