CS 229S Final Project Report

Sanjit Neelam
ICME
Stanford University
sneelam@stanford.edu

Anuj Shetty
ICME
Stanford University
anuj42@stanford.edu

John Winnicki
ICME
Stanford University
winnicki@stanford.edu

1 THE LEADERBOARD CATEGORY WE WANT IN OUR FINAL SUBMISSION IS: MEMORY USAGE

THE LEADERBOARD CATEGORY WE WANT IN OUR FINAL SUBMISSION IS: MEMORY USAGE

2 Introduction

We attempt to optimize the training and inference of Andrej Karpathy's nanoGPT by implementing quantization, pruning, and parallelism. We evaluate on `wikitext-103-v1` (`https://huggingface.co/datasets/wikitext/viewer/wikitext-103-v1`).

3 Quantization

3.1 Measuring perplexity

We calculate perplexity on 60967 tokens of `wikitext-103-v1` as follows. We split up the sequence of tokens into chunks of length 1024, stack `batch_size` chunks of length 1024 along the first (batch) dimension, then forward this batch through the model. The next-token-prediction for around half the tokens in the batch is made without even half the max context of 1024, but this seems to be the way perplexity is calculated in Radford et al. [2019]. We infer this from the example at `https://huggingface.co/docs/transformers/perplexity`.

This method to measure perplexity is much quicker to compute, especially for large models and large validation sets. Moreover, with this method we get a measure of perplexity that is more correlated with downstream applications than calculating perplexity by ensuring each prediction was made with the max available context. In practice, many downstream tasks are performed without even getting close to using a model's max context.

3.2 Post-training simulated linear quantization

The three main variants of quantization are post-training dynamic quantization (PTD), post-training static quantization (PTS), and quantization-aware training (QAT). PTD consists of quantizing the model weights, then quantizing the activations during runtime by e.g. calculating the maximum and minimum values in the activations during runtime to determine the scale and zero-point. PTS learns scales and zero-points needed to quantize activations by forwarding representative batches of tokens through the model in a calibration phase, and does not update these parameters during runtime. QAT simulates the noise injected by quantization at inference time during the training process by quantizing and dequantizing tensors at various points in the forward pass.

PyTorch does not natively support INT8 matrix multiplications, so we only implement a *simulated* variant of post-training dynamic quantization where all matrix multiplications are between `bfloat16` tensors. Specifically, we first quantize all model parameters, then dequantize each parameter as and when it's needed to perform a matrix multiplication with activations from the previous layer. This increases latency to be even higher than the un-quantized model, but results in lower peak GPU memory allocated since at any point in time, only a subset of the parameters are `bfloat16` and the rest are INT8. On the other hand, quantizing the activations at the end of a layer and dequantizing them at the beginning of the next layer will not reduce peak memory usage but it will significantly reduce perplexity as in Figure 1 of Park et al. [2022]. We do not quantize activations in our implementation but note that it would be necessary to accurately simulate PTD.

Linear quantization (dequantization) is an affine mapping of $x \in [\alpha, \beta] \subset \mathbb{R}$ to (from) $x_q \in [\alpha_q, \beta_q] \subset \mathbb{Z}$. It is the most simple form of quantization since elements of the set of possible quantized values are equally spaced. A more complex method of quantizing may be to implement a variant of Huffman Coding (Han et al. [2015]), which can represent more common values with a higher resolution. Linear quantization can be symmetric if we enforce $\alpha_q = \beta_q$ and affine otherwise. More details about quantization can be found at `https://leimao.github.io/article/Neural-Networks-Quantization/`.

Symmetric quantizing LayerNorm weights results in an increase of around 4 perplexity after all other parameters have been quantized. On the other hand, affine quantizing LayerNorm results in an increase of around 0.6 perplexity after all other parameters have been quantized. Based on this observation, we use symmetric quantization for all weights (except LayerNorm weight) and affine quantization for all biases. We also noticed that letting $(\alpha_q, \beta_q) = (-127, 127)$ instead of $(-128, 127)$ when quantizing all parameters increases perplexity by around 2.2.

3.3 Speculative decoding

Speculative decoding (Chen et al. [2023]) is a technique to speed up autoregressive decoding with a target GPT M by decoding with a smaller, faster draft GPT D. We tried speculative decoding with M = gpt2-medium and D = gpt2, which reduced latency from 0.0550 seconds per token to 0.0427 seconds per token (details in Appendix). We also tried speculative decoding with M = gpt2 and D = gpt2 quantized, which increased latency to 0.0968 seconds per token. This is expected since our quantized implementation is slowed down by dequantizing and quantizing weights, but speculative decoding also runs for 26 iterations rather than the previous 21 due to the degradation in quality from quantizing. These results may change with a different random seed.

4 Pruning

In the pruning stage of the project, we embarked on a nuanced exploration of memory-efficient transformer architecture neural networks. Here, we implemented iterative pruning, a technique that enhances model performance by strategically eliminating less significant weights based on norm calculations. Importantly, we completed this task without the use of external libraries, such as the PyTorch implementation of pruning (which would not have achieved the intended outcome of the model anyway). This allowed us to challenge ourselves to fully understand the intricacies in actually optimizing and neural network in the real world. Our iterative pruning involves a three-stage approach, which we will discuss in more detail in the upcoming sections: first, we trained the model on the WikiText103 dataset for 100 iterations to refine refine the pre-trained model's comprehension capabilities. Next, we pruned the lowest magnitude weights using two methods: a granular approach targeting specific weights and a broader strategy removing entire rows of the neural network weight matrix. The first approach was achieved via masking, though with significant changes to the training strategy. The second approach required significant modifications in the training strategy and model architecture. The final stage entailed retraining the pruned model to recover and enhance its capabilities, a crucial step to offset the impact of pruning and improve the model's efficiency and accuracy.

