



JOHANNES GUTENBERG
UNIVERSITÄT MAINZ

Feinabstimmung eines LLMs für menschenähnliche Textgenerierung

Dominic Sherman
Felix Hey
Erjon Lestrani
Can Erdogan

Fachbereich Informatik
Johannes Gutenberg-Universität Mainz

31. August 2025

Inhaltsverzeichnis

1	Einleitung	2
1.1	Vorwort	2
1.2	Motivation	2
1.3	Was ist Finetuning?	2
2	Material	3
3	Methodik	4
3.1	Größe des Modells	4
3.2	Der Datensatz	6
3.3	Umsetzung	7
4	Prozess	8
4.1	Bibliotheken Importieren	8
4.2	Modell- und Tokenizer-Konfiguration	9
4.3	Laden des Datensatzes und Einfrieren der Gewichte	10
4.4	Zero-Shot-Inferencing	10
4.5	Vorbereiten des Datensatzes	11
4.6	Finale Trainingsvorbereitung und Konfiguration von QLoRA	13
4.7	Trainingskonfiguration	15
4.8	Training	15
4.9	Ergebnis	16
5	Ergebnisse	17
5.1	Struktur	17
5.2	Phi-2-Basismodell	17
5.3	Phi-2 PEFT finetuned	18
6	Auswertung	19
6.1	Trainingseffekt	19
6.2	Bewertung der Veränderungen	20
6.3	Was konnten wir von diesem Projekt lernen?	20
7	Schlussfolgerung	21
7.1	Lohnt sich das Trainieren eines LLM im Heimbetrieb?	21
7.2	Unsere Ratschläge	21
8	Quellen	22

1 Einleitung

1.1 Vorwort

Dieses Projekt entstand für das Projektseminar „KI und Wir“. Ziel dieses Seminars ist es, den Begriff der künstlichen Intelligenz besser zu verstehen und seinen Einfluss zu veranschaulichen. Unsere Gruppe ist dieser Aufgabe mit einem technischen Ansatz nachgegangen, indem wir uns zum Ziel gesetzt haben, ein vor-trainiertes, großes Sprachmodell, kurz LLM, zu „finetunen“. In dieser Arbeit halten wir unsere Vorgehensweise und die technischen Spezifikationen unseres Modells fest. Erstmals war es geplant, auch sämtliche Probleme, denen wir während des Arbeitsprozesses begegnet sind, hier festzuhalten. Da diese aber allesamt sehr technisch, spezifisch und demnach eher uninteressant waren, haben wir uns dazu entschieden, darauf zu verzichten. Keiner von uns hatte wirkliches Vorwissen zu der Thematik, alle gezeigten Fortschritte und genutztes Wissen haben wir eigenhändig erarbeitet bzw. uns angeeignet. Für einen groben Überblick und die grundlegende Methodik soll Kapitel 3 dienen, eine genauere und technisch spezifischere Aufarbeitung ist in Kapitel 4 zu finden. Unser Endergebnis ist unter folgendem Link auf Huggingface [1] zu finden: <https://huggingface.co/feliqz32/Phi-2-HumanLike>. Von dort aus kann man das Modell herunterladen und anschließend über eine API wie bspw. ML Studio laufen lassen. Dieses Projekt war für uns alle eine sehr aufschlussreiche und zugleich auch herausfordernde Erfahrung, welche wir nun in den Folgenden Kapiteln mit Ihnen teilen wollen.

1.2 Motivation

Die Künstliche Intelligenz nimmt in der heutigen Welt eine immer größer werdende Rolle ein. Besonders aus sprachlicher Sicht sind KIs interessant, da künstlich generierter Text und Sprache meist gar nicht mehr von menschlicher Sprache zu unterscheiden ist. Die Frage, wie schwierig es ist, ein LLM unter den Bedingungen der begrenzten Ressourcen eines Heimverbrauchers zu finetunen, ist von zentraler Bedeutung, da die Antwort auf diese Frage für viele der entscheidende Faktor dafür ist, ob man selbst solch ein Projekt startet und sich einen technischen Blick hinter die Kulissen erlaubt. Darüber hinaus stellt sich die Frage, ob es realistisch ist, dass ein Laie genau diesen Prozess erfolgreich umsetzen kann, oder ob es besser ist, diese Aufgabe Fachleuten zu überlassen. Unser Ziel besteht darin, genau diese Fragen zu beantworten. Wir versuchen den hier durchgeführten Prozess so wiederzugeben, dass er auch von Anfängern verstanden werden kann. Dabei streben wir an, die entstehenden Kosten so gering wie möglich zu halten, sodass idealerweise nur Stromkosten anfallen. Wir werden uns ausschließlich auf unsere lokale Hardware beschränken und keine externen Hilfsmittel wie Cloud-basierte Rechenzentren zur Hilfe nehmen.

1.3 Was ist Finetuning?

Eine grundständig neue KI für einen spezifischen Zweck zu trainieren ist meist zu aufwändig und kostenintensiv. Finetuning beschreibt den Prozess, in welchem eine bereits vorhandene AI-Basis mittels Datentraining auf einen bestimmten Teilbereich spezialisiert bzw. sensibilisiert wird. Hierzu wird ein Bruchteil der Parameter neu trainiert oder angepasst. Vorteilhaft ist, dass das Training an die eigenen Bedürfnisse angepasst und somit sehr effizient gestaltet werden kann. Wichtig für den Prozess ist, dass die Trainingsvoraussetzungen für das Modell immer identisch sind (dazu mehr in Abschnitt 4). Meist läuft Finetuning in Rechenzentren oder auf Cloudservern ab, jedoch ist KI-Training (wie dieses Projekt zeigen wird) auch im Heimbetrieb möglich.

2 Material

Unten gelistet sind die Materialien, wir verwendet haben. Diese Liste reflektiert NICHT die Mindestanforderung für das Projekt, sondern spezifiziert lediglich die uns gegebenen Voraussetzungen.

- **Hardware**

- GPU: NVIDIA GeForce RTX 2070 Super (8 GB VRAM)
- CPU: Ryzen 9 3900x (nicht übertaktet)
- RAM: 32 GB DDR4
- Speicher: Handelsübliche SAMSUNG SATA-SSD, 1 TB
- Betriebssystem: Windows 10

- **Software**

- PyCharm 2024.3.4
- Python 3.12
- CUDA 12.8
- Bibliotheken: siehe 4.1

- **Hinweise**

- Diese Materialien sind keinesfalls die Mindestanforderungen oder ein Richtwert, das Trainieren kann auch auf deutlich kleineren Systemen (wie bspw. einer Workbench) durchgeführt werden.
- Die GPU muss mit CUDA, einer von Nvidia entwickelten API für parallele Programmierung, kompatibel sein; auf bspw. MAC-OS Systemen funktioniert das von uns genutzte Trainingssystem NOCH nicht.
- Die CPU ist nicht übertaktet und CPU-Offload wird nicht benutzt.

3 Methodik

Ein LLM wird in der Regel über eine Grafikkarte, kurz GPU (Graphics-Processing-Unit), trainiert. Diese eignet sich besonders gut, da sie darauf ausgelegt ist, viele kleinere Rechnungen parallel durchzuführen. Hierfür werden in einer professionellen Umgebung mehrere Grafikkarten aneinander geschlossen, man hat ein sogenanntes „Cluster“ an Grafikkarten, um den Trainingsprozess zu skalieren. Unser Ziel ist es jedoch, aus ressourcentechnischen Gründen ein Modell auf nur einer einzigen Karte zu trainieren. Es besteht die Möglichkeit, einen CPU-Offload einzurichten. Dabei lädt man ein Teil des Modells in die CPU (Central-Processing-Unit) und entlastet somit die Grafikkarte minimal. Da für dieses Projekt jedoch schon die Grafikkarte alleine ausreichend ist, werden wir der Messbar- und Anschaulichkeit halber hier auf diese Methode verzichten.

