

Bachelorarbeit

Optimierung der ABAP-Code-Generierung eines Large Language Models durch Fine-Tuning am Beispiel von LLaMA3

Zur Erlangung des Grades Bachelor of Science

Vorgelegt an der Fakultät für Informatik und Ingenieurwissenschaften
der Technischen Hochschule Köln
im Studiengang Wirtschaftsinformatik

Vorgelegt von: Tim Köhne
Matrikel-Nr.: xxxxxxxx
Adresse: xxxxxxxx
xxxxxxx
E-Mail: xxxxxxxx

Erstprüfer: Prof. Dr. Westenberger
Zweitprüfer: Prof. Dr. Victor

xxxxxxx, 19. August 2024

Inhaltsverzeichnis

Abbildungsverzeichnis	II
1 Einführung	1
1.1 Überblick über ABAP	1
1.2 Überblick über Llama3	1
1.3 Motivation zur Nutzung von Llama3 zur ABAP-Code-Generation	1
1.4 Zielsetzung dieser Arbeit	2
2 Hintergrund	3
2.1 Funktionsweise von LLMs	3
2.1.1 Tokenisierung	3
2.1.2 Embedding	4
2.1.3 Transformation	7
2.1.4 Un-Embedding	9
2.2 Trainingsprozess von LLMs	9
2.2.1 Pre-Training	9
2.2.2 Fine-Tuning	12
2.2.3 Fine-Tuning Varianten	13
2.3 Betrachtung vorheriger Versuche der LLM Programmcode Generierung . . .	15
3 Methodik	17
3.1 Entwicklung der Auswertungsverfahren	17
3.2 Vorbereitung des Fine-Tuning Prozesses	18
3.3 Untersuchung der Hyperparameter-Einstellungen	19
3.4 Entwicklung des Fine-Tuning Datensatzes	22
3.5 Inferenz mit den Modellen	24
4 Durchführung des Fine-Tunings	25
4.1 Optimierung	25
4.2 Datensatz DS3	26
4.3 Datensatz DS2	29
4.4 Datensatz DS1	33
4.5 Untersuchung der Ergebnisse	35
5 Fazit und Ausblick	39
5.1 Fazit	39
5.2 Diskussion möglicher Limitationen und zukünftiger Forschungsrichtungen .	40
Literaturverzeichnis	41
Anhang	44
Eigenständigkeitserklärung	46

Abbildungsverzeichnis

1	Beispielhafte Tokenisierungsverfahren	4
2	Beispiel für One-Hot Encoding als bekanntes Embedding-Verfahren	5
3	Beispielhafte Darstellung von syntaktischen Beziehung von Vektoren nach dem Embedding [Mik+13]	6
4	Beispielhafte Darstellung von semantischen Beziehung von Vektoren nach dem Embedding. [Mik+13]	6
5	Schematische Darstellung der Funktionsweise von LoRA [Hu+21; Hug24] . .	14
6	Vorlage zur Formatierung der Fine-Tuning Daten [Tao+23]	22
7	Beispielhafte Anfrage an das Metas Llama3.1:8b-text-q4_K_M	25
8	Beispielhafte Anfrage an Metas Llama3.1:8b-instruct-q4_K_M	26
9	Beispielhafte Anfrage an das Modell ds3-v1-llama3.1:8B-q4_K_M	26
10	Beispielhafte Anfrage an das Modell ds3-v2-llama3.1:8B-q4_K_M	27
11	Beispielhafte Anfrage an das Modell ds3-v3-llama3.1:8B-q4_K_M	27
12	Darstellung vom Loss der verschiedenen Modellversionen beim Fine-Tuning mit Datensatz DS3	28
13	Beispielhafte Anfrage an das Modell ds2-v1-llama3.1:8B-q4_K_M	29
14	Beispielhafte Anfrage an das Modell ds2-v1-llama3.1:8B-q4_K_M	30
15	Darstellung vom Verlust der verschiedenen Modellversionen beim Fine-Tuning mit Datensatz DS2	30
16	Beispielhafte Anfrage an das Modell ds2-v5-llama3.1:8B-q4_K_M	31
17	Darstellung vom Loss von Version eins und Version sechs beim Fine-Tuning mit Datensatz DS2	32
18	Beispielhafte Anfrage an das Modell ds2-v6-llama3.1:8B-q4_K_M	32
19	Beispielhafte Anfrage an das Modell ds1-v1-llama3.1:8B-q4_K_M	33
20	Darstellung des Loss im Training von den letzten Modellversionen jedes Datensatzes	34
21	Benchmarkergebnis: Funktionserstellungen von Metas Llama3.1:8B-Instruct	35
22	Benchmarkergebnis: Funktionsaufrufe von Metas Llama3.1:8B-Instruct . . .	35
23	Benchmarkergebnis: Funktionserstellungen von DS1-v1	36
24	Benchmarkergebnis: Funktionsaufrufe von DS1-1	36
25	Benchmarkergebnis: Funktionserstellungen von DS2-v6	37
26	Benchmarkergebnis: Funktionsaufrufe von DS2-6	37
27	Benchmarkergebnis: Funktionserstellungen von DS3-v3	38
28	Benchmarkergebnis: Funktionsaufrufe von DS3-3	38

1 Einführung

1.1 Überblick über ABAP

ABAP (Advanced Business Application Programming) ist eine proprietäre Programmiersprache von SAP die erstmals 1983 veröffentlicht wurde und zur Programmierung von Anwendungen im SAP-Umfeld dient.

Programme liegen als Quellcode in der SAP Datenbank vor, diese können zu einem Bytecode kompiliert werden, welcher auch innerhalb der SAP Datenbank abgelegt wird. Dieser Bytecode kann dann von der ABAP-Laufzeitumgebung innerhalb der SAP GUI interpretiert werden. Der Programmierer greift über die SAP GUI auf die ABAP Workbench zu, diese dient als Code-Editor. Die Programmiersprache bietet typische Variablen, Datentypen, Operationen und Kontrollstrukturen wie If-Then-Else, Do-While und Case-Anweisungen. Außerdem gibt es Funktionsbausteine und Standardfunktionsbausteine die häufige Aufgaben der Datenverarbeitung optimieren. Seit 1999 unterstützt ABAP auch objektorientierte Entwicklung durch hinzufügen von Klassen, Objekten und Vererbung. Eine der wichtigsten Funktionen von ABAP ist die Datenbankinteraktion, hier bietet ABAP mit OpenSQL eine Abstraktionsebene um sämtliche Datenbankinteraktionen unabhängig von der Datenbankplattform zu machen, und Zugriffe zu optimieren. So werden unter anderem Puffer eingesetzt um Zugriffszeiten zu verbessern, Daten werden dynamisch zwischen Arbeitsspeicher und Massenspeicher verschoben um die Zugriffszeit und die Speichermenge abzustimmen und es wird Online-Transaction-Processing eingesetzt um vielen Nutzern gleichzeitig Zugriff zu ermöglichen, ohne dass es zu Konflikten kommt.

1.2 Überblick über Llama3

Llama3 ist eine Gruppe von Large Language Models (LLMs) von Meta AI welche im April 2024 erstmals veröffentlicht wurde. Es wurden erst Modelle mit acht Milliarden (8B) und 70 Milliarden (70B) Parametern veröffentlicht, später dann auch eines mit 405 Milliarden (405B). Die Llama3 Modelle sind derzeit sehr beliebt da sie sehr gute Ergebnisse erzielen, lokal ausgeführt werden können und ihre Lizenz Entwicklern viele Freiheiten bietet.

Die Trainingsdaten von Llama3 umfassen 15 Billionen (15T) Tokens welche, nach Meta, aus öffentlich verfügbaren Quellen stammen. Ungefähr 5% der Trainingsdaten sind hochqualitative nicht englischsprachige Trainingsdaten. Zur Erstellung der Trainingsdaten wurden verschiedene Filterungsprozesse eingesetzt um eine gute Qualität zu gewährleisten, so wurden unter anderem heuristische Filter und semantische Deduplikation genutzt, zusätzlich wurde auch Llama2 eingesetzt um die Textqualität von potenziellen Trainingsdaten zu klassifizieren. [Dub+24]

1.3 Motivation zur Nutzung von Llama3 zur ABAP-Code-Generation

Die Automatisierung spielt seit Jahren eine zunehmend wichtige Rolle in der Effizienz- und Produktivitätssteigerung der Softwareentwicklung. Insbesondere die jüngsten Fortschritte im Bereich der Large Language Models (LLMs), wie ChatGPT und Llama, haben gezeigt, dass diese Modelle leistungsfähige Werkzeuge für die automatische Code-Generierung in

weit verbreiteten Programmiersprachen wie Python und JavaScript sind. Im Gegensatz dazu bleibt die Automatisierung in proprietären Sprachen wie ABAP, die eine zentrale Rolle bei der Entwicklung von SAP-ERP-Systemen spielen, eine Herausforderung. Die Generierung von hochwertigem ABAP-Code erfordert tiefes Fachwissen und umfangreiche Programmiererfahrung, was aktuelle LLMs vor erhebliche Schwierigkeiten stellt. Diese Herausforderungen sind bislang kaum erforscht, wie die geringe Anzahl relevanter Publikationen vermuten lässt. Eine mögliche Erklärung könnte sein, dass die existierenden LLMs überwiegend auf allgemeinen Datensätzen trainiert wurden und daher spezialisiertes Wissen für Nischenbereiche wie ABAP fehlt. Vor diesem Hintergrund zielt diese Untersuchung darauf ab, das Sprachmodell Llama3 durch gezieltes Fine-Tuning zu optimieren, um dessen Fähigkeit zur ABAP-Code-Generierung zu verbessern.

1.4 Zielsetzung dieser Arbeit

Im Rahmen dieser Untersuchung sollen folgende Forschungsfragen beantwortet werden:

- RQ1: Mit welchen Metriken kann die Effizienz und Genauigkeit eines generierten ABAP-Codes sinnvoll und praxisnah gemessen werden?
- RQ2: Wie kann das Fine Tuning die Effizienz und Genauigkeit der ABAP-Code-Generierung durch das Llama3-Modell verbessern?
- RQ3: In welchem Maße ist der Einsatz des spezialisierten Llama3-Modells als alltägliches Hilfsmittel für ABAP-Entwickler sinnvoll und praktikabel?

2 Hintergrund

2.1 Funktionsweise von LLMs

Large Language Models sind künstliche neuronale Netze mit sehr vielen Parametern die mit Deep-Learning-Techniken durch selbst-überwachtes Lernen anhand von riesigen Mengen von Trainingsdaten trainiert werden. Sie werden eingesetzt um die menschliche Sprache zu verstehen und Text zu generieren, worin sie sich als besser als alle bisherigen Methoden erwiesen haben. Sie haben bekannte Tests wie den Turing-Test bestanden und schneiden auch in vielen Prüfungen, Tests und Examen über verschiedene Bereiche wie Recht, Sprachen, Mathematik, Naturwissenschaften sowie Wirtschaftswissenschaften und vielen anderen besser ab als der durchschnittliche menschliche Prüfling. [Ope+24]

2.1.1 Tokenisierung

Die Tokenisierung dient dazu Text in sogenannte Tokens zu überführen, Tokens sind kleinere Texteinheiten, welche es dem LLM in späteren Schnitten erlauben Muster zu finden. Es gibt viele Möglichkeiten wie Text in Tokens aufgeteilt werden kann, einfachste Verfahren sind beispielsweise die Aufteilung des Textes in Wörter oder Buchstaben, jedes Verfahren hat Vor- und Nachteile.

Die Aufteilung in Wörter ist sinnvoll wenn man die semantischen Informationen die im Token enthalten sind betrachtet, anhand der das LLM später lernen soll. Wörter enthalten viel mehr semantische Informationen als Buchstaben (beziehungsweise Zeichen in einem Zeichensatz wie ASCII), dementsprechend könnte das LLM aus Wörtern mehr lernen als aus Zeichen. Die Aufteilung in Wörter ist auch von Vorteil für die Kontextmenge des LLMs, diese ist üblicherweise begrenzt und mit Wörtern kann diese mehr Informationen enthalten als mit Zeichen. Außerdem sind zusätzliche Tokens mit zusätzlicher Rechenleistung, und damit zusätzlichen Kosten, verbunden.

Die Aufteilung in Zeichen ist sinnvoll wenn man die Menge der möglichen Tokens (das Vokabular) betrachtet, in einem Zeichensatz wie ASCII würden genau 128 Token benötigt werden um sämtlichen Text abzubilden. Bei der Aufteilung in Wörter müsste das Vokabular um einen Faktor von ungefähr 4000 größer sein.¹ Dann können jedoch immer noch Probleme mit Wörtern entstehen, die bei der Erstellung des LLMs nicht beachtet wurden, beispielsweise durch neue Wörter oder sehr seltene Wörter. In Sprachen wie Deutsch ist es außerdem nahezu unmöglich alle Wörter abzubilden, da durch zusammengesetzte Wörter eine nahezu unbegrenzte Menge gebildet werden kann.

Moderne LLMs nutzen einen hybriden Ansatz indem Teilworte als Tokens eingesetzt werden um die Vorteile zu maximieren und die Nachteile zu minimieren. Tiktoken² ist ein beliebter Open-Source Tokenizer von OpenAI der von vielen LLMs genutzt oder erweitert wird. Dieser nutzt Byte-Pair Encoding um Text in Tokens zu übersetzen, dabei wird ein repräsentativer Text (oder der gesamte Textkorpus des LLMs) zu Grunde gelegt. Zu Beginn wird jedem Zeichen ein Token zugewiesen. Daraufhin wird iterativ die häufigste aufeinanderfolgende Kombination von zwei Tokens zu einem neuen Token zusammenge-

¹Das Oxford English Dictionary (<https://www.oed.com/>) enthält mehr als 500000 Wörter.

²<https://github.com/openai/tiktoken>

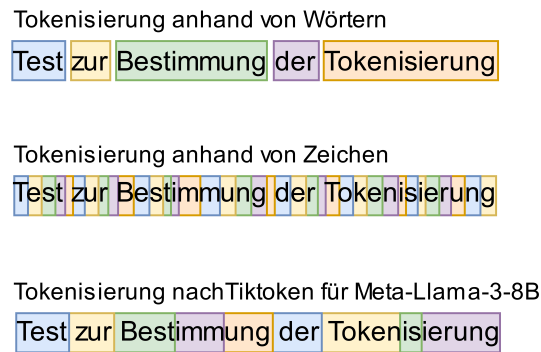


Abbildung 1 Beispielhafte Tokenisierungsverfahren

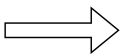
fasst, welches dann die ursprünglichen Tokens ersetzt. Dieser Vorgang kann durchgeführt werden bis eine gewünschte Vokabulargröße erreicht ist und erlaubt so genaue Kontrolle. Außerdem können auf diese Art neue Wörter immer auf Tokens abgebildet werden, da der gesamte Zeichensatz als einzelne Tokens zur Verfügung steht. Nachdem die Standard-Tokens bearbeitet sind werden einige besondere Tokens hinzugefügt, beispielsweise um den Start und das Ende eines Textabschnitts anzugeben. Abschließend werden allen Tokens einzigartige Token-IDs zugewiesen um diese numerisch zu identifizieren.

Ein Vergleich dieser drei Tokenisierungsverfahren ist in Abbildung 1 dargestellt. Die Tokenisierung anhand von Wörtern führt hier zu fünf Tokens, Tokenisierung anhand von Zeichen führt zu 37 Tokens und die Tokenisierung nach Tiktoken für die Meta-Llama-3 Familie führt zu neun Tokens. Bei Tiktoken fällt auf, dass häufige Worte und Teilworte zu einem Token zusammengefasst sind, und seltenere aus mehreren Tokens aufgebaut sind. Tokens wie “Test”, “Best” und “Token” sind häufig da sie auch im Englischen vorkommen und der Meta-Llama-3 Datensatz größtenteils aus englischen Text besteht. Die Endung “ung” ist eine der häufigsten Wortendungen in der deutschen Sprache, was vermutlich der Grund dafür ist, dass diese als Token zusammengefasst ist.

2.1.2 Embedding

Das bekannteste Beispiel für ein Embedding-Verfahren ist das One-Hot Encoding welches in vielen Machine-Learning Projekten genutzt wird. Dies ist oft nötig, da kategorische Parameter in numerische Werte umgewandelt werden müssen damit mit ihnen gearbeitet werden kann. Ohne One-Hot Encoding werden die einzelnen Kategorien einfach durchnummeriert, was zu Problemen führen kann, da die Daten dann eine Rangfolge enthalten, welche vorher nicht zwingend vorhanden war. Bei One-Hot Encoding wird die ursprüngliche Spalte verworfen und stattdessen für jede Kategorie eine zusätzliche Spalte hinzugefügt, die neuen Spalten werden mit Nullen aufgefüllt außer an der Stelle, wo die Kategorie übereinstimmt, hier wird eine Eins eingetragen. Dies verhindert die versehentliche Implementation einer Rangfolge, dafür müssen aber genauso viele neue Spalten hinzugefügt werden wie es Kategorien gibt was den Datensatz eventuell erheblich vergrößert.

In Abbildung 2 wird dargestellt wie der kategorische Parameter “Wohnort” durch One-Hot Encoding in drei numerische Parameter überführt wird. An diesem einfachen Beispiel ist bereits zu erkennen wie schnell die Datenmenge ansteigt, hier wurden aus drei Einträgen



Name	Wohnort
Max Mustermann	Köln
Erika Mustermann	Berlin
John Doe	New York City

Name	Wohnort-Köln	Wohnort-Berlin	Wohnort-NYC
Max Mustermann	1	0	0
Erika Mustermann	0	1	0
John Doe	0	0	1

Abbildung 2 Beispiel für One-Hot Encoding als bekanntes Embedding-Verfahren

mit einem kategorischen Parameter, drei Einträge mit jeweils drei numerischen Parametern. Die Menge der Datenfelder hat sich also von drei auf neun erhöht, allgemein gilt

$$\text{Anzahl Datenfelder} = \text{Anzahl Einträge} \cdot \text{Anzahl Kategorien}$$

One-Hot Encoding eignet sich nicht wenn die Anzahl an Kategorien sehr hoch ist, da da die Datenmenge steigt und das Training schwieriger wird und länger dauert. Außerdem sorgt One-Hot Encoding für eine geringe Datendichte, was, wenn nicht genügend Trainingsdaten vorhanden sind, schnell zu Overfitting (Überanpassung) des Machine Learning Modells führt. Im Bereich der natürlichen Sprachverarbeitung wie mit LLMs wird One-Hot Encoding aus diesen Gründen nicht genutzt.