4.1 Background

Our iterative pruning strategy was a three stage process: first we finetuned our model on the WikiText103 dataset for 100 iterations. Fine-tuning involves refining a pre-trained model on a specific dataset to adapt it to particular tasks, a practice that has become mainstream in the field and enables practitioners to achieve high levels of performance without re-inventing the wheel. This approach, as described by Howard and Ruder (2018) in their seminal paper "Universal Language Model Fine-tuning for Text Classification," leverages the knowledge a model has gained from a large, diverse dataset and applies it to a more focused domain. Many papers that we looked at in this topic was finetuning in the context of large language models like GPT and BERT. Devlin et al. (2018) in their groundbreaking work on BERT ("BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding") demonstrated how fine-tuning these models on task-specific datasets can significantly improve performance across a wide range of tasks. The basic idea of finetuning is outlined by Radford et al. (2018) in their study on GPT ("Improving Language Understanding by Generative Pre-Training"), where one makes slight adjustments to a model's weights based on a more directed dataset, ensuring that the model retains its general linguistic understanding while becoming more adept at handling the nuances of the target task.

Subsequently, we identify and prune the weights with the lowest magnitude. Pruning in neural networks is a technique used to reduce model complexity and improve efficiency, primarily by removing weights or neurons deemed less important. The main paper we read was on the pioneering work by Han et al. (2015), titled "Learning both Weights and Connections for Efficient Neural Networks". Here, they demonstrated that pruning redundant connections in neural networks can significantly reduce their size without compromising accuracy. This concept was further expanded by Frankle and Carbin (2018) in "The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks," where they introduced the idea that a small, pruned network (a 'winning ticket') can perform comparable to the original pre-pruned network. Molchanov's 2016 paper ("Pruning Convolutional Neural Networks for Resource Efficient Inference" showed that the technique not only reduces the computational burden but can also lead to models that are more generalizable.

In this pruning stage, we applied two different approaches: a highly granular approach operating in an "unstructured" fashion, and a less granular approach operating in a "structured" fashion, with visual description seen in pruning figure 1. In the unstructured pruning, each layer is represented in pytorch by a weight matrix. We iterate line by line in each row, setting a specified percentage of the weights in each row with lowest absolute value are set to zero. In order to implement this technique, at each pruning stage, the original weights are saved as parameters, along with the mask (matrix with same dimension as weights with corresponding 0/1 values), with the pruning applied as a hook. During the forward pass, the hook is triggered and the pruned network is used. However, during backprop, the original weight matrix is used. The rationale for employing the original, unpruned network during the backward pass hinges on the optimization process, which predominantly focuses on the unpruned weights. This is due to the fact that the pruned weights have a gradient value of zero and therefore will not be the focus the training process. Utilizing the original network for gradient calculation simplifies implementation and ensures the preservation of the network's structural integrity. Such consistency is pivotal in the training regimen, fostering a more gradual and controlled learning curve. This approach maintains a balanced and stable dynamic within the network, facilitating its adaptation to incremental changes and complexities inherent in the training process.

The other approach was a less granular approach, known as structured pruning, whereby a specified percentage of of rows are removed altogether from the neural network weight matrix, as seen in pruning figure 1. This strategy is significantly more complex, requiring modification of the model architecture during the training process. In this stage of the project, both the row of the weight matrix, as well as the corresponding column of the output were pruned, as seen in figure 2. Here, we chose to prune the QKV matrix, splitting the QKV weight matrix into the Q, K, and V matrices. Then, we applied the pruning step. Finally, the projection weights were pruned with the V matrix, since the transformer architecture applies the projection layer by applying the V matrix. One important implementation detail was that the number of rows was always made divisible by the number of heads, in order to facilitate multiheaded attention (since QKV has to be split across the heads).

Pruning specifically the QKV matrices in a transformer architecture, rather than other parts, was a strategic choice aimed at optimizing the most computationally intensive component of the model. The QKV matrices are integral to the attention mechanism; by focusing pruning on these matrices, we most directly reduced the computational burden and memory usage of the architecture, leading to a more efficient model with potentially less impact on overall performance. This approach also addresses the well-known redundancy often found in attention mechanisms, where not all elements of the QKV matrices are equally critical for performance. Pruning here can maintain or even improve the model's effectiveness by retaining only the most informative aspects of the attention process.

Additionally, other parts of the transformer, like feed-forward layers, may not have the same level of redundancy and parts like the embedding layer are more sensitive to pruning, making them less ideal candidates for extensive pruning. Thus, targeting the QKV matrices can be argued to be the most effective strategy for balancing model efficiency with performance in transformer architectures.

The final step is to retrain the model, where the pruned model undergoes fine-tuning to recover and potentially enhance its learning capabilities. This step is pivotal to ensure that the model not only retains but possibly improves its efficiency and accuracy, thereby compensating for the initial impacts of weight pruning.

4.2 When and How Much To Prune

We tested **extensively** how frequently and by how much to prune. In particular, we went through all of the combinations of 10, 50, 100, 200 iterations between each prune, in combination with pruning by 5, 7, 10, 15 percent of model pruned per iteration. We found after this testing that 10 percent of models pruned every 100 iterations performed the best.

4.3 Extra Credit

We also completed the extra credit portion of the project: In the initial training phase, the model undergoes 50 iterations of finetuning using Wikitext, a popular dataset for language modeling. The next phase involves a selective pruning process. Here, neurons are systematically removed from the model, with the constraint that the resulting model's loss does not exceed a threshold of 3.2. We applied unstructured pruning at 10 percent reduction pet pruning stage. Pruning based on threshold is crucial for balancing model efficiency with performance: by pruning only when necessary, we are able to more accurately prune away unnecessary neurons when the model is satisfactorily performing while also not pushing forward with pruning too aggressively when the model hasn't yet adapted to the changes. The model undergoes an iterative training phase thereafter, which includes an additional 25 iterations of finetuning and repeating the above process. This cycle is repeated until the model is reduced to 10 percent of its original size. Our methodology here emphasizes a balanced approach to model optimization, prioritizing both efficiency and effectiveness.