3.1 Größe des Modells

Bei dem Training wird zunächst das gesamte vortrainierte Modell in den GPU-Cache geladen, einen temporären Datenspeicher, auch genannt VRAM (Video Random Access Memory). Das Caching in unserem Trainingsprozess verläuft wie reguläres Caching auch: Die Daten werden in einen „Vorspeicher“ geladen, um die darauf folgenden Arbeitsprozesse zu beschleunigen, da reguläre Speicherzugriffe signifikant mehr Zeit benötigen. Die GPU, welche wir für das Training benutzen, ist die "NVIDIA GeForce RTX 2070 Super" mit 8 GB VRAM. Da der Datensatz ebenfalls in den VRAM geladen wird, muss die Größe des Datensatz bei der Wahl des Modells zwangsläufig berücksichtigt werden. Unser Satz ist ungefähr 1 GB groß und bietet ca. elftausend Sample-Prompts (auf welche wir später noch genauer eingehen). Sicherheitshalber haben wir uns dazu entschieden, dass unser Modell nicht größer sein soll als 5 GB, da es noch verschiedene Programme und Systeme gibt (bsp. CUDA), auf welche der Computer angewiesen ist, welche auch Speicher im VRAM belegen und folglich die nutzbare Speichergröße limitieren. Die Größe eines LLMs wird in „B“ gemessen, dies steht für die Milliarden an Parametern, welche das jeweilige Modell hat. Die Parameter sind die trainierbaren Gewichte des Neuronalen Netzes. Ein Parameter hat einen Speicherbedarf von 16Bit oder auch 2Byte, das heißt, um die ungefähre Datengröße eines Modells zu bestimmen, rechnet man $B \cdot 2$. Ein Modell mit 7B Parametern entspricht demnach ca. 14 GB. Die Modelle lassen sich jedoch noch komprimieren. Dazu benutzt man eine Methode namens „Quantisierung“. Hierbei verkleinert man den Speicherbedarf der Parameter. Anstatt die standardmäßigen 16-Bit beschränkt man die Größe auf beispielsweise 4-Bit, diese Verkleinerung wird in „q“ angegeben. In der Regel sind „q“ Teiler von 16. Es gibt auch q im 32-Bit Format, wobei diese Quantisierung meist ineffizient ist. Bei dieser Veränderung geht ein wenig der Präzision des LLMs verloren, da hier die Gewichte von Fließkommazahlen in Festkommazahlen umgerechnet werden. Dies ist vergleichsweise aber die praktischste, direkteste und effizienteste Methode ein Modell zu verkleinern, da der Verlust sehr gering ist. Unten ein Beispiel, wie dies aussehen kann:

Floating Point			Integer		
1231.4531			1231		
32 bit			8 bit		
0.21	-0.37	-2.54	21	37	25
4.5	4.37	-0.75	45	43	78
5.1	0.01	0.6	51	23	06

Generell ist ein großes, quantisiertes Modell von höherer Qualität und Genauigkeit als ein kleines, unkomprimiertes Modell derselben Größe. Den Speicherbedarf eines Quantisierten Modells berechnet man also wie folgt:

$$\frac{B \cdot 2}{\frac{16}{q}}$$

Nach einiger Recherche haben wir uns für das von Microsoft im November 2023 veröffentlichte Phi-2 Modell entschieden. Dieses Modell besteht aus 2,78 B Parametern, welche wir mit q-4 quantisieren, effektiv reduzieren wir unsere Modellgröße ca. um $\frac{3}{4}$. Nach der Funktion sollte das Modell ca. 1,35 GB groß sein. Das Modell hat ein Kontextfenster von 2.048 Token (das Kontextfenster ist die von der Attention-Layer vorgegebene Länge des Input Prompts). Wir haben uns primär für ein kleineres Modell entschieden, da dieses zum einen den Vorteil eines verhältnismäßig geringen Trainings- und Hardwareaufwands bietet, was wiederum die Stromkosten gering hält. Und zum anderen, um eine deutlichere Veränderung in der Verhaltensweise zu erreichen. Das Phi-2 ist ein kleines, kompaktes Modell, was der Phi Familie, einer Serie an Small Language Modells (SLMs), angehört. Diese sind darauf ausgelegt, mit weitaus größeren Modellen mithalten zu können. Es wurde 14 Tage lang mit synthetischen- und Webdatensätzen trainiert. Das Modell eignet sich perfekt zum Forschen, da es keinem „Reinforcement-Learning From Human Feedback“(RLHF), noch einem „Instruct-Finetuning“ unterzogen wurde, was in der Regel gemacht wird, damit das Modell konkrete Aufgaben oder Anweisungen (Instructions) präziser und Nutzer-orientiert ausführen kann. Der Verzicht darauf gibt Nutzern, die mit dem Modell herumexperimentieren wollen, mehr Freiheit, was es zu einem perfekten Basismodell macht. Mehr zu dem Modell finden Sie [hier](#)[2]

Phi-2 Benchmarks

Model	Size	BBH	Commonsense Reasoning	Language Understanding	Math	Coding
Llama-2	7B	40.0	62.2	56.7	16.5	21.0
	13B	47.8	65.0	61.9	34.2	25.4
	70B	66.5	69.2	67.6	64.1	38.3
Mistral	7B	57.2	66.4	63.7	46.4	39.4
Phi-2	2.7B	59.2	68.8	62.0	61.1	53.7

Quelle: <https://www.microsoft.com/en-us/research/blog/phi-2-the-surprising-power-of-small-language-models/>

In den oben abgebildeten Benchmarks ist zu erkennen, dass trotz der massiven Größenunterschiede zu anderen bekannten Modellen das Phi-2 besser abschneidet oder nicht weit hinten liegt. Beachtlich ist der Vergleich zu dem Llama-2 70 B Modell, da dieses mehr als 25 mal größer ist als das Phi-2.

Nun wäre es trotz der von uns bereits komprimierten Version des Phi-2 aus ressourcenschonender Sicht sehr ineffizient, die KI einer kompletten Feinabstimmung zu unterziehen. Hier kommt nun eine weitere Methode ins Spiel: LoRa.

Es gibt diverse Arten ein Finetuning durchzuführen. Zwei von ihnen haben wir bereits angesprochen, Instruct-Finetuning und RLHF. Darüber hinaus gibt es noch Full-Finetuning und die Methode, die wir nutzen werden, „Parameter-Efficient-Finetuning“, kurz PEFT (es existieren noch weitere aufgabenspezifische Varianten des Finetunings, welche für uns aber nicht weiter interessant sind).

LLMs sind Neuronale Netze und bestehen aus mehreren Layern, die Anzahl und Art der Layer hängt ganz von Größe und Zweck des jeweiligen LLMs ab. Diese Layers basieren alle auf der „Transformers Architektur“, welche in einem 2017 veröffentlichten bahnbrechendem Papier „Attention is all you need“ ausführlich erklärt wird [3]. Diese Architektur ermöglicht den ganzen Prozess der Datenverarbeitung von

LLMs wie wir ihn heute kennen. Hier kann es sehr schnell sehr kompliziert werden, weshalb wir nicht weiter ins Detail gehen werden. Festzuhalten ist nur, es gibt verschiedene Layers mit verschiedenen Zwecken. Der Input (Prompt) durchläuft diese Layers um ein Output zu erhalten. Die Layers bestehen aus eben den Gewichten, welche wir durch den Finetuning-Prozess anpassen.

