CS229S Project Report

MICHELLE FU, GRIFFIN BRYAN HOLT, RAE WONG, Stanford University, United States

This paper explores various methods for efficient machine learning systems and concludes with insights gained from implementing them.

1 QUANTIZATION

1.1 Introduction

1.1.1 Memory constraint. A dominating problem is the memory constraint of devices. With the increased number of parameters, the memory consumption of weight matrices would limit the size of ML models that we can run. A common solution is to employ model parallelism [6] where we can use multiple devices to train a model, each storing a subset of weights. However, there are instances where we would have to fit an entire model into one device, such as to perform inference on personal mobile devices.

1.1.2 Quantization. Quantization is a method to compress weights to a low precision data type, hence consuming less memory. For instance, an 8-bit quantization maps the range of the original (32/16fp) values to a range of 8-bit int type.

There are various points in the ML pipeline where we can perform quantization but we will be focusing on post-training quantization, which is sufficient to enable inference given a memory constraint. [3] Post-training quantization performs quantization on the weights frozen after training and use the quantized weights for inference on a smaller device. There are training aware quantization that may be used during the training phase which we will not be exploring in this paper.

1.2 Background

1.2.1 Zero-point quantization. Zero-point quantization considers the range of a particular tensor, which achieves better precision using the same number of quantization bits.
Zero-point b-bit quantization scales a tensor T by

$$scale = (T_{max} - T_{min})/(2^b - 1)$$

computes zero-point

$$zeroPt = -round(T_{min} * scale) - 2^{b-1}$$

The quantization and dequantization functions are as follows.

$$quantize(T) = clamp(round(T/scale) + zeroPt)$$
$$dequantize(T_q) = (T_q - zeroPt) * scale$$

1.2.2 Dynamic quantization of activations. In order to perform 8-bit inference, we also have to quantize activations other than the weights. A straight forward way is to keep activations in full precision and only perform dynamic quantization of activations into 8-bit integers during computation with the weights.

Author's address: Michelle Fu, Griffin Bryan Holt, Rae Wong, Stanford University, 450 Jane Stanford Way, Stanford, United States.

1.2.3 Simulated quantization. To support actual quantized inference, computation kernels have to be adapted for 8-bit inputs and interact with hardware instructions that performs arithmetic on 8-bit integers. To avoid dealing with hardware-specific intrinsics, we instead implement simulated quantization. Weights are quantized post-training and loaded in 8-bit to reduce memory usage, but are dequantized on-the-fly to perform floating point computation. This also avoids the need to quantize activations. However, there is a latency cost to this dequantization step which shows in our experiments.

1.3 Experimental Setup and Results

Quantization experiments are done using GPT2-small on a single T4 instance. Inference runs are conducted with a sequence length of 1024 and speculative inference latency are recorded for generating 50 tokens with batch size 1.

Model	Batch size	Train Perplexity	Test Perplexity
GPT2-Quantized	4	58.021	58.212
GPT2-Quantized	12	58.015	58.038
GPT2	4	**56.334**	**56.475**
GPT2	12	56.560	56.520

Table 1. Perplexities on WikiText for GPT2-Quantized and GPT2

Model	Batch size	Memory after model load	Memory after forward pass
GPT2-Quantized	4	**838740992**	**2935009792**
GPT2-Quantized	12	**838740992**	7102312960
GPT2	4	1215850496	3564008960
GPT2	12	1215850496	7728379392

Table 2. Memory usage on WikiText for GPT2-Quantized and GPT2

Main Model	Draft Model	Inference Latency (tok/sec)	Speedup
GPT2-medium	- (naive)	58.069995	-
GPT2-medium	GPT2-small	39.149959	0.674
GPT2-large	- (naive)	28.010873	-
GPT2-large	GPT2-small	41.168315	1.47

Table 3. Speculative decoding

1.4 Experimental Analysis

1.4.1 Perplexity. Perplexity is calculated as follows:

$$exp\left\{-\frac{1}{t}\sum_{i=1}^{t}\log p_\theta(x_i|x_{<i})\right\}$$

Main Model	Draft Model	Inference Latency (tok/sec)	Speedup
GPT2-small	- (naive)	98.926075	-
GPT2-small	GPT2-small quantized	21.925317	0.222
GPT2-small	GPT2-small	45.350442	0.458
GPT2-small	GPT2-small quantized + kv cache	34.261246	0.346

Table 4. Speculative decoding with quantization

Across both batch sizes, we found that GPT2-quantized had similar, but slightly higher, perplexity on WikiText than GPT2. This is to be expected, due to the lower precision of the model weights.