4.4 Pruning Experimental Setup

Initially, the model was trained on the WikiText103 dataset for 100 iterations, using a batch size of 8, a block size of 1024, and with gradient accumulation set to 40. Starting at this point, we would apply the pruning strategies to reduce the size by 10 percent and retrain for 50 iterations. We did this for 9 rounds until the model was at 10 percent of its original size. We experimented with many different forms, and ultimately, due to the highly expensive nature of the training and limited Google Cloud Compute credits, we found that this setting resulted in the best performance. The experiments were conducted using a machine configured with an n1-standard-4 type, powered by an Intel Skylake CPU and equipped with 1 x NVIDIA T4 GPU, and a 50gb disk. We used 6 layer, 6-headed MHA architecture, with embedding size of 384.

4.5 Pruning: Experimental Results and Discussion

In figures 4 through 10, we present the our experimental results for quality (loss), inference latency, and training time, for both unstructured and structured pruning.

4.6 Important Note about Results

First, it should be noted that although we are certain about the implementation, having thoroughly worked through the code and tested it, we were not able to run it on as large of a network as we wanted, due to memory restraints. In particular, for structured pruning, since we are modifying the dimensions of the weight matrix directly, the memory requirements are substantially higher compared to unstructured pruning. We completely exhausted our provided Google Cloud credits, and even dipped into personal funds by spinning up an AWS instance with T4 GPU to complete the final portion of the training (see pruning figure 3 for proof). Therefore, although we exactly followed the instructions, we may get results that would be different than if we trained on a more expressive model.

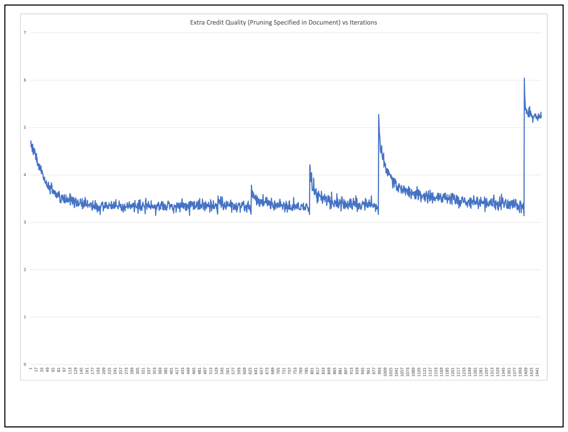


Figure 1: Pruning Figure 10 – Extra Credit Adaptive Pruning Strategy Quality

4.7 Extra Credit Pruning Technique

We begin with the extra credit pruning technique. In pruning figure 10, we see the results of our extra credit pruning model, whereby, we take a more balanced and adaptive approach to training, and prune only after reaching a certain level of loss, and then waiting until we re-attain this level. In the figure, we see that every time the model dips below the 2.3 level, there are increasingly higher spikes in loss, due to pruning happening at that stage. Clearly, our strategy works as intended. In addition, it's immediately clear that initially, pruning does not seem to affect the quality of the model. However, as we increase the percent of the model pruned, the model takes a longer time to adapt and the validation loss initial spike in loss grows significantly larger. Intuitively, this makes sense: we are pruning away increasingly important connections, due to the magnitude-based heuristic in our pruning strategy, and it will take longer for our model to recover from removing critical connections.

4.8 Model Size vs Quality

In pruning figures 4 and 5, we compare model size to loss. In our figure, we provide the loss at each time step, where every 10 units in the figure, we prune the model by 10 percent. As we can see in both figures, the model is still converging when we begin converging. Again, unfortunately due the constraints of the problem and more notably, the compute constraints due to running out of money and not being able to secure a machine with enough memory to properly perform structured pruning, we were forced to perform our experiments in such settings. However, our results still match the results provided by "Structured Pruning for Efficient Generative Pre-trained Language Models", where we find that pruning our model does not adversely affect the quality up to 90 percent pruned! Interestingly, we find that both the unstructured and structured pruning perform similarly, attaining levels around loss of 5. However, we note that unstructured appears to have less volatility in loss levels, as well as converging to this value faster. This makes sense, since the approach is more granular: unstructured pruning individually selects weights in each layer to remove. We would expect better performance, since this approach is more individualized, and in our small model, we would not expect increased generalization to help as much.

Another explanation to this difference might be because unstructured pruning maintains the original network topology. By deactivating individual weights using masks, unstructured pruning preserves the intricate connections and patterns learned during initial training. This method ensures that the critical pathways and complex interactions within the network remain largely intact, allowing the network to retain more of its learned capabilities and nuances. Furthermore, since unstructured pruning does not fundamentally alter the network's architecture, it avoids the potential inefficiencies and irregularities in computation and memory access that can arise from the more disruptive changes in structured pruning. As a result, unstructured pruning enables more gradual and controlled fine-tuning of the network, leading to a more stable and reliable performance even as the pruning level increases. This stability and reliability manifest in convergence to high quality and lower volatility.