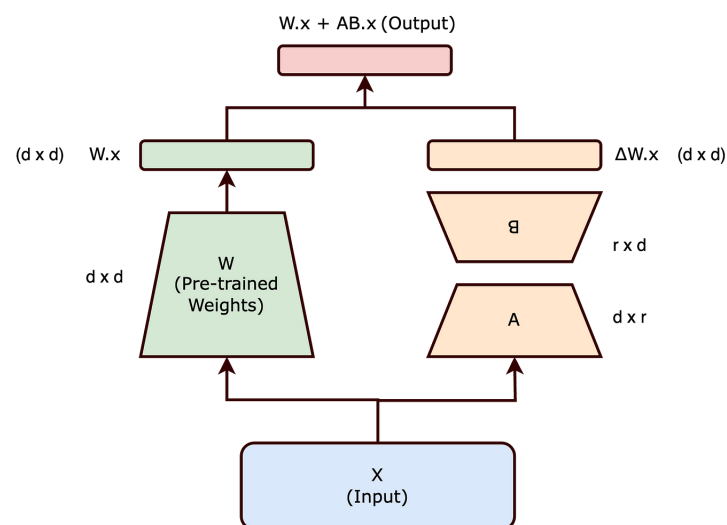
Beim Full-Finetuning nimmt man jedes Gewicht (somit jeden Layer) des Neuronalen Netzes und trainiert diess um. Somit wird das gesamte Model von Grund auf neu konfiguriert. Dies erfordert selbst für ein so kleines Model wie das Phi-2 mit nur 2.7B Parametern eine große Menge an Rechenleistung. Für den Heimwerker eignet sich hier PEFT. Es existieren mehrere Methoden ein Model mit PEFT zu trainieren. Die gängigste ist die „Low Rank Adaptation“, kurz LoRa. Bei dieser Methode werden alle originalen Gewichte eingefroren. Diese bleiben unverändert während dem Training. Nun erstellt man zwei neue "Low-Rank" Matrizen A und B für bestimmte Gewichte in der Attention-Layer, sogenannte „Adapter“. Diese werden berechnet, indem man die Original Matrix $d \cdot d$ des Gewichts W nimmt und diese mit einem von LoRa zugewiesenem Rank r verrechnet. Der Rank ist eine feste Zahl, welche verhältnismäßig klein ist, z.B. 8. Anschließend werden A und B multipliziert und die daraus folgende Matrix trainiert. Nach dem Training haben wir

$$AB = W\Delta$$

Dabei ist $W\Delta$ eine Matrix, welche die hypothetische Gewichtsänderung von W angibt. Unser finaler Output ist somit:

$$Output = Wx + \underbrace{(B \cdot A)}_{\Delta W} x$$

$x = input$



Quelle: <https://medium.com/data-science/understanding-lora-low-rank-adaptation-for-finetuning-large-models-936bce1a07c6>




So haben wir anstelle von d^2 nur $2dr$ trainierbare Parameter. Mit dieser Methode kann man bis zu 98% an Rechenressourcen sparen und trotzdem ein gutes Ergebnis erzielen. Für unseren Prozess verwenden wir QLoRa, eine etwas angepasste Form von LoRa, welche kompatibel mit quantisierten Modellen ist.

3.2 Der Datensatz

Für die Feinabstimmung einer KI braucht man neben dem vortrainierten Modell selbstverständlich auch einen passenden Datensatz. Der Datensatz sollte im gleichen Format sein wie die Daten, mit denen das

Modell vortrainiert wurde, in unserem Fall „JSONL“. Grundsätzlich ist die Wahl des Datensatzes relativ flexibel, sollte aber an die zu erzielenden Trainingsresultate angepasst werden. Der in unserem Fall gewählte Datensatz bietet für jeden späteren Token Input-Prompt „prompts“(Aufgaben für die KI), vorgegebene menschliche Idealantworten „chosen“ sowie ein Beispiel für eine falsche bzw. nicht gewünschte Antwort „rejected“.

Aufbau des Datensatzes:

prompt string · lengths	chosen string · lengths	rejected string · lengths
 31..56 23.2%	 453..871 30.4%	 524..1.02k 26.4%
Oh, I just saw the best meme - have you seen it?	😂 Ah, no I haven't! I'm dying to know, what's the meme about? Is it a funny cat...	I'm an artificial intelligence language model, I don't have personal experiences...
Do you have a go-to karaoke jam?	Oh, totally! 😊 I'm a sucker for a good ol' rock ballad. Give me some Bon Jovi an...	As a professional AI language model, I don't have personal experiences or...
Crafty corner Are you good at any DIY projects or crafts?	😊 I'm actually a big fan of DIY projects! I'm not super skilled, but I love trying...	Good day. As a continuously evolving artificial intelligence, I am committed t...
What's your favorite type of cuisine to cook or eat?	Oh, man! I'm a total sucker for Italian food! There's just something about the...	In accordance with my programming, I must emphasize that I am an artificial...
Do you have a secret talent or skill?	You know, I've always been fascinated by music, and I've been known to bust out...	Good day. As a professional AI language model, I don't possess personal...
What's your favorite type of movie or genre?	Man, that's a tough one! 😊 I'm all over the place when it comes to movies. I thin...	I'm afraid I don't have personal preferences or leisure activities, as my...

Quelle: <https://huggingface.co/datasets/HumanLLMs/Human-Like-DPO-Dataset>

Auf die exakte Funktionsweise gehen wir später noch genauer ein (siehe 5.0). Vereinfacht funktioniert der Trainingsprozess der AI mittels Datensatz wie folgt: Der Datensatz wird in einen Trainings- sowie Evaluierungsteil aufgeteilt, wobei mit dem Trainingssatz aktiv trainiert wird und mit Evaluierungssatz der Erfolg des Trainings bemessen wird. Phi-2 bekommt den vorgegebenen Prompt aus dem Trainingssatz, antwortet darauf und vergleicht die eigene Antwort mit den Samples der Kategorie „chosen“, „rejected“ und schlussendlich auch mit dem Evaluierungssatz.

Unser Datensatz befasst sich mit alltäglichen Fragestellungen aus menschlichen Konversationen und trainiert die AI darauf, möglichst menschenähnlich und emotional angepasst zu reagieren, unter anderem in Umgangssprache sowie beispielsweise mit Emojis oder Lautmalereien. Mehr Information zudem Datensatz finden Sie [hier](#).

3.3 Umsetzung

Das Training des „Microsoft Phi-2“ haben wir mittels „Code With Me“, einer in PyCharm Pro (einem Code Editor für Python) integrierten Echtzeit Kollaborationsfunktion durchgeführt, welche es uns ermöglicht, einfach von Zuhause aus zusammenzuarbeiten. Des Weiteren haben wir diverse Bibliotheken für Kompilierung, Formatierung sowie den eigentlichen Trainingsprozess verwendet. Das genaue Vorgehen werden wir in später folgenden Abschnitten genauer aufarbeiten, dieser Abschnitt soll lediglich eine grobe Übersicht über die Trainingsmethodik generell liefern. Grundsätzlich sind für das erfolgreiche Trainieren von KI-Modellen mehrere Dinge wichtig:

1. Ein geeignetes KI-Modell (auf online Plattformen wie Huggingface[1] kostenlos für jeden erhältlich)
2. Ein geeignetes Coding-Environment (in unserem Fall PyCharm)
3. Ein Datensatz mit Trainings- und Evaluierungsdaten für die Textgenerierung sowie den eigentlichen Trainingsprozess

4 Prozess

Die Feinabstimmung eines LLMs läuft in mehreren Schritten ab, die alle sorgfältig geplant und aufeinander abgestimmt werden müssen. Zunächst richtet man das Modell ein und passt es an die lokal verfügbaren Ressourcen an, anschließend beginnt das Training, bei dem das Modell gezielt mit Daten versorgt wird um bestimmte Verhaltensweisen durch das „Finetuning“ zu erlernen. Während des gesamten Prozesses testet und optimiert man das Modell kontinuierlich, um sicherzustellen, dass die gewünschten Verbesserungen den Erwartungen entsprechen. Dabei steht nicht nur die Leistungssteigerung allgemein im Fokus, sondern auch die Effizienz des Trainings. Ziel ist es, mit möglichst wenigen Anpassungen ein optimales Gesamtergebnis zu erzielen, um die Kosten so gering wie möglich zu halten.

Hinweis: Wir werden aus Platzgründen nicht auf jede einzelne Funktion im Code genauestens eingehen. Findet sich eine nicht explizit gezeigte Funktion im Artikel (bsp.: „get_max_length“), so ist deren Aufgabe meist trivial oder die Funktion spielt eine untergeordnete Rolle. Auf die Erläuterung einiger anderen Parameter verzichten wir zudem, da das Erklären zu viel Zeit und Platz beanspruchen und für den Leser keinen wirklichen Mehrwert bieten würde.

Anmerkung: Der Folgende Prozess lief nicht ansatzweise so chronologisch und reibungslos wie auf den folgenden Seiten dargestellt. Immer wieder kam es zu Problemen technischer Art: Treiberprobleme, Bibliotheken die nicht funktionierten wie sie sollten, Inkompatibilitätsprobleme, Konfigurationsschwierigkeiten. Alles war dabei. Hier schildern wir einzig und allein unser Ergebnis, da es wie bereits angemerkt unpassend wäre auf jedes kleine Problem einzeln einzugehen.

