CSCE 689: Special Topics in Programming Large Language Models (LLMs)
Homework 2
Student: Mu-Ruei Tseng
UIN: 133007868

Overview

In this homework, I implemented a variant of GPT-2 from scratch, based on the Python implementation from Andrej Karpathy's llm.c repository. Given the limited dataset size (fineweb10B) and training resources (A100 GPU for 24 hours), it was crucial to choose a suitable number of parameters and implement efficient techniques to ensure the model converged faster. According to the scaling laws for large language models (3), there is a trade-off between model size and dataset size: as one increases, the other should also increase proportionally to avoid underfitting or overfitting. Since our dataset is relatively small (10B tokens) and training time is short, I experimented with two smaller model sizes: 124M and 350M. The checkpoint corresponding to the step with the lowest validation loss was selected for evaluation. Both models were evaluated on the HellaSwag dataset, yielding normalized accuracies of **30.78%** and **33.52%** for the 124M and 350M models, respectively.

Model and Architectures

Based on the original GPT-2 structure (1), I made five key modifications: switching from post-normalization to pre-normalization, using Rotary Position Embedding (RoPE) in place of traditional global positional embedding, incorporating Grouped Query Attention (GQA), modifying the Feed-Forward Layer to use SwiGLU rather than GELU, and replacing LayerNorm with RMSNorm.

- ***Pre-normalization*** Pre-normalization was used in GPT-3 (2), where normalization is performed before the residual connection. Using pre-normalization helps increase the stability for deeper models and provides better gradient flow since the inputs to both the self-attention and feed-forward layers are normalized before any transformations. This prevents activations from growing too large or small, thus reducing the risk of exploding or vanishing gradients.

- ***Rotary Position Embedding*** Su et al. (5) proposed Rotary Position Embedding (RoPE) to effectively leverage positional information as an alternative to the learned global positional embeddings used in the GPT-2 structure. RoPE introduces positional information by applying a rotational transformation to the query and key vectors within the self-attention mechanism. The rotation matrix is fixed and can be pre-computed, which differentiates it from learned positional embeddings that require additional parameters. A major limitation of using global positional embeddings is that they do not directly encode the relative distance between tokens. Instead, they only provide information about the absolute positions of tokens in a sentence, forcing the model to infer relative distances implicitly, which becomes more difficult with longer sequences. When we understand a sentence, we typically focus on the relationships between words rather than their absolute positions. RoPE addresses this issue by directly encoding relative distances into the attention calculation for the query (Q) and key (K) vectors, making it more effective for capturing long-range dependencies.

- ***Grouped Query Attention*** Grouped Query Attention (GQA), introduced by Ainslie et al. (6), offers a trade-off between Multi-Query Attention (MQA) and Multi-Head Attention (MHA). The core idea is to group several query heads into subgroups and compute the attention using a single shared key and value head for each group. By grouping the query heads, GQA reduces the total number of key (K) and value (V) parameters, which results in fewer key and value projections, reducing both memory usage and computation time. This not only makes the model more efficient

but also helps to reduce the overall model size. Moreover, GQA enhances the model's ability to handle relative attention across multiple heads, as the grouping mechanism allows for more effective focus on the relationships between tokens, rather than just their individual positions. This is especially beneficial in tasks requiring long-range dependencies, where relative positioning is crucial for understanding context. In this model, we set the group size to 2, meaning that every two query heads share the same key and value head.

- ***FFN with SwiGLU*** The Feed-Forward Network (FFN) with SwiGLU (7) replaces the original Gaussian Error Linear Units (GELU) activation function with Swish Gated Linear Units (SwiGLU). SwiGLU is a neural network layer that applies a component-wise product of two linear transformations of the input, with one being activated by the Swish function. Additionally, it reduces the hidden dimension to ⅔ of the original size to maintain consistency with the original FFN architecture. This configuration has demonstrated superior average performance on the SuperGLUE benchmark (8).
- ***Replacing LayerNorm with RMSNorm*** LayerNorm computes the mean and standard deviation across the feature dimension of a specific input, then normalizes the input by subtracting the mean and dividing by the standard deviation. This process normalizes both the mean and variance of the features—in the case of LLMs, the embeddings. RMSNorm (9), on the other hand, normalizes the features based only on the calculated root mean square (RMS). It is a simpler operation compared to LayerNorm, as it only has one learnable parameter (scale), while LayerNorm has two learnable parameters: scale and shift. As a result, RMSNorm typically requires less computation and can be more stable in certain cases because it avoids the need to subtract the mean.

