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 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.

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

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

3.2 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)
        (k_proj): Linear(in_features=2048, out_features=256, bias=False)
        (v_proj): Linear(in_features=2048, out_features=256, bias=False)
        (o_proj): Linear(in_features=2048, out_features=2048, bias=False)
        (rotary_emb): LlamaRotaryEmbedding()
      )
      (mlp): LlamaMLP(
        (gate_proj): Linear(in_features=2048, out_features=5632, bias=False)
        (up_proj): Linear(in_features=2048, out_features=5632, bias=False)
        (down_proj): Linear(in_features=5632, out_features=2048, bias=False)
        (act_fn): SiLU()
      )
      (input_layernorm): LlamaRMSNorm((2048,), eps=1e-05)
      (post_attention_layernorm): LlamaRMSNorm((2048,), eps=1e-05)
    )
  )
```

```
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.

3.3 Pruning

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

3.3.1 Framework

Für die Durchführung des Prunings standen zwei Frameworks zur Auswahl. *LLM-Pruner* und *Wanda* (Pruning by Weights and Activations). Während der LLM-Pruner nur das im vorherigen Kapitel beschriebene strukturierte Pruning unterstützt, setzt die Wanda-Methode auf unstrukturiertes Pruning.

3.3.2 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.

4 Ergebnisse

Hier kommt alles zu den Ergebnissen hin!

5 Diskussion

Hier kommt alles zur Diskussion hin!