Stattdessen benutzen LLMs üblicherweise semantische Enkodierungstechniken, hier werden Neuronale Netze genutzt um die Token-IDs in eine reelle Vektorrepräsentationen zu überführen, welche die semantische Bedeutung der Tokens im Vektorraum darstellt. Die entstehenden Vektoren haben eine hohe Datendichte, eine festlegbare Anzahl an Dimensionen und können dementsprechend viele verschiedene sprachliche und semantische Merkmale des Tokens als auch Position und Beziehung zu anderen Tokens einfangen. Die hohe Anzahl an Dimension im Vektorraum erlaubt es viele Merkmale abzubilden und jedes Token entspricht einem Vektor innerhalb des Vektorraums. Die Distanz zwischen zwei Vektoren im Vektorraum entspricht also der Ähnlichkeit die Tokens zueinander haben, basierend auf den gelernten Merkmalen. [GB16]

Diese Ähnlichkeit ist beispielhaft in Abbildungen 3a, 3b, 4a und 4b dargestellt, welche anhand der Ergebnisse von [Mik+13] zeigen, dass diese Embedding Methode es ermöglicht syntaktische und semantische Beziehungen zwischen Wörtern zu finden. Diese Darstellungen vereinfachen das grundlegende Konzepts, sie zeigen Embedding-Beziehungen in einem dreidimensionalen Raum wohingegen diese Untersuchung in bis zu 600 Dimensionen stattgefunden hat. Moderne LLMs nutzen mehrere Tausend Dimensionen für diese Aufgabe.[Bro+20; Dub+24]

In Abbildung 3a wird vereinfacht die syntaktische Beziehung von Wörtern zwischen der Gegenwartsform und der Vergangenheitsform dargestellt. So zeigt das Embedding, dass die Wörter “walking” und “walked” in der gleichen Beziehung zueinander stehen wie die Wörter “swimming” und “swam”. Abbildung 3b zeigt die syntaktische Beziehung zwischen Wörtern und ihrem Gegenteil. So zeigt das Embedding, dass die Wörter “possibly” und “impossibly” in der gleichen Beziehung zueinander stehen wie die Wörter “ethical” und “unethical”. In Abbildung 4a wird vereinfacht dargestellt, dass die Embeddings der Wörter “Woman” und “Man” im gleichen Verhältnis zueinander stehen wie die Embeddings der

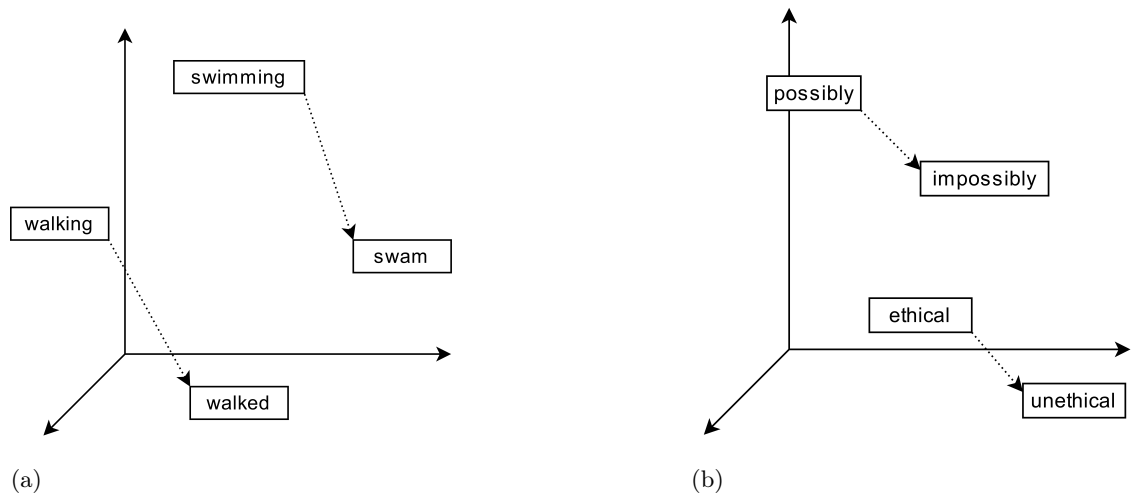


Abbildung 3 Beispielhafte Darstellung von syntaktischen Beziehung von Vektoren nach dem Embedding [Mik+13]

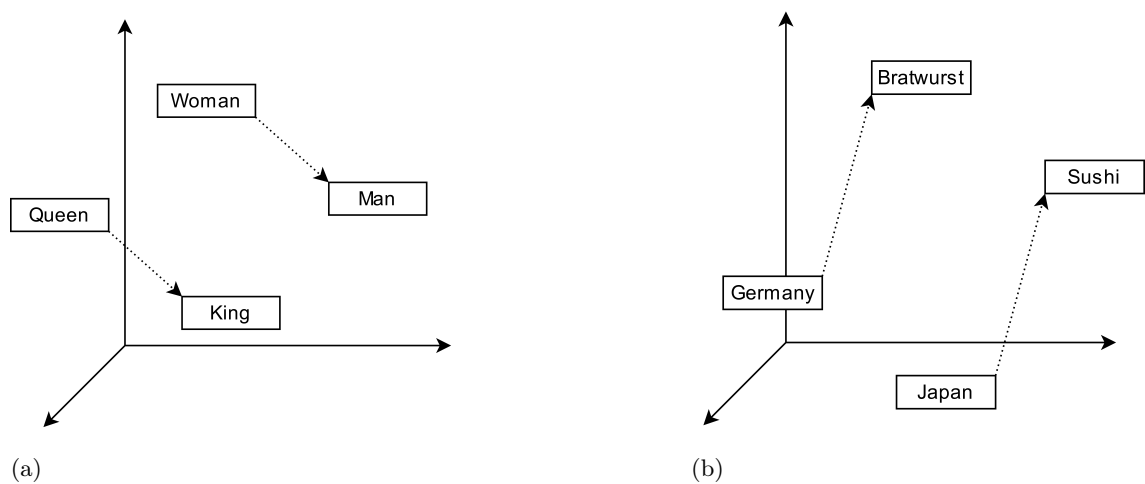


Abbildung 4 Beispielhafte Darstellung von semantischen Beziehung von Vektoren nach dem Embedding. [Mik+13]

Wörter “Queen” und “King”. Dieses Beispiel ist bekannt dafür, dass es zeigt, dass Vektorarithmetik wie

$$\text{vector}(\text{“King”}) - \text{vector}(\text{“Man”}) + \text{vector}(\text{“Woman”}) \approx \text{vector}(\text{“Queen”})$$

mit Embedding möglich ist und auch semantische Zusammenhänge approximieren kann. Abbildung 4b zeigt nach dem gleichen Prinzip das die Beziehung der Embeddings von “Germany” zu “Bratwurst” im gleichen Verhältnis zueinander stehen wie die Embeddings von “Japan” und “Sushi”.

In LLMs wird das ganze durch Embedding Layer umgesetzt, welche am Anfang des Neuralen Netzes durchlaufen werden und die Tokens in ihre entsprechenden Vektoren transformieren. Dies geschieht durch eine Matrixmultiplikation mit der Embedding Matrix, deren Spalten die Embeddings von jedem Token des Vokabulars sind. Llama3 hat ein Vokabular von 128256 Tokens und eine Embedding-Vektorgröße von 4096 bei der 8B Variante, 8192

bei der 70B Variante und 16384 bei der 405B Variante was dafür sorgt, dass die Embedding Matrix eine Dimensionen von 4096×128256 bei der 8B Variante, 8192×128256 bei der 70B Variante und 16384×128256 bei der 405B Variante hat. [Dub+24]

2.1.3 Transformation

Zur eigentlichen Vorhersage nutzen Transformer-Modelle zwei Konzepte: das Attention-Modul und das Feed-Forward-Modul. Für jeden Layer des Models werden sowohl ein Attention-Modul als auch ein Feed-Forward-Modul durchlaufen. Llama3 nutzt unterschiedliche Anzahlen an Layern für die verschiedenen Modellvarianten, 8B hat 32 Layer, 70B hat 80 Layer und 405B hat 126 Layer. Außerdem befindet sich üblicherweise eine Normalisierungsfunktion nach den Modulen um die Stabilität beim Training zu erhöhen.

2.1.3.1 Attention-Modul

Das Attention-Modul dient dazu, das Embedding von Tokens anzupassen, basierend auf anderen Tokens die im Kontext enthalten sind. Dieser Ansatz nutzt den Attention-Mechanismus der erstmals in [BCB16] verwendet wurde und durch die Transformer-Architektur von [Vas+23] bekannt gemacht wurde, sodass heute alle beliebten LLMs auf diesem aufbauen. In der Transformer-Architektur werden sogenannten Attention-Heads eingesetzt, diese dienen dazu, Verknüpfungen zwischen verschiedenen Embeddings zu finden und dafür zu sorgen, dass diese sich gegenseitig beeinflussen können. Als vereinfachtes Beispiel könnte ein Attention-Head dafür verantwortlich sein, die Farbe von einem Gegenstand zu beschreiben. Dementsprechend würde der Attention-Head nur Embeddings beachten die Farben oder Gegenstände enthalten und dann das Embedding der Gegenstände so modifizieren, dass die Farbe Teil des Embeddings ist. Aus Embeddings für “Rot” und “Auto” würde er also ein Embedding für “Rotes Auto” erstellen.

Umgesetzt wird dies mit drei Matrizen die im Training angepasst werden, Q ist die Query-Matrix oder Abfragematrix, K ist die Key-Matrix oder Schlüsselmatrix und V ist die Value-Matrix oder Wertematrix. Die Größe dieser Matrizen ist abhängig von dem Vokabular und der Embeddinggröße.

Wenn die Token-Embeddings als \vec{x}_i gegeben sind, lassen diese sich folgendermaßen beschreiben:

- Die Query-Matrix dient dazu, für Embeddings herauszufinden welche anderen Embeddings Einfluss auf dieses haben sollten. Hierfür wird der Query-Vektor $\vec{q}_i = \vec{x}_i \cdot Q$ berechnet, welcher als eine Art Anfrage dient.
- Die Key-Matrix dient zur Berechnung der Antwort auf die Query. Es wird der Key-Vektor $\vec{k}_i = \vec{x}_i \cdot K$ berechnet.
- Die Value-Matrix dient zur Berechnung welchen Einfluss eine Embeddings auf andere haben soll. Es wird der Value-Vektor $\vec{v}_i = \vec{x}_i \cdot V$ berechnet.

Dann können alle Query-Vektoren und Key-Vektoren miteinander verglichen werden indem Skalarprodukte zwischen ihnen gebildet werden. Die Ähnlichkeit von Query-Vektor und

Key-Vektor gibt an, wie stark das Embedding vom Key-Vektor das Embedding des Query-Vektors beeinflussen soll.

$$a_{ij} = \vec{q}_i \cdot \vec{k}_j$$

Zur Stabilisierung während dem Training wird dies üblicherweise noch durch die Quadratwurzel der Länge der Key-Vektoren geteilt. Außerdem wird die Softmax Funktion auf das Ergebnis angewendet um die Gewichtung zwischen Null und Eins zu skalieren.

$$\tilde{a}_{ij} = \text{softmax} \left(\frac{a_{ij}}{\sqrt{\dim(k)}} \right)$$

Das Ergebnis ist eine Matrix \tilde{a}_{ij} von Attention-Gewichten, die beschreiben wie stark ein Embedding \vec{x}_j ein anderes Embedding \vec{x}_i beeinflussen soll. Abschließend können die Attention-Gewichte mit den Value-Vektoren multipliziert werden um die neuen Embeddings, mit Einfluss aller anderen Embeddings, zu berechnen.

Diese Schritte werden üblicherweise als “Standard Scaled Dot-Product Attention” bezeichnet und in folgender Formel zusammengefasst. [Vas+23]

$$\text{Attention}(Q, K, V) = \text{softmax} \left(\frac{QK^T}{\sqrt{d_k}} \right) V$$

In der Praxis wird üblicherweise Multi-Head-Attention genutzt, hier werden mehrere Attention-Heads mit jeweils eigenen Query-, Key- und Value-Matrizen eingesetzt die parallel arbeiten um die Lernfähigkeit um ein vielfaches zu verbessern.

Llama3 nutzt für die unterschiedlichen Modelle auch unterschiedlich viele Attention-Heads, die 8B Variante nutzt 32 Attentions-Heads, 70B nutzt 64 Attention-Heads und 405B nutzt 128 Attention-Heads. [Dub+24]

2.1.3.2 Feed-Forward-Modul

Die Standard-Transformer Architektur nach [Vas+23] nutzt Zwei Feed-Forward-Layer um lineare Transformationen mit Gewichtung und Bias durchzuführen welche ein vollständig verbundenes Neuronales Netz bilden und durch die eine Aktivierungsfunktion wie ReLU verbunden sind, um Nichtlinearität zu ermöglichen, was die Lernfähigkeit erhöht. Llama3 hingegen nutzt Drei Feed-Forward-Layer die ohne Bias arbeiten und nur Gewichtungen ausführen. Als Aktivierungsfunktion wird SwiGLU eingesetzt, nachdem [Sha20] gezeigt hat, dass Transformer-Modelle mit dieser damit bessere Ergebnisse liefern. Die Aktivierungsfunktion wird nur zwischen dem ersten und zweiten Layer eingesetzt. [Dub+24]

Die Feed-Forward-Layer transformieren die Embeddings anhand von gelernten Matrizen welche auf alle Embeddings gleich angewendet werden, hier hat der Kontext also keinen weiteren Einfluss. Die Dimension dieser Feed-Forward Matrizen ist üblicherweise größer als

die der Embeddings, was komplexeres Lernverhalten ermöglicht. [Vas+23]

Diese Feed-Forward Matrizen machen einem Großteil der Parameter von Transformer-Modellen aus. [Dub+24]

Llama3 nutzt hier wieder unterschiedliche Dimensionen je nach Modellvariante, die 8B Variante nutzt eine Embedding-Dimension von 4096 und eine Feed-Forward-Dimension von 14336, die 70B Variante nutzt eine Embedding-Dimension von 8192 und eine Feed-Forward-Dimension von 28672 und die 405B Variante nutzt eine Embedding-Dimension von 16384 und eine Feed-Forward-Dimension von 53248.

2.1.4 Un-Embedding

Das Un-Embedding dient dazu, die Embeddings wieder zurück in Tokens zu überführen, hierfür wird ein Unembedding-Layer genutzt. Dieser führt zuerst eine lineare Transformation aus um die Embeddings zurück auf das Vokabular abzubilden, gefolgt von folgender leicht modifizierte softmax-Funktion, wobei T als Temperatur bezeichnet wird:

$$\frac{e^{\frac{x_n}{T}}}{\sum_{n=0}^{N-1} e^{\frac{x_n}{T}}}$$

Diese dient dazu, die Ausgabewahrscheinlichkeiten über dem Vokabular zu bestimmen. Abschließend findet ein Auswahlverfahren statt, welches letztendlich das folgende Token bestimmt. Die Modifizierung der softmax-Funktion erlaubt es eine Temperatur T festzulegen, welche die Ausgabewahrscheinlichkeiten anpasst. Bei einer hohen Temperatur wird die Wahrscheinlichkeitsverteilung flacher und gleichmäßiger, bei einer geringen Temperatur wird die Verteilung steiler. Ist die Temperatur $T = 1$, ist die softmax-Funktion unverändert, eine Temperatur von $T = 0$ gewichtet den höchsten Eintrag mit 100%. Ein übliches Maximum der Temperatur ist $T = 2$, was keinen technischen Grund hat, sondern nur dazu dient die Qualität zu gewährleisten, da höhere Temperaturen oft zu schlechten und zusammenhanglosen Texten führen.

2.2 Trainingsprozess von LLMs

2.2.1 Pre-Training

Das Pre-Training (Vortraining) ist der wohl aufwändigste Schritt in der Erstellung eines LLMs. Es dient dazu, dem Modell das allgemeine Verständnis für Sprache mit Grammatik und Vokabular sowie Faktenwissen und logisches Denken beizubringen. Nach dem Pre-Training wird das Modell als base-model (Basis-Modell) bezeichnet, dieses ist nur darauf trainiert das nächste Token vorherzusagen. Von hier kann es dann durch Fine-Tuning (Feinabstimmung) weiterentwickelt werden.

2.2.1.1 Ablauf

Zu Beginn wird ein Datensatz und ein initialisiertes Modell benötigt, die Parameter des

Modells werden oft zufällig initialisiert. Dann kann ein sogenannter Forward-Pass durchgeführt werden. Dabei werden Teile des Datensatzes in das Modell gegeben, diese werden wie zuvor beschrieben tokenisiert, embedded, transformiert und un-embedded bis eine Vorhersage über das folgende Token gemacht wird. Dann wird eine Loss-Calculation (Verlustrechnung) durchgeführt indem eine Loss-Function (Verlustfunktion) genutzt wird, um den Loss (Fehler, Verlust) zwischen der Vorhersage und dem eigentlichen Token zu berechnen.

Durch Backpropagation (Rückpropagierung oder Fehlerrückführung) werden dann für alle Parameter aus der Embedding-Matrix, aus allen Query-, Key- und Value-Matrizen im Attention-Modul, aus allen Feed-Forward-Matrizen sowie aus der Un-Embedding Matrix Gradienten berechnet, welche angeben wie die Parameter angepasst werden müssen um den Verlust zu minimieren. Daraufhin werden die Parameter mit einem Optimierungsalgorithmus angepasst um das Ergebnis zu verbessern.

Diese Abfolge von Forward-Pass, Verlustrechnung, Backpropagation und Parameteroptimierung ist im Deep-Learning üblich und wird durchgeführt bis die geplante Trainingsdauer erreicht ist, das Modell die Erwartungen erfüllt, die Ressourcenkosten erreicht sind oder bis die Leistungsfähigkeit des Modells ein Plateau erreicht.

2.2.1.2 Datensatz

Der Datensatz ist von großer Wichtigkeit für die Qualität des darauf basierenden LLMs. Beliebte LLMs nutzen immer größere Datensätze um immer bessere Ergebnisse zu erzielen, GPT3 wurde mit 499 Milliarden Tokens trainiert [Bro+20] und GPT4 wird auf 13 Billionen geschätzt.³ Llama2 wurde auf zwei Billionen Tokens trainiert [Tou+23] und Llama3 auf 15 Billionen. [Dub+24]

Trainingsdaten sind üblicherweise zusammengetragen aus verschiedenen öffentlich verfügbaren Datensätzen. Bekannte Quellen hierfür sind unter anderem:

- CommonCrawl⁴ ist einer der größten öffentlichen Datensätze im Internet, er enthält mehrere Petabyte an Daten von mehreren Milliarden Webseiten die seit 2008 durch Webcrawler gesammelt wurden. Hier können mehrere TB an reinen Text-Daten extrahiert werden.
- Wikipedia stellt ihren gesamten Datensatz öffentlich zur Verfügung, hier sind ungefähr 100GB an hochqualitativem Text verfügbar. Es steht auch der gesamte Revisionsverlauf zur Verfügung welcher mehrere TB umfasst.⁵
- GitHub ermöglicht es nicht direkt alle Daten herunterzuladen, sie bieten jedoch öffentlichen Zugriff auf die einzelnen Repositories sowie eine API, über die Repositories aufgelistet, gefiltert und heruntergeladen werden können. Dadurch gibt es Tools wie den Github-Downloader⁶, welche mit Standardeinstellungen gefiltert den Download

³OpenAI veröffentlicht kaum Informationen zum Architektur von GPT4 und anderen neueren Modellen. Spekulationen u.a. von Semianalysis (<https://www.semianalysis.com/p/gpt-4-architecture-infrast-structure>) schätzen die Trainingsdaten jedoch auf 13 Billionen Tokens.

⁴<https://commoncrawl.org/>

⁵https://en.wikipedia.org/w/index.php?title=Wikipedia:Database_download

⁶<https://github.com/EleutherAI/github-downloader>

von ungefähr 95GB in 192000 Repositories ermöglicht.

- Project Gutenberg⁷ ist eine Sammlung von über 70000 öffentlich verfügbaren eBooks. Ähnliche Datensätze wie “books1” und “books2” wurden in der Vergangenheit für viele Modelle genutzt, sind aber wegen Urheberrechtsverletzungen im Datensatz nicht mehr verfügbar.