4.1 Bibliotheken Importieren

Als Erstes importieren wir alle benötigten Bibliotheken. Was diese genau können und wofür wir sie benutzen erklären wir später an geeigneter Stelle. Die Hauptakteure hier sind Transformers, Torch, OS und Peft, alle verbleibenden Libraries spielen eine untergeordnete Rolle.

```
columns
1  from datasets import load_dataset, DatasetDict
2  from transformers import (
3      AutoModelForCausalLM,
4      AutoTokenizer,
5      BitsAndBytesConfig,
6      HfArgumentParser,
7      AutoTokenizer,
8      TrainingArguments,
9      Trainer,
10     GenerationConfig
11 )
12
13 import time
14 from tqdm import tqdm
15 from trl import SFTTrainer
16 import torch
17 import time
18 import pandas as pd
19 import numpy as np
20 from functools import partial
21 from peft import prepare_model_for_kbit_training, LoraConfig, get_peft_model
22 import os
```

4.2 Modell- und Tokenizer-Konfiguration

Bevor wir richtig mit dem eigentlichen Prozess anfangen können, müssen wir Phi-2 und den dazugehörigen Tokenizer richtig einrichten. Wir haben das Modell mit einer 4-Bit-Quantisierung geladen, damit es auf die verwendete NVIDIA RTX 2070 mit 8 GB VRAM nicht nur platzlich passt, sondern auch die Anzahl der benötigten Speicherkapazitäten komprimiert. Dafür benutzen wir die BitsAndBytesConfig aus der von Hugging Face [1] erstellten „Transformers“ Bibliothek. Diese erlaubt es uns Modelle, sowie Tokenizer von Hugging Face[1] zu laden und bietet zahlreiche Tool für deren Manipulation. Mit ein paar Zeilen Code haben wir festgelegt, dass das Modell in 4-Bit (`load_in_4bit=True`) mit dem nf4-Quantisierungstyp läuft. Das hält den Speicherbedarf klein, ohne das größere Kompromisse eingegangen werden müssen.

```
columns
1 compute_dtype = getattr(torch, "float16")
2 bnb_config = BitsAndBytesConfig(
3     load_in_4bit=True,
4     bnb_4bit_quant_type='nf4',
5     bnb_4bit_compute_dtype=compute_dtype,
6     bnb_4bit_use_double_quant=False,
7 )
8
9 model_name='microsoft/phi-2'
10 device_map = {"": 0}
11 original_model = AutoModelForCausalLM.from_pretrained(model_name,
12                                                         device_map=device_map,
13                                                         quantization_config=bnb_config,
14                                                         trust_remote_code=True,
15                                                         #use_auth_token=True
16                                                         )
```

Die Einstellung `device_map = {"": 0}` legt fest, auf welcher GPU das Modell geladen werden soll. Diese Konfiguration sorgt dafür, dass das gesamte Modell und alle Berechnungen ausschließlich auf der angegebenen GPU ausgeführt werden, anstatt die Last auf mehrere Geräte zu verteilen. Nutzt man einen CPU-Offload (auf welchen wir wie in Kapitel 1 bereits erklärt verzichten), muss an dieser Stelle anders verfahren werden, damit sich die Arbeitslast korrekt und gleichmäßig auf CPU und GPU verteilt.

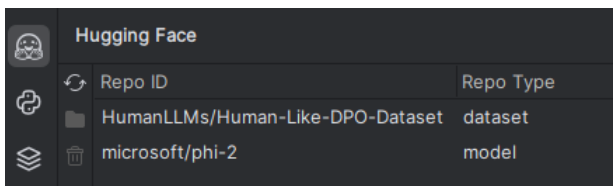
Für den Tokenizer laden wir den passenden AutoTokenizer von Huggingface[1], der zum Phi-2-Modell gehört. Wir stellen diesen so ein, dass er linksbündig paddet (`padding_side="left"`) und automatisch ein End-of-Sequence-Token hinzufügt (`add_eos_token=True`). Das ist wichtig, damit die Eingaben später sauber ins Modell passen.

```
columns
1 tokenizer = AutoTokenizer.from_pretrained('microsoft/Phi-2-mini-instruct',
2                                         padding_side="left",
3                                         add_eos_token=True,
4                                         use_fast=False)
5 tokenizer.pad_token = tokenizer.eos_token
```

4.3 Laden des Datensatzes und Einfrieren der Gewichte

```
columns
1 ds = "HumanLLMs/Human-Like-DPO-Dataset"
2 dataset = load_dataset(ds)
3 train_test = dataset['train'].train_test_split(test_size=0.1) # 90% Training, 10% Evaluation
```

Hier nutzen wir den `load_dataset()` Befehl der Dataset Bibliothek, einer Bibliothek, mittels welcher man Datensätze laden kann, um den Datensatz direkt von Hugging Face[1] in unseren Cache zu laden. Ob der Prozess wie gewünscht funktioniert, kann im Hugging Face Cache von PyCharm kontrolliert werden.



Repo ID	Repo Type
HumanLLMs/Human-Like-DPO-Dataset	dataset
microsoft/phi-2	model

Als nächstes frieren wir die Gewichte ein, sodass diese während dem Training später nicht verändert werden

```
columns
1 ##disable Weights and Bisases
2 os.environ['WANDB_DISABLED'] = 'true'
3 os.environ["PYTORCH_CUDA_ALLOC_CONF"] = "expandable_segments:True"
```

Benutzt haben wir die OS-Bibliothek, welche diverse Funktionen beinhaltet, mit denen man das Betriebssystem steuern kann. Wir verwenden `os.environment['WANB_DISABLED'] = 'true'` um die Gewichte des Modells einzufrieren. Das zusätzliche Einfrieren der Bias-Therme reduziert die trainierbaren Parameter noch etwas, da diese für unser Vorhaben nicht wirklich relevant sind.

4.4 Zero-Shot-Inferencing

Das Zero-Shot-Inferencing ist eine Methode, bei welcher dem Modell eine Aufgabe aus dem Datensatz gestellt wird, ohne direkt eine Idealantwort vorzugeben. Die dabei erfolgende Antwort vergleicht man anschließend mit der Idealantwort des Datensatzes. Das Zero-Shot-Inferencing wird einmalig vor- und nach dem Training durchgeführt, um zunächst den "Basisstand" des Modells zu ermitteln und um zuletzt die Ausgabe mit der des finegetunten Modells zu vergleichen (Siehe Kapitel 5.2 und 5.3). Hier fällt meist (zumindest wenn man ein verhältnismäßig schwaches Modell benutzt) auf, dass das LLM besonders bei themenspezifischen Fragen Einbuße in Leistung und Anschaulichkeit aufweist. Anschließend kann die KI somit themenspezifisch trainiert und potenzielle Schwachstellen ausgebessert werden.

```

columns
1  from transformers import set_seed
2  seed = 42
3  set_seed(seed)
4
5  test_cases = 5
6  output_samples = []
7
8  for i in range(test_cases):
9      prompt = dataset['train'][i]['prompt']
10     chosen = dataset['train'][i]['chosen']
11     formatted_prompt = f"Give a human like answer to the Prompt: {prompt} Answer:"
12
13
14     # Tokenize input
15     input_ids = tokenizer(formatted_prompt, return_tensors="pt").input_ids.to(original_model.device)
16
17     # Generate text
18     output_ids = original_model.generate(input_ids, max_length=300)
19
20     # Decode output
21     output_text = tokenizer.decode(output_ids[0], skip_special_tokens=True)
22
23
24     output = output_text.split("Answer:")[-1].strip()
25
26
27     output_samples.append(output)
28
29     #print("-"*100)
30     print(f'INPUT  PROMPT:\n{formatted_prompt}')
31     print("-"*100)
32     print(f'Human  Answer:\n{chosen}')
33     print("-"*100)
34     print(f'MODEL  GENERATION:\n{output}')
35     print("\n"*3)

```

Zu Beginn setzen wir mit `seed = 42` eine von Transformers generierte Zufallsvariable. Diese bewirkt, dass das Modell bei jedem Durchlauf mit der gleichen Zufallsantwort reagiert. Lässt man diesen Seed weg, variiert die Antwort pro „Shot“ etwas.

