Big-Data-Analyseprojekt

Analyse der Auswirkungen verschiedener Pruning-Methoden an *Large Language Models*

Kai Daniel Herbst und Sebastian Viet

Oktober 2024 - März 2025

Inhaltsverzeichnis

Ziel: 3600 Wörter

1 Einleitung

Hier kommt die Einleitung hin!

2 Theorie

Hier kommt alles zur Theorie hin!

3 Methodik

3.1 Literaturrecherche

3.2 Arbeitsorganisation

Für die Zusammenarbeit und Organisation der anstehenden Aufgaben und zu verfassenden Kapitel wurde das Notizenprogramm *Notion* verwendet. Notion erlaubt das Erstellen kollaborativer Notizbücher, die parallel von mehreren Personen bearbeitet werden können. Über die Datenbankfunktion lassen sich sowohl Meilensteinpläne als auch Task-Boards erstellen, die zur Planung der Arbeit verwendet wurden.

3.3 Entwicklungsumgebung

Da sowohl das Pruning von Large Language Models als auch die anschließende Evaluierung der geprunten Modelle erhebliche Rechenressourcen erfordern, wurde entschieden, diese Prozesse auf einem leistungsstarken Server in der Cloud durchzuführen. Insbesondere das Testen der resultierenden Modelle stellt eine hohe Belastung für die verfügbaren Ressourcen dar und ist auf herkömmlicher Hardware nicht effizient durchzuführen. Zu diesen Zweck wurden EC2-Instanzen des Cloud-Providers AWS (Amazon Web Services) aus der *g*-Instanzfamilie verwendet. Diese Instanzfamilie ist speziell auf rechenintensive Anwendungen ausgelegt und bietet GPUs (Graphics Processing Units), die für parallele Berechnungen optimiert sind und den Einsatz von CUDA, dem von *NVIDIA* entwickelten Toolkit zur Verwendung von GPUs in allgemeinen Rechenaufgaben, ermöglichen.

Im Detail wurden Instanzen des Typs *gd4n.xlarge* verwendet, die mit 16 GiB Hauptspeicher, einem leistungsstarken Prozessor der *Intel Xeon Family* sowie einer *NVIDIA T4 Tensor Core* Grafikeinheit ausgestattet sind. Für den Start der Instanzen wurde das speziell auf Deep-Learning-Anwendungen zugeschnittene *Deep Learning OSS Nvidia Driver AMI GPU PyTorch 2.5.1 (Ubuntu 22.04) 20241208* Ubuntu-Image verwendet. In diesem Image ist das benötigte Python-Package *PyTorch* bereits vorinstalliert und ist zusätzlich mit *CUDA* ausgestattet.

Verwendet wurden im Detail Instanzen des Types *gd4n.xlarge*, die mit 16GiB Hauptspeicher, einem physischen Prozessor der *Intel Xeon Family* und einer *NVIDIA T4 Tensor Core* Grafikeinheit kommen. Gestartet wurden die Instanzen mit dem *Deep Learning OSS Nvidia Driver AMI GPU PyTorch 2.5.1 (Ubuntu 22.04) 20241208* Ubuntu Image, das bereits das benötigte Python-Package *PyTorch* vorinstalliert hat. Diese, auf Deep Learning ausgerichteten, Images haben neben PyTorch bereits *CUDA* vorinstalliert und ermöglichen so einen einfachen Start in das Arbeiten mit Programmen, die die GPU benötigen.