1.4.2 Memory Usage. Memory was calculated using a `torch.cuda.max_memory_allocated()` call after `model.to(device)` and another call after evaluation. We found that memory usage for GPT2-Quantized after loading the model to device was 69.00% of the memory usage for vanilla GPT2, and the memory usage post-inference was 82.35% of GPT2 with batch size 4/91.90% of GPT2 with batch size 12.

While the memory usage for quantized model is lower, it wasn't as low as we were expecting given the 50% decrease in the size of the weights (torch.int8 vs torch.float16). We hypothesize a few reasons for this behavior.

First, when loading a quantized model from a `.pt` file, we must convert the weights to torch.int8. The amount of memory used before sending GPT2-Quantized to the device, calculated using a `torch.cuda.max_memory_allocated()` call, is 175188992 bytes. The amount of memory used to before sending the vanilla GPT2 to device is 0 bytes. Additionally, some amount of memory (although likely negligible) is used to store the scale and offset for zero-point quantization; these values are stored in torch.float32.

Secondly, we need to dequantize the weights to use them in inference, and the dequantized weights are stored in torch.float32. This likely contributes to the higher proportional memory usage for post-inference.

1.4.3 Speculative decoding. Speculative decoding is a strategy to speedup inference latency by having a smaller draft model predict the next x tokens for the larger main model to verify. [4] In a single speculative iteration, we can obtain up to $x + 1$ tokens depending on how accurate the prediction was with respect to the tokens that the main model would have computed in an auto-regressive manner. This off-loads the auto-regressive iterations to the draft model that is smaller and hence faster. The effectiveness of speculative decoding relies on two main properties: the reduction in size of model between the draft and main and the number of tokens the models agree upon.

A demonstration of speculative decoding (where $x = 3$) is recorded in Table 3. We observe that using GPT2-small as a draft model for GPT2-medium does not achieve any improvement in inference latency as the overhead of speculative decoding is higher than the efficiency gained. However, when using GPT2-small as a draft model for GPT2-large, we observe a latency speedup of 1.47×. This implies that the difference in the size of models is an important factor, which is reasonable as the size of the draft model contributes to the overhead of token prediction.

1.4.4 Speculative decoding with quantization. We investigate the possibility of using the quantized version of the main model as its draft model based on two theories: the quantized model is smaller and it displays good agreement with the main model since quantization can be seen as a form of training with explicit signal [1]. Experiments are done on GPT2-small models.

In Table 4, we observe the contrary where there is an increase in inference latency when using the quantized GPT2-small model to predict for a GPT2-small main model. The reasons for this can be attributed to our simulated quantization. Firstly, computation kernels during inference of the quantized model are still done in the original precision (fp16), which gives no run-time speedup as compared to the theoretically faster computations in 8-int. Secondly, there is additional overhead from dequantization on-the-fly in the simulated quantization. Hence, we are effectively using a slower draft model.

To investigate the upper-bound of using a same sized draft model without the dequantization overhead, we recorded the results of using a GPT2-small model as both the draft and main models. We observe that the upper-bound is at 0.458×, which implies that latency should approximately doubled. This is expected as each speculative iteration now performs $x + 1 = 4$ inference loops while obtaining around 2 tokens each time.

Next, we explored ways to overcome the dequantization overhead by implementing other optimizations. We first do a simple optimization of saving the quantized tensors before dequantization to reduce the need for re-quantization after computations. Next, we implemented KV-caching in our simulated quantization. KV-caching reduces the number of FLOPs during computation and hence reduces latency. As shown in Table 4, this increased the speedup from 0.222× to 0.346×, closer to the theoretical upper-bound of 0.458×.

2 PRUNING

2.1 Introduction

The large quantity of parameters within GPT2 brings about an obvious cost in FLOPs and throughput. This cost naturally raises the question: can we both benefit from the performance achieved by finetuning GPT2, but also decrease the number of FLOPs required to run the model? One method by which we may accomplish this goal is *pruning*: the systematic sparsification of a model's parameters after or in conjuction with fine-tuning [5].

2.2 Background

For this project, we implemented two methods of pruning: *single-element pruning* for sparsification of model parameters; and *row-wise pruning* for direct reduction of model dimensions. The two methods are described in detail in the subsequent subsections.

2.2.1 Single-Element Pruning. Each iteration k of our algorithm for single-element pruning is as follows:

(1) Finetune the GPT2 model for 100 iterations on the Wikitext dataset/task.
(2) For the parameters $W \in \mathbb{R}^{n \times m}$ each linear layer in the model:
(3) If $|W_{ij}| < r_k$, where r_k is the pruning-threshold for iteration k, prune the parameter W_{ij}.
(4) Repeat Step #1 the model reaches the desired size ($\approx$ 10% of the model's original size).

The pruning-thresholds r_k were chosen at runtime after each iteration so that approximately 10% of the model's original parameters were pruned at each iteration. After the model size was reduced to 10% of its original total, the model was finetuned once more for 100 iterations.