Außerdem enthalten Trainingsdaten oft nicht-öffentlich verfügbare Daten, dies könnte unter anderem folgendes enthalten:

- Eigene Firmendaten. Fast jedes Unternehmen hat Firmendaten die zum Training eines LLMs nützlich sein könnten. Beispielsweise hat Bloomberg [Wu+23] ein eigenes LLM trainiert, dessen Trainingsdaten zu mehr als 50% aus privaten Finanzdaten besteht. Das Ergebnis ist ein Modell das in allgemeinen Aufgaben ähnlich gut ist wie andere, ähnlich große Modelle, jedoch im Finanzbereich viel bessere Ergebnisse erzielt.
- Lizenzierte Daten. Große LLM-Anbieter nutzen häufig Daten die von anderen Firmen erworben wurde. So können entweder große Mengen an Daten erhalten werden, wie beispielsweise die Lizenzvereinbarungen zwischen sowohl OpenAI als auch Google mit Reddit, welche ihnen unbegrenzten Zugriff auf Reddit-Daten gibt.⁸

Andererseits werden kleinere, aber hochqualitative Daten lizenziert, OpenAI hat beispielsweise Lizenzverträge mit Medienunternehmen wie Time, The Atlantic, The Financial Times, Axel Springer und vielen mehr.⁹

- Nutzer-Interaktionsdaten. Anbieter von Chatbots wie OpenAI haben große Mengen an Chatverläufen von Nutzer mit ihren Chatbots, diese können explizit zum Training verwendet werden.¹⁰ Andere Firmen haben Telefongespräche, Chat-, und Emailverläufe vom Kundensupport die einen ähnlichen Zweck erfüllen können.
- Unberechtigter Zugriff auf Daten. Die Trainingsdaten von Modellen sind häufig nicht öffentlich, aber es wird vermutet, dass diese viele Copyright-geschützten Daten enthalten, da oft Web-Scraping eingesetzt wird um diese Daten aus öffentlichen Webseiten auszulesen. Beispielsweise hat die New York Times diesbezüglich eine Anklage gegen OpenAI und Microsoft eingereicht¹¹ Ein weiterer öffentlicher Vorwurf ist, dass OpenAI die Transkriptionssoftware Whisper entwickelt hat um Transkriptionen von Youtube Videos als Trainingsdaten für ihre anderen Modelle nutzen zu können, und dass GPT4 bereits anhand von Millionen von Stunden an Youtube Videos trainiert

⁷<https://www.gutenberg.org/>

⁸<https://www.reuters.com/technology/reddit-ai-content-licensing-deal-with-google-sources-say-2024-02-22/>

⁹Eine Liste von bekannten Lizenzvereinbarungen wurden von Variety veröffentlicht unter <https://variety.com/vip/breaking-down-ai-content-licensing-all-the-publisher-deals-training-ai-models-1236093395/>

¹⁰OpenAI nutzt ChatGPT Chatverläufe um ihre Modelle zu trainieren. <https://help.openai.com/en/articles/5722486-how-your-data-is-used-to-improve-model-performance>

¹¹<https://www.nytimes.com/2023/12/27/business/media/new-york-times-open-ai-microsoft-law-suit.html>

wurde.¹²

Neben der Menge der Trainingsdaten ist auch die Qualität der Daten sehr wichtig. Da die meisten Daten von Webseiten stammen, enthalten diese große Mengen an HTML, CSS und Javascript, welche, zumindest in diesem Umfang, nicht Teil der Trainingsdaten sein sollen. Auch sind Daten häufig schlecht formatiert, oder enthalten irrelevante Informationen wie Werbung in der Mitte einer Konversation, Beschriftungen von Textfeldern, Buttons oder ähnlichem, welche aus den Trainingsdaten entfernt werden sollten. Außerdem müssen personenbezogene Daten, Hassrede, Illegale Inhalte und altersbeschränkte Daten herausgefiltert werden. Um die Qualität zu verbessern werden auch qualitativ schlechte Trainingsdaten sowie veraltete Informationen und Missinformationen entfernt oder gekennzeichnet. Duplikate in den Trainingsdaten können zu Problemen mit Overfitting (Überanpassung) führen, weshalb oft Fuzzy-Deduplication (Unschärfe Deduplizierung) Verfahren angewendet werden um diese zu entfernen. Um all diese Ziele zu erreichen werden häufig komplexe Datenreinigungs- und Filterabläufe entwickelt, welche die riesigen Mengen an Trainingsdaten stark verkleinern. Die Datensätze und deren Entwicklungsprozesse von den aktuell besten LLMs sind nicht öffentlich, für GPT3 wurden jedoch 45TB an komprimiertem Text zu nur noch 570GB unkomprimiertem Text gefiltert. [Bro+20; Dub+24]

2.2.2 Fine-Tuning

Fine-Tuning ist ein Verfahren um das Verhalten von LLMs anzupassen, nachdem ein Modell das Pre-Training abgeschlossen hat ist es üblich dieses zu Fine-Tunen. Das Basis-Modell hat zwar bereits viel Wissen gesammelt, für die meisten Anwendungen ist es aber nicht geeignet, da es nur das nächste Token vorhersagen kann. Durch Fine-Tuning kann das Modell dann angepasst werden um gewünschte Aufgaben zu erledigen. Chatbots werden beispielsweise durch Instruction-Tuning darauf trainiert, ein Prompt mit einer Anfrage zu bekommen, und darauf zu antworten. Auf diese Weise können auch andere Ziele erreicht werden, oft werden durch Fine-Tuning der Sprach- oder Antwortstil eines LLMs angepasst oder zusätzliches Wissen für eine Spezialisierung hinzugefügt. Teil des Fine-Tunings ist üblicherweise auch das AI-Alignment was häufig mit RLHF (Reinforcement Learning with Human Feedback) umgesetzt wird. Hierfür werden dem LLM übliche Aufgaben gegeben und Menschen bewerten die Antworten anhand vorher definierter Kriterien wie Höflichkeit, Humor, Empathie, Hilfsbereitschaft, Nützlichkeit, Bias, Halluzinationen und ähnlichem, daraufhin wird das Modell anhand dieser Bewertung angepasst.

Datensätze zum Fine-Tuning unterscheiden sich stark von denen, die für das Pre-Training genutzt werden. Sie haben einen viel kleineren Umfang, Meta empfiehlt ein-paar Tausend hochqualitative Beispiele¹³, OpenAI empfiehlt sogar mit nur 50 bis 100 hochqualitativen Beispielen anzufangen und größere Mengen auszuprobieren wenn dies erfolgreich war¹⁴.

Der Ablauf des Fine-Tuning Prozesses ist ähnlich zu dem Pre-Training. Es wird ein Modell

¹²<https://www.nytimes.com/2024/04/06/technology/tech-giants-harvest-data-artificial-intelligence.html>

¹³<https://ai.meta.com/blog/how-to-fine-tune-llms-peft-dataset-curation/>

¹⁴<https://platform.openai.com/docs/guides/fine-tuning/preparing-your-dataset>

benötigt das bereits das Pre-Training durchlaufen hat sowie der spezialisierte Datensatz. Je nach Aufgabe können dann Modelleigenschaften oder einzelne Layer angepasst werden, für die meisten Textgenerierungsaufgaben ist dies jedoch nicht nötig. Daraufhin kann das Modell trainiert werden, hierfür wird die Learning-Rate (Lernrate) typischerweise geringer eingestellt als im Pre-Training da das grundlegende Wissen bereits vorhanden ist und nur noch kleine Anpassungen gemacht werden müssen.

Llama3 enthält sowohl das Basis-Modell als auch eine Instruct-Variante für jede Modellgröße. Die Instruct-Varianten wurden durch Instruction-Tuning und AI-Alignment verfeinert und entsprechen einem Chatbot wie man ihm beispielsweise von ChatGPT kennt.

2.2.3 Fine-Tuning Varianten

Die Standardmethode zum Fine-Tuning wird heute oft als Full Fine-Tuning bezeichnet, hier werden während dem Lernen alle Parameter angepasst. Dadurch kann der größtmögliche Trainingserfolg erzielt werden, es entsteht jedoch auch ein sehr hoher Rechenaufwand und Speicherbedarf, da das gesamte Modell, wie im Pre-Training, geladen werden muss. Wegen diesem hohen Aufwand wird Full-Fine-Tuning von großen Modelle heutzutage nur noch selten durchgeführt. Der Modell-Entwickler nutzt Full-Fine-Tuning üblicherweise für seine Fine-Tuning Anforderungen, aber die meisten Drittanbieter oder Privatpersonen nutzen üblicherweise andere Verfahren.

PEFT (Parameter-Efficient Fine-Tuning) ist eine Gruppe von Fine-Tuning Verfahren, bei denen nicht alle Parameter angepasst werden, sondern nur eine kleinere Teilmenge. Dadurch kann der Fine-Tuning Prozess viel effizienter ablaufen, und der Einfluss der etwas geringeren Qualitäten rechtfertigt meisten den eingegangenen Kompromiss. Vor allem der viel geringere Speicherbedarf und die geringeren Trainingskosten leiten die meisten Entwickler dazu PEFT Verfahren einzusetzen, dies erlaubt auch Hobby-Entwicklern das Fine-Tuning von recht großen LLMs an eigener Hardware oder durch kostengünstige Cloud-Lösungen. Außerdem zeigen einige PEFT Verfahren nur sehr geringe oder keine Leistungsnaht gegenüber Full-Fine-Tuning. [Wey+24; Hu+21; Det+23]

Unter PEFT fallen unter anderem folgende Verfahren:

Adapter sind ein Fine-Tuning Verfahren, bei dem zusätzliche Layer in das Modell hinzugefügt werden. Diese können dann trainiert werden, während der Rest des Modells eingefroren (unveränderbar) ist. Die genaue Implementation ist hier nicht festgelegt und erfordert spezialisierte Planung für die gewünschte Aufgabe. Generell fügen Adapter nur eine geringe Anzahl an Parametern hinzu die trotzdem gute Fine-Tuning Ergebnisse liefern. Oft entsteht durch Adapter jedoch eine Latenz bei der Interferenz im fertigen Modell da zusätzliche Schritte durchlaufen werden müssen.

LoRA (Low Rank Adaption) [Hu+21] ist ein Fine-Tuning Verfahren das auf Adaptern basiert. Es werden sogenannte Update-Matrizen genutzt welche auf das vortrainierte Modell aufaddiert werden können um das Training effizienter zu machen. Hierfür werden die ursprünglichen Gewichte eingefroren und stattdessen werden die kleineren Update-Matrizen trainiert. Die Größe der Update-Matrizen kann über einen Parameter r eingestellt wer-

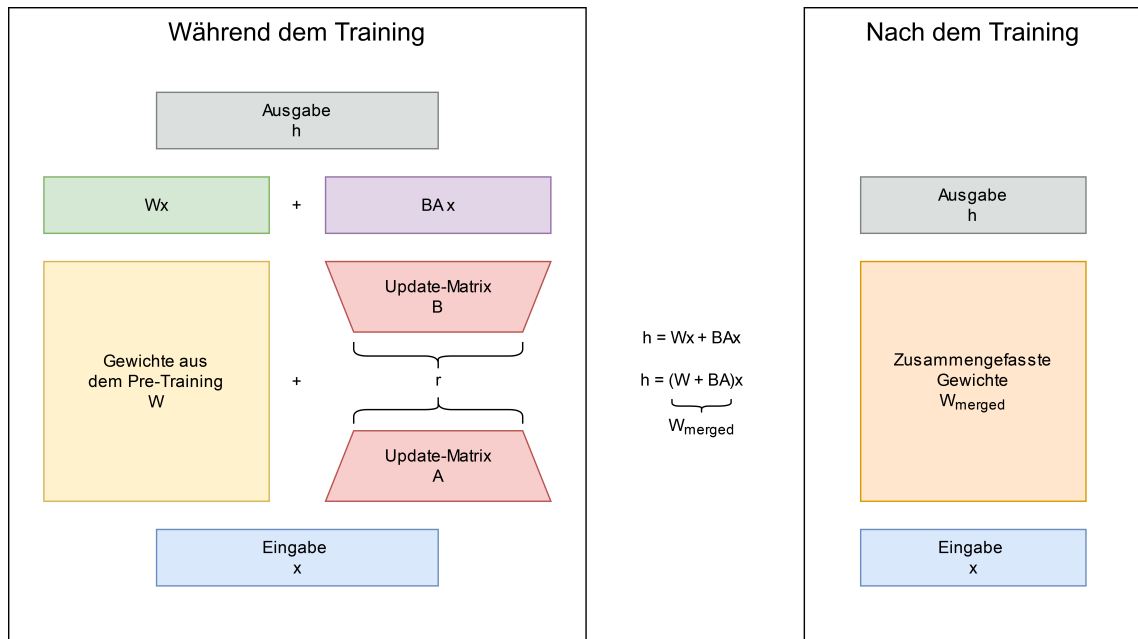


Abbildung 5 Schematische Darstellung der Funktionsweise von LoRA [Hu+21; Hug24]

den und bestimmt die Anzahl der neuen Gewichte die das Fine-Tuning beeinflussen sollen. Am Ende des Fine-Tunings können die ursprünglichen Gewichte und die Gewichte aus der Update-Matrix zu einem neuen Modell zusammengefügt werden, so entsteht keine zusätzliche Latenz bei der Inferenz wie bei vielen anderen Adaptern. LoRA erzielt nach [Hu+21] sehr gute Ergebnisse, für GPT3 konnte es die Parameterzahl um einen Faktor von 10000 reduzieren und den Speicherbedarf auf ein Drittel verringern ohne Qualitätsverluste zu erleiden.

QLoRA [Det+23] ist eine Erweiterung von LoRA, welchen die Hardwareanforderungen noch weiter reduziert indem das Modell durch Quantization (Quantisierung) in eine geringere Präzision überführt wird. Modelle nutzen üblicherweise den Datentyp float16, wobei die 16 für die Anzahl der Bits steht, die dieses Modell für die Speicherung jedes Parameters nutzt. Durch Quantization kann der Datentyp in einen anderen, kleineren, überführt werden [Fra+23; Det+22]. Beliebte LLMs werden oft zu den Datentypen int8, int6, int5, int4, int3 und int2 quantisiert, diese werden dann mit fp16, q8, q6, q5, q4, q3 und q2 im Modellnamen gekennzeichnet. Durch die Umwandlung in Datentypen mit geringerer Präzision gehen Daten verloren, deshalb sollte immer das Modell mit der höchsten Präzision genutzt werden, welches die Hardware erlaubt. QLoRA nutzt dieses Prinzip und führt Fine-Tuning auf einem quantisierten Modell aus. In [Det+23] wurde QLoRA Fine-Tuning erfolgreich auf viele quantisierte 4-Bit Modelle angewendet ohne das ein Leistungsabfall im Vergleich zum Full-Fine-Tuning der gleichen float16-Modelle festzustellen ist. Bei Modellen mit 65 Milliarden Parametern (65B) konnte so der durchschnittliche Speicherbedarf von 780GB auf unter 48GB gesenkt werden.

2.3 Betrachtung vorheriger Versuche der LLM Programmcode Generierung

StarCoder [Li+23a] ist ein LLM das zur Code Generierung entwickelt wurden, es wurden sowohl ein Basis-Modell als auch Fine-Tuning Varianten veröffentlicht. Der Datensatz basiert auf “The Stack”, einem öffentlichen Datensatz von Code. Um den Datensatz zu verbessern wurde dieser ausgiebig gefiltert. Im ersten Schritt wurde sich auf die beliebtesten Programmiersprachen, und die mit den meisten verfügbaren Daten, fokussiert. Dadurch wurden die Trainingsdaten von 358 Programmiersprachen auf nur noch 86 gefiltert, ABAP wurde in diesem Schritt verworfen. Es wurden insgesamt 2.58 Millionen Dateien zufällig ausgewählt und in Dateiendungen kategorisiert wobei maximal 1000 pro Dateiendung behalten wurden. Für jede Dateiendung wurden dann 50 bis 100 manuell inspiziert um sicherzustellen, dass es sich um hochqualitativen Code handelt und nicht um Rohdaten, Text oder automatisch generierten Code und um das beste Dateifilterungsvorgehen für die weiteren Schritte zu bestimmen. Darauf basierend wurden einige Dateiendungen komplett entfernt, außerdem wurden Dateien mit Zeilen mit mehr als 1000 Zeichen, und Dateien mit weniger als 25% alpha-numerischer Zeichen, entfernt. Es wurde zusätzlich eine Deduplikation durch Hashing anhand von Minhashes und Locally Sensitive Hashing mit 5-grams und einer Jaccard Ähnlichkeit von 0.7 durchgeführt um sicherzustellen, dass der Datensatz nicht zu viele Duplikate enthält.

Um persönlich identifizierbare Informationen auszufiltern wurde ein separates Modell trainiert, dass diese Daten finden und durch Platzhalter ersetzen soll. Hierfür wurde ein Datensatz von persönlichen Informationen in Source-Code in kleine Teilmengen unterteilt und über eine Online-Plattform an insgesamt 1399 Arbeitskräfte verteilt, die die Problemstellen markieren, wodurch das Modell lernt. Außerdem wurde RegEX eingesetzt um Keys, Passwörter, Email-Adressen, IP-Adressen und ähnliches zu finden. Schließlich wurde sichergestellt, dass keine Kontamination des Datensatzes mit den später verwendeten Benchmarks vorhanden ist, indem diese Daten gefunden und entfernt wurden.

StarCoder2 [Loz+24] ist das Nachfolge-Modell dazu, dieses ist größer und wurde mit viel mehr Daten trainiert, verfolgt aber die gleichen Filterungsschritte.

DeepSeekCoder [Guo+24] ist eine Reihe von Modellen in unterschiedlichen Größen, welche auf den DeepSeek Modellen basieren und Fine-Tuning für die Code-Generierung durchlaufen haben. Der Datensatz umfasst 87% Source-Code, 10% englischer Text aus Programmierungsthemen und 3% Daten von hochqualitativen chinesischem Text. Zur Filterung wurden: 1) die maximale Zeilenlängen auf 1000 begrenzt, 2) die durchschnittliche Zeilenlänge auf 100 begrenzt, 3) der Anteil an alpha-numerischen Zeichen auf mindestens 25% begrenzt, 4) HTML Dateien wurden entfernt wenn der sichtbare Text weniger 80% ist und 5) wurden JSON und YAML Dateien entfernt die weniger als 50 oder mehr als 5000 Zeichen enthalten und den Fokus auf die Struktur zu legen, nicht auf den Inhalt.

Außerdem wurde Repository-Level Deduplikation durchgeführt und Code mit geringer Qualität wurde gefiltert indem ein Compiler, ein Qualitätsmodell und Heuristische Regeln eingesetzt wurden. Abschließend wurden n-Gram Filter eingesetzt um Kontamination der Trainingsdaten mit den Benchmarkdaten zu verhindern. Insgesamt wurden diese Modelle

anhand von zwei Milliarden Token verfeinert.

CodeLlama [Roz+24] ist eine Familie von Modellen die auf Llama2 basieren und zur Code-Generierung entwickelt wurde. Der Datensatz ist in drei Teile aufgeteilt: Proprietäre Daten, das sind hier hauptsächlich Daten die durch RLHF zum Fine-Tuning des Instruct-Varianten gesammelt wurden. Sie enthalten tausende Beispiele mit Prompt und Antwort und mehrere Millionen Beispiele die nicht erlaubt werden sollen (Rejection Sampling). Der Zweite Teil des Datensatzes sind Wiederholungen aus den vorherigen Daten, hiermit soll sichergestellt werden, dass die Codellama-Instruct Variante nicht zu viel von seinem Code-Verständnis vergisst während es auf Instructions feinabgestimmt wird. Der dritte und größte Teil umfasst Beispiele für Programmieraufgaben. Hierfür wurde eine vorherige Version von Llama2 genutzt um 62000 Fragen im Stil von einem Vorstellungsgespräch für eine Rolle als Softwareentwickler zu generieren. Dann wurden Duplikate entfernt und Unit-Tests sowie Lösungen für diese Aufgaben generiert. Die Unit-Tests wurde genutzt um die Lösungen zu verifizieren. Wenn eine erfolgreiche Lösung gefunden wurde, wurde der gesamte Test mit Unit-Tests und Lösung in den Datensatz aufgenommen.

