Default Project: Optimizations on Large Language Models

Jason Wang* jsywang@stanford.edu
Zhoujie Ding* d1ng@stanford.edu
Yiming Ni* yimingni@stanford.edu

1 Introduction

This project is the default project of CS229S. It explores various techniques of optimization large language models during training and inference time to improve the model's efficiency and hardware utilization. We focused on optimization for GPT-2, which builds on the open-sourced nanoGPT [1] and tested these methods through training and evaluating on the Wikitext103 causal language modeling task. Specifically, we applied quantization and pruning, which are the required methods, as well as a variety of other techniques that we will detail in section 2.

2 Background and Methods

2.1 Quantization

We implemented 8-bit quantized inference, post-training quantization on the model parameters with the GPT-2 pretrained model [2]. Our implementation achieved similar inference perplexity to the original model (section 3.1)

Our quantization method works on quantizing weights in Linear layers only, ignoring bias in Linear layers and parameters in LayerNorm to preserve nanoGPT's performance. For a given linear layer, we quantize every row of its weight matrix following symmetric quantization mode.

$$
\begin{aligned}
q &= 2^{8-1} - 1 \\
s &= \frac{q}{\text{absmax}(x)} \\
Q(x) &= \text{int}_8(\text{clamp}(\text{round}(x \cdot s), -q, q)) \\
Q^{-1}(z) &= \frac{\text{float}_{16}(z)}{s}
\end{aligned}
$$

This compresses the range from $\pm\text{absmax}(x)$ to $\pm q$. During forward operation, the integer weights are dequantized back to floating point weights to perform matrix multiplication with the input. This is also known as W8A16 (8-bit weights and 16-bit activations).

2.2 Pruning

Iterative Magnitude-based Pruning decreases the model size and compresses the model size [3]. In the method, the model is iteratively finetuned and pruned. In this project, we implemented individual weights pruning where weights with the lowest absolute value were set to zero, and dimension pruning where the dimension with the lowest L2 norm was pruned. We also tried two different protocols for pruning. In the first protocol, we prune a certain portion of the original model's weights. In the second protocol, we continued to prune an arbitrary amount of the weights until the validation loss exceeded a threshold. We include more details on our implementation in section 3.2. The evaluation includes a scrutiny of model size, quality, and a qualitative comparison between pruning individual weights and dimensions.

*Equal contribution

2.3 KV Cache

We used the KV cache technique to speedup inference performance. KV cache works by caching all the previous keys and values so that the model only needs to take in the last token (query) to generate a new token.

2.4 Speculative Decoding

In the decoding steps, the dominant cost is moving memory around because for each token sampled, we need to read multiple blocks of multi-headed attention and multilayer perceptron layers. Hence the process is memory bound. To utilize the spared compute, we used speculative decoding [4][5] to assemble bigger batches when the batch size is small. In particular, we use a smaller draft model to guess the next few tokens and add them to the batch. Then we can run the entire batch in parallel on the main model. While decoding, we check if the main model agrees on the guessed tokens.

2.5 Kernel Fusion

We noticed a lot of operations within the transformer model can be fused together. Therefore, we experimented with a few of the options by implementing fused kernels using OpenAI Triton. Unfortunately, these kernels all turned out to not provide many speedups, so we decided to not use them in our final submission.

2.5.1 W8A16 Linear

As mentioned above, during forward operation, the quantized linear weights are first dequantized, and then normal 16-bit matrix multiplication is carried out. Fusing these two operations can save some memory bandwidth for writing and reading the dequantized weights in memory. This kernel can be applied to all linear layers since we quantize all linear layers.

2.5.2 Linear + Bias + Activation

The usual forward operation of a linear layer is matrix multiplication, adding bias, then performing activation function. These three operations can be naturally fused together to save memory bandwidth for writing and reading intermediate values. This kernel can be applied to all first layers in the MLP block.

2.5.3 Bias + Dropout + Residual

The last operations in an attention block is usually a linear layer followed by a dropout layer and then adding residual connection. This kernel fuses the bias part of the linear layer with dropout and residual connection, which can be applied to the end of attention block as well as the MLP block.

2.5.4 Linear + Bias + Dropout + Residual

Following similar idea, we also tried fusing the linear operation with the previous kernel.