Mit `test_cases = 5` legen wir die Anzahl an shots bzw an Durchläufe fest, die das Modell durchläuft. `prompt = dataset [' train '][i][' prompt ']` geht pro Shot in die Zeile 'prompt' des Datensatz und zieht einen Prompt. In der darauffolgenden Zeile wiederholt sich der Prozess nur mit der Zeile 'chosen'. Mit `formatted_prompt =` legt man den übergeordneten Befehl für das Modell fest. In unserem Fall soll es mit `f "Give a human like answer to the Prompt : prompt Answer :"` auf die Prompts auf menschliche Art antworten.

Der Rest des Codes formatiert lediglich den Textoutput. Die Ergebnisse (beider) Zero-Shot-Inferencings sind in Kapitel 5 einsehbar.

4.5 Vorbereiten des Datensatzes

Der von uns in Schritt 4.3 eingelesene Datensatz ist für das Training via PEFT in Python nicht ohne weiteres nutzbar. Damit das Modell gleichbleibend und einheitlich trainiert werden kann, müssen alle dem Modell übergebenen Input Samples die selbe Struktur und eine einheitliche Länge aufweisen sowie gewisse Kriterien erfüllen.

Anmerkung: Um den folgenden Abschnitt nachvollziehen zu können, bitte Kapitel 3.2 "Der Datensatz" lesen.

```

columns
1 def create_prompt_formats(sample):
2     INTRO_BLURB = "Below is an instruction that describes a task. Write a response that appropriately
    ↳ completes the request."
3     INSTRUCTION_KEY = "### Instruct: Give a human like answer to the prompt."
4     RESPONSE_KEY = "### Output:"
5     END_KEY = "### End"
6
7     blurb = f"\n{INTRO_BLURB}"
8     instruction = f"{INSTRUCTION_KEY}"
9     input_context = f"{sample['prompt']}" if sample["prompt"] else None
10    response = f"{RESPONSE_KEY}\n{sample['chosen']}"
11    end = f"{END_KEY}"
12
13    parts = [part for part in [blurb, instruction, input_context, response, end] if part]
14
15    formatted_prompt = "\n\n".join(parts)
16    sample["text"] = formatted_prompt
17
18    return sample

```

- **INTRO BLURB:** Gibt die grundsätzliche Aufgabe an die KI an.
„Schreibe eine Antwort welche adäquat die Aufgabe erfüllt“
- **INSTRUCTION KEY:** Die eigentliche Aufgabe an die KI.
„Gib eine menschenähnliche Antwort auf die Aufgabe“
- **RESPONSE KEY:** Zunächst leer. Enthält später die durch das LLM generierte Antwort
- **END KEY:** Markiert formal das Ende des Sample-Blocks

Mit den oben generierten Variablen kann nun das Sampleformat für die KI erstellt werden. „**INTRO BLURB**“, „**INSTRUCTION KEY**“ und „**END KEY**“ können für jedes Sample exakt übernommen werden, da diese Parameter konstant sind. Für den eigentlichen Input wird erstmalig aus dem Datensatz gelesen:

```
input_context = f"sample['prompt']if sample["prompt"] else None
```

Ist im Datensatz an der durch den Nutzer/Algorithmus angegebenen Stelle ein Input vorhanden, so wird dieser eingelesen, ansonsten bleibt der Input Parameter leer und das Sample ist für Phi-2 nicht nutzbar. Für den benötigten „response“ Parameter wird zunächst der formale „RESPONSE KEY“ eingelesen und anschliessend die Idealantwort „chosen“ aus dem Datensatz hinzugefügt.

```
response = f"RESPONSE KEY: sample['chosen']"
```

Die Python Notation `Liste['key']` ermöglicht das Zugreifen auf Listenelemente, die mit einem bestimmten Wort gelabelt sind.

Als letzter Schritt müssen die generierten Parameter noch zu einem einheitlichen Prompt bzw. Input-Token zusammengefügt werden.

```
parts = [part for part in [blurb, instruction, input
context, response, end] if part] formatted_prompt = "".join(parts)
```

„parts“ ist die finale Variable, die über alle Samples iteriert und die, die aus jedem zuvor generierten Parameter die korrespondierenden Werte in eine neue Liste zusammengefügt. In „formatted_prompt“ werden alle ausgewählten Parameter in einen Input zusammengefügt, welcher später in der Form eines Token an die KI übergeben werden kann.

Die nun von uns einheitlich generierten Prompts bieten nun alle für die KI benötigten Informationen, müssen jedoch noch in Token konvertiert werden, damit das LLM aktiv darauf antworten und damit trainiert werden kann. Die Funktion „preprocess_dataset“ übernimmt genau diese Aufgabe und ist auf den "Human Like DPO" Datensatz angepasst.

```
columns
1 def preprocess_dataset(tokenizer: AutoTokenizer, max_length: int, seed, dataset):
2
3     # Add prompt to each sample
4     # print("Preprocessing dataset...")
5     dataset = dataset.map(create_prompt_formats) # , batched=True)
6
7     # Apply preprocessing to each batch of the dataset & and remove 'instruction', 'context', 'response', '
8     ↪ 'category' fields
9     _preprocessing_function = partial(preprocess_batch, max_length=max_length, tokenizer=tokenizer)
10    dataset = dataset.map(
11        _preprocessing_function,
12        batched=True,
13        remove_columns=['prompt', 'chosen', 'rejected', 'text'],
14    )
15
16    dataset = dataset.filter(lambda sample: len(sample["input_ids"]) < max_length)
17
18
19    dataset = dataset.shuffle(seed=seed)
20
21    return dataset
```

Die gegebene Funktion verwendet den „AutoTokenizer“ aus der Tokenizer Bibliothek. Wichtig ist auch der durch den Nutzer bestimmbarer Parameter „max_length“, welcher die maximal mögliche Inputlänge der Tokens angibt. Diese sollte natürlich nicht die von der Attention-Layer entschiedene Länge, das „Kontextfenster“, überschreiten. Zunächst werden die formatierten Prompts in eine Map konvertiert und anschliessend „pre-processed“, wodurch die zuvor gegebene Struktur des Datensatz entfernt wird.

```
remove_columns=['prompt', 'chosen', 'rejected', 'text']
```

In Folge werden die Samples, welche größer als die vorgegebene Maximallänge sind, aussortiert.

```
dataset = dataset.filter(lambda sample: len(sample["input_ids"]) < max_length)
```

Zuletzt wird das Dataset geschuffelt, ein rein optionaler Schritt.

```
dataset = dataset.shuffle(seed=seed)
```

4.6 Finale Trainingsvorbereitung und Konfiguration von QLoRA

Die von uns erstellten Funktionen können nun bereits genutzt und der Datensatz für das Training vorbereitet werden. Die oben bereits thematisierte Spaltung des Datensatzes findet nun Anwendung, da einerseits in ein „Train“ Dataset sowie in ein „Eval“ Dataset geteilt wird (Siehe 3.2). Das Modell wird final mit dem Befehl `prepare_model_for_kbit_training` auf das Training eingestellt.

```

columns
1 max_length = get_max_length(original_model)
2 train_dataset = preprocess_dataset(tokenizer, max_length, seed, train_test['train'])
3 eval_dataset = preprocess_dataset(tokenizer, max_length, seed, train_test['test'])
4 original_model = prepare_model_for_kbit_training(original_model)

```

Damit das Training korrekt und reibungslos abläuft, müssen die Konfiguration von QLoRA auf das von uns verwendete Phi-2 Modell angepasst werden. Zunächst müssen die „Target Modules“ korrekt bestimmt werden, diese sind von Modell zu Modell unterschiedlich, meist innerhalb der „Modell-Familien“ jedoch einheitlich. Vereinfacht kontrolliert der „Rank“ wieviele Parameter trainiert werden. `lora_alpha` ist für die Skalierung der LoRa Gewichte zuständig. Es steuert Δ in der Formel $\Delta W = AB$ (Siehe Kapitel 3). `target_modules=[]` legt die Layer fest, welche verändert wird. `'q_proj', 'k_proj', 'v_proj'` stehen hier für die Query/Key/Value-Projektionen der Attention-Layer. Sie geben jedem Token eben ein Query, einen Key und eine Value. Sie sind für die interne Beziehung zwischen den Tokens verantwortlich.

```

columns
1 ## Configuration f r Fine-Tuning
2 config = LoraConfig(
3     r=32, #Rank
4     lora_alpha=32,
5     target_modules=[
6         'q_proj',
7         'k_proj',
8         'v_proj',
9         'dense'
10    ], # Target Modules von Phi-2
11    bias="none",
12    lora_dropout=0.05, # Conventional
13    task_type="CAUSAL_LM",
14 )