In [NZ24] wurden mehrere Basis-Modelle von CodeGen und Gemma durch Fine-Tuning auf die Verilog-Code-Generation spezialisiert. Für den Versuchsaufbau wurden hier fünf LLMs verfeinert, eines als Experte zur Klassifizierung der Daten und die restlichen zur Code Generierung für Anfänger, Mittelmäßige, Fortgeschrittene und Experten. Der Datensatz stammt hier von öffentlichen Github Projekten, welche durch GPT3.5 sowohl in diese Kategorien eingeteilt als auch mit Beschreibungen versehen wurden.

3 Methodik

3.1 Entwicklung der Auswertungsverfahren

Um die Veränderung der Ergebnisse durch Fine-Tuning festzustellen muss ein einheitlicher Analyseprozess eingesetzt werden, der die Ergebnisse vor und nach dem Fine-Tuning möglichst objektiv miteinander vergleicht.

Um dies zu gewährleisten sollten folgende Punkte beachtet werden:

- ABAP ist eine komplexe Programmiersprache welche mit Variablen, Datentypen, Kontrollstrukturen, Funktionsblöcken, Operation, Objektorientierung, Fehlerbehandlung sowie der Standardbibliothek und mehr, sehr viele Funktionen hat. Um diese alle abzudecken wird eine große Anzahl an verschiedenen Testszenarien benötigt.
- Da die Antworten von LLMs nicht-deterministisch sind (zumindest nicht insoweit Computer dazu in der Lage sind) ist es notwendig viele Wiederholungen der Testszenarien durchzuführen. Dies dient dazu, sicherzustellen, dass Zufälle einen möglichst kleinen Einfluss auf das Ergebnis haben da die Analyse dann anhand von Durchschnittsn durchgeföhrt werden kann.
- Das Ziel ist es Programmcode zu generieren, hierfür ist der Auswertungsprozess komplex, da das Ergebnis nicht einfach mit einer Musterlösung verglichen werden kann. Es gibt nahezu unendlich viele Wege das gleiche Problem durch Code zu lösen, beispielsweise durch unterschiedliche Variablennamen. Es muss also ein intelligenterer Auswertungsprozess eingesetzt werden um Code auszuwerten.

Da die manuelle Analyse von Code und die manuelle Auswertung der Ergebnisse sehr aufwendig sind, besteht der vermutlich beste Weg, einen solchen Auswertungsprozess umzusetzen, darin, ihn zu automatisieren. Wenn ein Prozess zur automatisierten Ausführung und Auswertung der LLM-Antworten gefunden wurde, kann die Anzahl der Wiederholungen beliebig hoch skaliert werden, um repräsentative Durchschnitte zu betrachten. Außerdem entfallen Fehler durch menschliche Ungenauigkeit und Unachtsamkeit welche, vor allem bei größeren Mengen, die Auswertung beeinflussen können. Diese Automatisierung kann durch Unit-Tests realisiert werden, hier stehen vorher erwartete Ein- und Ausgaben für den Code fest, dann wird der Code automatisiert ausgeführt und die erwarteten Ausgaben werden mit denen nach der Ausführung verglichen. Problematisch an dieser Automatisierung ist jedoch, dass ABAP ausschließlich innerhalb der SAP Umgebung läuft wodurch die Automatisierung schwierig wird, da diese vielen Einschränkungen hat.

Zur Umsetzung dieser Automatisierung wird das Benchmarking-Framework¹⁵ eingesetzt, an welchem ich im Praxisprojekt mitgearbeitet habe. Dieses diente ursprünglich dafür, die Fähigkeiten von ChatGPT in der ABAP-Entwicklung zu analysieren. Durch etwas Anpassung kann hier stattdessen ein beliebiges anderes LLM getestet werden, in diesem Fall eine Standard-Llama3 Variante und die, die durch Fine-Tuning für ABAP optimiert wurde. Es nutzt den bekannten HumanEval Benchmark [Che+21], welcher 164 Aufgaben,

¹⁵<https://github.com/timkoehne/Praxisprojekt-ABAP-und-ChatGPT>

Musterlösungen und Unit-Tests für die Python-Programmierung zur Verfügung stellt, und adaptiert diesen für ABAP Code. HumanEval hat sich als einer der Standard-Benchmarks für LLMs etabliert, es gibt für nahezu alle LLMs öffentliche Ergebnisse von HumanEval für Python und Untersuchen wie [Cas+22] und [Zhe+24] adaptieren ihn auch für anderen Programmiersprachen. Außerdem wird durch pyrfc¹⁶ eine Verbindung zu einem lokalen ABAP-Entwicklungsserver hergestellt, auf dem die Tests dann automatisiert vom originalen ABAP Compiler ausgeführt werden können. Diese Schnittstelle erlaubt jedoch nur Zugriff auf ABAP-Funktionsbausteine, was den testbaren Bereich der ABAP-Funktionen etwas einschränkt, vor allem wenn man beachtet, dass Funktionsbausteine weitgehend nicht mehr genutzt werden und stattdessen Objektorientierung mit Klassen und Methoden genutzt wird. Hier konnte jedoch kein besseres oder vergleichbares Automatisierungstool zur Ausführung von beliebigem ABAP-Code gefunden werden, welches Objektorientierung unterstützt, deshalb wird diese Untersuchung Objektorientierung nicht betrachten. Der Großteil der Funktionen, wie Variablen und Kontrollstrukturen können trotzdem getestet werden und funktionieren in beiden Fällen gleich, dementsprechend sollte diese Einschränkung keinen allzu großen Einfluss haben.

3.2 Vorbereitung des Fine-Tuning Prozesses

Entscheidungen über die exakte Modellauswahl und die Details des Ablaufs sind schwierig. Man möchte das größtmögliche Modell auswählen, da dieses am intelligentesten ist, allerdings müssen auch die verfügbaren Ressourcen betrachtet werden. Für diese Untersuchung war eine Limitierung auf eine lokal verfügbare Nvidia RTX 4070 mit 12GB VRAM geplant, wie sich in der Auswertung in Abschnitt 4 zeigen wird, wurde diese Limitierung jedoch leicht abgeändert. Stattdessen wurde das Fine-Tuning über Google Colab Notebooks auf einer virtuellen Maschine von Google Compute-Engine durchgeführt da mehr VRAM nötig war. Das fertige Modell läuft nach dem Fine-Tuning trotzdem lokal mit unter 12GB VRAM.

Als Fine-Tuning-Verfahren ist bei so begrenzten Ressourcen nur QLoRA sinnvoll und möglich. Dieses Verfahren konnte, wie in Abschnitt 2.2.3 beschrieben, die Ressourcenkosten erheblich reduzieren und führt im optimal Fall zu nur geringen bis gar keinen Einbußen in der Qualität. Es gibt viele Möglichkeiten QLoRA durchführen, dafür werden die folgenden Tools betrachtet:

- Torchtune¹⁷ ist ein Python-Bibliothek, die PyTorch erweitert und der Erstellung und dem Fine-Tuning von LLMs dient. Es ermöglicht Full-Fine-Tuning, LoRA und QLoRA Fine-Tuning innerhalb von PyTorch. Es gibt an, dass Llama3-8B im QLoRA Fine-Tuning eine Obergrenze von ca. 9GB VRAM erreicht, bei LoRA erreicht es ca. 19GB.
- Axolotl¹⁸ ist ein Tool, dass Fine-Tuning für eine Vielzahl von Konfigurationen und Architekturen ermöglicht. Es unterstützt alle gängigen Fine-Tuning Verfahren, einige

¹⁶<https://github.com/SAP/PyRFC>

¹⁷<https://github.com/pytorch/torchtune>

¹⁸<https://github.com/axolotl-ai-cloud/axolotl>

Optimierungen und lokales- sowie Cloud-Training. Außerdem hat es Konfigurationen für eine große Menge an Modellen, unter anderem die Llama Familie.

- Unsloth¹⁹ ist ein Startup, dass mehrere Fine-Tuning Methoden anbietet. Ihre Open-Source-Lösung ist sehr beliebt, sie bietet durch verschiedene Optimierungen die größten Reduktionen der Geschwindigkeit, und vor allem die größte Verringerung des Speicherbedarfs. Es unterstützt alle Fine-Tuning Verfahren, spezialisiert sich jedoch auf 4-Bit QLoRA, wo die meisten Optimierungen stattfinden. Außerdem gibt es vorgefertigte, spezialisierte, Konfigurationen für nahezu alle beliebten Modelle, inklusive Llama3. Nach eigenen Angaben kann so QLoRA Fine-Tuning von Llama3-8B auf Grafikkarten mit nur 8GB VRAM durchgeführt werden. Das größere 70B Modell wird trotzdem erst ab 48GB VRAM empfohlen.

Damit steht fest, dass für diese Untersuchung nur das Llama3-8B Modell in Frage kommt. Als Tool wird Unsloth eingesetzt, da es dieses Modell explizit unterstützt, der Ressourcenverbrauch geringer ist als bei den Alternativen und die Beliebtheit darauf schließen lässt, dass Nutzer zufrieden sind.

Wenn das Fine-Tuning abgeschlossen ist, wird der LoRA-Adapter mit dem Basis-Modell zusammengefügt, sodass ein komplett eigenständiges Modell entsteht. Dieses wird üblicherweise im GGUF-Dateiformat gespeichert, welches sowohl das gesamte Modell mit Gewichten, als auch dazugehörige Metadaten, enthält.

3.3 Untersuchung der Hyperparameter-Einstellungen

Hyperparameter sind Einstellungen welche die Details des Lernprozesses definieren, diese werden vor dem Training oder vor dem Fine-Tuning festgelegt und kontrollieren dann den Lernprozess. Sie werden nicht, wie normale Parameter, vom Modell gelernt, stattdessen werden Hyperparameter vom Entwickler eingestellt da sie üblicherweise nicht aus Daten ableitbar sind. Genauer betrachtet werden hier nur diese, die für das Fine-Tuning relevant sind.

Modell-spezifische Hyperparameter beschreiben den Aufbau von Modellen, hiervon werden folgende betrachtet:

- `r` [Hu+21] beschreibt die Rang der LoRA Update-Matrizen, je kleiner dieser ist, desto weniger Informationen können in den Update-Matrizen gespeichert werden, aber desto größer die Ressourcenersparnisse. Typische sind hier Werte von acht bis 128 wobei 16 von Unsloth für Llama3 empfohlen wird.
- `lora_alpha` [Hu+21] ist ein Faktor, der den Einfluss der LoRA Update-Matrizen skaliert. Ein hoher `lora_alpha` Wert würde also einen großen Einfluss auf die Update-Matrizen legen. Üblich ist es hier einen Wert zu wählen, der dem doppelten von `r` entspricht. Unsloth empfiehlt hier jedoch auch 16 für Llama3.
- `lora_dropout` [Lin+24] dient dazu, Overfitting (Überanpassung) des Modells zu verhindern. Hierbei werden mit geringer Wahrscheinlichkeit einzelne Parameter der

¹⁹<https://github.com/unslothai/unsloth>

Update-Matrix, für einzelne Trainingsschritte, auf null gesetzt. Dadurch wird sichergestellt, dass das Modell keinen zu starken Einfluss auf einzelne Parameter legt und stattdessen alle Parameter gleich gewichtet. Der Parameter kann beliebige float Werte zwischen null und eins annehmen, von Unsloth wird für Llama3 ein Wert von null empfohlen.

- `target_modules`²⁰ gibt an, an welchen Stellen LoRA Update-Matrizen genutzt werden sollen. Dadurch ist es möglich nur bestimmte Teile des Modells durch Fine-Tuning anzupassen. Unsloth empfiehlt hier `["q_proj", "k_proj", "v_proj", "o_proj", "gate_proj", "up_proj", "down_proj"]`. Wobei Q , K , V und O die Query-, Key- und Value-Matrizen sowie eine Output-Matrix aus dem Attention-Block sind und $Gate$, Up und $Down$ Teile des Feed-Forward-Blocks sind.
- `sequence_length` ist die Kontextlänge die das Modell zulässt. Hier sollte entweder der gleiche Wert aus dem Pre-Training verwendet werden, oder ein kleinerer, da so Ressourcen gespart werden können. Bei begrenzten Ressourcen können auch Out-of-Memory Fehler entstehen können, da höhere `Sequence_length` Werte auch mehr Speicher während dem Training benötigen. Unsloth stellt den Wert standardmäßig auf 2048 um Speicher zu sparen, schlägt aber vor diesen nach Bedarf manuell zu erhöhen.
- `use_rslora` kann genutzt werden um Rank-stabilized LoRA [Kal23] zu aktivieren, hierdurch werden mit höhere Ränge für r bessere Ergebnisse erzielt, dafür entsteht jedoch auch ein höherer Ressourcenaufwand. Unsloth empfiehlt dies für Llama3 auszuschalten.
- `loftq_config` kann genutzt werden um die Nutzung vom LoftQ Framework [Li+23b] zu aktivieren. Dieses berechnet die Quantisierung und die LoRA Update-Matrizen gleichzeitig, hierdurch können, vor allem bei kleineren Präzisionen, wie 2-Bit, qualitativ bessere Ergebnisse erzielt werden. Unsloth empfiehlt dies für Llama3 auszuschalten.

Optimierungs-spezifische Hyperparameter beschreiben das Lernverhalten von Modellen, hiervon werden folgende betrachtet:

- `Learning_rate` kontrolliert wie stark Parameter angepasst werden, es legt fest, wie groß die Schritte sind die vom Optimierungsalgorithmus gemacht werden um die Loss-Funktion zu reduzieren. Zum Fine-Tuning wird diese üblicherweise kleiner eingestellt, als beim Pre-Training. Unsloth empfiehlt hier 0.0002.
- `optim` gibt den Optimierungsalgorithmus an der genutzt werden soll um die Gewichte anzupassen. Die bekanntesten sind hier wahrscheinlich SGD (Stochastic Gradient Descent) und Adam (Adaptive Moment Estimation) da diese in Machine Learning generell viel genutzt werden. Unsloth empfiehlt hier "adamw_8bit".
- `weight_decay` reguliert die Größe von Gewichten indem sehr große Parametergewichtung bestraft wird, dadurch kann Overfitting verhindert werden. Unsloth empfiehlt

²⁰https://huggingface.co/docs/peft/main/en/conceptual_guides/lora

einem Wert von 0.01.

- `lr_scheduler_type` bestimmt ein Verfahren, nachdem die Learning Rate im Laufe des Trainings verändert werden soll. Dies ist üblich, da man am Anfang des Trainings das Verhalten in größeren Schritten anpassen will, da das Modell vermutlich noch sehr weit vom gewünschten Verhalten entfernt ist. Je länger das Training läuft desto kleiner sollen die Anpassungsschritte werden, da das Modell näher am Ziel ist. Hierfür empfiehlt Unsloth “linear”.
- `warmup_steps` wird genutzt um die Learning Rate am Anfang des Trainings zu verringern. Dies wird zu Beginn des Fine-Tunings oft gemacht, da der Datensatz sich vermutlich recht stark von dem des Pre-Trainings unterscheidet, und man einen sanfteren Übergang der Gewichte erreichen möchte. Unsloth empfiehlt hier einen Wert von fünf.
- `per_device_train_batch_size` ist Teil des Gradient Accumulation Verfahrens und legt fest, wieviele Mini-Batches genutzt werden. Für Llama3 empfiehlt Unsloth hier zwei.
- `gradient_accumulation_steps` ist Teil des Gradient Accumulation Verfahrens um legt die Anzahl der Schritte fest, die pro Mini-Batch durchlaufen werden. Unsloth empfiehlt hier für Llama3 einen Wert von vier.

Ablauf-spezifische Hyperparameter beschreiben den Ablauf des Trainings, betrachtet werden hier:

- `epoch` gibt an, wie oft der gesamte Datensatz während dem Training durchlaufen wird. Hier muss beachtet werden, dass zu große Werte zu Overfitting und zu kleine Werte zu Underfitting führen, beides möchte man soweit wie möglich vermeiden. Die richtige Anzahl an Epochen kommt auf das Modell und die Trainingsdaten an, manchmal ist nur eine Epoche die beste Wahl, manchmal sind zehn optimal.
- `Batch size` bestimmt, wie viele Daten im Training gleichzeitig das Modell durchlaufen. Ein Durchlauf ist hier ein Step (Schritt) und die Daten sind zusammengefasst eine Batch. Für jeden Schritt wird ein Forward-Pass, eine Loss-Berechnung, die Backpropagation und ein Aufruf des Optimizers mit der aktuellen Batch ausgeführt. Wenn mit kleinen Batch sizes trainiert wird, wird Speicher gespart und das Modell lernt üblicherweise schneller, da der Optimizer öfter aufgerufen wird. Größere Batch sizes haben zwar einen höheren Speicherverbrauch, aber auch einen geringeren Rechenaufwand, da der Optimizer seltener aufgerufen wird.

In Unsloth wird die Batch size nicht manuell festgelegt, stattdessen wird ein Verfahren namens Gradient Accumulation eingesetzt, dieses dient dazu, mit weniger Speicherbedarf größere Batch sizes zu simulieren. Es werden sogenannte Mini-Batches genutzt, diese sind kleiner als ursprüngliche Batches und durchlaufen nacheinander das Modell. Erst nachdem eine festgelegte Anzahl an Mini-Batches das Modell durchlaufen haben wird der Optimizer aufgerufen. Es gilt

$$\text{Batch size} = (\text{Anzahl der Mini-Batches}) \cdot (\text{Größe der Mini-Batches})$$

Below is an instruction that describes a task. Write a response that appropriately completes the request.

```
#### Instruction:
{instruction}
```

```
#### Response:
{response}
```

Abbildung 6 Vorlage zur Formatierung der Fine-Tuning Daten [Tao+23]

Gradient Accumulation wird eingesetzt um den Speicherbedarf während dem Training zu senken, ohne die benötigte Rechenleistung zu erhöhen.

Die bereits benannten Hyperparameterempfehlungen stammen aus dem Unsloth Llama3.1-8B Fine-Tuning Notebook.²¹ Für diese Untersuchung werden die meisten Hyperparameterempfehlungen von Unsloth übernommen, da diese explizit für Llama3 entwickelt wurden und vermutlich einen guten Startpunkt bieten. Eine Ausnahme ist hier die Kontextlänge, diese wird von 2048 auf 8192 angehoben um einen längeren Kontext zu erlauben, welcher für das Training anhand von langen ABAP-Dateien notwendig ist. Außerdem werden Hyperparameter wie die Lernrate und die Anzahl der Epochen im Laufe des Trainings angepasst um das bestmögliche Lernergebnis zu erzielen.

3.4 Entwicklung des Fine-Tuning Datensatzes

Die Fine-Tuning Datensätze die hier betrachtet werden dienen zur Interaktion mit Menschen im Chatbot-Format. Dafür müssen die Daten einem Format entsprechen, dass dem eines Gesprächs möglichst ähnlich ist. Es kann nicht einfach unstrukturierter Text genutzt werden, da das Modell sonst eher wie eine Autovervollständigung agiert, nicht wie ein Chatbot der auf Anfragen antwortet. Um dies zu erreichen wird die Vorlage in Abbildung 6 von Alpaca [Tao+23] eingesetzt, diese nimmt eine Instruction (Instruktion) und eine Response (Antwort) entgegen.

In dieser Untersuchung werden folgende drei Datensätze betrachtet um herauszufinden wie wichtig Datenmenge und Datenqualität für das Ergebnis ist:

DS1 Dieser Datensatz ist der Umfangreichste, er basiert auf 644MB ABAP-Code, aus insgesamt 59628 Dateien. Diese Daten stammen aus “The Stack v2” [Loz+24], einem Code-Datensatz vom BigCode Project welches insgesamt über 32TB in mehr als drei Milliarden Dateien aus über 600 Programmier- und Markup-Sprachen anbietet. Sie sind erreichbar über einen AWS S3 Bucket wofür sowohl eine Authentifizierung über AWS, als auch über HuggingFace nötig ist. Dort stehen diese, unter anderem, nach Programmiersprachen sortiert und komprimiert zur Verfügung und können über ein kurzes Python-Skript einfach heruntergeladen werden. Die relevanten Dateien wurden mithilfe von RegEx-Mustern in folgende Kategorien klassifiziert: “Function Module”, “Class”, “Report”, “Include”, “Interface”, “Method”, “Form”, “Module” und

²¹<https://colab.research.google.com/drive/1Ys44kVvmeZtnICzWz0xgpRnrIOjZAuxp>

“Unknown”.