2.6 torch.compile

Following the idea of KV cache, we found that it can be decomposed into two operations - prefill and generation. Prefill is the first forward pass, saving the prompt into the KV cache. Generation is the stage where only the last token is provided and the model predicts the next token using the KV cache.

Note that the generation operation has a fixed size input of (batch size, 1) where 1 is the sequence length. Therefore, we can use torch.compile to compile specifically for this fixed size input output operation in 'reduce-overhead' mode to save memory bandwidth when the input is a vector.

Batch Size	With Quantization (seconds)	Without Quantization (seconds)
4	41.8604 (train) 43.0131 (val)	41.9369 (train) 43.0793 (val)
12	41.8611 (train) 42.4083 (val)	42.5210 (train) 41.9354 (val)

Table 1: Wikitext perplexity score at inference time (deliverable for part 1.1)

Batch Size	With Quantization (model size 0.412 GB)	Without Quantization (model size 1.42 GB)
4	2.51 GB (after inference)	4.23 GB (after inference)
12	6.67 GB (after inference)	8.42 GB (after inference)

Table 2: Memory usage at inference time (deliverable for part 1.2)

3 Experiment Setup and Results

3.1 Training and Evaluation Based on Quantization (Deliverable Part 1)

Our implementation of post-training quantization achieved similar perplexity score at inference time compared to the original model. For more details, we evaluated at inference time with batch sizes 4 and 12 and reported the respective wikitext perplexity score in table 1. We provide a detailed explanation on how our implementation prevents information loss in section 4.1.1.

With our quantization, the model used significantly less memory during inference evaluation. We also reported the memory usage for both batch sizes 4 and 12 in table 2. In section 4.1.2 we explain why we achieved a memory reduction.

Inspired by the empirical observation that draft models trained with signals from the main model display increased agreement that can speed up speculative decoding, we tested using our quantized model as the draft model and GPT-2 Medium as the main model. We compared the standard decoding using GPT-2 Medium, speculative decoding with GPT-2 Small as the draft model and with our quantized GPT-2 Medium as the draft model (table 3). Intuitively, we would expect our method to be faster than the others due to higher agreement. However, we found the latency of our method to be the highest. We provide a detailed explanation on this observation in section 4.1.3.

3.2 Training and Evaluation Based on Pruning (Deliverable Part 2)

To provide qualitative evaluation to our pruning method, we finetuned our model for a total of 100 iterations with gradient accumulation of 40. During our iterative pruning process, we gradually incremented our pruning ratio until we reached 10% of the original model size. We reported a plot of pruned model size, which is the number of non-zero parameters against the quality on the wikitext task which we used the validation loss in this case (fig 1). We include more insights on this plot in section 4.2.

We also studied the impact of pruning dimensions. Instead of pruning individual weights with the lowest magnitudes, we pruned the dimensions of the weights along the model dimension with the lowest L2 norm.

For every linear layers' weight matrix (with shape out features times in features), we prune the columns with the lowest L2 norm. We implemented a custom linear layer class which slices the input's column and the weight's column in the forward pass. We noticed that this resulted in lower throughput in training and inference. This is because of the extra slicing operations which introduces more memory usage, which could dominant the forward operation. To mitigate this issue, we

Decoding	Main Model	Draft Model	Latency (seconds)
Standard	GPT-2 Medium	-	0.812
Speculative	GPT-2 Medium	GPT-2 Small	0.555
Speculative	GPT-2 Medium	Quantized GPT-2 Medium	0.848

Table 3: Inference latency for generating 50 tokens with a prompt of length 1024 with batch size 1 with different decoding methods and models (deliverable for part 1.3 and 1.4)