Nachdem über SSH (Secure Shell) mit der jeweiligen Instanz verbunde wurde, wurde zunächst die vorinstallierte PyTorch-Umgebung gestartet und anschließenden das Reposi-

Verwendete EC2-Instanz	
Instanz-Typ	gd4n.xlarge
Hauptspeicher	16GiB
vCPU	4
Clock Speed	2.5GHz
GPU	nvidia t4 tensor core
CPU	Intel Xeon Family
AMI	Deep Learning OSS Nvidia Driver AMI GPU PyTorch 2.5.1
AMI-ID	ami-0fa5e5fd27b3e163a

Tabelle 1: Attribute der verwendeten Hardware

tory des verwendeten Frameworks geklont.

```
$ source activate pytorch
$ git clone https://github.com/horseee/LLM-Pruner.git
```

Da die auf dem Image vorinstallierte PyTorch-Umgebung mit dem Conda-Package-Manager bereitgestellt wird, wurden die im Repository in der *requirements.txt*-Datei angebenen Package über Conda installiert.

```
$ conda install transformers sentencepiece datasets wandb ...
```

Damit ist die Versuchsumgebung identisch zu der in dieser Arbeit verwendeten Umgebung.

3.4 Verwendetes Large Language Model

Aufgrund von finanziellen und zeitlichen Einschränkungen wurde für diese Analyse das TinyLlama-Modell (*TinyLlama/TinyLlama-1.1B-Chat-v1.0*) als Basismodell gewählt. Mit 1,1 Milliarden Parametern ist es im Vergleich zu modernen, größeren Modellen wie *Llama 3.1* – das mit 405 Milliarden Parametern deutlich umfangreicher ist – relativ klein. Das verwendete TinyLlama wurde auf einem Datensatz von drei Milliarden Token vortrainiert und basiert auf der gleichen Architektur wie die *Llama 2*-Modelle. Diese Wahl ermöglichte es, innerhalb der verfügbaren Ressourcen eine Analyse durchzuführen, während gleichzeitig die Komplexität des Modells berücksichtigt wurde.

Der nachfolgende Auszug zeigt die detailierte Architektur des Modells:

```
LlamaModel(
  (embed_tokens): Embedding(32000, 2048)
  (layers): ModuleList(
    (0-21): 22 x LlamaDecoderLayer(
      (self_attn): LlamaSdpaAttention(
        (q_proj): Linear(in_features=2048, out_features=2048, bias=False)
```

```
7           (k_proj): Linear(in_features=2048, out_features=256, bias=False)
8           (v_proj): Linear(in_features=2048, out_features=256, bias=False)
9           (o_proj): Linear(in_features=2048, out_features=2048, bias=False)
10          (rotary_emb): LlamaRotaryEmbedding()
11        )
12        (mlp): LlamaMLP(
13          (gate_proj): Linear(in_features=2048, out_features=5632, bias=False)
14          (up_proj): Linear(in_features=2048, out_features=5632, bias=False)
15          (down_proj): Linear(in_features=5632, out_features=2048, bias=False)
16          (act_fn): SiLU()
17        )
18        (input_layernorm): LlamaRMSNorm((2048,), eps=1e-05)
19        (post_attention_layernorm): LlamaRMSNorm((2048,), eps=1e-05)
20      )
21    )
22    (norm): LlamaRMSNorm((2048,), eps=1e-05)
23    (rotary_emb): LlamaRotaryEmbedding()
24  )
```

Die Architektur besteht aus einer Embedding-Schicht, über die die Eingabesequenzen in Vektoren der Dimension 2048 umwandelt werden. Die Token-Embeddings basieren auf einem Vokabular von 32.000 Wörtern. Darauf folgt eine Modul-Liste mit 22 LlamaDecoderLayer, die jeweils aus mehreren Submodulen bestehen. Die Decoder-Schicht enthält einen Self-Attention-Mechanismus (LlamaSdpaAttention).

Um die grundlegenden Funktionen der ersten und letzten Schichten nicht zu beeinflussen, wurden sowohl im Attention-Abschnitt als auch im MLP-Abschnitt ausschließlich die Layer 4 bis 18 für das Pruning verwendet. Die Schichten der Architektur außerhalb des Decoder-Layers wie bspw. die Embedding-Schichten werden grundsätzlich nicht berührt, da sie für die Umwandlung der Texteingaben in die korrekte Repräsentation nötig sind.