DS1 enthält nur die Kategorien “Function Module”, “Class” und “Report” da die meisten anderen Kategorien nur wenige Daten enthalten. Die “Unknown” Kategorie enthält noch fast 7500 Dateien, diese sind jedoch hauptsächlich schlecht formatiert, Binärdateien, oder aus anderen Gründen nicht brauchbar. Einige dieser Dateien könnten durch weitere Vorverarbeitung wahrscheinlich brauchbar gemacht werden, davon wurde hier jedoch abgesehen da der voraussichtliche Aufwand dies nicht rechtfertigt.

Es handelt sich jedoch nur um einzelne Dateien, und zum Fine-Tuning wird, wie oben beschrieben, sowohl eine Instruction als auch eine Response benötigt. Da das Modell ABAP-Code generieren soll, werden die ABAP-Dateien also als Response in das Template eingesetzt. Es fehlt also noch die Instruction, da diese Daten jedoch nicht für diesen Zweck gemacht wurden, gibt es keine Instructions dazu. Um trotzdem diese große Anzahl an Daten zum Fine-Tuning nutzen zu können, werdend diese mit Llama3 generiert. Dies sollte dem Modell viel einfacher Fallen, als Code zu generieren, da der ganze Code als Kontext mit angegeben werden kann und so in die Generierung der Instruction einfließt.

Nach der Generierung der Instructions wurden diese noch einmal gefiltert, um sicherzustellen, dass die offensichtlich schlechten Prompts nicht im Datensatz vorkommen. Außerdem wurde die Länge von sowohl Prompt als auch Response auf die Kontextlänge von 8192 Tokens begrenzt. Nach dieser Filterung sind noch 29477 Einträge im Datensatz.

DS2 Dieser Datensatz beruht auf Code-Alpaca [Cha24], einem Datensatz für Code in mehreren verschiedenen Programmiersprachen. Es wurden zufällig 1000 Einträge ausgewählt und durch Llama3 von anderen Programmiersprachen nach ABAP übersetzt. Da dies auf einem bestehenden Fine-Tuning Datensatz beruht, gibt es hier für jeden Eintrag sowohl eine Instruction als auch eine Response, wobei die Respose den Code enthält und übersetzt wird. Hier wird die Vorlage leicht abgeändert, bei Code-Alpaca gibt es zusätzlich noch ein Input-Feld, welches in der Vorlage genauso aussieht, wie die für Instruction und Response. Es wird für manche Einträge genutzt um Ausführungsbeispiele anzugeben. Auch hier ist zu erwarten, dass die Generierung recht gute Ergebnisse liefert, da das LLM zur Übersetzung bestehende Lösungen in anderen Programmiersprachen als Kontext nutzt.

DS3 Bei diesem Datensatz handelt es sich um den einzigen ABAP Fine-Tuning Datensatz der öffentlich verfügbar gefunden werden konnte.²² Es gibt keine weiteren Informationen zur Herkunft des Datensatzes, der Autor ist nur unter einem Pseudonym bekannt und hat bisher keine anderen Datensätze veröffentlicht. Der Datensatz umfasst 248 Beispiele zur Generierung von kurzen ABAP Reports anhand von kurzen Instructions. Die Beispiele sehen qualitativ hochwertig aus, es wurde jedoch keine ausführliche Prüfung der Korrektheit der Beispiele durchgeführt.

Auf weitere Filterung wird bei diesen Datensätzen verzichtet. DS1 und DS2 basieren auf

²²<https://huggingface.co/datasets/smjain/abap>

beliebten Datensätzen die bereits ausgiebig gefiltert wurden, was in [Loz+24] und [Cha24] ausführlich beschrieben ist. Bei DS3 ist dies zwar nicht bekannt, der Datensatz ist jedoch klein genug, dass dies nicht notwendig ist, bei einer manuellen Überprüfung wurden hier keine Probleme erkannt.

3.5 Inferenz mit den Modellen

Zur Inferenz wird Ollama²³ lokal ausgeführt, dies bietet API Zugriff auf die Modelle für viele Programmiersprachen, Tools und ähnliches.

Für die automatisierte Auswertung wird die Python-Bibliothek ollama-python²⁴ genutzt um aus Python auf die Modelle zuzugreifen, sehr ähnlich zu bekannten Schnittstellen wie der von OpenAI.

Manuelle Interaktion mit den Modellen ist über die Kommandozeile möglich, hier wird stattdessen jedoch Open-WebUI²⁵ genutzt um eine gewohnte Chat-Oberfläche zu haben, ähnlich wie der von ChatGPT. Nach dem Fine-Tuning stehen die Modelle im GGUF-Dateiformat zur Verfügung, dieses kann dann durch eine einfache Ollama-Modelfile in Ollama importiert werden.

²³<https://github.com/ollama/ollama>

²⁴<https://github.com/ollama/ollama-python>

²⁵<https://github.com/open-webui/open-webui>

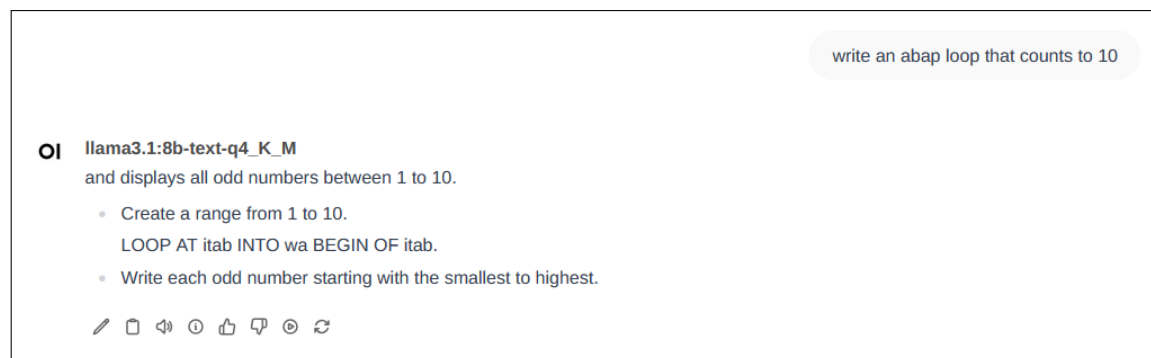


Abbildung 7 Beispielhafte Anfrage an das Metas Llama3.1:8b-text-q4_K_M

4 Durchführung des Fine-Tunings

Bei ersten Experimenten ist schnell aufgefallen, dass es zwar möglich ist Llama3-8B mit nur 12GB VRAM durch Fine-Tuning zu trainieren, aber einige Probleme entstehen, welche mit mehr VRAM nicht auftreten. Das Training mit den Unsloth Standardeinstellungen funktioniert mit weniger als 12GB VRAM, der Export des fertigen Modells enthält jedoch einen Zwischenschritt der 12GB überschreitet. Es ist zwar möglich den Export auszulassen, und stattdessen den LoRA Adapter bei der Inferenz mit aufzurufen, hierdurch entsteht jedoch ungewollte Latenz bei jedem Inferenz-Aufruf. Außerdem können die 12GB VRAM im Training nur erreicht werden, wenn der Kontext ziemlich kurz ist, Unsloth verspricht nur 2048 Token. Eine so kurze Kontextlänge ist ein Problem für einen Großteil der Trainingsdaten, vor allem DS1 enthält viele lange ABAP-Dateien welche nicht problemlos aufgeteilt werden können. Aus diesem Grund wurde die Ausführung stattdessen auf Google Colab und eine virtuelle Maschine von Google Compute Engine verlegt, dort wurden sowohl eine Nvidia Tesla T4 mit 16GB VRAM, als auch eine Nvidia L4 mit 24GB VRAM genutzt.

4.1 Optimierung

Bei der Optimierung wurde mit DS3 begonnen, danach DS2 und schließlich DS1 betrachtet. Diese Reihenfolge wurde gewählt, da so zuerst mit den kleineren Datensätzen gearbeitet wird, und erste Erfahrungen anhand dieser kleineren, schnelleren Fine-Tuning Prozesse gesammelt werden können, welche dann bei der Optimierung der Größeren helfen. Dadurch werden mehrere Versionen mit den kleinen Modellen durchlaufen um die Parameter kennenzulernen, sodass der große Datensatz im ersten Versuch erfolgreich funktioniert. Die Versionen werden zu Beginn manuell bewertet und mit einander verglichen, die beste Version jedes Modells wird anschließend durch den Benchmark ausgiebig und methodisch geprüft.

Als grober Vergleich ist in Abbildungen 7 und 8 jeweils eine beispielhafte Antwort für das Prompt "write an abap loop that counts to 10" von dem Basis-Modell und dem Instruct-Modell von Meta Llama3:8B auf diese einfache Anfrage gegeben. Hier ist zu sehen, dass das Basis-Modell die Aufgabe nicht erfüllt und stattdessen den Text versucht weiterzuführen. Das Instruct-Modell stattdessen gibt eine gute Antwort.

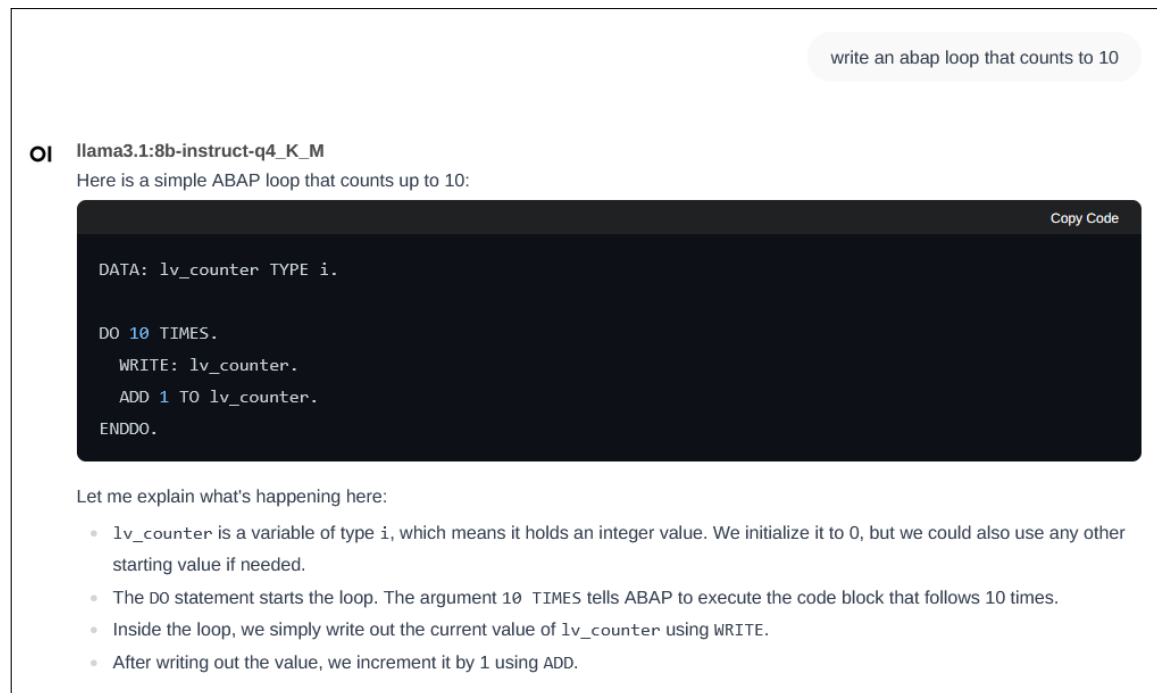


Abbildung 8 Beispielhafte Anfrage an Metas Llama3.1:8b-instruct-q4_K_M



Abbildung 9 Beispielhafte Anfrage an das Modell ds3-v1-llama3.1:8B-q4_K_M

4.2 Datensatz DS3

Im ersten Versuchsaufbau wurde für eine Epoche mit dem Datensatz DS3 trainiert. Das resultierende Modell ist bereits überraschend gut. Es hat schon das Instruct-Prinzip verstanden und antwortet sehr zuverlässig auf die Prompts. Die Qualität der Antwort ist jedoch noch nicht gut, in Darstellung 9 ist beispielhaft dargestellt, wie es vergisst die Variable zu definieren. Durch unstrukturierte Tests scheint es so, als würde das Modell in ca. 80% der Fälle mit Code antworten der Syntaxfehler enthält. Die häufigsten Fehler sind fehlende Variablendefinition und Fehler im Schleifensyntax. Bei Prompts die keine Aufforderung zur Code-Generierung sind, kommt es manchmal zu Artefakten wo Teile des Prompts oder seiner eigenen Antwort wiederholt werden.

Da der Datensatz recht klein ist, und das Ergebnis nicht zufriedenstellend ist, ist zu erwarten dass diese eine Epoche nicht ausreicht um das Modell ausreichend zu verfeinern.

Für die zweite Version wurden drei Epochen durchlaufen, außerdem wurde eine, für LLMs übliche, Code-Formatierung zu den Antworten der Trainingsdaten hinzugefügt damit das LLM diese auch lernt. Das Modell scheint in dieser Version viel über ABAP-Syntax gelernt



Abbildung 10 Beispielhafte Anfrage an das Modell ds3-v2-llama3.1:8B-q4_K_M



Abbildung 11 Beispielhafte Anfrage an das Modell ds3-v3-llama3.1:8B-q4_K_M

zu haben, Abbildung 10 ist eine beispielhafte Antwort dieser Version. Es antwortet auf dieses einfache Beispiel-Prompt nun in ca. 80% der Fälle mit syntaktisch korrektem Code, die häufigsten Fehler scheinen immer noch fehlende Variablendeklarationen zu sein. Die Wiederholungen, welche bei Prompts ohne Aufforderung zur Code-Generierung aufgetreten sind, scheinen in dieser Version nicht mehr vorzukommen. Andere Verhaltensänderungen konnten in dieser Version nicht gefunden werden. Die drei Akzentsymbole (backtick) am Ende des Codeblocks entstanden durch eine leicht inkorrekte Implementierung der Code-Formatierung bei der Formatierung der Trainingsdaten, dies wird in der nächsten Modell-Version behoben.

Die dritte Version dieses Modells hat zehn Epochen durchlaufen. Es kann keine weitere großartige Veränderung der Fähigkeiten festgestellt werden. Das Instruct-Prinzip funktioniert einwandfrei, ABAP Code Generierung funktioniert ähnlich gut wie in der vorherigen Version und auch Fragen nach Themen die nichts mit ABAP zu tun haben, beantwortet diese Version recht gut.

Allgemein hat DS3 bisher bessere Ergebnisse geliefert als erwartet. Es handelt sich um einen sehr kleinen und spezialisierten Datensatz, welcher es trotzdem geschafft hat, dem Modell das Instruct-Prinzip für ABAP-Aufgaben, aber auch für allgemeine Themen, beizubringen.

Insgesamt wurden Modelle basierend auf Datensatz DS3 bis zu zehn Epochen trainiert,

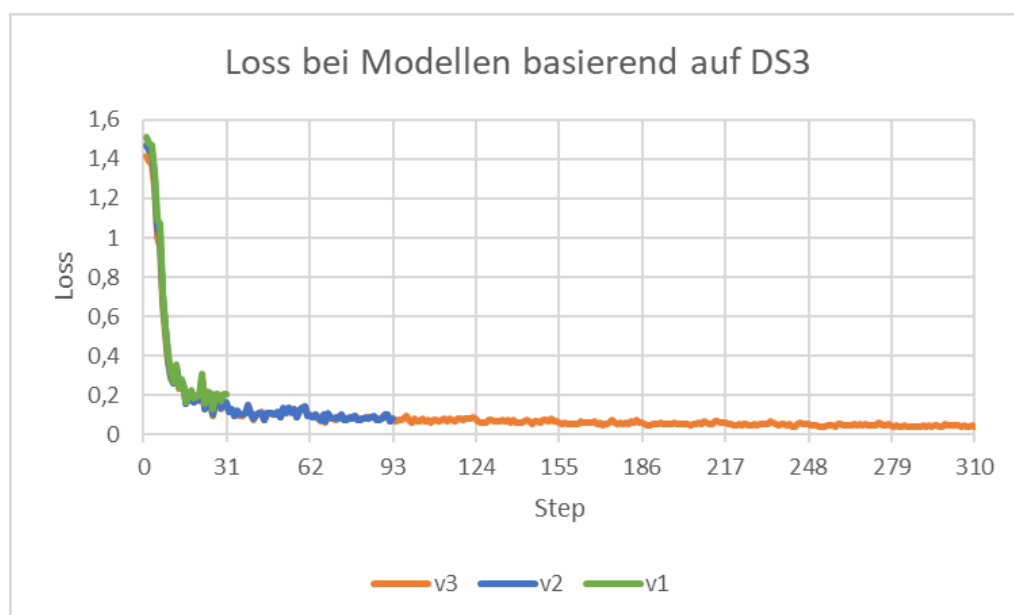


Abbildung 12 Darstellung vom Loss der verschiedenen Modellversionen beim Fine-Tuning mit Datensatz DS3

dabei wurden maximal 310 Trainingsschritte durchlaufen. Das Training wurde auf einer Tesla T4 mit 16GB VRAM durchgeführt, es hat durchschnittlich 7GB VRAM genutzt, beim Modellexport wurde kurzzeitig ein Höchstwert von 13.5GB vRAM erreicht. Die Trainingsdauer für Modelleversionen eins, zwei und drei betrug 3, 10 und 30 Minuten. Der berechnete Loss dieser Durchläufe ist in Abbildung 12 dargestellt, ganz klar zu erkennen ist, dass ein Plateau nach ungefähr 30 Schritten erreicht ist. Von dort an sinkt der Loss nur noch sehr langsam, bei Schritt 30 ist ein Wert von ca. 0.16 erreicht, dieser sinkt bis Schritt 90 nur auf ca. 0.08 und bis Schritt 310 auf 0.04.

Besonders Interessant an allen Modellversionen die anhand von DS3 trainiert wurden ist, dass die Antworten eine erkennbare Struktur aufweisen, welche auf den kleinen, spezialisierten Datensatz zurückzuführen ist. Das Modell fasst sich sehr kurz und gibt keine zusätzliche Erklärungen, wie man das von Instruct-Modelle gewöhnt ist. Wenn das Prompt eine Antwort mit Code verlangt, wird diese ausschließlich Code enthalten, wenn allgemeinere Fragen oder Instruktionen gestellt werden wird nahezu immer in nur einem Satz geantwortet. Außerdem fällt bei Prompts zur ABAP Generierung auf, dass entweder nur der geforderte Teil, oder ein ganzer Report generiert wird, niemals andere ABAP Dateien. Bei direkter Nachfrage nach Funktionsbausteinen werden diese üblicherweise generiert, sie enthalten jedoch sehr viel häufiger Syntaxfehler als Reports.

Da manuell kein weiterer Fortschritt festgestellt werden konnte und der Loss ein Plateau erreicht hat, wird an dieser Stelle nicht weiter mit diesem Datensatz trainiert. Stattdessen wird Version drei des Modells nun durch den Benchmark ausgiebig getestet, die Ergebnisse dazu sind in Abschnitt 4.5 zu finden. Nach persönlichem empfinden könnte auch Version zwei ausgewählt werden da sie sehr ähnliche Ergebnisse liefern. Hier wird jedoch Version drei durch den Benchmark geprüft, da der Verlust trotzdem weiter gesunken ist und keine Merkmale von Overfitting zu erkennen sind.