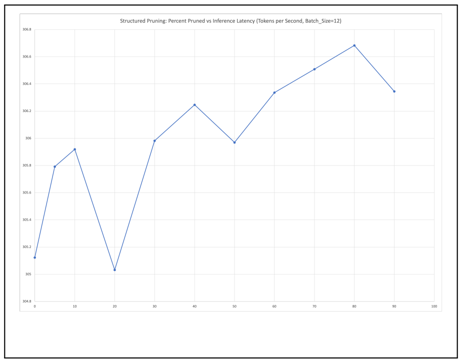


Figure 2: Pruning Figure 9 – Structured Pruning Speed of Execution, Batch Size 12

4.9 Model Size vs Inference Latency

In pruning figures 6 and 7, we observe structured and unstructured quality vs inference latency, measured in tokens per second for batch size 1. Firstly, we can see that structured pruning trends upwards. This matches our intuition, since we would expect that as the network size decreases, the inference should be faster. Secondly, we can see that unstructured pruning stays in roughly at the same level of inference speed. Intuitively, one would expect this to be the case, since unstructured pruning is achieved by a mask, and so the actual network is not much smaller, apart from sparsity. We do see an initial spike in inference latency, and so we conjecture that the gains from sparsity are mostly realized in the first stage of pruning. Afterwards, perhaps the overhead of maintaining the sparse matrix operations does not significantly improve performance. Finally, in comparing both of the figures, we notice that structured pruning only achieves the same inference latency as unstructured pruning at higher levels of inference. This initial performance gap might be attributed to the inefficiencies in GPU utilization and irregular memory access patterns caused by the architectural changes in structured pruning. However, as the level of pruning increased, the performance of structured pruning improved, eventually matching the consistent performance level of unstructured pruning. Unstructured pruning, which deactivates weights using masks but retains the original network structure, maintained a steady performance throughout, leveraging efficient GPU utilization and memory access patterns. The convergence of performance between the two methods at higher pruning levels highlights how structured pruning can adapt and overcome its initial inefficiencies. In pruning figures 8 and 9, we see the exact same trends for batch size 12, although the attributes mentioned above are even more prominently shown. We believe this is due to the fact that the differences manifest themselves more explicitly due to the increased requirement for parallelization, and thus, the differences in sparsity and architecture integrity become more important.

5 Pipeline parallelism

5.1 Background

Pipeline parallelism is a distributed computing technique used to address the computational and memory challenges in training large neural network models, and is the advanced form of earlier methods like data and model parallelism. Data parallelism involves splitting the training data across multiple processors that each hold a complete copy of the model, thereby allowing simultaneous processing of different data subsets, which is required in the age of big data. However, as models grew in size, this approach started hitting limitations, particularly in terms of memory requirements.

Model parallelism was introduced as a solution to this. It involves partitioning the model itself across different processors, with each processor responsible for a subset of the model's layers or parameters. While this method addresses the memory constraint by distributing the model's parameters, it introduces challenges in terms of synchronization and communication overhead, especially for models with intricate and dense layers.

Pipeline parallelism takes these concepts further by combining aspects of both data and model parallelism. It splits the model into segments, with each segment on a different GPU, and then processes different mini-batches of data simultaneously in different stages of the pipeline. Each segment of the model processes a different mini-batch of data at any given time and this minimizes the idle time of each GPU observed in model parallelism.

5.2 Experimental setup

Despite the $50 Google Cloud credits, we were not able to get any instances with 2 GPUs after repeated attempts (even spot instances). We managed to get a spot instance with 2 T4 GPUs (each with 8GB RAM) on AWS, using our personal funds.

There exist a few packages for parallelism like Pipe on PyTorch tor, however we have created our own implementation. The NanoGPT architecture (`gpt-medium`) roughly consists of the embedding layer, 24 layers of self-attention + MLP, and the `lm_head`. We split these layers into 2 partitions, the first one containing the embedding and half of the self-attention + MLP layers, and the second one containing the rest of the self-attention + MLP layers and the `lm_head` layer. We then modify the forward pass to run the first partition on GPU 0. then send the intermediate activation tensor from GPU 0 to GPU 1 using `torch.distributed.broadcast`, and then complete the rest of the forward pass on GPU 1. We then calculate the loss and similarly propagate the gradients backward across both GPUs. We also split each micro-batch of the NanoGPT implementation into smaller batches of size 2 to fit in our GPU RAM.

We use `torch.distributed.launch` to launch the script with a different process for each node, and appropriately manage the control flow for both nodes using the same script.

We also implement model parallelism as a comparison, following a similar procedure above, except running a single process where we move tensors from one GPU to the other simply by using `.to('cuda:1')`.

We've used the `gpt2-medium` model, on the `Wikitext` dataset, with a block size of 1024 and batch size of 4 for these runs, with other parameters remaining the same as the default.

5.3 Results

We use `torch.profiler.profile` to record the event trace of the 5 training iterations. The result of model parallelism is shown in Figure 3, where we can see that tensors are passed from one GPU to the other, with no concurrent computations happening on the GPUs, resulting in a lot of idle time, effectively just doubling our GPU memory in some sense rather than speeding up the training.

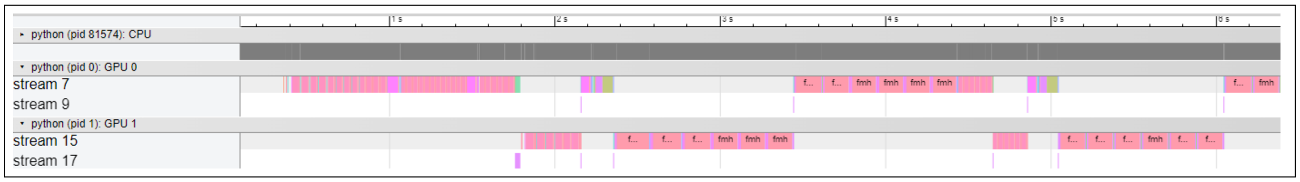


Figure 3: Model Parallelism - No concurrency

The result of pipeline parallelism are shown in Figure 4, where we can see that after intermediate activations are passed from GPU 0 to GPU 1, GPU 0 then fetches the next batch and concurrently starts the forward pass on it, thus minimizing the 'bubble' of idle time. We also modified where we create the partition in the model to create different levels of concurrency.