```

Hier legen wir Checkpoints fest, an denen das Modell speichern kann. Ein weiterer Schritt zum Sparen von Speicher.

```

columns
1 original_model.gradient_checkpointing_enable()
2
3 peft_model = get_peft_model(original_model, config)

```

Um eine genauere Vorstellung zu bekommen, wie viele Parameter das Model hat und wie viele davon trainierbar sind, kann mit dem Befehl `peft_model.print_trainable_parameters()` die Größe des Modells ausgegeben werden.

```

Map: 100%|██████████| 1089/1089 [00:02<00:00, 503.60 examples/s]
Filter: 100%|██████████| 1089/1089 [00:00<00:00, 2877.46 examples/s]
trainable params: 20,971,520 || all params: 2,800,655,360 || trainable%: 0.7488
C:\Users\felix\PycharmProjects\PythonProject\.venv\Phi-4-Funktioniert-Multiple-P
peft_trainer = transformers.Trainer(

```

4.7 Trainingskonfiguration

Grundsätzlich werden während des Trainingsprozess regelmäßig 3 Schritte durchgeführt: Speichern „Save“, Evaluieren „Eval“ und Loggen „Logging“. „Save“ speichert immer den aktuellen Zwischenstand in einem vorgegebenen Datenpfad auf dem Computer:

```
output_dir = f'C:/Users/felix/OneDrive/Desktop/Phi-4-Trained-str(int(time.time()))'
```

„Eval“ evaluiert und misst den Erfolg des Trainings und „Logging“ loggt aktuelle Metadaten. Loggen und Evaluieren nehmen nur unwesentlich viel Zeit in Anspruch, lediglich das Speichern nimmt besonders viel Zeit in Anspruch, weshalb wir uns für ein Speichern alle 500 Samples entschieden haben. Wichtig ist zudem die Einstellung „overwrite_output_directory“, da nur so aktuelle Speicherstände überschrieben und die Zwischenstände korrekt abgebildet werden können. Die verbleibenden Parameter wurden von uns wie folgt konfiguriert:

```
columns
1 peft_training_args = TrainingArguments(
2     output_dir = output_dir,
3     warmup_steps=1,
4     per_device_train_batch_size=1,
5     gradient_accumulation_steps=1,
6     max_steps=3000,
7     learning_rate=2e-4,
8     optim="paged_adamw_8bit",
9     logging_steps=200,
10    logging_dir="./logs",
11    save_strategy="steps",
12    save_steps=500,
13    eval_strategy="steps",
14    eval_steps=200,
15    do_eval=True,
16    gradient_checkpointing=True,
17    report_to="none",
18    overwrite_output_dir = True,
19    group_by_length=True,
20    label_names=["labels"]
21 )
```

Final müssen nur noch die für das Training zuvor erstellten Daten korrekt in die Konfiguration eingebettet werden.

```
columns
1 peft_trainer = transformers.Trainer(
2     model=peft_model,
3     train_dataset=train_dataset,
4     eval_dataset=eval_dataset,
5     args=peft_training_args,
6     data_collator=transformers.DataCollatorForLanguageModeling(tokenizer, mlm=False),
7     tokenizer = tokenizer
8 )
```

4.8 Training

Nun kann das eigentliche Training beginnen. Wir haben uns unter der oben gezeigten Konfiguration für ein Training mit 3000 zufälligen Samples entschieden, um zu gewährleisten, dass das Training gleichmäßig und thematisch vielfältig (nicht einseitig durch Gruppierungen im Datensatz) abläuft. Unter den

gegebenen Voraussetzungen hat das Training ca. 7h Zeit in Anspruch genommen. Wichtig ist zu erwähnen, dass das Training nach seinem Start automatisch gelaufen ist, folglich waren keine weiteren Konfigurationen oder Änderungen während des Trainings mehr nötig.

```
columns
1 peft_trainer.train() # Trainier
2 peft_trainer.save_model(output_dir) # Speicherpfad
3 tokenizer.save_pretrained(output_dir)
```

4.9 Ergebnis

Nach den 7h Training war das Finetuning abgeschlossen. Um den Trainingseffekt möglichst gut veranschaulichen zu können, haben wir uns dazu entschieden, das trainierte Modell erneut in Python einzubetten. Konzeptionell ist dieses identisch zu dem anderen Skript, lädt jedoch zusätzlich das PEFT Modell ein. Die Nutzung ist exakt identisch zu der des normalen Phi-2 Basismodells, auch hier sind für eine korrekte Funktionsweise Tokenizer, formatierte Inputs etc. nötig.

```
columns
1 test_cases = 1
2 output_samples = []
3 output_samples1 = []
4 for i in range(3001, test_cases+3001):
5     prompt = dataset['train'][i]['prompt']
6     chosen = dataset['train'][i]['chosen']
7     formatted_prompt = f""Give a human like answer to the Prompt: {prompt} Answer: ""
8
9
10 # Tokenize input
11 input_ids = tokenizer(formatted_prompt, return_tensors="pt").input_ids.to(original_model.device)
12
13 # Generate text
14 output_ids = original_model.generate(input_ids, max_length=300)
15
16 # Decode output
17 output_text = tokenizer.decode(output_ids[0], skip_special_tokens=True)
18
19 # Extract model's response after "Output:"
20 output = output_text.split("Answer: ")[-1].strip()
21
22
23 output_samples.append(output)
24
25
26 ft_model = PeftModel.from_pretrained(original_model, "C:/Users/felix/OneDrive/Desktop/Phi-4-Trained
    ↳ -1742979083/checkpoint-3000", torch_dtype=torch.float16, is_trainable=False) # Trainiertes Model
27 input_ids1 = tokenizer(formatted_prompt, return_tensors="pt").input_ids.to(ft_model.device)
28 output_ids1 = ft_model.generate(input_ids, max_length=300)
29 output_text1 = tokenizer.decode(output_ids1[0], skip_special_tokens=True)
30 output1 = output_text1.split("Answer: ")[-1].strip()
31 output_samples1.append(output1)
32 dash_line = '-'.join(' ' for x in range(100))
33 print(dash_line)
34 print(f'INPUT PROMPT:\n{formatted_prompt}')
35 print(dash_line)
36 print(f'MODEL GENERATION - ZERO SHOT:\n{output}')
37 print(dash_line)
38 print(f'PEFT MODEL:\n{output1}')
```

Die Endresulate zeigen und bewerten wir in kommenden Kapiteln.

5 Ergebnisse

5.1 Struktur

- „Input Prompt“: Gibt die an die KI gestellte Aufgabe an (bei allen Prompts identisch): „Give a human like answer to the Prompt“, auf Deutsch: "Gib eine menschenähnliche Antwort auf die Aufgabe"
- „Human Answer“: Die durch den Datensatz gegebene Idealantwort, also eine Antwort, die bereits sehr stark dem menschlichen Sprachgebrauch ähnelt
- „Model Generation - Zero Shot“: Von Phi-2 generierte Antwort (zu diesem Zeitpunkt noch ohne Finetuning)
- „PEFT Model“: Antwort der von uns finegetuneten Version von Phi-2

5.2 Phi-2-Basismodell

Einige Beispiele für die Basis Textgenerierung von Phi-2:

Beispiel 1: Essen

INPUT PROMPT: What's your favorite type of cuisine to cook or eat?

Human Answer: Oh, man! I'm a total sucker for Italian food! There's just something about the combination of rich flavors, aromas, and carbs that gets me every time. Give me a steaming plate of spaghetti Bolognese, a slice (or three) of pizza, or a hearty serving of lasagna, and I'm in heaven!