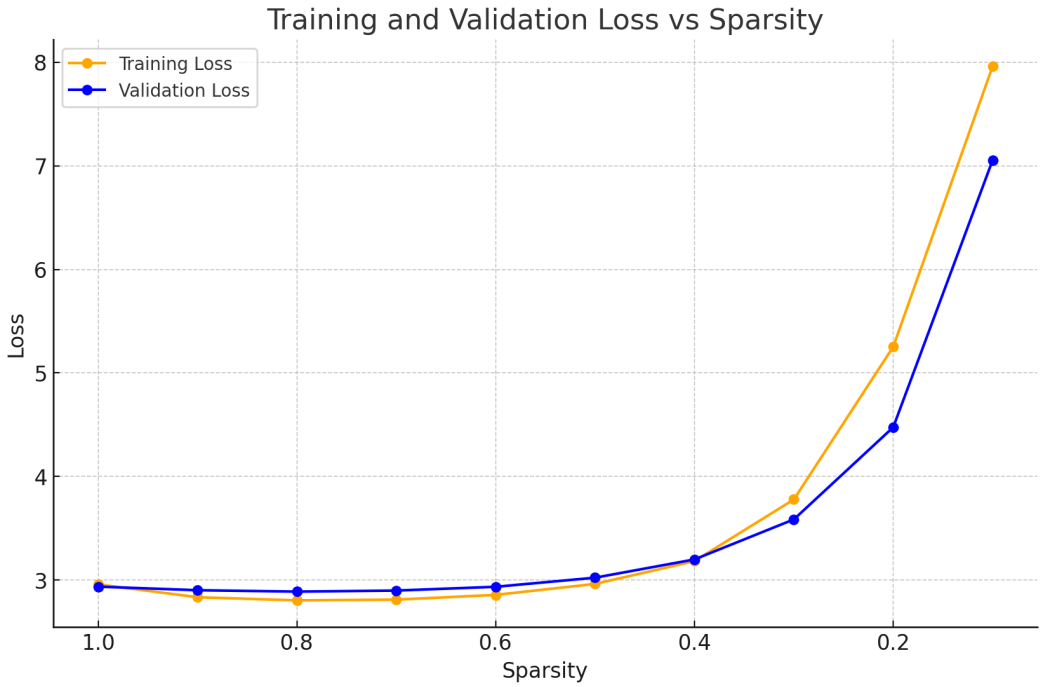

Figure 1: Plot of pruned model size vs. the quality on the wikitext task for pruning individual weights (deliverable for part 2.1). **Note that sparsity denotes the ratio of the pruned model size to the original model size.**

precompute the sliced weights before inference, thus saving around 50% of the extra slicing memory usage. We were then able to observe speedup in throughput when model size decreases.

We compared the pruned model size against the quality as well as against the speed, in this case the number of tokens generated per second (fig 3).

We repeated the above two studies with a different pruning protocol. Specifically, after the initial 50 iterations of training, we prune the model slightly and finetuned it for 25 more iterations. We repeated these steps until the training loss is above 3.2. We reported the model size against the validation loss for both pruning individual weights and dimensions, as well as the model size against the inference speed for pruning dimensions (fig 2 and 4).

3.3 Overall Training and Evaluation (Leaderboard)

For the leaderboard submission, we focused on optimizing inference throughput.

Our final leaderboard submission contains the following techniques from Section 2.

- A fixed-size KV cache is used for attention. To ensure correct result is generated, we used a mask in flash attention to mask out the unused positions in key and value.
- KV decoding is split into two stages: prefill and decode where prefill has dynamic shape input and decode is fixed size input.
- The model is compile by torch.compile in "reduce-overhead" model to optimize memory usage. Since KV decoding has a fixed size input for all the modules in the decode stage now, we can compile end-to-end while disabling dynamic shapes, which will allow the model to produce more optimized kernels.

We summarized our model performance in table 3.3.