2.2.2 Row-wise Pruning. Each iteration k of our algorithm for single-element pruning is as follows:

(1) Finetune the GPT2 model for 100 iterations on the Wikitext dataset/task.
(2) For the parameters $W \in \mathbb{R}^{n \times m}$ each linear layer in the model:
(3) If $\frac{1}{m}\|W_i\|_2 < m_k$, where r_k is the pruning-threshold for iteration k, prune the row W_i.
(4) Repeat Step #1 the model reaches the desired size ($\approx$ 10% of the model's original size), but now training for 50 iterations (instead of the full 100).

Besides the first 100 iterations of finetuning prior to any pruning, we only pruned for 50 iterations in between pruning. this was because we chose the pruning-thresholds $r_k$at runtime after each iteration so that approximately 5% of the model's original parameters were pruned at each iteration (instead of 10% as in single-element pruning). This was because pruning entire rows was much more aggressive than pruning single elements, so we wanted to move slower through pruning but without compromising on training time. Thus, we decreased the number of iterations of finetuning in between pruning to 50. After the model size was reduced to 10% of its original total, the model was finetuned once more for 50 iterations.

We did not prune a row W_i by the magnitude of norm $\|W_i\|_2$, but instead by the norm divided by the size of the row (in number of elements) $\frac{1}{m}\|W_i\|_2$. This was done to prevent rows of smaller dimension from getting pruned first, as they will naturally have smaller norms that rows of larger dimensions in other layers. Arguably, the rows of smaller dimensions would need less pruning than rows of larger dimension, since they rely on fewer values in the first place.

2.3 Experimental Setup and Results

Both single-element pruning and row-wise pruning were accomplished via the introduction of our custom `PrunableLinear` PyTorch module (see our code submission). This class inherited from the `torch.nn.Linear`, accomplishing the same task as a standard linear layer, with the exception of elementwise-multiplying the linear layer's parameters by a "pruned" mask before multiplying against the input. Each mask M was the same size and shape as the linear layer's parameters W: the entry in the mask M_{ij} was equal to 0 if element W_{ij} of the weight matrix was pruned; otherwise, $M_{ij} = 1$. Every linear layer throughout the GPT2 model was replaced with one of our `PrunableLinear` layers, including the Language Modeling Head.

Our experiments were run on a single T4 Google Colab GPU with 15 GB RAM.[1] Due to the GPU RAM restrictions, and the fact that the pruning masks nearly doubled the size of the GPT2 model, we had to decrease the batch size from $B = 8$ to $B = 4$ in order for the model to run.

2.3.1 Single-Element Pruning. The results for single-element pruning are presented in both Figure 1a and Table 5 (located in the Appendix). Table 5 includes the pruning magnitudes r_k determined at each time step to specifically decrease the model size by 10% if its original size. Model performance was measured by validation loss.

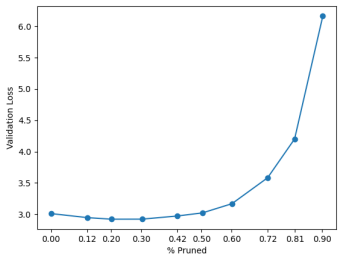
2.3.2 Row-wise Pruning. The results row-wise pruning are presented in Figures 1b and 1c; as well as Table 6 (located in the Appendix). Table 6 includes the pruning magnitudes r_k determined at each time step to specifically decrease the model size by 5% if its original size. Model performance was measured by validation loss.

Note that, because of the note in the instructions, "Do not worry about compression", we did not compress the row-wise-pruned models.[2] As a result, we measured model speed according to the number of seconds required per training iteration.

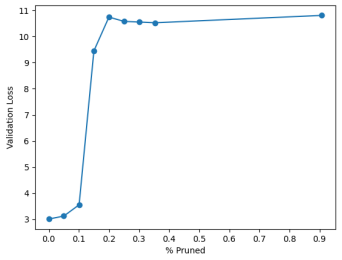
You will also notice a jump in the percentage pruned from 35% at iteration 450 to 90% at iteration 500: this was because the model was performing so badly by 35%-pruned, that we decided it wouldn't make any significant difference to jump straight to 90%-pruned from there (and we were correct).

[1]We tried for weeks to get GPU resources on Google Cloud but were unsuccessful. We even wrote a script that utilized the `gcloud` API to continuously ping the Google Cloud server to start up the requested VM. The script rotated between different regions with each rejection. We ran it for a straight 24 hours one time and we were never granted resources. Thus, we had to resort to a smaller GPU RAM single GPU on Google Colab.

