GPT (to be changed)

Michael Gathara
University of Alabama at Birmingham
mikegtr@uab.edu

Yang (Jason) Liu
University of Alabama at Birmingham
yliu0602@uab.edu

Vaishak Menon
University of Alabama at Birmingham
vmenon19@uab.edu

Elisabeth Molen
University of Alabama at Birmingham
emolen@uab.edu

Trenton Davis
University of Alabama at Birmingham
trentd@uab.edu

Akshar Patel
University of Alabama at Birmingham
akshar2020@uab.edu

***Abstract*—Large language models (LLMs) are a subset of artificial intelligence (AI) models that have the ability to intake, understand, and generate human readable text. Since their inception, these models have found rapid success and widespread adoption in a vast range of industries and use cases including software engineering, healthcare, research, and supply-chain management. A specific type of LLM- generative pretrained transformers (GPTs)- stand out as one of the most used and adopted models as they are tuned to be highly effective at understanding conversational context and generating coherent and relevant text. Built on top of the transformer architecture, GPTs are often more finely tuned to perform specific tasks after first being trained on a large corupus of less specific textual data- hence the pre-trained portion of the name. While the transformer has remained the core of these models, many different optimizations have been proposed and implemented over the past several years including flash attention and multi-layer attention (MLA). In this work, we implement a scratch-built GPT equipped with some of these optimizations, train it on the fineweb dataset, and then test it on several tasks and analyze some of the features of the model.**

I. Introduction

In recent years, artificial intelligence (AI) models have been gaining popularity in both personal use as well as in industry adoption. Among these models, none seem to have risen to such a high level of popular adoption as large language models (LLMs) which are AI models that can intake, understand, and generate human-readable text [1]. Usage of these has gained traction in a broad range of industries including research, healthcare, supply-chain, and software engineering [2] [3] [4]. The most powerful LLMs are often built on an architectural framework called a transformer which calculates importance of different parts of the input and weighs the relationship between them in order to produce more coherent and relevant outputs. In their seminal 2017 paper, Vaswani et al. introduced the transformer architecture and set the stage for the many iterations of LLMs that have come in the years since [5]. This original paper focused on the core attention component of the transformer which calculates the interactions between different input tokens and uses this contextual knowledge to more dynamically generate relevant output text. While a transformative architecture, the core attention mechanism does suffer from a few key drawbacks that have motivated the need for some important optimizations in the years since. Because it calculates interactions between every input token pair, the attention mechanism suffers from a quadratic time and memory complexity leading to some amnesia inducing context length limitations. Additionally, models immediately following the transformer architecture follow a single uniform application of the attention mechanism at their different layers. This lack of heterogenity and depth didn't allow the models to fully capture the nuanced and dynamic interactions between the input tokens.

Generative pre-trained transformers (GPTs) are a specific instantiation of LLMs that use the transformer architecture as a backbone. GPTs are engineered to be highly effective at generating relevant text and at performing conversationally oriented tasks by first pre-training them on a huge corpus of text and then fine tuning them for their specific tasks- a process that was introduced by Radford et al. [6]. Following that initial introduction, GPTs utilizing magnitudes more parameters began to be explored- consistently increasing their usefulness in conversational tasks. Specifically, post-training capacity of the models to learn and respond to instructions deliverd to them as inputs was detailed by Brown et al. in their 2020 paper introducing GPT-3 [1]. While making significant bounds in model performance and practical usefulness, these architectures still suffered from the same core limitations of the original transformer setup that they are built on top of.

In the present work, we implement a from-scratch trained GPT architecture but we also explore some of the optimizations that have been introduced to address the aformentioned limitations of the transformer architecture. Two of the core optimizations that we utilize are flash-attention and multi-layer attention. Flash attention addresses the issue of quadratic memory complexity in the original attention mechanism- an issue that significantly slows down the computation as expensive GPU memory transfers abound with full attention scores being computed and scored. Flash attention focuses on making the attention mechanism input/output (IO) aware and splits the computations into tiles that can fit on a GPUs on-chip memory and eschew temporally costly IO operations [7]. In our setup, we use flash attention and see significant improvements in our model training time as well as inference.

II. Methods

We present a transformer-based language model that employs a novel attention mechanism called Multi-headed Latent Attention (MLA). This section details the architecture of our model and the training methodology.