Abbildung 13 Beispielhafte Anfrage an das Modell ds2-v1-llama3.1:8B-q4_K_M

Links zu den Trainingsdurchläufe für alle Modelle basierend auf DS3 sind im Anhang zu finden.

4.3 Datensatz DS2

Für das Training anhand von DS2 wird in der ersten Version wieder nur eine Epoche durchlaufen. Die Ergebnisse von Abschnitt 4.2 haben zwar gezeigt dass eine Epoche dort nicht ausgereicht hat, der Grund hierfür steht jedoch noch nicht fest. Es ist möglich, dass der Datensatz DS3 einfach zu klein war um innerhalb von einer Epoche genügend zu lernen. Ein Modell basierend auf DS2 durchläuft in einer Epoche mehr Schritte als eines basierend auf DS3 in drei Epochen. Da Abschnitt 4.2 gezeigt hat, dass ein Modell basierend auf DS3 nach drei Epochen gute Ergebnisse liefert, wird hier in der ersten Version erst einmal eine Epoche durchlaufen werden um festzustellen, ob die Wiederholung der Daten durch Epochen, oder die höhere Anzahl an Trainingsschritten für die Verbesserung verantwortlich ist.

Die erste Modellversion die anhand von DS2 trainiert wurde zeigt erstaunlich gute Ergebnisse in ABAP, das Beispieldprompt “write an abap loop that counts to 10” wird nahezu immer richtig beantwortet. Das etwas schwerere Prompt “print odd numbers until 20” kann, wie in Abbildung 13, manchmal erfolgreich beantwortet werden, manchmal treten jedoch Syntaxfehler, vor allem in Schleifendefinitionen, auf. Auch oft enthalten die Antworten hier Endlosschleifen, das Modell generiert viele Do-Schleifen ohne angegebene Grenze. Teilweise wird die Abbruchbedingung komplett weggelassen und teilweise führen inkorrekte If-Abfragen innerhalb der Schleife zu Endlosschleifen.

Bei Prompts die nichts mit ABAP zu tun haben generiert das Modell hier trotzdem manchmal ABAP Code um diese zu beantworten, teilweise wird hier eine angebrachte Antwort als ABAP Code dargestellt und teilweise werden Antworten mit ABAP Code komplett ohne Zusammenhang zum Thema generiert. Dies ist in Abbildung 14 dargestellt.

Die erste Version dieses Modells hat gezeigt, dass der Syntax hier auch mit nur einer Epoche ein ähnliches Niveau erreicht, wie mit mehreren Epochen bei Datensatz DS3. Dementsprechend scheint die Anzahl der Trainingsschritte wichtiger zu sein als die Wiederholung der Daten. Es fällt jedoch auf, dass der Loss hier weit höher ist, als der, der DS3 Modelle. In gewissem Umfang ist dies zu erwarten, da der Datensatz viel größer ist und vor allem

Tell me a random fun fact about the Roman Empire

ds2-v1-llama3.1:8b-Q4_K_M

```

abap
Data: lv_fact TYPE string.

lv_fact = |The Roman Empire was one of the most powerful and influential empires in history, ruling over

WRITE: / lv_fact.

```

Abbildung 14 Beispielhafte Anfrage an das Modell ds2-v1-llama3.1:8B-q4_K_M

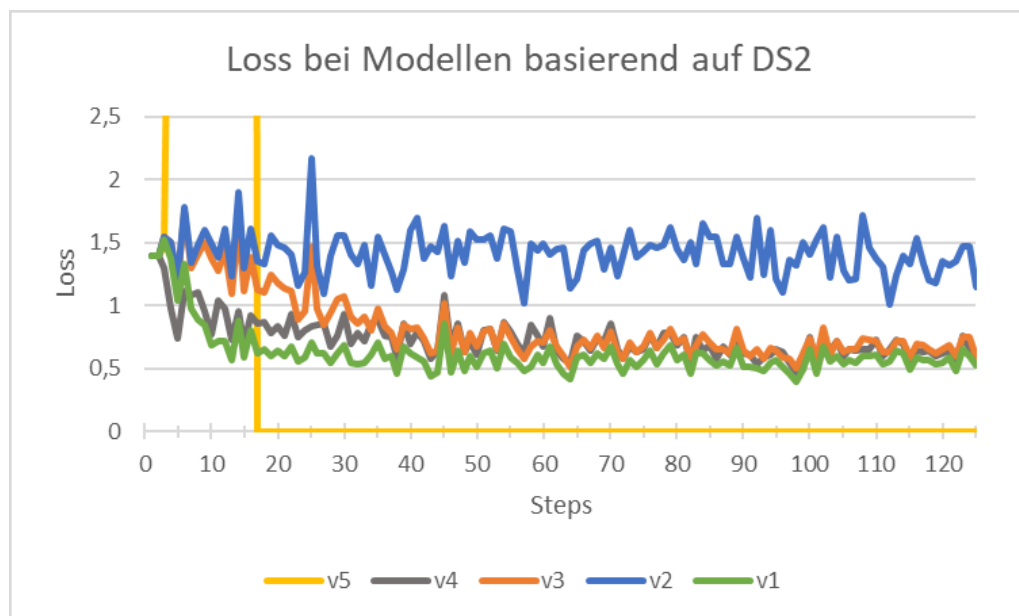


Abbildung 15 Darstellung vom Verlust der verschiedenen Modellversionen beim Fine-Tuning mit Datensatz DS2

weil die Einträge sich in DS2 mehr voneinander unterscheiden als in DS3. Trotzdem ist es möglich, dass durch eine andere Lernrate ein tieferer Verlust erreicht werden kann.

Um den Einfluss der Lernrate zu untersuchen werden Modelleversionen zwei, drei, vier und fünf trainiert. Modell Version eins hat die Standard-Lernrate von 0.0002, Version zwei und drei haben tiefere mit 0.000002 und 0.00002, Version vier und fünf haben höhere mit 0.002 und 0.02. So wird ein Wertebereich von zwei Größenordnungen um den Wert der ersten Version geprüft. Der Verlust dieser Modelle wird zusammen in Abbildung 15 dargestellt und mit Version eins verglichen.

Version zwei hat mit einer Lernrate von 0.000002 definitiv zu wenig gelernt, es antwortet auf einfache Instruktionen zur ABAP Code Generierung in nur ca. 10% der Fälle richtig, meistens antwortet das Modell hier mit Python oder nur mit einem Schleifenkopf. In weiteren 10% der Fällen wird zusammenhangloser Text generiert. Der Verlust in Abbildung 15 zeigt eindeutig, dass diese Version nicht genügend gelernt hat. Das Fine-Tuning müsste viel mehr Schritte durchlaufen um einen ähnlichen Loss zu erreichen wie die erste

write an abap loop that counts to 10

```
OI ds2-v5-llama3.1:8b-Q4_K_M
#HeaderCodeactionDate RuntimeObjectURLExceptionException###_TisgMapsInitialied#/*
$LANG#iParam##GuidIdstyleType#RuntimeObject#CppClass#####tolua##actionDate#gc#####BitFields#####URLEx
ception#####HeaderCode##### addCriterion_Tis####togroup/*
#
RuntimeObject#####actionDate##$LANGInitialied###GuidIdCppClass#iParamstyleTypegMaps##.scalablytypedIIK#####
actionDatetolua##URLExceptionException##KANJl#####HeaderCode#####_Tis#####URLExceptionException##gc#####$LANG##Runtime
ObjectInitialiedCppClass RuntimeObject##togroup.scalablytyped#####gMaps
```

Abbildung 16 Beispielhafte Anfrage an das Modell ds2-v5-llama3.1:8B-q4_K_M

Version.

Version drei hat mit einer Lernrate von 0.00002 nahezu das gleiche Verhalten wie Modell Version 1. Der einzige auffällige Unterschied ist, dass Version drei nur sehr selten die Code-Formatierung verwendet wohingegen Version eins diese nahezu immer benutzt wenn ABAP Code vorkommt. Die Ähnlichkeit der Modelle ist auch im Loss zu sehen, Version drei hat etwas länger gebraucht, jedoch trotzdem schnell das gleiche Plateau erreicht wie die erste Version.

Version vier hat mit einer Lernrate von 0.002 ein extrem ähnliches Verhalten wie die erste Modellversion, es konnten keinerlei Unterschiede festgestellt werden. Auch hier zeigt der Loss, dass Version vier das gleiche Plateau erreicht hat wie die erste Version.

Version fünf ist mit einer Lernrate von 0.02 komplett nutzlos geworden, es generiert, ohne jemals zu stoppen, sinnlosen Text wie in Abbildung 16 dargestellt. Der Loss zeigt diese Probleme im Lernprozess, er steigt anfangs extrem stark bis zu einem Maximum von ca. 42.3 und fällt kurz darauf auf 0.0, wo er für den Rest des Trainings bleibt.

Die Betrachtung der Modelleversionen zwei, drei, vier und fünf haben gezeigt, dass die empfohlene Lernrate gut gewählt ist, aber auch, dass der exakte Wert nicht sonderlich wichtig ist. Eine große Reichweite von Lernraten werden das gleiche Plateau, in einer ähnlichen Anzahl an Schritten, erreichen. Welche Lernrate genau ausgewählt wird macht dementsprechend keinen großen Unterschied, es sei denn man wählt einen sehr falschen Wert, wobei dies in der Auswertung dann schnell festgestellt werden kann.

Darauf basierend wird in Version sechs des Modells mit der ursprünglichen Lernrate von 0.0002 weitergearbeitet. Version sechs wird fünf Epochen, und damit insgesamt 625 Schritte, durchlaufen um zu sehen ob es auf diese Weise weiter lernt. Der Loss ist in Abbildung 17 dargestellt, und zeigt eine abrupte Abnahme nach jeder Epoche. Die Qualität der ABAP Code Generation hat sich, gefühlt, kaum verändert. Instruktionen zu anderen Themenbereichen als der ABAP Code Generierung werden in dieser Version fast ausschließlich durch ABAP Code beantwortet, beispielhaft zu sehen in Abbildung 18. Erstaunlicherweise ist die Idee hinter der Antwort in den meisten Fällen gut und der ABAP Code oft ausführbar.

Insgesamt wurden sechs Modell Versionen mit Datensatz DS2 trainiert, das Training wurde auf einer Nvidia Tesla T4 mit 16GB VRAM durchgeführt und hat durchschnittlich 10.2GB

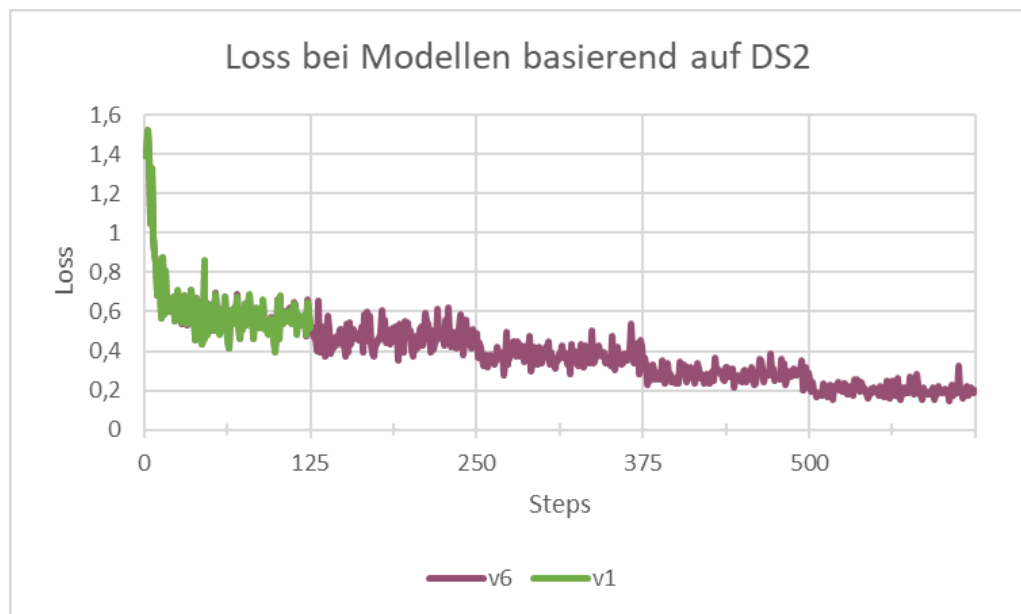


Abbildung 17 Darstellung vom Loss von Version eins und Version sechs beim Fine-Tuning mit Datensatz DS2

What are 5 creative things I could gift my friend for their birthday.

ds2-v6-llama3.1:8b-Q4_K_M

```
abap                                     Copy Code
DATA: lr_string TYPE string.

lr_string = 'A handmade card'.
WRITE: / lr_string.
lr_string = 'Tickets to their favorite show or concert'.
WRITE: / lr_string.
lr_string = 'A personalized book'.
WRITE: / lr_string.
lr_string = 'An experience like a cooking class together, or a pottery workshop'.
WRITE: / lr_string.
lr_string = 'A surprise party with their closest friends'.
WRITE: / lr_string.
```

Abbildung 18 Beispielhafte Anfrage an das Modell ds2-v6-llama3.1:8B-q4_K_M

write an abap loop that counts to 10

✎ □

OI

ds1-v1-llama3.1:8b-Q4_K_M

abap
Copy Code

```

REPORT z_test_for.

DO 5 TIMES.
    WRITE: / sy-index.
ENDDO.

DO 3 TIMES WHILE sy-index < 10.
    WRITE: / sy-index.
ENDDO.

DO 1 UNTIL sy-index = 7.
    IF sy-index > 4 AND sy-index <= 6. "AND or OR
        CONTINUE.
    ENDIF.
    WRITE: / sy-index.
ENDDO.

LOOP AT i FROM 3 TO 8 WHILE sy-tabix < 10. "sy-tabix ist Zeile in Tabelle
    WRITE: / sy-index.
ENDLOOP.
```

Abbildung 19 Beispielhafte Anfrage an das Modell ds1-v1-llama3.1:8B-q4_K_M

VRAM genutzt, beim Modellexport wurde kurzzeitig ein Höchstwert von 13.5GB VRAM erreicht. Die Trainingsdauer betrug knapp 20 Minuten für Versionen eins bis fünf, und bei Version sechs knapp 90 Minuten. Es hat sich gezeigt, dass die Lernrate nur in Extremfällen nennenswerten Einfluss auf das Ergebnis des Fine-Tunings hat. Da in Version sechs der Loss zwar bei jeder Epoche weiter gesunken ist, die Antwortqualität jedoch nicht nennenswert besser geworden ist, werden für diesen Datensatz keine höheren Epochenzahlen mehr getestet. Stattdessen wird Version sechs des Modells nun durch den Benchmark ausgiebig getestet, die Ergebnisse dazu sind in Abschnitt 4.5 zu finden.

Links zu den Trainingsdurchläufen für alle Modelle basierend auf DS2 sind im Anhang zu finden.

4.4 Datensatz DS1

Für das Training anhand von DS1 wird in der ersten Version nur eine Epoche durchlaufen. Das resultierende Modell ist unerwartet schlecht, es löst die einfache Aufforderung “write an abap loop that counts to 10” nur in ca. 25% der Versuche. Teilweise werden stattdessen Leere Reports, Hello-World-Programme oder sonstige Test-Programme generiert. Oft versucht es sehr lange und komplexe Antworten zu generieren, diese enthalten dann nahezu immer Syntaxfehler und, auch wenn Syntaxfehler ignoriert werden, ergeben sie semantisch meistens keinen Sinn. Dies ist beispielhaft in Abbildung 19 zu sehen, das Modell versucht vier Schleifen zu nutzen, um bis zehn zu zählen, und baut dabei in drei Schleifen Syntaxfehler ein.

Der Verlust ist in Abbildung 20 dargestellt und mit denen von DS2-v6 und DS3-v3 vergli-

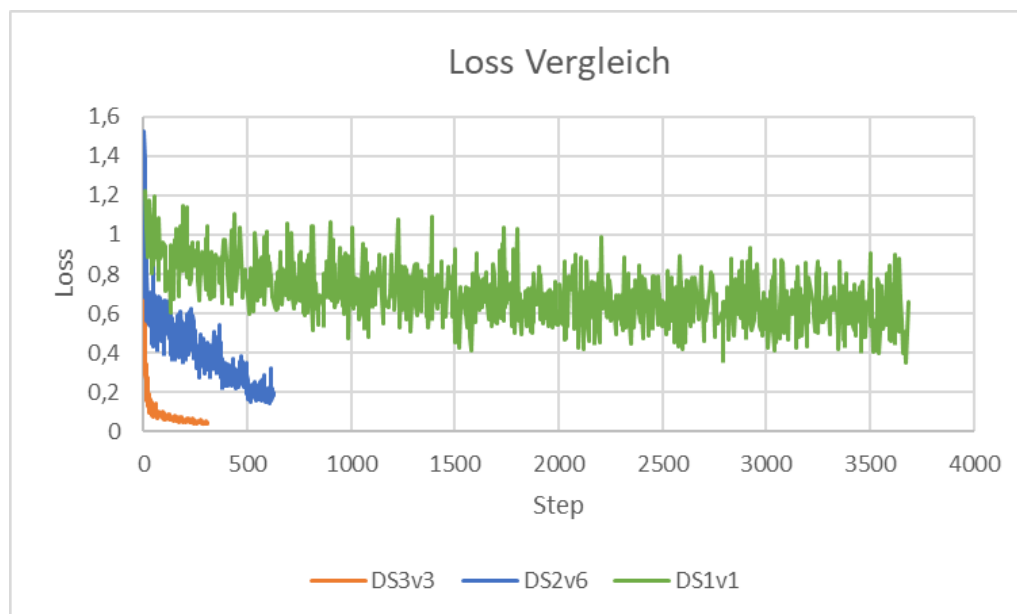


Abbildung 20 Darstellung des Loss im Training von den letzten Modellversionen jedes Datensatzes

chen. Es fällt auf, dass der Verlust von DS1-v1 über den gesamten Trainingszeitraum viel höher bleibt, als die der anderen Modelle, obwohl diese viel kürzer trainiert wurden. Außerdem fällt auf, dass zwar alle drei Modelle ein Plateau erreicht²⁶ haben, DS1-v1 jedoch bei weitem die größten Abweichungen, um das Plateau herum, hat.

Die Probleme der ersten Version dieses Modells deuten eindeutig auf den grundlegenden Datensatz DS1 als Verursacher der Probleme hin. Die sehr langen und komplexen Antworten des Modells können nur auf die Trainingsdaten zurückgeführt werden, genauso das häufige Vorkommen von Test- und Hello-World-Programmen. Aus diesem Grund wird an dieser Stelle keine weitere Modell Version anhand von DS1 trainiert. Die starke Streuung des Loss ließe sich vermutlich verringern, indem eine niedrigere Lernrate genutzt wird, der erneute Trainingsaufwand ist dies hier jedoch nicht Wert, da die Trainingsdaten vermutlich ein größeres Problem darstellen.

Es wurde also nur eine Modell Version mit Datensatz DS2 trainiert. Das Training wurde diesmal auf einer Nvidia L4 mit 24GB VRAM durchgeführt da mit nur 16GB ein Out-Of-Memory Fehler aufgetreten ist, vermutlich da die einzelnen Einträge der Trainingsdaten hier länger sind als in den anderen Datensätzen. Es wurden durchschnittlich 20.2GB VRAM genutzt und ein Höchstwert von 20.4GB VRAM erreicht. Die Trainingsdauer betrug knapp unter 18 Stunden.

Ein Link zum Trainingsdurchlauf von Modellversion eins basierend auf DS1 ist im Anhang zu finden.

²⁶Bei DS2-v6 ist das Plateau hier nicht offensichtlich erkennbar, in Abbildung 17 ist dieses jedoch gut zu erkennen.