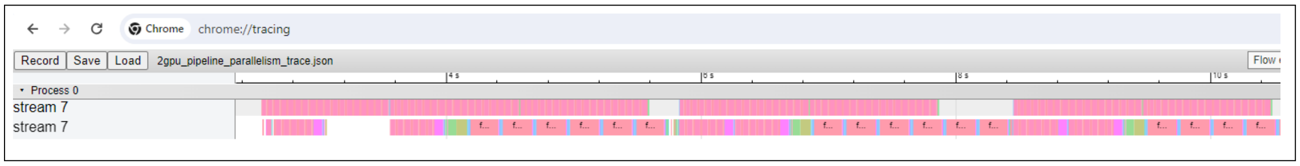


Figure 4: Pipeline Parallelism - Concurrency

We then ran this with different batch sizes, recording the training throughput from iterations 20 to 40 specifically, with results shown in table 1.

Batch Size	Peak Memory Usage		Training Throughput (tokens/second)
	GPU1	GPU2	
4	4.04GiB	7.16GiB	221.3719
12	4.04GiB	7.16GiB	418.4762

Table 1: Training throughput and memory usage for different batch sizes

References

Pipeline parallelism in pytorch. URL `https://pytorch.org/docs/stable/pipeline.html`.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.

Song Han, Huizi Mao, and William J Dally. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*, 2015.

Minseop Park, Jaeseong You, Markus Nagel, and Simyung Chang. Quadapter: Adapter for gpt-2 quantization. *arXiv preprint arXiv:2211.16912*, 2022.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

6 Appendix

7 Leaderboard

```
{
    "loss": 3.0250,
    "inference_latency_1": 178.9039,
    "inference_latency_12": 351.4810,
    "training_throughput_4": 221.3719,
    "training_throughput_12": 418.4762,
    "memory_usage": 0.7257
}
```

7.1 Quantization

The following is the output of running `_eval_quantized.py`.

```
--------------------------------------------------------------------------------
Inference: without quantization
--------------------------------------------------------------------------------
loading weights from pretrained gpt: gpt2
forcing vocab_size=50257, block_size=1024, bias=True
overriding dropout rate to 0.0
number of parameters: 123.65M

GPU memory allocated after calling model.to(device) 0.4643 GB
GPT-2 perplexity on wikitext/val.bin, batch_size=4: 56.6525
Peak GPU memory allocated: 3.0414 GB

GPT-2 perplexity on wikitext/val.bin, batch_size=12: 56.6610
Peak GPU memory allocated: 7.6879 GB


--------------------------------------------------------------------------------
Inference: with quantization
```

```
--------------------------------------------------------------------------------
loading weights from pretrained gpt: gpt2
forcing vocab_size=50257, block_size=1024, bias=True
overriding dropout rate to 0.0
number of parameters: 123.65M

GPU memory allocated after calling model.to(device) 0.1248 GB
GPT-2 quantized perplexity on wikitext/val.bin, batch_size=4: 63.7557
Peak GPU memory allocated: 2.4533 GB

GPT-2 quantized perplexity on wikitext/val.bin, batch_size=12: 63.7438
Peak GPU memory allocated: 7.1007 GB


--------------------------------------------------------------------------------
Inference latency: decoding with M and speculative decoding with M, D
--------------------------------------------------------------------------------
Loading target model M (gpt2-medium)
loading weights from pretrained gpt: gpt2-medium
forcing vocab_size=50257, block_size=1024, bias=True
overriding dropout rate to 0.0
number of parameters: 353.77M

Loading draft model D (gpt2)
loading weights from pretrained gpt: gpt2
forcing vocab_size=50257, block_size=1024, bias=True
overriding dropout rate to 0.0
number of parameters: 123.65M

Standard decoding with M (gpt2-medium)
Inference latency, batch size 1: 0.0550 sec/tok (50 tokens, context size 1024)

Speculative decoding with M (gpt2-medium), D (gpt2)
Ran for 21 iterations generated 50 tokens with num_speculative=3
Inference latency, batch size 1: 0.0427 sec/tok (50 tokens, context size 1024)


--------------------------------------------------------------------------------
Inference latency: decoding with M and speculative decoding with M, M quantized
--------------------------------------------------------------------------------
Loading target model M (gpt2)
loading weights from pretrained gpt: gpt2
forcing vocab_size=50257, block_size=1024, bias=True
overriding dropout rate to 0.0
number of parameters: 123.65M

Loading draft model D (gpt2)
loading weights from pretrained gpt: gpt2
forcing vocab_size=50257, block_size=1024, bias=True
overriding dropout rate to 0.0
number of parameters: 123.65M
Quantizing draft model D (gpt2)

Standard decoding with M (gpt2)
Inference latency, batch size 1: 0.0203 sec/tok (50 tokens, context size 1024)

Speculative decoding with M (gpt2), D (gpt2 quantized)
Ran for 26 iterations generated 50 tokens with num_speculative=3
Inference latency, batch size 1: 0.0968 sec/tok (50 tokens, context size 1024)
```

```
--------------------------------------------------------------------------------
Leaderboard: memory usage for inference without quantization
--------------------------------------------------------------------------------
loading weights from pretrained gpt: gpt2
forcing vocab_size=50257, block_size=1024, bias=True
overriding dropout rate to 0.0
number of parameters: 123.65M

GPU memory allocated after calling model.to(device) 0.5900 GB
GPT-2 perplexity on wikitext/val.bin, batch_size=1: 56.6595
Peak GPU memory allocated: 1.3174 GB


--------------------------------------------------------------------------------
Leaderboard: memory usage for inference with quantization
--------------------------------------------------------------------------------
loading weights from pretrained gpt: gpt2
forcing vocab_size=50257, block_size=1024, bias=True
overriding dropout rate to 0.0
number of parameters: 123.65M

GPU memory allocated after calling model.to(device) 0.1248 GB
GPT-2 quantized perplexity on wikitext/val.bin, batch_size=1: 63.7593
Peak GPU memory allocated: 0.7257 GB
```