Model Training Details

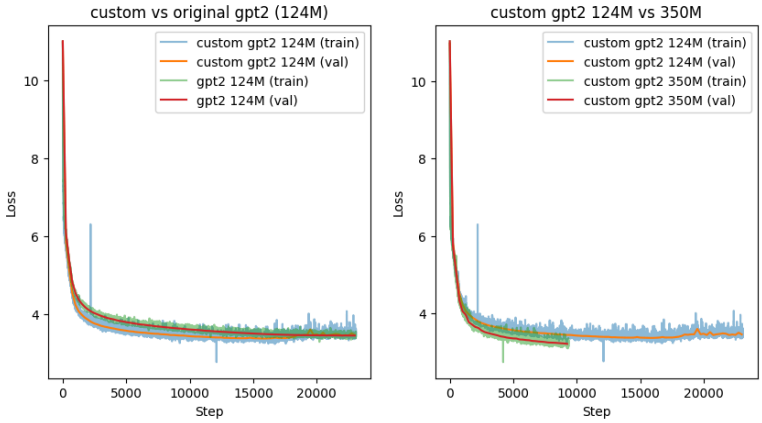
I followed the hyperparameter suggestion from the GPT3 paper (2). The batch size was set to 524288 (~0.5M tokens), as larger batch sizes provide more stable gradient estimates and enable larger models to converge faster. Additionally, the batch size should scale with model size to efficiently utilize computational resources (3). However, due to limited GPU memory, I set the mini-batch size to 8 and used gradient accumulation to achieve the same effective batch size. With a sequence length of 1024, each mini-batch processes approximately 8192 tokens. For the optimizer, I used AdamW instead of the standard Adam optimizer, as it decouples weight decay from the gradient updates, which improves Adam's generalization performance (4). β_1, β_2, and ϵ is set to 0.9, 0.95 and 10^{-8}respectively. For the learning rate, I adopted the cosine decay schedule that reduces the learning rate to 10% of its maximum value after 19073 steps (roughly 1 epoch, 10B tokens). A linear LR warmup is also used, starting from the 10% of the maximum learning rate and reaches the maximum after 715 steps (roughly 375M tokens). The maximum learning rate is set to 0.0012. Additionally, a weight decay of 0.1 was chosen to provide regularization.

To improve token throughput and reduce the time needed to process each step, several implementation techniques can be used. First, I utilized torch.compile() to compile the LLM, reducing computational overhead. Additionally, I adjusted the precision settings from "highest" (float32) to "high" for faster calculations. With "high" precision, float32 matrix multiplications use the TensorFloat32 data type (10 mantissa bits). This adjustment maintains a good balance between speed and accuracy, allowing efficient internal computations with reduced precision. Lastly, I used torch.autocast with dtype=bfloat16 to perform mixed precision computation, which automatically uses lower precision for specific operations.

Results

In this section, we present the loss graph for the fine-tuned 10B dataset and the evaluation results on HellaSwag. First, I compared the loss progression between the original GPT-2 model and our custom 124M-parameter variant, as shown in the figure below. The custom GPT-2 architecture demonstrates faster convergence and achieves a lower validation loss given the same number of steps. These modifications result in a slight accuracy improvement on HellaSwag, increasing from the original GPT-2's 29.4% to 30.78%, even with limited computational resources.

In line with scaling laws, model size plays a crucial role in improving performance. To explore this, I trained the custom GPT-2 model with 350M parameters under the same conditions. Despite completing only around 10,000 steps, the larger model shows significantly lower loss compared to the 124M version. This larger model further boosted the accuracy to 33.52%, approaching the performance of GPT-3's 124M model.



Model Card and Code Repository

Here, I provided the link towards both the models:

- gpt2-124M-gqa: git clone https://huggingface.co/Morris88826/Mu-Ruei_Tseng_133007868_124M
- gpt2-350M-gqa: git clone https://huggingface.co/Morris88826/Mu-Ruei_Tseng_133007868_350M

Code: https://github.com/Morris88826/CSCE-689-HW2

Problems and Difficulties

One major issue for this assignment is to convert the custom gpt2 variant model's checkpoint into suitable format that can be used for the lm-evaluation-harness repository. The custom model's configuration must inherit from transformers.PretrainedConfig, and the model itself should inherit from transformers.PreTrainedModel. This is necessary to integrate the model into Hugging Face's framework. After modifying the model, It is also important to save the config (GPTConfig), model (GPT), and tokenizer (GPT2Tokenizer).

Since lm-evaluation-harness repository doesn't natively support custom models. To evaluate my model, I need to modify the main.py by registering your model and config. Specifically, in main.py, register the custom model and config as follows:

AutoConfig.register(<model_type>, GPTConfig)
AutoModelForCausalLM.register(GPTConfig, GPT)

This solution allows for custom PyTorch models to be evaluated with lm-evaluation-harness by utilizing Hugging Face's framework for handling model saving/loading and evaluation.

References

[1] Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). Language Models are Unsupervised Multitask Learners.

[2] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D., Wu, J., Winter, C., … Amodei, D. (2020). Language Models are Few-Shot Learners. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, & H. Lin (Eds.), Advances in Neural Information Processing Systems (Vol. 33, pp. 1877–1901). Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[3] Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., & Amodei, D. (2020). Scaling Laws for Neural Language Models. ArXiv, abs/2001.08361. https://api.semanticscholar.org/CorpusID:210861095

[4] Loshchilov, I., & Hutter, F. (2019). Decoupled Weight Decay Regularization. https://arxiv.org/abs/1711.05101

[5] Su, J., Lu, Y., Pan, S., Murtadha, A., Wen, B., & Liu, Y. (2023). RoFormer: Enhanced Transformer with Rotary Position Embedding. https://arxiv.org/abs/2104.09864

[6] Ainslie, J., Lee-Thorp, J., de Jong, M., Zemlyanskiy, Y., Lebrón, F., & Sanghai, S. (2023). GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints. https://arxiv.org/abs/2305.13245

[7] Shazeer, N. (2020). GLU Variants Improve Transformer. https://arxiv.org/abs/2002.05202

[8] Wang, A., Pruksachatkun, Y., Nangia, N., Singh, A., Michael, J., Hill, F., Levy, O., & Bowman, S. R. (2020). SuperGLUE: A Stickier Benchmark for General-Purpose Language Understanding Systems. https://arxiv.org/abs/1905.00537

[9] Zhang, B., & Sennrich, R. (2019). Root Mean Square Layer Normalization. https://arxiv.org/abs/1910.07467