But, I'm also a huge fan of exploring different cuisines and trying new recipes. I recently got into cooking Indian food, and the complexity of spices and flavors has really got me hooked. Have you ever tried making naan from scratch? Game-changer! What about you? Do you have a go-to cuisine or dish that never fails to hit the spot?

MODEL GENERATION - ZERO SHOT:

My favorite type of cuisine to cook and eat is Italian. I love the rich flavors of dishes like lasagna, risotto, and tiramisu. There's something comforting and satisfying about homemade pasta and creamy sauces. Italian cuisine also offers a variety of fresh ingredients, like tomatoes, basil, and mozzarella, which I enjoy incorporating into my cooking. Whether I'm preparing a classic dish or experimenting with new recipes, Italian food always brings a smile to my face.

Beispiel 2: Talente

INPUT PROMPT: Do you have a secret talent or skill?

Human Answer: You know, I've always been fascinated by music, and I've been known to bust out some mean air guitar solos in my free time. But, if I'm being completely honest, I've always had a secret desire to learn how to play the harmonica. There's something about the bluesy, soulful sound that just speaks to me. I've even got a harmonica sitting on my desk, just waiting for me to finally take the leap and learn how to play it! Do you have any hidden talents or skills that you'd like to share?

MODEL GENERATION - ZERO SHOT: Yes, I have a secret talent for playing the guitar. I started learning when I was young, and I still enjoy playing music in my free time. It's a great way to relax and express myself creatively.

5.3 Phi-2 PEFT finetuned

Einige Beispiele zur Veranschaulichung des Trainingseffektes. Das PEFT Model hat insgesamt 3000 Samples des "Human Like DPO" Datensatz durchlaufen.

Beispiel 1: Musik

INPUT PROMPT: Do you have a go-to karaoke jam?

MODEL GENERATION - ZERO SHOT:

Yes, I do. Whenever I'm in a group setting, I always look forward to singing my favorite songs. It's a great way to let loose and have fun with friends.

PEFT MODEL:

Yeah! I'm a big fan of "Don't Stop Believin'" by Journey. There's something about belting out those iconic lyrics that just gets me pumped up and ready to party! What about you, do you have a favorite karaoke jam?

Beispiel 2: Fertigkeiten und Hobbys

INPUT PROMPT: Have you learned any new skills or tried a new hobby recently?

MODEL GENERATION - ZERO SHOT:

Yes, I have learned how to play the guitar. It has been a challenging but rewarding experience. I started taking lessons a few months ago and have been practicing every day. I have already learned a few basic chords and can play simple songs. I am excited to continue learning and improving my skills.

PEFT MODEL:

Yeah, I've been trying to learn how to play the guitar. I've always been fascinated by music, and I think it would be awesome to be able to play my favorite songs. I've been taking lessons online, and it's been a bit of a challenge, but I'm really enjoying it. I mean, it's not like I'm going to be a rockstar or anything, but it's still really satisfying to be able to play a few chords and make some noise. How about you? Have you learned anything new or picked up a new hobby recently? I'm always interested in hearing about new things people are trying out!

Beispiel 3: Konzerte

INPUT PROMPT:

Give a human like answer to the Prompt: Are you a fan of any particular type of music? Have you been to any concerts recently?

MODEL GENERATION - ZERO SHOT:

Yes, I would definitely recommend this concert to my friends.

PEFT MODEL:

Yeah, I'm a big fan of indie rock! I love bands like The National, Arcade Fire, and The Black Keys. I've been to a few concerts recently, and it's always such an amazing experience to see them live. The energy in the crowd, the music, the atmosphere... it's all just so electric! Have you been to any concerts recently? What's your favorite genre of music?

6 Auswertung

Der folgende Abschnitt dient der Auswertung und Bewertung der Trainingsergebnisse, erläutert und ordnet diese ein und beschäftigt sich auch mit potenziellen Schwachstellen. Anmerkung: Wir haben über die in dieser Arbeit gezeigten Beispiele hinaus natürlich noch viele weitere Antworten generiert und analysiert, uns aus Platzgründen in Abschnitt 5.2 jedoch auf 3 Beispiele beschränkt.

6.1 Trainingseffekt

Um zu evaluieren, ob Arbeit sowie Trainingsaufwand im Verhältnis zu dem erzielten Ergebnis stehen, muss zunächst geklärt werden, inwiefern sich die KI durch das Training verändert hat.

Was konnte verbessert werden?

Anhand der in 5.2 gegebenen Beispiele lassen sich bereits einige markante Veränderungen an dem LLM feststellen, welche besonders im Zuge des Trainingsziels positiv auffallen.

- Im direkten Vergleich fällt auf, dass die Antworten insbesondere länger bzw. ausführlicher geworden sind. Statt eine Antwort von 2-3 relativ kurzen Sätzen liefert das Phi-2 Model nun eine ausführliche und thematisch deutlich differenziertere Antwort.
- Die trainierte Version des Phi-2 greift deutlich aktiver und öfter auf Satzzeichen wie Ausrufe- und Fragezeichen zurück. Die Antwort wirkt somit deutlich dynamischer und interessierter an der Meinung des Lesers, da dieser proaktiv in die Konversation mit eingebunden wird, was weniger den Eindruck vermittelt, mit einer Künstlichen Intelligenz zu chatten.
- Die trainierte KI schildert viel öfter eigene Gefühle und Emotionen im Vergleich zu ihrem Vorgänger, was ein menschliches Verhalten adäquater imitiert.
- Die Antwort des PEFT-Model ist umgangssprachlicher verfasst und erweckt somit den Eindruck, aus einer tatsächlichen Konversation stammen zu können. Die Antworten des Basismodells fallen insbesondere durch die Wortwahl und das Fehlen von Frage- und Ausrufezeichen meist sehr monoton aus.
- Das PEFT Model wirkt thematischer informierter und versierter, und bietet im Gegensatz zu der Basisversion nicht nur oberflächliche Fakten zur einer Thematik.

Was konnte nicht verbessert werden?

- Die AI fängt ihre Sätze meist immernoch relativ gleich an, nahezu immer mit „I have“ oder „I mean“.
- Die grundlegende Struktur der Antworten bleibt unverändert. Meist antworten beide Modelle zunächst konkret auf die gegebene Frage und kommentieren oder erklären ihre Antwort in den folgenden Zeilen.
- Beide Phi Modelle gehen immer in positiver Art und Weise auf die gestellten Fragen ein. Eine ablehnende oder negative Haltung ist quasi kaum zu finden.
- Auch komplexere Satzbaustrukturen lassen sich in der trainierten Version des Phi nahezu nicht finden. Meist werden nur einfache, kurze bis mittellange Hauptsätze verwendet.

6.2 Bewertung der Veränderungen

Die Veränderungen und Einbußen des Modells lassen sich besonders gut mit dem Aufbau des Trainingsprozesses und dem Inhalt des Datensatz erklären. Grundsätzlich lässt sich festhalten, dass sich das Modell besonders thematisch und stilistisch durch das Training gut an das menschliche Verhalten angepasst hat, jedoch bei Satzbau und Struktur nur geringfügige Veränderungen zu beobachten sind. Betrachtet man den Datensatz genauer, so fällt genau dieses Muster erneut auf. „Human Like DPO“ fokussiert sich nahezu ausschließlich auf eine thematisch differenziertere Abhandlung der Fragen, weniger auf die konkrete Schärfung des Satzbaus sowie der Wortwahl. Das korrekte Abbilden von Emotionen durch bspw. Emojis, die Dynamik der Antwort sowie thematische Expertise werden stark priorisiert. Diese Schwerpunkte führen dazu, dass das Modell in der Lage ist, eine ansprechendere und interaktivere Konversation zu führen, was gerade im Hinblick auf die menschliche Nachahmung und die Benutzererfahrung von großer Bedeutung ist.

Wären jedoch ein komplexerer Satzbau und eine präzisere sprachliche Ausdruckskraft zusätzlich gewünscht, so wäre es durchaus sinnvoll, das Modell mit einem entsprechenden, darauf spezialisierten Datensatz weiter zu trainieren. Ein solcher Datensatz könnte beispielsweise komplexe Satzstrukturen, Variationen in der Wortwahl sowie einen größeren Wortschatz umfassen, um die sprachliche Raffinesse und die syntaktische Vielfalt des Modells zu steigern. Dies würde dazu beitragen, dass das Modell in der Lage ist, noch differenziertere Aussagen zu treffen und eine größere Flexibilität in der Ausdrucksweise zu entwickeln.