7.2 Pruning

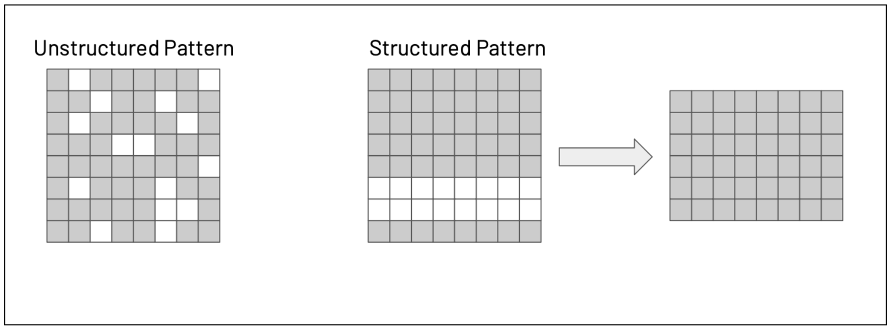


Figure 5: Pruning Figure 1 – Unstructured vs Structured Pruning Diagram

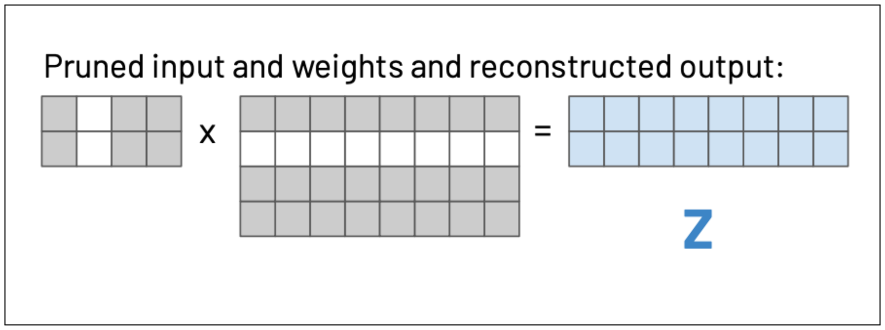
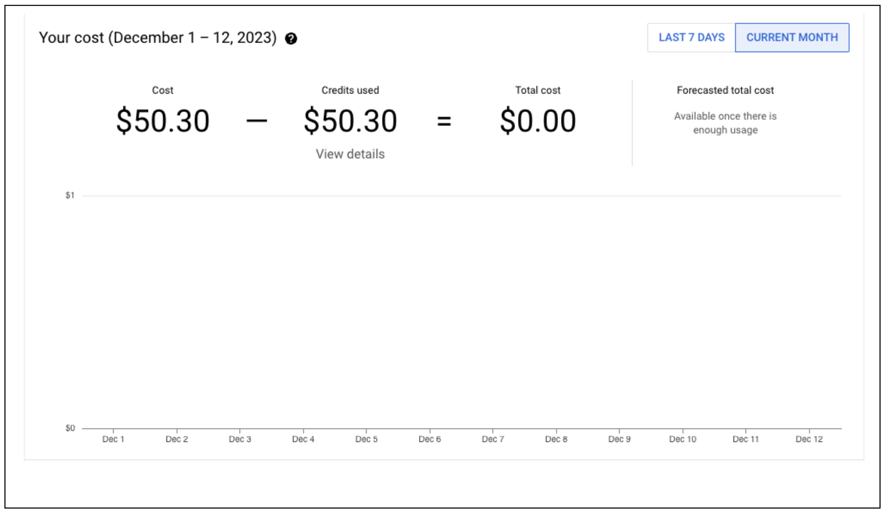


Figure 6: Pruning Figure 2 – Structured Pruning Input and Output


Your cost (December 1 – 12, 2023)