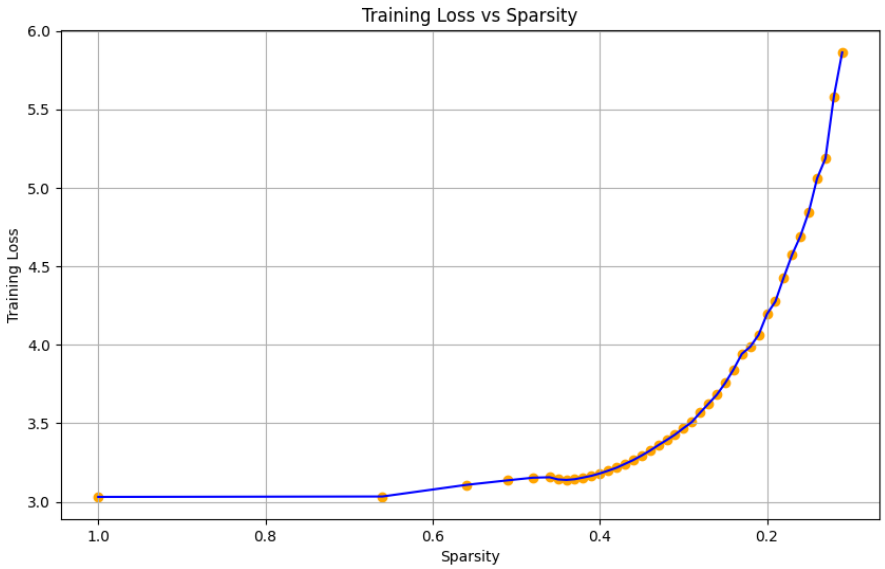

Figure 2: Plot of pruned model size vs. the quality on the wikitext task for pruning individual weights (deliverable for part 2.1). **Note that sparsity denotes the ratio of the pruned model size to the original model size.**

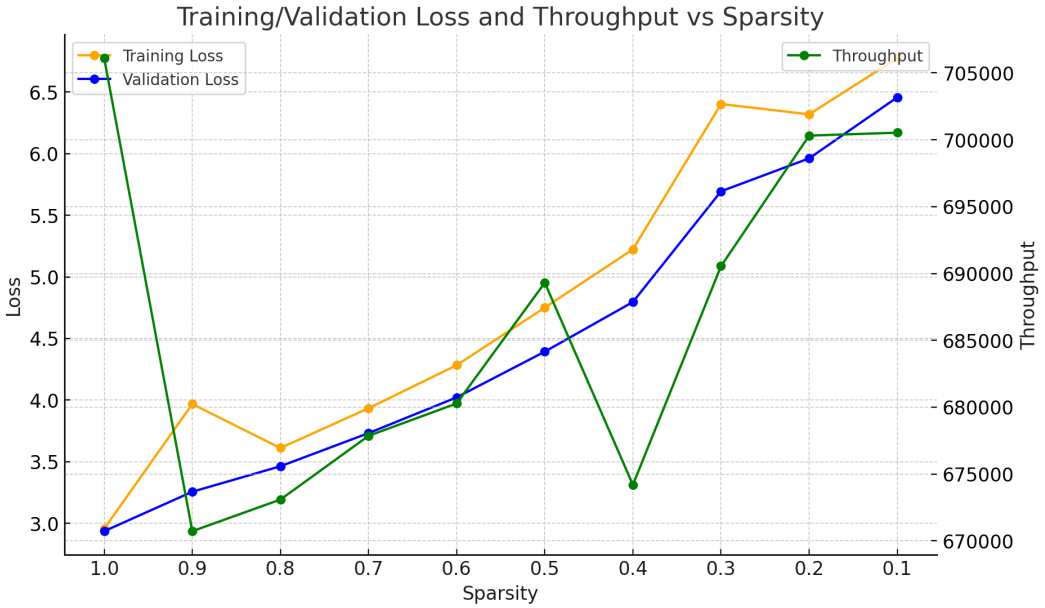

Figure 3: Plot of pruned model size vs the quality and the pruned model size vs the speed for pruning dimension (deliverable for part 2.2). **Note that sparsity denotes the ratio of the pruned model size to the original model size.**

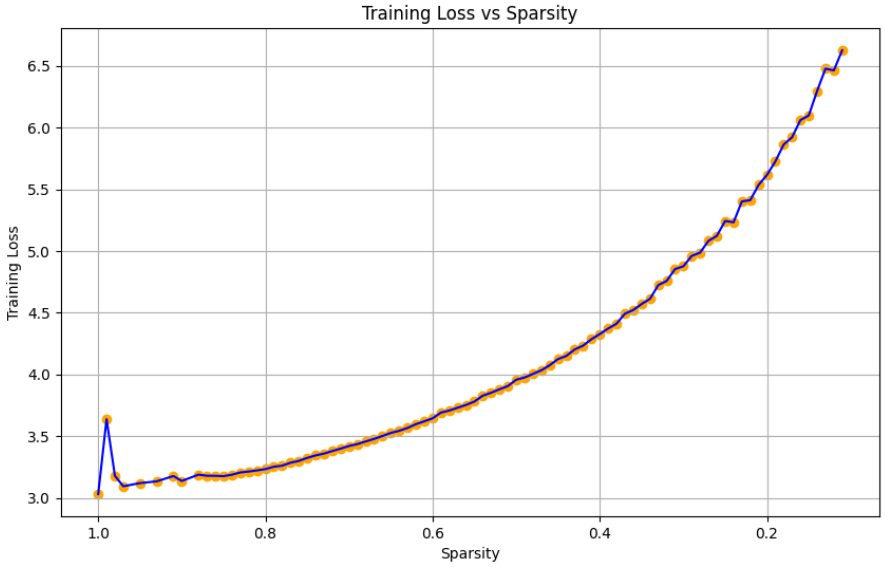

Figure 4: Plot of pruned model size vs the quality and the pruned model size vs the speed for pruning dimension (deliverable for part 2.2). **Note that sparsity denotes the ratio of the pruned model size to the original model size.**