3.5 Pruning

Das nachfolgende Kapitel beinhaltet die Vorgehensweise und verwendeten Technologien, die für das Pruning des TinyLlama-Modells verwendet wurde.

3.5.1 Verwendetes Framework

Für die Durchführung des Prunings standen zwei Frameworks zur Auswahl: *LLM-Pruner* und *Wanda* (Pruning by Weights and Activations). Während der *LLM-Pruner* ausschließlich das im vorherigen Kapitel beschriebene strukturierte Pruning unterstützt, bietet *Wanda* zusätzlich die Möglichkeit, unstrukturiertes Pruning durchzuführen. Trotz dieser zusätzlichen Funktionalität wurde für die Umsetzung dieser Untersuchung aus verschiedenen Gründen ausschließlich der *LLM-Pruner* verwendet.

Ein wesentlicher Faktor für diese Entscheidung war die Aktualität der Projekte. Die letzten Updates im *Wanda*-Projekt wurden Ende 2023 vorgenommen. Dies lag zum Zeitpunkt des Verfassens dieser Arbeit bereits mehr als ein Jahr zurück. Im Gegensatz dazu wird der *LLM-*

Pruner weiterhin aktiv weiterentwickelt, was eine aktuellere und besser unterstützte Basis bietet. Ein weiterer wichtiger Aspekt ist die Unterstützung spezifischer Modelle. Während beim *LLM-Pruner* explizit die Kompatibilität mit dem *TinyLlama*-Modell hervorgehoben wird, fehlt eine solche Aussage im Fall von *Wanda*. Es ist zwar anzunehmen, dass die Methoden von *Wanda* aufgrund der Unterstützung von *Llama-2* – dessen Architektur dem *TinyLlama* ähnlich ist – ebenfalls auf das *TinyLlama*-Modell anwendbar sein könnten. Allerdings bleibt dies ungewiss, da eine direkte Unterstützung nicht garantiert wird.

Ein weiterer entscheidender Punkt war, dass trotz mehrerer Versuche mit *Wanda* kein erfolgreiches Pruning durchgeführt werden konnte. Angesichts dieser Schwierigkeiten und unter Berücksichtigung der bereits genannten Argumente fiel die Entscheidung, sich ausschließlich auf den *LLM-Pruner* zu fokussieren.

3.5.2 Durchführung des Prunings

Das Pruning wurde, wie bereits erläutert, anhand des *TinyLlama*-Modells getestet und untersucht. Dabei wurden drei unterschiedliche Pruning-Ratios gewählt, um die Auswirkungen verschiedener Verkleinerungsgrade des Modells zu analysieren. Da das *LLM-Pruner*-Projekt bereits eigene Ergebnisse für das *TinyLlama*-Modell mit einer Pruning-Ratio von 20% veröffentlicht hatte, wurde dieses Verhältnis in der vorliegenden Untersuchung nicht erneut evaluiert. Stattdessen wurden die bereits existierenden Ergebnisse mit den Verhältnissen von 30%, 40% und schließlich 70% verglichen. Obwohl eine Verkleinerung um 70% bei einem ohnehin bereits kompakten Modell wie *TinyLlama* als unrealistisch erscheint, wurde dennoch im Rahmen der Untersuchung überprüft, welche Ergebnisse das Modell bei einer solch extremen Reduktion in den Tests liefert.

Das Pruning erfolgt stets über den folgenden Befehl, der ausgeführt werden kann, sobald sich im LLM-Pruner-Repository auf der höchsten Ebene befindet:

```
$ python llama3.py
    --base_model TinyLlama/TinyLlama-1.1B-Chat-v1.0
    --pruning_ratio [PRUNING_RATIO]
    --device cuda
    --eval_device cuda
    --block_wise
    --block_mlp_layer_start [START_MLP_LAYER]
    --block_mlp_layer_end [END_MLP_LAYER]
    --block_attention_layer_start [START_ATTENTION_LAYER]
    --block_attention_layer_end [END_ATTENTION_LAYER]
    --save_ckpt_log_name [SAVE_PATH]
    --pruner_type [PRUNER_TYPE]
    [--taylor param_first]
    --save_model
    --max_seq_len 2048
    --test_after_train
```