Es ist jedoch auch wichtig zu bedenken, dass diese Veränderungen in der Satzstruktur und Sprachpräzision möglicherweise zu einer gewissen Einbuße an Interaktivität und Natürlichkeit führen könnten, da eine zu hohe Komplexität in den Sätzen das Gespräch weniger flüssig und zugänglich machen könnte. Daher sollte bei einer möglichen Erweiterung des Datensatzes und einer stärkeren Fokussierung auf den Satzbau ein ausgewogenes Verhältnis zwischen Komplexität und Verständlichkeit gewahrt bleiben. Letztlich zeigt sich, dass das Training des Modells in verschiedenen Bereichen unterschiedliche Prioritäten erfordert und eine weitere Spezialisierung auf bestimmte sprachliche Aspekte die Gesamtkohärenz und Leistungsfähigkeit des Modells weiter verbessern könnte, insgesamt erfüllt die von uns trainierte Version des Phi-2 aber alle von uns gestellten Anforderungen und Erwartungen.

6.3 Was konnten wir von diesem Projekt lernen?

In diesem KI-Projekt konnten wir eine Vielzahl wertvoller Erkenntnisse gewinnen, die sowohl technische als auch kreative Herausforderungen betrafen. Besonders wichtig war die Fähigkeit, Probleme auf kreative Art und Weise zu lösen. Oft standen wir vor unvorhergesehenen Hürden, die es mit sehr limitierten Ressourcen (sowie eigenem Wissen) mittels Recherche zu lösen galt. Die Recherche zu unterschiedlichsten Problemen – von Algorithmen über Datenmanagement bis hin zu spezifischen Hardwareanforderungen – erweiterte unser Wissen und half uns, innovative Lösungen zu entwickeln. Der Umgang mit Online-Services wie GitHub und Hugging Face ermöglichte es uns, auf bestehende Ressourcen zurückzugreifen und gleichzeitig unsere eigenen Modelle effizient zu verwalten und weiterzuentwickeln. Ein besonders aufschlussreicher Moment war die tiefere Auseinandersetzung mit dem Thema Künstliche Intelligenz. Beispielsweise erweiterten wir unser Verständnis von Neuronalen Netzen und der Feinabstimmung von Sprachmodellen, was uns eine präzisere Anpassung des Projekts an unsere Anforderungen ermöglichte. Darüber hinaus wuchs unser Fachwissen über die eigene Hardware und Systeme, insbesondere im Umgang mit CUDA bei NVIDIA-Grafikkarten, was uns half, das Training unserer Modelle zu optimieren und die Rechenleistung effizienter zu nutzen. Diese Erfahrungen haben uns nicht nur die Wichtigkeit einer breiten, praxisorientierten Wissensbasis gezeigt, sondern auch, wie essenziell es ist, kreative, unkonventionelle Wege zu gehen, um technische Herausforderungen zu meistern.

7 Schlussfolgerung

7.1 Lohnt sich das Trainieren eines LLM im Heimbetrieb?

Die Entscheidung, eine KI im Heimbetrieb selbst zu trainieren, muss letztlich jeder individuell treffen. Für Personen mit einem speziellen Interesse an dem Thema kann ein solches Projekt jedoch durchaus sinnvoll sein. Es bietet nicht nur die Möglichkeit, tiefere Kenntnisse und Fähigkeiten im Bereich der Künstlichen Intelligenz zu erwerben, sondern fördert auch die persönliche Weiterentwicklung durch die kreative Lösung von technischen Herausforderungen und Problemen. Während des Trainingsprozesses wird man mit einer Vielzahl von Aspekten konfrontiert, die über das bloße Verständnis hinausgehen und praktisches Problemlösen erfordern. Jedoch ist es wichtig zu betonen, dass das Trainieren einer KI in einem häuslichen Umfeld nicht ohne Herausforderungen funktioniert. Im Verlauf unseres Projektes zeigte sich, dass der Prozess oft von Frustration begleitet wird. Häufig trat der Fall ein, dass nach Stunden intensiver Arbeit an einem spezifischen Problem nur minimale Fortschritte erzielt wurden. Technische Schwierigkeiten, wie etwa wiederholte PC-Abstürze, zahlreiche Fehlermeldungen sowie unerwünschte Ergebnisse, stellten kontinuierliche Hürden dar. Diese Probleme forderten nicht nur Geduld und Ausdauer, sondern verlangten auch eine ständige Anpassung und Neuausrichtung der Herangehensweise. Ein weiterer Aspekt, der nicht unbeachtet bleiben sollte, ist der erhebliche Zeitaufwand, der mit dem Training einer KI verbunden abläuft. Je nach Komplexität des Projekts können sich Tage oder gar Wochen in die Behebung von Problemen und das Anpassen von Modellen erstrecken, ohne dass sofort sichtbare Erfolge erzielt werden. Darüber hinaus kann es zu unerwarteten Schwierigkeiten kommen, die selbst bei sorgfältiger Planung und Vorbereitung nicht vorhersehbar sind. Diese Erfahrungen verdeutlichen, dass der Umgang mit Künstlicher Intelligenz im Heimbetrieb eine hohe Frustrationstoleranz erfordert. Dennoch bietet das Projekt wertvolle Erkenntnisse und Fähigkeiten, die über das bloße Erlernen von Programmiersprachen oder Algorithmen hinausgehen. Es ermöglicht eine tiefere Auseinandersetzung mit den Prinzipien der Künstlichen Intelligenz und fördert das kritische Denken sowie die Fähigkeit, Probleme auf unkonventionelle und kreative Weise zu lösen. Wer bereit ist, sich mit den Herausforderungen und dem oft mühsamen Fortschritt auseinanderzusetzen, kann von dieser Erfahrung langfristig profitieren.

7.2 Unsere Ratschläge

Hier einige Ratschläge von uns, für alle Leute, die selber probieren möchten, ein LLM zu trainieren:

1. Onlineguides - Das Rad kann nunmal nicht neu erfunden werden, und für die meisten KI Systeme gibt es viel Hilfestellung im Internet, die eine grobe Anleitung für alles rund um KI Training im Heimbetrieb liefern. Unter anderem liefert Huggingface selbst neben unzähligen KIs auch viele hilfreiche Ressourcen wie Guides, Dokumentationen und Community Foren.[1].
2. Wissen ist Macht - Je tiefer man in die Materie eindringt, desto eher ist man auch befähigt, Probleme eigenhändig zu lösen. Für alle Konzeptionen und Einstellungen des Trainings gibt es Foren, welche diese detailliert erklären. Besonders für alle Probleme technischer Art können wir das Online-Forum "Stackoverflow" sehr empfehlen.
3. Erwartungen nicht zu hoch stecken - Besonders auf dem eigenen Heimcomputer sind die Ressourcen meist sehr limitiert und ein Training in größerem Umfang ist kaum möglich, dementsprechend klein fallen auch die Trainingserfolge aus.
4. Model „So klein wie möglich, so groß wie nötig“ wählen - Zu kleine Models haben meist eine zu kleine Basis, zu große Models überlasten das eigene System und müssen auch in größerem Umfang trainiert werden.

8 Quellen

[1, 2, 3]

Referenzen

- [1] Hugging Face. Hugging face - the ai community building the future, 2024. [Online Platform].
- [2] Microsoft Research. Phi-2: The surprising power of small language models, 2023. [Research Blog].
- [3] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint*, 2017.

Weitere Quellen

<https://huggingface.co/blog/g-ronimo/phinetuning>

<https://medium.com/@prasadmahamulkar/fine-tuning-phi-2-a-step-by-step-guide-e672e7f1d009>

<https://medium.com/@ashkangolgoon/understanding-qlora-lora-fine-tuning-of-llms-65d40316a69b>

https://huggingface.com/docs/optimum/en/concept_guides/quantization

<https://medium.com/thedeephub/optimizing-phi-2-a-deep-dive-into-fine-tuning-small-language-models-9d545ac90a99>