Model Performance	Value
Loss	3.11
Training throughput (batch size 4)	8426.24
Training throughput (batch size 12)	111773.49
Inference throughput (batch size 4)	789.12
Inference throughput (batch size 12)	5157.07

Table 4: (deliverable for part 3)

4 Experiment Analysis

4.1 Evaluation Analysis Based on Quantization (Deliverable Part 1)

4.1.1 Perplexity

From Table 1, we observe that the perplexity scores of the quantized model is very close to the scores. This is because we quantized the weights in groups. If we instead directly quantize the whole weight matrix, the performance will be worse because the dynamic range of the entire weight matrix could be larger than per-row dynamic range. The outliers of the weight matrix will lead to a larger range such that the quantized distribution will be skewed. Performing per-row quantization is less sensitive to outliers, so the performance (perplexity) is more stable.

4.1.2 Memory usage

From Table 2, we can see that quantizing the model from FP32 to Int8 saves lots of memory usage at inference time. The reason why it is not an exact 75% reduction is because we are only quantizing the weights in the model but not the entire model pipeline. The intermediate computations are still carried out in BF16.

To verify that the quantized model is indeed in 8-bit, we also looked at the model size, which is the memory usage before inference (and after initializing the model). We observe that the model size of the quantized model is about 29% (0.412GB vs 1.42GB) of the size of the FP32 model.

4.1.3 Speculative Decoding Latency

From Table 3, we can see that speculative decoding produces a 1.46x speedup when the draft model is gpt2-small. However, the inference latency became worse when using quantized gpt2-medium as the draft model. This is because our quantized model doesn't have any actual speedup over the original model (because all the computation is still carried out in 16-bit). If the draft model is not faster than the main model, then speculative decoding wouldn't give any speedup.

However, we were able to confirm that if the draft models have been trained with explicit signal from the main model (in our case, a quantized version of the main model), then it tend to display increased agreement with the main model. When using gpt2-small as the draft model to generate 50 tokens, speculative model runs for 17 steps, which means every speculative decoding iteration, approximately 2.94 tokens were decoded. When using the quantized gpt2-medium as the draft model to generate 50 tokens, speculative model runs for 13 steps instead, which means every speculative decoding iteration, approximately 3.84 tokens were decoded. This is around 1 more token to decode per iteration.

4.2 Training and Evaluation Based on Pruning (Deliverable Part 2)

From the four plots, we observe that the quality of the wikitext task degrades as model size decreases. This is because pruning the model essentially means using a smaller network. Iterative pruning helps mitigate this issue by iteratively fintuning the model on the dataset to restore some performance.

From figure 3, we also observe that the throughput generally increases as model size decreases if we ignore the data point where we don't prune the model at all. This is expected when we delete the rows for both model inputs and weight matrices so the speed for matrix multiplication gets faster. When we don't prune the model, we suspect that using the full active indices to slice the matrices is well-optimized and hence the throughput is higher than the other cases.

To compare the effect of pruning dimensions versus individual weights, we examined the model quality at the same model size. When the model size is 10% of its original size, individual pruning achieves validation loss of 7 while dimension pruning achieves validation loss of 6.5. This suggests that dimension pruning is a better method. However, we noticed that individual pruning has performance degrade at around 50% of the original model size while the performance degrade for dimension started at 90% of the original model size. This is because dimension pruning prunes out entire rows, which hurts performance earlier in the pruning process.

References

[1] Andrej Karpathy. nanogpt: The simplest, fastest repository for training/finetuning medium-sized gpts. `https://github.com/karpathy/nanoGPT`, 2023.

[2] Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 8-bit optimizers via block-wise quantization, 2022.

[3] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks, 2015.

[4] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding, 2023.

[5] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling, 2023.