A. Model Architecture

Our model follows the general transformer architecture [5] with several key modifications. The model consists of 12 transformer layers, each containing a multiheaded latent attention module and a feed-forward network. We use an embedding dimension of 768 and 12 attention heads throughout the model. The context window is limited to 512 tokens. The input token sequence is first embedded using a token embedding layer and combined with learned positional embeddings:

$$h_0 = W_e \cdot x + W_p \tag{1}$$

where $W_e \in \mathbb{R}^{V \times d}$ is the token embedding matrix for vocabulary size V and embedding dimension d, and $W_p \in \mathbb{R}^{n \times d}$ is the positional embedding matrix for a maximum sequence length n.

1) Multi-headed Latent Attention (MLA): Jason do you want to write about MLA here?

2) Feed-Forward Networks: After the attention layer, each transformer block contains a feed-forward network using SwiGLU activation [8], which has been shown to outperform ReLU in language models. The feed-forward network is defined as:

$$\text{Swish}(xW_1) \otimes (xW_2)\ \text{FFN}(x) = \text{SwiGLU}(x)W_3 \tag{2}$$

where $W_1, W_2 \in \mathbb{R}^{d \times 4d}$ and $W_3 \in \mathbb{R}^{4d \times d}$ are learned parameter matrices, $\text{Swish}(x) = x \cdot \sigma(x)$ is the swish activation function, and $\otimes$ represents element-wise multiplication. Each attention and feed-forward sublayer is wrapped with a residual connection and layer normalization:

$$h'_i = \text{LN}(h_i) \tag{3}$$

$$h_{i+\frac{1}{2}} = h_i + \text{MLA}(h'_i) \tag{4}$$

$$h''_{i+\frac{1}{2}} = \text{LN}(h_{i+\frac{1}{2}}) \tag{5}$$

$$h_{i+1} = h_{i+\frac{1}{2}} + \text{FFN}(h''_{i+\frac{1}{2}}) \tag{6}$$

where LN denotes layer normalization.

3) Custom Byte Pair Encoding:

a) Configuration and Hyperparameters: We train a custom Byte-Pair Encoding (BPE) tokenizer [9] on the Wikitext-103 corpus with a vocabulary size of 25,000. The tokenizer includes special tokens `[PAD]`, `[UNK]`, `[CLS]`, `[SEP]`, `[MASK]`, `[BOS]`, and `[EOS]`. We use whitespace pre-tokenization and set a minimum token frequency of 2 to filter out rare tokens.

b) Data Preprocessing: The raw Wikitext-103 training split is first cleaned to remove markup and normalize whitespace. To process the corpus efficiently, we perform the cleaning in grouped batches using multiprocessing. Each cleaned line is then split on whitespace to produce an initial sequence of byte tokens for BPE training. Our cleaning routine leverages pre-compiles regular expressions to strip wiki-specific formatting, correct hyphenation and punctuation spacing, and collapse multiple spaces. The resulting cleaned lines are saved as a newline-delimited file on disk, providing a stable noise-reduced input for the BPE trainer.

c) Tokenizer Construction: We instantiate a `Tokenizer` with a BPE model using `unk_token="[UNK]"` to represent out-of-vocabulary subwords. A `Whitespace` pre-tokenizer is attached to split the cleaned text into initial byte tokens. We then configure a `BpeTrainer` with our target vocabulary size (25 000), minimum merge frequency (2), and the reserved special tokens. The trainer drives the merge process, repeatedly identifying and merging the most frequent adjacent byte pairs until the vocabulary size is reached.

d) Tokenizer Training and Implementation: BPE training is performed in a single pass over the cleaned text, merging byte-pairs until the vocabulary criterion is met. The resulting vocabulary and merge rules are then saved as a JSON file for reproducible tokenization. The system is implemented with the Rust-optimized core of the Hugging Face Tokenizers library, which parallelizes intensive steps across CPU cores, and uses buffered I/O to efficiently manage large files.

e) Tokenizer Loading and Consistency: At inference, we load the JSON file produced by training; if it's missing, the cleaning and training pipeline is re-invoked automatically. This guarantees that both training and evaluation share an identical subword mapping.

B. Training

1) Dataset: We train our model on the Wikitext-103 corpus [10], a collection of good quality Wikipedia articles containing approximately 1.8 million tokens. Prior to tokenization, we apply a text cleaning procedure to remove markup elements and normalize whitespace. The cleaned text is then tokenized and chunked into sequences of 512 tokens for training.