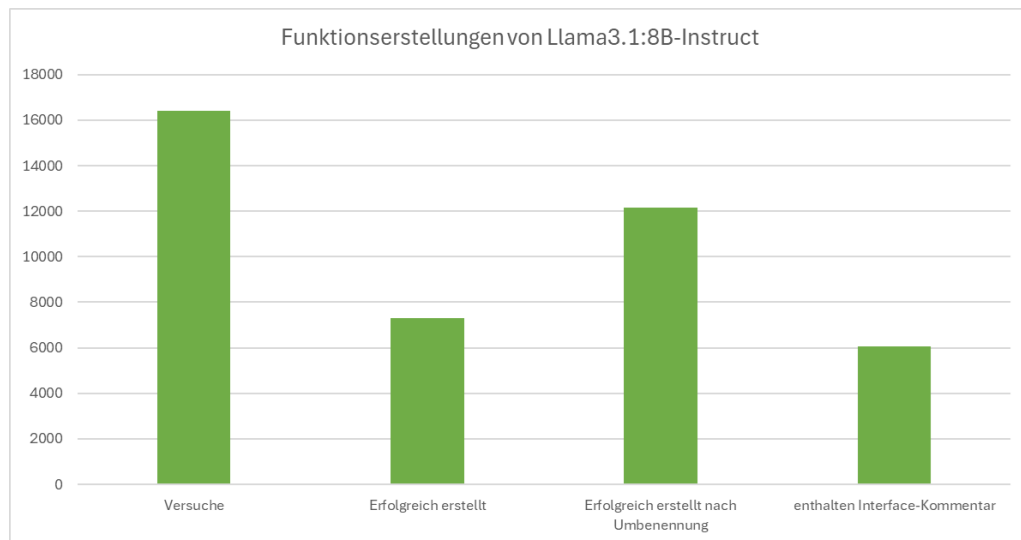


Abbildung 21 Benchmarkergebnis: Funktionserstellungen von Metas Llama3.1:8B-Instruct

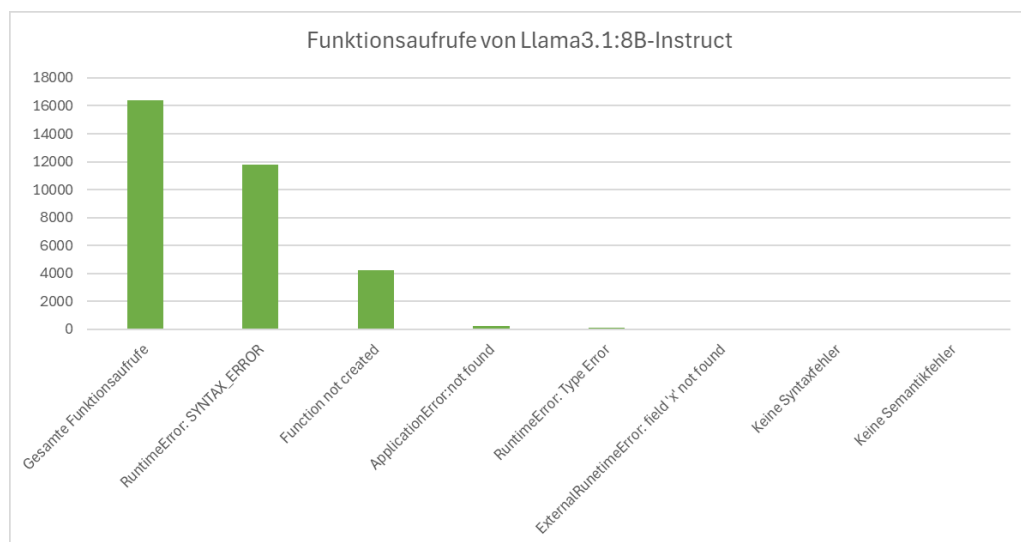


Abbildung 22 Benchmarkergebnis: Funktionsaufrufe von Metas Llama3.1:8B-Instruct

4.5 Untersuchung der Ergebnisse

Zum Vergleich werden zunächst die Benchmarkergebnis von Llama3.1:8B-Instruct betrachtet. Abbildung 21 zeigt die Funktionserstellungen, hier ist zu erkennen, dass von den 16400 Versuchen 7321 Funktionen erfolgreich erstellt worden sind. Da bei vielen Antworten schon der Funktionsname, über die ABAP-GUI, nicht zulässig wäre, wird dieser hier automatisch geändert um nicht direkt einen Großteil der Antworten zu verwerfen. Auf diese Weise können 12165 Funktionen erfolgreich erstellt werden. Insgesamt haben nur 6053 Funktionen einen Interface-Kommentar, da dieser für die Parameterübergabe der Unit-Tests genutzt wird, ist zu erwarten, dass die Tests ohne Interface-Kommentar auf jeden Fall fehlschlagen. Abbildung 22 zeigt die Funktionsaufrufe, hier ist zu erkennen, dass 11797 Funktionsaufrufe aufgrund eines Syntax-Errors abgebrochen sind. 4235 Aufrufe konnten nicht ausgeführt werden, da die Funktion nicht erstellt worden ist, andere Probleme machen hier nur einen sehr kleinen Teil aus. Kein Funktionsaufruf konnte erfolgreich, ohne Syntax- oder Semantiktfehler ausgeführt werden.

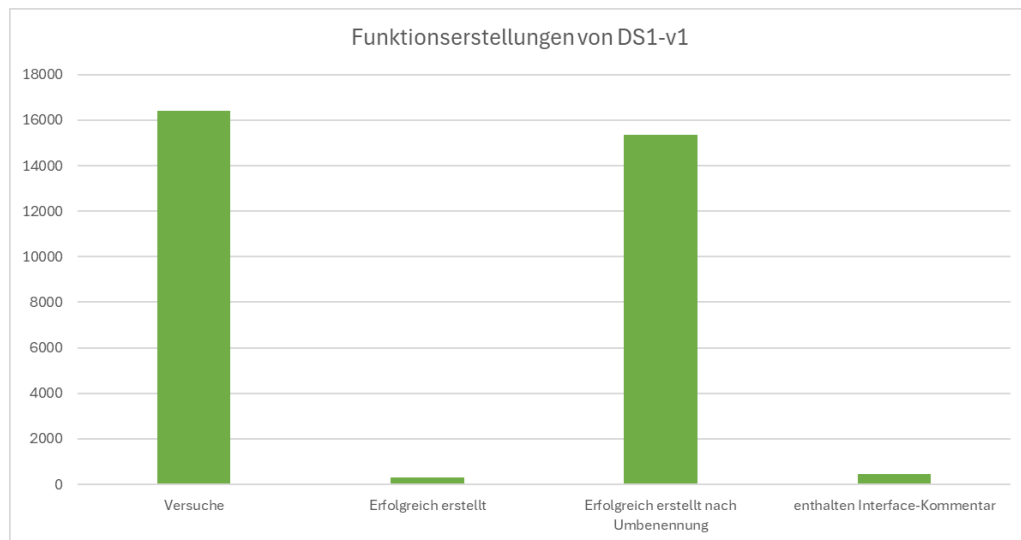


Abbildung 23 Benchmarkergebnis: Funktionserstellungen von DS1-v1

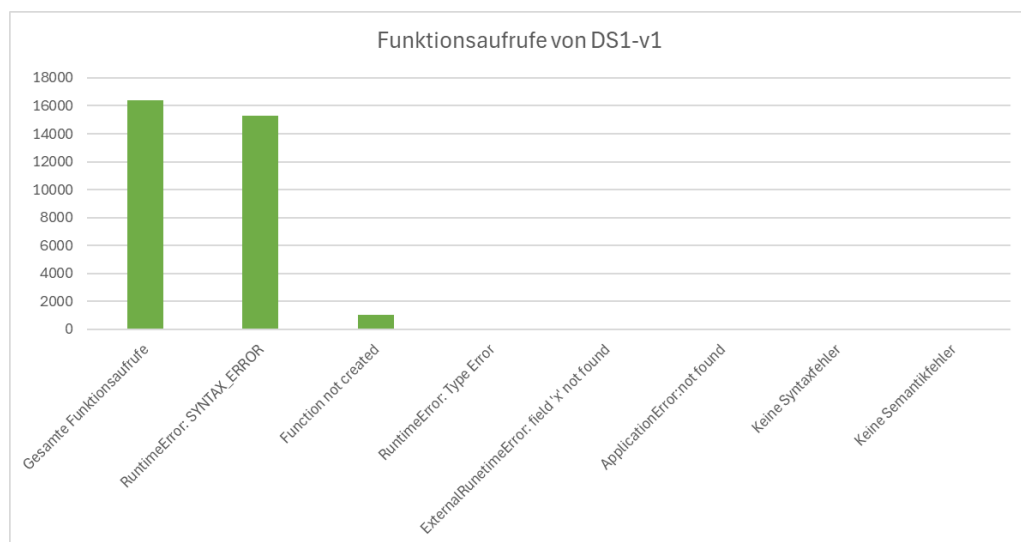


Abbildung 24 Benchmarkergebnis: Funktionsaufrufe von DS1-1

Die Benchmarkergebnis von DS1-v1 zeigen hier schlechtere Ergebnisse, Abbildung 23 zeigt die Funktionserstellungen, es ist zu erkennen, dass hier nur 284 Funktionen erfolgreich erstellt werden konnten. Nach der Funktionsnamenänderung konnten hier 15344 erstellt werden. Insgesamt hatten auch nur 464 Funktionen einen Interface-Kommentar. Abbildung 24 zeigt die Funktionsaufrufe, hier ist zu erkennen, dass 15303 Funktionsaufrufe aufgrund von Syntaxfehlern abgebrochen wurden. 1056 Aufrufe konnten nicht ausgeführt werden, da die Funktion nicht erstellt worden ist. Auch hier sind andere Fehler nur in sehr kleinen Mengen aufgetreten und insgesamt konnte kein Funktionsaufruf fehlerfrei durchgeführt werden.

Der Benchmark von DS2-v6 zeigt ähnlich schlechte Ergebnisse. In Abbildung 25 sind die Funktionserstellungen dargestellt, diese zeigen, dass ohne Eingriff nur 421 Funktionen erstellt werden konnten. Nach der Funktionsnamenänderung konnten 16271 erstellt werden. Es enthielten nur 46 der Funktionen einen Interface-Kommentar. Abbildung 26 zeigt die Funktionsaufrufe. Es ist erkennbar, dass, mit 16269 von 16400, nahezu alle aufgrund

von Syntaxfehlern fehlgeschlagen sind. Auch hier wurde kein Aufruf erfolgreich durchgeführt.

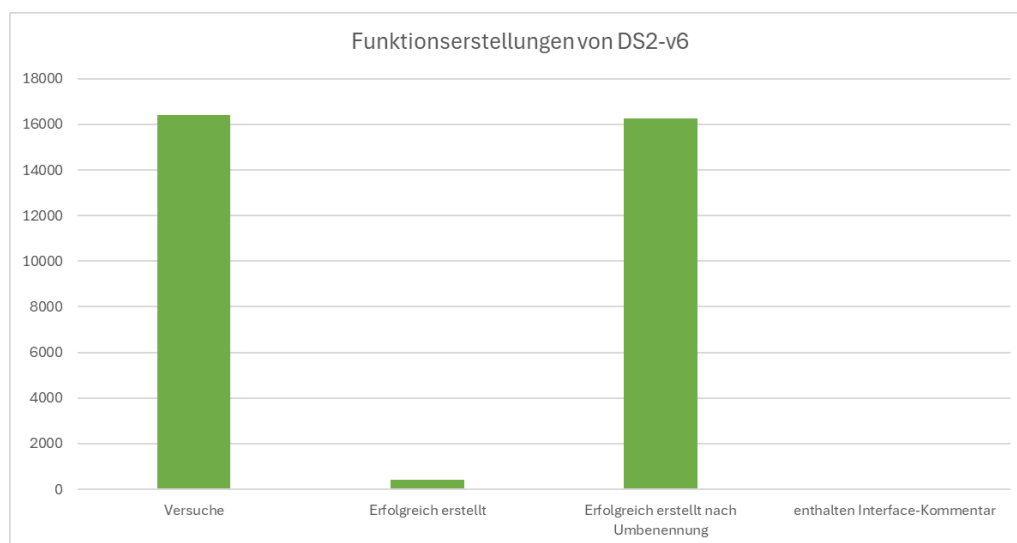


Abbildung 25 Benchmarkergebnis: Funktionserstellungen von DS2-v6

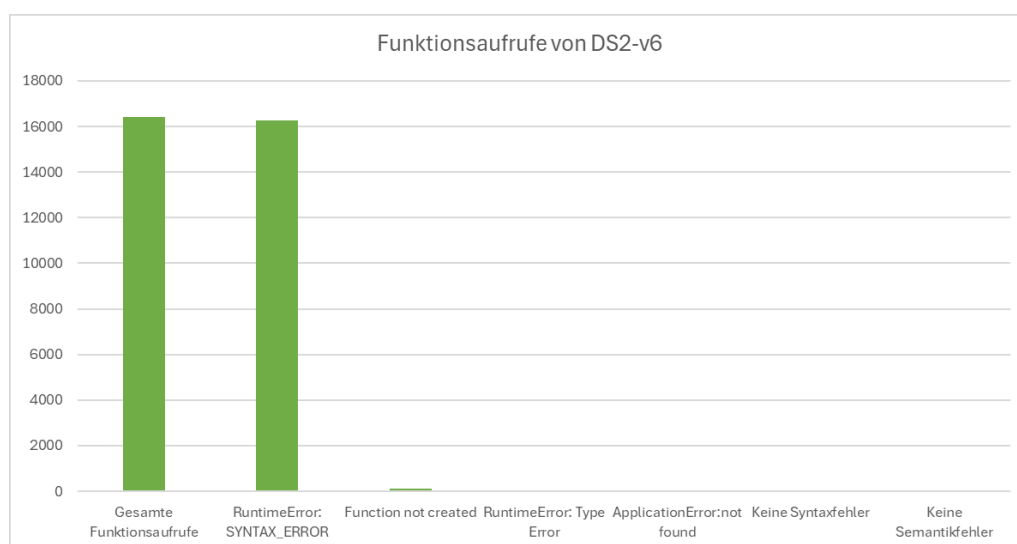


Abbildung 26 Benchmarkergebnis: Funktionsaufrufe von DS2-6

Eine Betrachtung der Ergebnisse des Benchmarks von Modell DS3-v3 zeigt ebenso schlechte Ergebnisse. Abbildung 27 zeigt die Erstellung der Funktionen, es konnten nur 48 Funktionen ohne Eingriffe erstellt werden, mit Umbenennung des Funktionsnamens konnten 13244 erstellt werden. Keine Funktion enthält hier einen Interface-Kommentar. In Abbildung 28 werden die Ausführungen der Funktionen dargestellt. Bei 13244 Funktionsaufrufen wurde dieser abgebrochen wegen Syntaxfehlern, 3156 konnten nicht ausgeführt werden, da die Funktion nicht erstellt werden konnte. Hier gab es gar keine anderen Fehlerquellen und auch keine erfolgreichen Testdurchläufe.

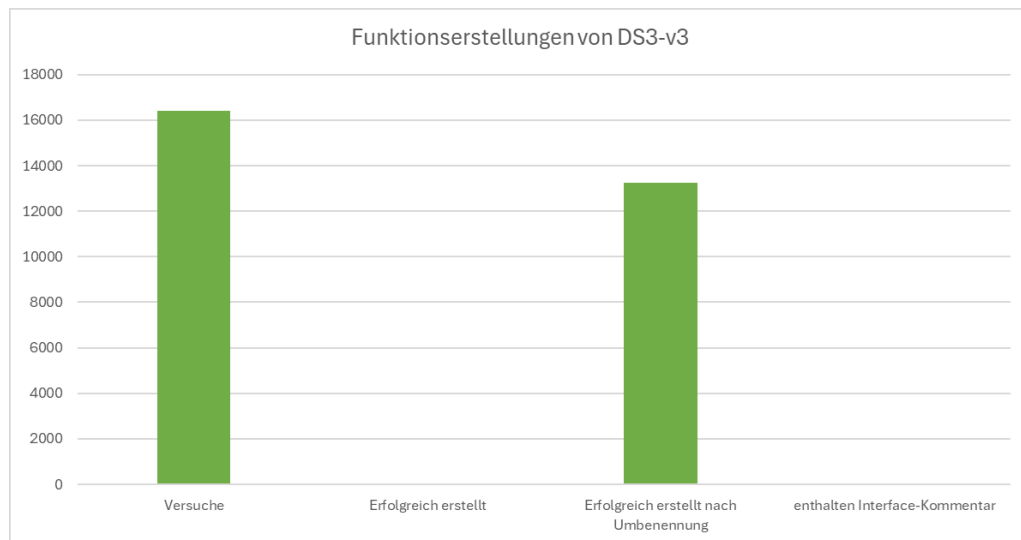


Abbildung 27 Benchmarkergebnis: Funktionserstellungen von DS3-v3

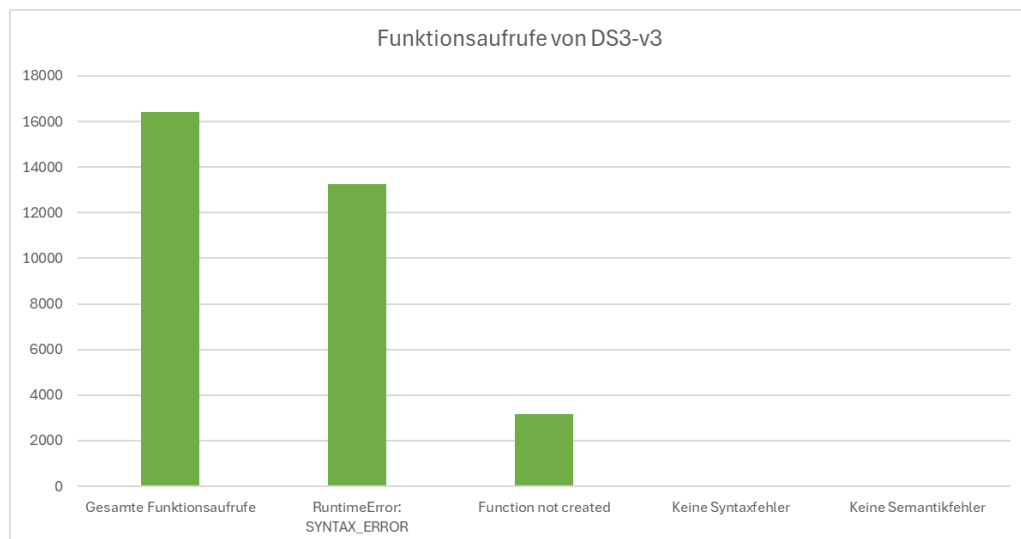


Abbildung 28 Benchmarkergebnis: Funktionsaufrufe von DS3-3

Insgesamt zeigen also alle Modelle keine nennenswerten ABAP-Fähigkeiten. Das häufigste Problem ist in allen Modellen, dass kein zulässiger ABAP-Funktionsname angegeben wird, aufgrund des Auswertungsverfahrens entsteht dieser Fehler auch wenn die Zeile, die den Namen festlegen würde, fehlt. Außerdem werden bei allen Modellen nur sehr selten Interface-Kommentare generiert, da der Benchmark über Unit-Tests funktioniert werden jedoch bei allen Funktionen Parameter benötigt. Ohne Parameter sind erfolgreiche Durchführungen also nicht möglich. Die häufigsten Laufzeitfehler sind Syntaxfehler, bei dem Vergleichsmodell Meta Llama3.1:8B-Instruct entstanden vergleichsweise die wenigsten, mit Syntaxfehlern in 71% der Tests. Bei DS2-v6 wurden diese in 99% der Testdurchläufe festgestellt, dort kommen sie am häufigsten vor.