Für das Pruning des *TinyLlama*-Modells wurde das Skript *llama3.py* verwendet, das speziell für das Pruning von *Llama3*-Modellen entwickelt wurde. In dieser Untersuchung

diente stets das zuvor beschriebene Modell *TinyLlama/TinyLlama-1.1B-Chat-v1.0* als *base_model*. Wie bereits erwähnt, wurden für die Pruning-Ratio die Verhältnisse 0.3, 0.5 und 0.7 ausgewählt, um unterschiedliche Stufen der Modellreduktion zu analysieren.

Alle Prozesse wurden in einer CUDA-fähigen Umgebung ausgeführt, weshalb die Anweisung, die GPU für die Berechnungen zu verwenden, stets mitgegeben wurde. Von den insgesamt 22 verfügbaren MLP- und Attention-Layern des Modells wurden für das Pruning jeweils die Layer vier bis 18 berücksichtigt.

Bezüglich der verfügbaren Pruner-Typen bot der *LLM-Pruner* vier verschiedene Optionen an: *Taylor*, *L1*, *L2* und *random*. Jede dieser vier Methoden wurde für jede der drei gewählten Pruning-Ratios getestet und evaluiert, um ihre jeweiligen Auswirkungen auf die Modellleistung zu untersuchen.

Zusätzlich wurde bei allen Experimenten das Argument *test_after_train* verwendet. Dadurch wurde nach jedem Pruning automatisch die Perplexity des Modells ermittelt, basierend auf den beiden Testdatensätzen *wikidata2* und *ptb* (*Penn Treebank*).

Um das TinyLlama-Modell zu 30% mit der *Taylor*-Methode zu prunen sieht der Befehl demnach beispielhaft wie folgt aus:

```
$ python llama3.py
    --base_model TinyLlama/TinyLlama-1.1B-Chat-v1.0
    --pruning_ratio 0.3
    --device cuda
    --eval_device cuda
    --block_wise
    --block_mlp_layer_start 4
    --block_mlp_layer_end 18
    --block_attention_layer_start 4
    --block_attention_layer_end 18
    --save_ckpt_log_name tinyllama_30_0616_prune_log
    --pruner_type taylor
    --taylor param_first
    --save_model
    --max_seq_len 2048
    --test_after_train
```

3.6 Fine-Tuning

Für das nach dem Pruning stattfindende Fine-Tuning wird vom LLM-Pruner *PEFT* (Parameter-Efficient Fine-Tuning) verwendet. PEFT stellt eine Methode dar, um große vortrainierte Modelle an spezifische Aufgaben anzupassen, ohne den gesamten Parameterraum des Modells zu optimieren. Stattdessen wird nur ein kleinerer Teil der Parameter während des Trainings modifiziert.

In den Evaluierungen der Modelle, die direkt im Anschluss an das Pruning durchgeführt wurden, hat sich bereits gezeigt, dass über die Pruning-Methode *Taylor* die vielversprechendsten Ergebnisse erzielt werden konnte. Diese geprunten Modelle konnten die höchsten

Werte in den Evaluierungen erreichen. Das rechen- und kostenintensive Fine-Tuning wurde daher nur für die Modelle durchgeführt, die zu 30% und 40% mit der *Taylor*-Methode geprunt wurden.

BEFEHL EINFÜGEN!

3.7 Evaluierung der Modelle

Als Basis jeglicher durchgeführter Tests dienten jeweils die Ergebnisse des selbst durchgeführten Benchmarks des TinyLlama-Basismodells. Alle erhobenen Werte und Ergebnisse werden in Relation dazu bewertet.