2) Training Process: The model is trained using distributed data parallelism across multiple NVIDIA H100 GPUs. We employ mixed precision training (FP16) with gradient scaling to improve computational efficiency while maintaining numerical stability. The training process uses a batch size of 72 per GPU with gradient accumulation over 4 steps, resulting in an effective batch size of 288. We implement model parallelism using PyTorch's DistributedDataParallel (DDP) framework. The dataset is partitioned across GPUs using a DistributedSampler to ensure each worker processes different data. The training loop includes the following steps:

1) Sample a batch (x, y) from the data loader
2) Perform a forward pass with mixed precision: $\text{logits}, \text{loss} = \text{model}(x, y)$
3) Scale the loss

4) Perform a backward pass with gradient scaling
5) If $i \bmod \text{accumulation\_steps} = 0$:
    a) Clip gradients to maximum norm 1.0
    b) Update model parameters
    c) Zero gradients
    d) Update learning rate with scheduler
6) If $i \bmod \text{eval\_interval} = 0$:
    a) Evaluate on validation set
    b) Save checkpoint if validation loss improved

This process continues for a maximum of 15,000 iterations or until convergence.

3) Optimization: We optimize our model using AdamW [11] with a weight decay of 1×10^{-4} and beta parameters $\beta_1 = 0.9$, $\beta_2 = 0.95$. The initial learning rate is set to 1×10^{-3}. Our learning rate schedule combines a linear warmup phase with a cosine decay phase:

$$\text{lr}(t) = \begin{cases} \text{lr}_{\text{max}} \cdot \frac{t}{\text{warmup}} & \text{if } t < \text{warmup} \\ \dots \end{cases} \tag{7}$$

where warmup $= 500$ iterations and max $= 15,000$ iterations. This schedule helps stabilize early training and gradually reduces the learning rate to achieve better convergence. where *warmup steps* eq 500 and *max steps* eq 15,000. This schedule helps stabilize early training and gradually reduces the learning rate to achieve better convergence. To prevent gradient explosion, we apply gradient clipping with a maximum norm of 1.0 before each optimizer step.

4) Hyperparameter Tuning: Elizabeth do you want to write about your hyperparameter tuning and experiments here?

5) Evaluation and Checkpointing: We evaluate the model on the validation set every 100 iterations by computing the average loss over multiple batches. The best model is selected based on the lowest validation loss achieved during training. Additionally, we save periodic checkpoints every 500 iterations to enable resumption of training if needed. To monitor training progress, we log key metrics including training loss, validation loss, learning rate, and throughput (tokens processed per second) using TensorBoard. After training, we generate sample text from the best checkpoint to qualitatively assess the model's capabilities.

C. Testing

For the testing of our model, we used two different primary metrics: loss and perplexity. The loss function that we utilized both in our model training as well as in the testing is the standard cross-entropy loss i.e.

$$\mathcal{L}_{\text{CE}} = -\frac{1}{N} \sum_{i=1}^{N} \sum_{c=1}^{C} y_{i,c} \log p_{i,c}, \tag{8}$$

where $y_{i,c} \in \{0, 1\}$ is the one-hot reference label for example i and class c, and $p_{i,c}$ is the model's predicted probability for that class. And the perplexity is defined as:

$$\text{PPL} = \exp\Big(-\tfrac{1}{T} \sum_{t=1}^{T} \log p(w_t \mid w_{<t})\Big), \tag{9}$$

where $p(w_t \mid w_{<t})$ is the model's predicted probability of token w_t given its history $w_{<t}$. In addition to these two metrics, we also qualitatively evaluate the model's performance by generating sample text from each of the model checkpoints that we test.

III. Results

IV. Discussion

V. Conclusion

REFERENCES

[1] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[2] Weixin Liang, Yaohui Zhang, Zhengxuan Wu, Haley Lepp, Wenlong Ji, Xuandong Zhao, Hancheng Cao, Sheng Liu, Siyu He, Zhi Huang, Diyi Yang, Christopher Potts, Christopher D Manning, and James Y. Zou. Mapping the increasing use of llms in scientific papers, 2024.

[3] Chao Zhang, Qingfeng Xu, Yongrui Yu, Guanghui Zhou, Keyan Zeng, Fengtian Chang, and Kai Ding. A survey on potentials, pathways and challenges of large language models in new-generation intelligent manufacturing. *Robotics and Computer-Integrated Manufacturing*, 92:102883, 2025.

[4] Ashok Urlana, Charaka Vinayak Kumar, Ajeet Kumar Singh, Bala Mallikarjunarao Garlapati, Srinivasa Rao Chalamala, and Rahul Mishra. Llms with industrial lens: Deciphering the challenges and prospects – a survey, 2024.

[5] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[6] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. OpenAI Technical Report, 2018.

[7] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022.

[8] Noam Shazeer. Glu variants improve transformer, 2020.

[9] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units, 2016.

[10] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models, 2016.

[11] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization, 2019.