5 Fazit und Ausblick

5.1 Fazit

Aus diesen Ergebnissen lassen sich folgende Antworten auf die anfangs gestellten Forschungsfragen ermitteln:

RQ1: Mit welchen Metriken kann die Effizienz und Genauigkeit eines generierten ABAP-Codes sinnvoll und praxisnah gemessen werden?

Die Auswertung von LLM generiertem ABAP Code ist schwierig, unter anderen, da ABAP viele Komponenten hat die ausgewertet werden müssen, da die Antwortgenerierung von LLMs durch den Zufall beeinflusst wird und da das SAP-Ökosystem, indem ABAP ausgeführt wird, viele Einschränkungen hat. Hier wird ein automatisiertes Framework zur Ausführung von ABAP-Funktionsbausteinen genutzt um LLM-generierten ABAP Code auszuführen, und so auf syntaktische und semantische Korrektheit zu prüfen. Dieses nutzt den beliebten Python LLM-Benchmark HumanEval und passt ihn für die ABAP-Code Generation an. Durch die Verwendung des bekannten Benchmarks werden viele Sprachfunktionen abdeckt und eine Vergleichsmöglichkeit der Ergebnisse mit anderen Programmiersprachen ermöglicht. Durch die Automatisierung der Ausführung kann der Einfluss des Zufalls minimiert werden, da durchschnittliche Ergebnisse betrachtet werden können. Außerdem werden menschliche Fehler in der Auswertung minimiert, dies spielt gerade bei großen Auswertungen eine wichtige Rolle. Insgesamt können so die durchschnittlichen Anzahlen an erfolgreich gelösten Aufgaben, sowie die Gründe für entstandene Fehler betrachtet werden.

RQ2: Wie kann das Fine Tuning die Effizienz und Genauigkeit der ABAP-Code-Generierung durch das Llama3-Modell verbessern?

Allgemein kann Fine-Tuning einen großen Einfluss auf das Verhalten von LLMs haben, viele Untersuchungen haben gezeigt, dass es durch Fine-Tuning möglich ist, die Fähigkeiten von LLMs in bestimmten Programmiersprachen stark zu verbessern. Außerdem ist klar, dass LLMs bereits intelligent genug sind um viele Programmieraufgaben, in beliebten Programmiersprachen wie Python, zu lösen.

Der Grundgedanke hinter dieser Untersuchung war, dass LLMs derzeit schlecht in ABAP sind, da ihnen Trainingsdaten dazu fehlen und sie für beliebte Programmiersprachen angepasst werden, aber nicht für ABAP. An dieser Stelle sollte Llama3 hier durch das Fine-Tuning verbessert werden und zusätzliche Trainingsdaten erhalten, die auf ABAP spezialisiert sind und dem Modell erlauben, seine allgemeinen Code-Verständnis Fähigkeit auf ABAP anzuwenden.

Leider konnte dies praktisch nicht umgesetzt werden. Es wurden zehn Llama3-8B Modelle durch Fine-Tuning trainiert, diese haben unterschiedliche Fähigkeiten in der ABAP-Code Generation gezeigt, es konnte jedoch kein Vorteil gegenüber dem Standard Llama3-8B-Instruct Modell erreicht werden. Grund hierfür sind vermutlich die Trainingsdaten, es konnte nicht genügend hochqualitativer ABAP-Code gefunden werden, der zum Fine-Tuning genutzt werden kann.

RQ3: In welchem Maße ist der Einsatz des spezialisierten Llama3-Modells als alltägliches Hilfsmittel für ABAP-Entwickler sinnvoll und praktikabel?

Derzeit ist die Nutzung von spezialisierten Llama3-Modellen für die ABAP-Entwicklung nicht sinnvoll. Die Modelle sind nicht gut genug um nützliche Antworten zu geben, sie enthalten nahezu immer sowohl Syntax-, als auch Semantikfehler und verursachen mehr Probleme als sie lösen.

5.2 Diskussion möglicher Limitationen und zukünftiger Forschungsrichtungen

Wie in vorherigen Abschnitten bereits beschrieben ist die Verfügbarkeit von hochqualitativen ABAP Trainingsdaten die größte Limitation dieser Untersuchung und wird auch zukünftige Untersuchungen in diese Richtung limitieren. Es gibt viel weniger ABAP Entwickler als für beliebtere Sprachen wie Python, dementsprechend ist auch die Menge der verfügbaren Daten geringer. Außerdem werden ABAP Daten üblicherweise in geschlossenen System innerhalb von Unternehmen verwaltet, es gibt nahezu keine ABAP Entwickler die ihre ABAP Dateien öffentlich ins Internet stellen. Die Daten die bereits auf Github oder ähnlichen Plattformen zu finden sind, sind oft sehr komplex und lang, was diese zu schlechten Trainingsdaten macht. Um bessere Fine-Tuning Ergebnisse mit ABAP zu erreichen muss viel mehr Aufwand in die Entwicklung der Trainingsdaten investiert werden, einfache Filterungsverfahren werden hier voraussichtlich nicht ausreichen, da die Datenmenge zu gering ist um viel mehr zu filtern als in dieser Ausarbeitung.

Stattdessen könnten Ansätze wie synthetische Datengeneration verfolgt werden, dabei werden häufig LLMs eingesetzt um Trainingsdaten zu generieren, mit Verfahren wie Self-Instruct [Wan+23] wird ein kleiner hochqualitativer Datensatz durch iterative Abänderung vergrößert. Der neu generierte Datensatz kann dann automatisiert gefiltert werden um Qualität zu gewährleisten. Diese Verfahren werden bereit viel eingesetzt, beispielsweise hat Nvidia Nemotron4 340B [Nvi+24] extra für diese Aufgabe entwickelt, in anderen Untersuchungen werden ähnliche Verfahren zur Generierung von Trainingsdaten mit Code genutzt [Yan+24; Maj+24].

Eine weitere nennenswerte Limitation in dieser Untersuchung waren die begrenzten Ressourcen, welche dazu geführt haben, dass Llama3-8B und QLoRA für diese Untersuchung ausgewählt wurden. Diese Einschränkungen haben definitiv Einfluss auf das Ergebnis gehabt, sind jedoch absolut gerechtfertigt, wenn man die hierfür eingesetzten monetären und zeitlichen Ressourcen mit denen vergleicht, die beispielsweise für ein Full-Fine-Tuning an Llama3-405B nötig wären. Außerdem würde auch ein viel größeres Modell anhand von schlechten Trainingsdaten keine optimalen Ergebnisse erzielen. Erst wenn ein sehr guter Datensatz entwickelt wurde, dieser auf größere Modelle skaliert werden kann, und intelligentere Modelle benötigt werden, wäre der Umstieg auf größere Modelle für solch eine Untersuchung sinnvoll.

Literaturverzeichnis

- [BCB16] Dzmitry Bahdanau, Kyunghyun Cho und Yoshua Bengio. *Neural Machine Translation by Jointly Learning to Align and Translate*. arXiv:1409.0473 [cs, stat]. Mai 2016. DOI: 10.48550/arXiv.1409.0473. URL: <http://arxiv.org/abs/1409.0473> (besucht am 5. Aug. 2024).
- [Bro+20] Tom B. Brown u. a. *Language Models are Few-Shot Learners*. arXiv:2005.14165 [cs]. Juli 2020. DOI: 10.48550/arXiv.2005.14165. URL: <http://arxiv.org/abs/2005.14165> (besucht am 6. Aug. 2024).
- [Cas+22] Federico Cassano u. a. *MultiPL-E: A Scalable and Extensible Approach to Benchmarking Neural Code Generation*. arXiv:2208.08227 [cs]. Dez. 2022. DOI: 10.48550/arXiv.2208.08227. URL: <http://arxiv.org/abs/2208.08227> (besucht am 6. Aug. 2024).
- [Cha24] Sahil Chaudhary. *sahil280114/codealpaca*. original-date: 2023-03-22T11:28:54Z. Aug. 2024. URL: <https://github.com/sahil280114/codealpaca> (besucht am 11. Aug. 2024).
- [Che+21] Mark Chen u. a. *Evaluating large language models trained on code*. arXiv:2107.03374 [cs]. Juli 2021. DOI: 10.48550/arXiv.2107.03374. URL: <http://arxiv.org/abs/2107.03374> (besucht am 10. Juni 2024).
- [Det+22] Tim Dettmers u. a. *LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale*. arXiv:2208.07339 [cs]. Nov. 2022. DOI: 10.48550/arXiv.2208.07339. URL: <http://arxiv.org/abs/2208.07339> (besucht am 8. Aug. 2024).
- [Det+23] Tim Dettmers u. a. *QLoRA: Efficient finetuning of quantized llms*. arXiv:2305.14314 [cs]. Mai 2023. DOI: 10.48550/arXiv.2305.14314. URL: <http://arxiv.org/abs/2305.14314> (besucht am 10. Juni 2024).
- [Dub+24] Abhimanyu Dubey u. a. *The Llama 3 Herd of Models*. arXiv:2407.21783 [cs]. Juli 2024. DOI: 10.48550/arXiv.2407.21783. URL: <http://arxiv.org/abs/2407.21783> (besucht am 6. Aug. 2024).
- [Fra+23] Elias Frantar u. a. *GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers*. arXiv:2210.17323 [cs]. März 2023. DOI: 10.48550/arXiv.2210.17323. URL: <http://arxiv.org/abs/2210.17323> (besucht am 8. Aug. 2024).
- [GB16] Cheng Guo und Felix Berkhahn. *Entity embeddings of categorical variables*. arXiv:1604.06737 [cs]. Apr. 2016. DOI: 10.48550/arXiv.1604.06737. URL: <http://arxiv.org/abs/1604.06737> (besucht am 4. Aug. 2024).
- [Guo+24] Daya Guo u. a. *DeepSeek-coder: When the large language model meets programming – the rise of code intelligence*. arXiv:2401.14196 [cs]. Jan. 2024. DOI: 10.48550/arXiv.2401.14196. URL: <http://arxiv.org/abs/2401.14196> (besucht am 10. Juni 2024).

- [Hu+21] Edward J. Hu u. a. *LoRA: Low-rank adaptation of large language models*. arXiv:2106.09685 [cs]. Okt. 2021. DOI: 10.48550/arXiv.2106.09685. URL: <http://arxiv.org/abs/2106.09685> (besucht am 12. Juni 2024).
- [Hug24] HuggingFace. *LoRA*. 2024. URL: https://huggingface.co/docs/peft/main/en/conceptual_guides/lora (besucht am 10. Juni 2024).
- [Kal23] Damjan Kalajdzievski. *A Rank Stabilization Scaling Factor for Fine-Tuning with LoRA*. arXiv:2312.03732 [cs]. Nov. 2023. DOI: 10.48550/arXiv.2312.03732. URL: <http://arxiv.org/abs/2312.03732> (besucht am 10. Aug. 2024).
- [Li+23a] Raymond Li u. a. *StarCoder: may the source be with you!* arXiv:2305.06161 [cs]. Dez. 2023. DOI: 10.48550/arXiv.2305.06161. URL: <http://arxiv.org/abs/2305.06161> (besucht am 10. Juni 2024).
- [Li+23b] Yixiao Li u. a. *LoftQ: LoRA-Fine-Tuning-Aware Quantization for Large Language Models*. arXiv:2310.08659 [cs]. Nov. 2023. DOI: 10.48550/arXiv.2310.08659. URL: <http://arxiv.org/abs/2310.08659> (besucht am 10. Aug. 2024).
- [Lin+24] Yang Lin u. a. *LoRA Dropout as a Sparsity Regularizer for Overfitting Control*. arXiv:2404.09610 [cs]. Apr. 2024. DOI: 10.48550/arXiv.2404.09610. URL: <http://arxiv.org/abs/2404.09610> (besucht am 10. Aug. 2024).
- [Loz+24] Anton Lozhkov u. a. *StarCoder 2 and the stack v2: The next generation*. arXiv:2402.19173 [cs]. Feb. 2024. DOI: 10.48550/arXiv.2402.19173. URL: <http://arxiv.org/abs/2402.19173> (besucht am 12. Juni 2024).
- [Maj+24] Somshubra Majumdar u. a. *Genetic Instruct: Scaling up Synthetic Generation of Coding Instructions for Large Language Models*. arXiv:2407.21077 [cs]. Juli 2024. DOI: 10.48550/arXiv.2407.21077. URL: <http://arxiv.org/abs/2407.21077> (besucht am 17. Aug. 2024).
- [Mik+13] Tomas Mikolov u. a. *Efficient estimation of word representations in vector space*. arXiv:1301.3781 [cs]. Sep. 2013. DOI: 10.48550/arXiv.1301.3781. URL: <http://arxiv.org/abs/1301.3781> (besucht am 4. Aug. 2024).
- [Nvi+24] Nvidia u. a. *Nemotron-4 340B Technical Report*. arXiv:2406.11704 [cs]. Aug. 2024. DOI: 10.48550/arXiv.2406.11704. URL: <http://arxiv.org/abs/2406.11704> (besucht am 17. Aug. 2024).
- [NZ24] Bardia Nadimi und Hao Zheng. *A multi-expert large language model architecture for verilog code generation*. arXiv:2404.08029 [cs]. Apr. 2024. DOI: 10.48550/arXiv.2404.08029. URL: <http://arxiv.org/abs/2404.08029> (besucht am 10. Juni 2024).
- [Ope+24] OpenAI u. a. *GPT-4 technical report*. arXiv:2303.08774 [cs]. März 2024. DOI: 10.48550/arXiv.2303.08774. URL: <http://arxiv.org/abs/2303.08774> (besucht am 29. Juli 2024).
- [Roz+24] Baptiste Rozière u. a. *Code llama: Open foundation models for code*. arXiv:2308.12950 [cs]. Jan. 2024. DOI: 10.48550/arXiv.2308.12950. URL: <http://arxiv.org/abs/2308.12950> (besucht am 10. Juni 2024).

- [Sha20] Noam Shazeer. *GLU Variants Improve Transformer*. arXiv:2002.05202 [cs, stat] version: 1. Feb. 2020. DOI: 10.48550/arXiv.2002.05202. URL: <http://arxiv.org/abs/2002.05202> (besucht am 6. Aug. 2024).
- [Tao+23] Rohan Taori u. a. *Stanford alpaca: An instruction-following LLaMA model*. 2023. URL: https://github.com/tatsu-lab/stanford_alpaca.
- [Tou+23] Hugo Touvron u. a. *Llama 2: Open Foundation and Fine-Tuned Chat Models*. arXiv:2307.09288 [cs]. Juli 2023. DOI: 10.48550/arXiv.2307.09288. URL: <http://arxiv.org/abs/2307.09288> (besucht am 6. Aug. 2024).
- [Vas+23] Ashish Vaswani u. a. *Attention is all you need*. arXiv:1706.03762 [cs]. Aug. 2023. DOI: 10.48550/arXiv.1706.03762. URL: <http://arxiv.org/abs/1706.03762> (besucht am 5. Aug. 2024).
- [Wan+23] Yizhong Wang u. a. *Self-Instruct: Aligning Language Models with Self-Generated Instructions*. arXiv:2212.10560 [cs]. Mai 2023. DOI: 10.48550/arXiv.2212.10560. URL: <http://arxiv.org/abs/2212.10560> (besucht am 17. Aug. 2024).
- [Wey+24] Martin Weyssow u. a. *Exploring parameter-efficient fine-tuning techniques for code generation with large language models*. arXiv:2308.10462 [cs]. Jan. 2024. DOI: 10.48550/arXiv.2308.10462. URL: <http://arxiv.org/abs/2308.10462> (besucht am 10. Juni 2024).
- [Wu+23] Shijie Wu u. a. *BloombergGPT: A Large Language Model for Finance*. arXiv:2303.17564 [cs, q-fin]. Dez. 2023. DOI: 10.48550/arXiv.2303.17564. URL: <http://arxiv.org/abs/2303.17564> (besucht am 7. Aug. 2024).
- [Yan+24] Jiayi Yang u. a. *Synthesizing Text-to-SQL Data from Weak and Strong LLMs*. arXiv:2408.03256 [cs]. Aug. 2024. DOI: 10.48550/arXiv.2408.03256. URL: <http://arxiv.org/abs/2408.03256> (besucht am 17. Aug. 2024).
- [Zhe+24] Qinkai Zheng u. a. *CodeGeeX: A Pre-Trained Model for Code Generation with Multilingual Benchmarking on HumanEval-X*. arXiv:2303.17568 [cs]. Juli 2024. DOI: 10.48550/arXiv.2303.17568. URL: <http://arxiv.org/abs/2303.17568> (besucht am 10. Aug. 2024).

Anhang

Trainingsdurchläufe zu Datensatz DS1

Version 1: https://colab.research.google.com/drive/148Fs_jdLHp-jFw5hL0rDtBKKsT18kC9i

Trainingsdurchläufe zu Datensatz DS2

Version 1: <https://colab.research.google.com/drive/1MxWdYx00gqMLyxXram8mdtvme3Wh9rVu>

Version 2: <https://colab.research.google.com/drive/1gXoeh8fy-HgPaVzgd46170ltNqPv8phb>

Version 3: <https://colab.research.google.com/drive/1Ww-p8gxD7KFQr3jVVV0ectCM9W5WuXSG>

Version 4: <https://colab.research.google.com/drive/19SgJv8ZiEvf6DWttQMAzl0M0j4WmelbR>

Version 5: <https://colab.research.google.com/drive/1M4xqtnUUK06VpGjyDhJIVRAODvI6Nyco>

Version 6: https://colab.research.google.com/drive/1uV_jSy7YxJYc4_Wk5B_JCJiKY1DEtblZ

Trainingsdurchläufe zu Datensatz DS3

Version 1: <https://colab.research.google.com/drive/1ylwJm1BW9pgACgomEoIyWf5my2R80F4c>

Version 2: <https://colab.research.google.com/drive/1Z4UOL8lzT5f0q8P0que13-Egvn1JvWwc?oid=115253896341743503689>

Version 3: <https://colab.research.google.com/drive/1XmElFbSiClIDoHxjhc1SUN0JFiWlUuzF?oid=115253896341743503689>

Datensätze

DS1: <https://huggingface.co/datasets/timkoehne/DS1>

DS2: <https://huggingface.co/datasets/timkoehne/DS2>

DS3: <https://huggingface.co/datasets/timkoehne/DS3>

DS1 Modell

ds1-v1: https://huggingface.co/timkoehne/ds1-v1-llama3.1-8b-Q4_K_M

DS2 Modelle

ds2-v1: https://huggingface.co/timkoehne/ds2-v1-llama3.1-8b-Q4_K_M

ds2-v2: https://huggingface.co/timkoehne/ds2-v2-llama3.1-8b-Q4_K_M

ds2-v3: https://huggingface.co/timkoehne/ds2-v3-llama3.1-8b-Q4_K_M

ds2-v4: https://huggingface.co/timkoehne/ds2-v4-llama3.1-8b-Q4_K_M

ds2-v5: https://huggingface.co/timkoehne/ds2-v5-llama3.1-8b-Q4_K_M

ds2-v6: https://huggingface.co/timkoehne/ds2-v6-llama3.1-8b-Q4_K_M

DS3 Modelle

ds3-v1: https://huggingface.co/timkoehne/ds3-v1-llama3.1-8b-Q4_K_M

ds3-v2: https://huggingface.co/timkoehne/ds3-v2-llama3.1-8b-Q4_K_M

ds3-v3: https://huggingface.co/timkoehne/ds3-v3-llama3.1-8b-Q4_K_M

Github

Github: <https://github.com/timkoehne/Llama3-Finetuning-for-ABAP>

Eigenständigkeitserklärung

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer oder der Verfasserin/des Verfassers selbst entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

xxxxxxx, 19. August 2024

Rechtsverbindliche Unterschrift