LAST 7 DAYS | CURRENT MONTH

Cost $50.30 — Credits used $50.30 = Total cost $0.00

View details

Forecasted total cost

Available once there is enough usage

Figure 7: Pruning Figure 3 – Proof of Google Cloud Limitations

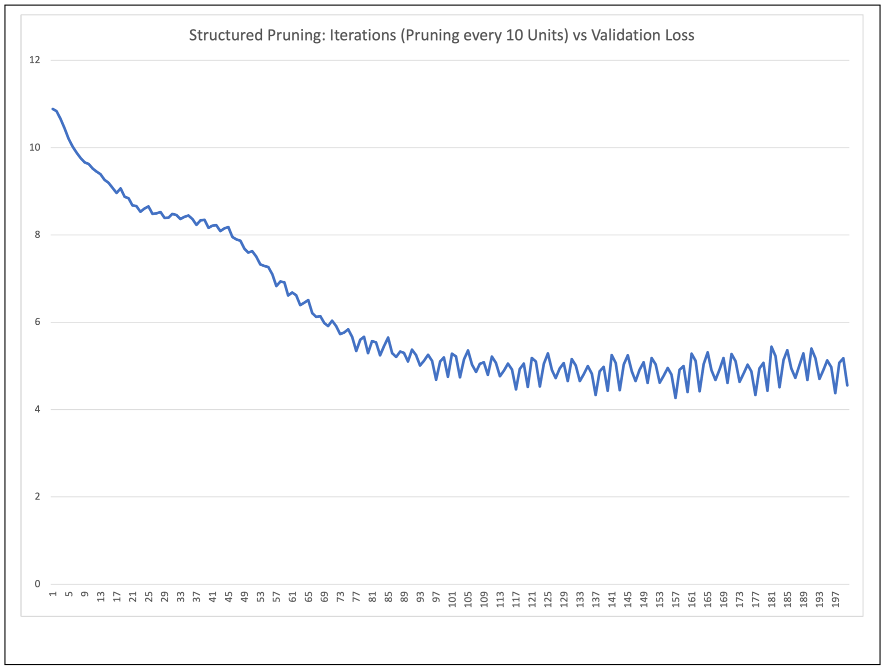


Figure 8: Pruning Figure 4 – Unstructured Pruning Quality

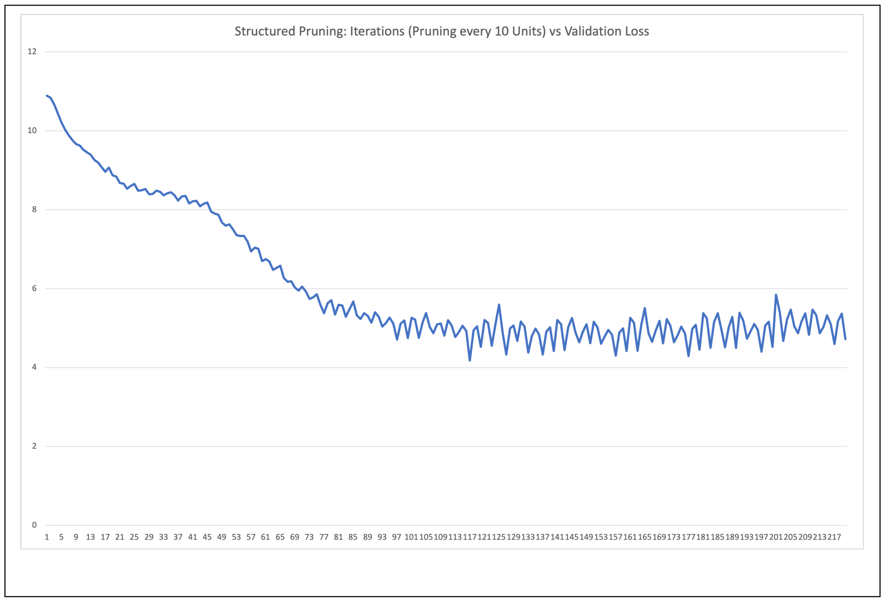


Figure 9: Pruning Figure 5 – Structured Pruning Quality

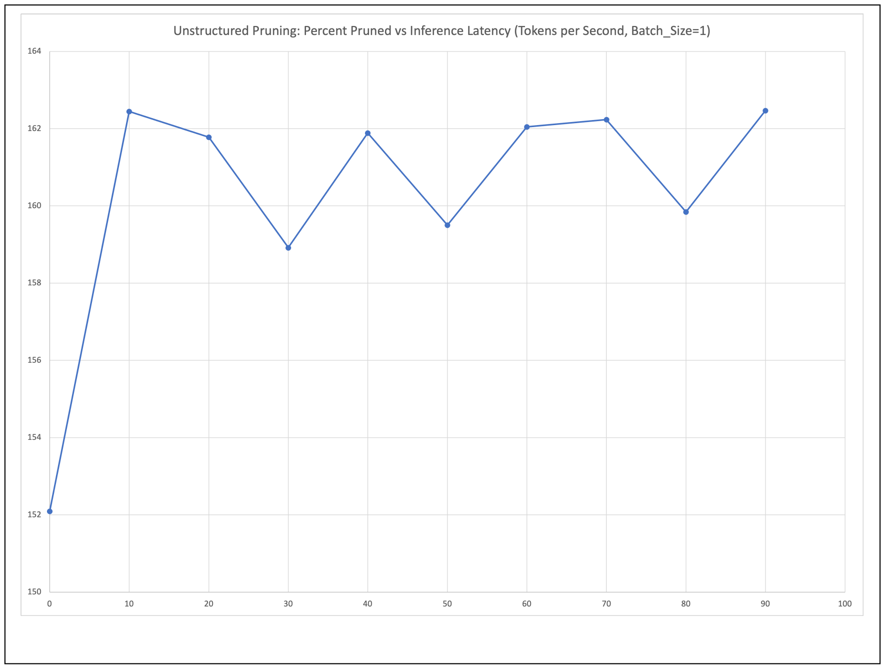


Figure 10: Pruning Figure 6 – Unstructured Pruning Speed of Execution, Batch Size 1

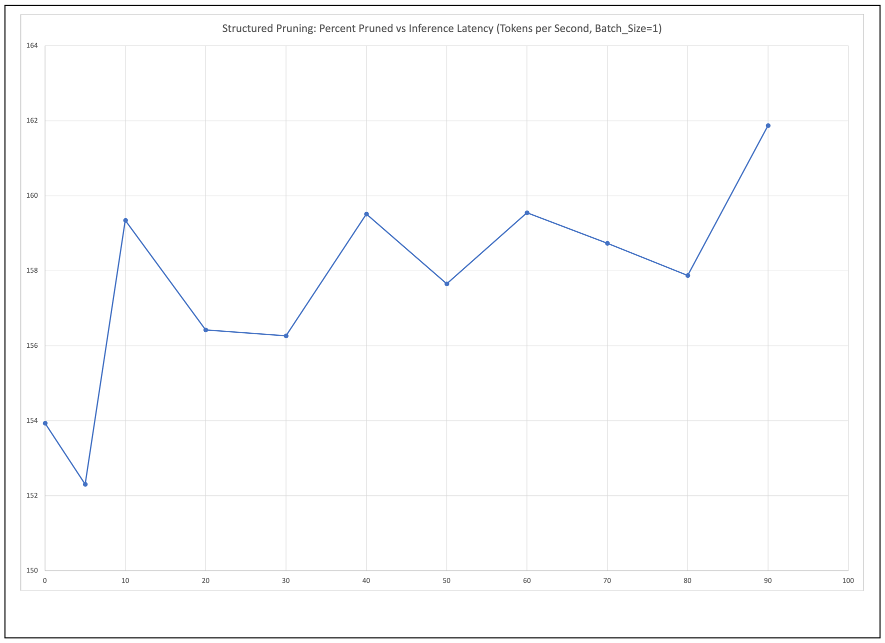


Figure 11: Pruning Figure 7 – Structured Pruning Speed of Execution, Batch Size 1