3.7.1 Modell-Benchmarks

Die Modelle, die durch das Pruning und das anschließende Fine-Tuning entstanden sind, wurden anschließend auf ihre verbliebenen Fähigkeiten hin untersucht. Zu diesem Zweck wurden sie anhand verschiedener Datensätze evaluiert, die jeweils unterschiedliche Aspekte der Leistungsfähigkeit von LLMs testen. Welche spezifischen Aspekte dabei geprüft wurden, wurde bereits in den vorherigen Abschnitten detailliert beschrieben.

- *openbookqa*
- *winogrande*
- *hellaswag*
- *arc_challenge*
- *boolq*

Die Evaluierung anhand dieser Datensätze wurde, wie beim Pruning, mit dem immer gleichen Befehl - angepasst an das jeweilige Modell - durchgeführt.

```
$ export PYTHONPATH='.'
$ python lm-evaluation-harness/main.py
    --model hf-causal-experimental
    --model_args checkpoint=[PATH_TO_MODEL]/pytorch_model.bin,
            config_pretrained=TinyLlama/TinyLlama-1.1B-Chat-v1.0
    --tasks openbookqa,winogrande,hellaswag,arc_challenge,boolq
    --device cuda:0
    --no_cache
    --output_path [PATH_TO_RESULTS]
```

Wie im Befehl ersichtlich, wurde das *lm-evaluation-harness*-Framework von OpenAI verwendet, das bereits im Repository enthalten ist. Dieses Framework dient der Evaluierung von Sprachmodellen anhand verschiedener Benchmarks bzw. *Tasks*. Durch die Nutzung des Frameworks in Kombination mit den definierten *Tasks* wird eine standardisierte Bewertung der Modelle ermöglicht, was wiederum den Vergleich unterschiedlicher Modelle erleichtert.

Das Argument *–model hf-causal-experimental* wird übergeben, um die Nutzung der GPU während der Tests zu gewährleisten. Zusätzlich ist die Angabe des Pfads zum geprunten bzw. nachtrainierten Modell erforderlich, ebenso wie die Grundarchitektur des Basismodells. Die im Befehl spezifizierten *Tasks* entsprechen dabei den zuvor beschriebenen.

Ein beispielhafter Aufruf zur Evaluierung des Modells, das zu 30% geprunt wurde, sieht wie folgt aus:

```
$ export PYTHONPATH='.'
$ python lm-evaluation-harness/main.py
    --model hf-causal-experimental
    --model_args checkpoint=prune_log/tinyllama_30_0418_l1_prune_log/pytorch_model.bin,
        config_pretrained=TinyLlama/TinyLlama-1.1B-Chat-v1.0
    --tasks openbookqa,winogrande,hellaswag,arc_challenge,boolq
    --device cuda:0
    --no_cache
    --output_path results/tinyllama_30_0418_l1
```

3.7.2 Modellperfomance

Neben den verbliebenen Fähigkeiten des geprunten LLMs soll zusätzlich dessen Performance getestet werden. Die Messung der Performance erfolgt durch die Durchführung der zuvor beschriebenen Benchmarks. Während dieser Tests werden sowohl die benötigte Zeit als auch der in Anspruch genommene Speicher berücksichtigt, um eine Bewertung der Effizienz des Modells zu ermöglichen.

Für diese Messungen wird der in der *zsh*-Shell integrierte *time*-Befehl verwendet, der sowohl Angaben zur verbrauchten Zeit als auch zum genutzten Speicher liefert. Um diese Daten zu erfassen, wird der *time*-Befehl einfach den zuvor beschriebenen Befehlen zur Durchführung der Benchmarks vorangestellt. Ein entsprechender Aufruf zur Messung der Performance sieht wie folgt aus:

```
$ export PYTHONPATH='.'
$ time python lm-evaluation-harness/main.py
    --model hf-causal-experimental
    ...
```

4 Ergebnisse

Hier kommt alles zu den Ergebnissen hin!

5 Diskussion

Hier kommt alles zur Diskussion hin!