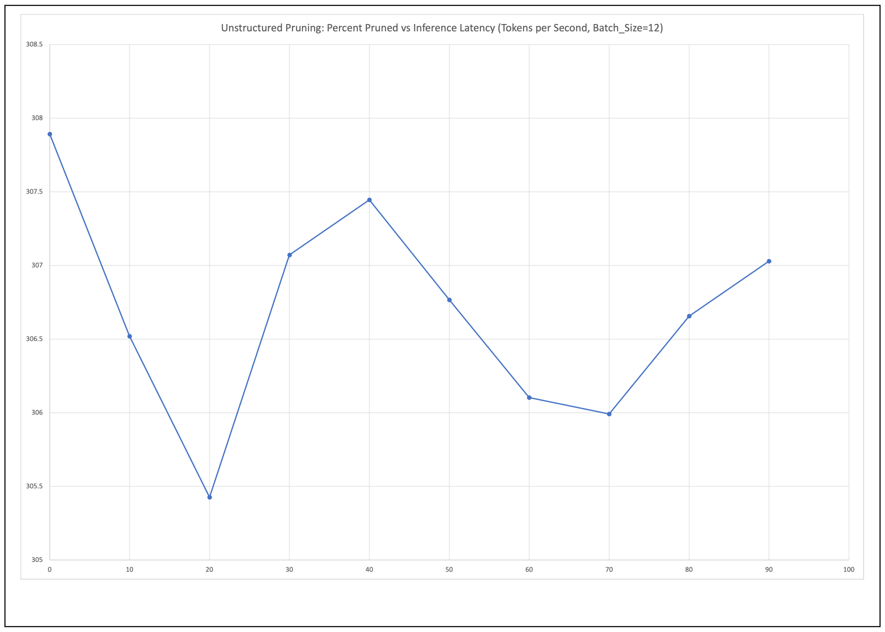


Figure 12: Pruning Figure 8 – Unstructured Pruning Speed of Execution, Batch Size 12

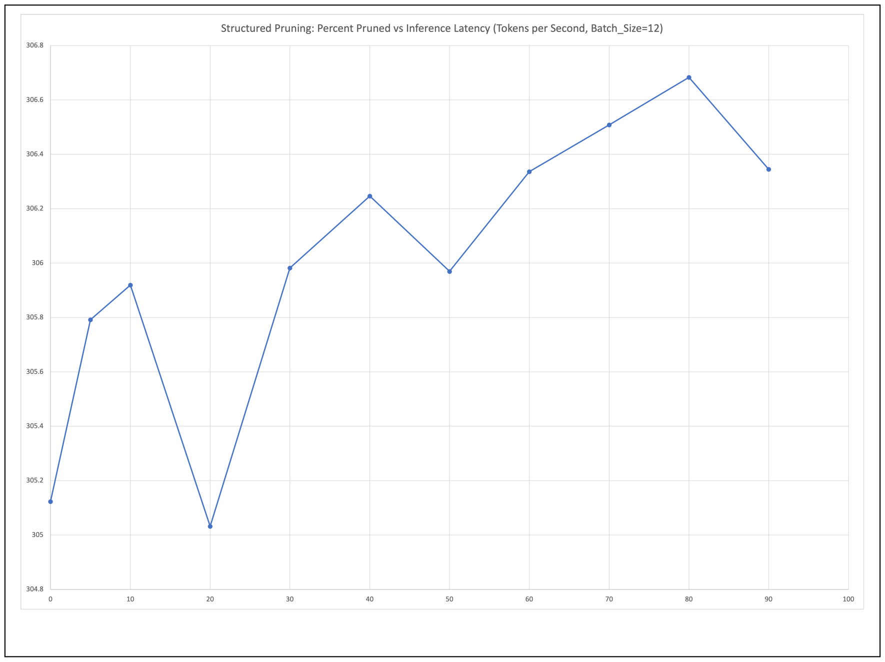


Figure 13: Pruning Figure 9 – Structured Pruning Speed of Execution, Batch Size 12

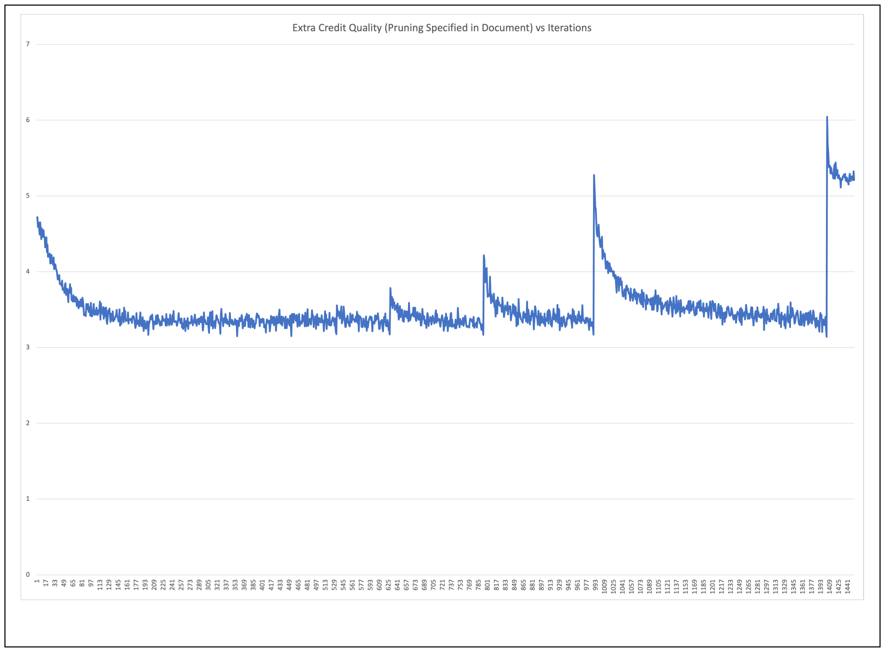


Figure 14: Pruning Figure 10 – Extra Credit Adaptive Pruning Strategy Quality