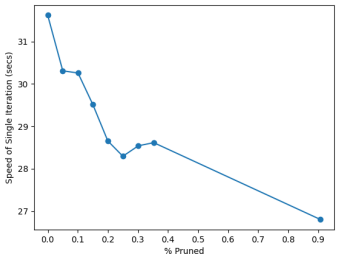
[2]However, this wasn't for lack of trying. Our `CompressedLinear` class just wasn't behaving correctly and we had to move on to other parts of the project.



(a) Validation loss vs. total percentage of pruned parameters for single-element pruning.



(b) Validation loss vs. total percentage of pruned parameters for row-wise pruning.



(c) Speed (seconds per training iteration) vs. total percentage of pruned parameters for row-wise pruning.

Fig. 1. Performance of the single-element and row-wise pruned models

2.4 Experimental Analysis

Single-element pruning was much more effective than row-wise pruning in maintaining model performance while increasing parameter sparsity. This is due to the less-aggressive nature of single-element pruning. In fact, we were able to reduce the model to about 50% of its original size using single-element pruning before seeing a more significant increase in validation loss (as compared to row-wise pruning which saw a significant increase in performance after only decreasing model size to 10%).

The poor performance of the row-wise pruning could be due to two decision we made in our approach:

- our decision to prune the Language Modeling Head as well (which could be much more sensitive to row-pruning, as it is uniquely important to the model); and/or
- our decision to prune according to the value of the size-adjusted $\frac{1}{m}\|W_i\|_2$ instead of just the norm $\|W_i\|$ of each row.

In fact, after checking the 90%-pruned model, we found that the language modeling head had almost been completely pruned. However, without additional testing against control models (models without those two decision), we don't know for certain.

We did see more success with the row-wise pruned model by decreasing the aggressiveness with which we pruned, from 10% to 5% at each iteration. We started row-wise pruning with 10%-reduction, but found that it jumped straight to about 7.5 validation loss; however, when we switched to 5%-reduction at each iteration, we were able to keep the model under 4 validation loss by the time we reached 10% total reduction again. Thus, the less aggressive the model is pruned at each iteration, the more likely the model optimizer will be able to adjust for the introduced sparsification.

Fortunately, even without compression, we did see a linear speedup in our row-wise pruned model as the percentage of pruned parameters increased.

We did also notice something interesting about model parameter updates during pruning: because of the saved optimizer state, the momentum parameters of the AdamW algorithm continued to update some of our pruned parameters even after they had been pruned (as the momentum carried over directions from pre-pruning updates). However, because those parameters were zeroed out by the pruning masks anyways, they still had no affect on the model output.

2.5 Conclusions & Future Work

Clearly, single-element pruning has a significant performance advantage over row-wise pruning. However, if the speedup-performance tradeoff obtained from row-wise pruning only 10% of the model parameters is acceptable, row-wise is advantageous in the fact that it is easier to compress (by removing dimensions from weight matrices). To more fully take advantage of the speedup from single-element pruning, it would be necessary to convert the parameters to PyTorch sparse matrices – which simply won't be that effective until the matrices are *much* sparser than 90%.

If we had the time to continue this project, we would recommend the following improvements or experiments:

- conduct row-wise pruning on all linear layers *except* the language modeling head;
- conduct row-wise pruning using just the row norm $\|W_i\|$ instead of the adjusted row norm $\frac{1}{m}\|W_i\|$;
- try more strategically choosing magnitudes r_k, perhaps even customizing magnitudes per each parameter matrix (e.g., for one layer have one magnitude threshold, and for another use a different threshold) by analyzing the distribution of row magnitudes in each layer; and
- analyze the speedup performance that can be achieved by converting single-element-pruned layers into PyTorch sparse matrices.

3 LEADERBOARD

3.1 KV-cache

We implemented KV-caching in our leaderboard submission to improve inference latency. It achieves a 2× speedup from a throughput of 427.8 to 929.3 tokens per second on batch size 12 using GPT2.

3.2 LoRA

3.2.1 Introduction. Low Rank Adaptation (LoRA) freezes the original model weights and reduces the number of trainable parameters by learning pairs of rank-decomposition matrices. This greatly improves training throughput without sacrificing accuracy [2].

3.2.2 Experiments and results. We created our own custom `LoraLinear` layers (see `lora.py`) and replaced every linear layer in the original GPT2 with a `LoraLinear` layer for training. We set the A and B matrices of each LoRA layer trainable and froze the rest of the model weights; after training, the trained LoRA matrices were merged back with the initial pretrained linear weights.

In practice, we found that using LoRA with the default hyperparameter settings degraded the performance and did not result in significant improvements to throughput (the improvement was only about 1 second per iteration). We hypothesize that this is because LoRA is intended to be used in finetuning, not training.

3.3 Investigating inference precision

We explored the effects of using different precision on inference latency to verify the speedup of using fp16 over fp32. This applies to hardware that supports faster half precision computation over full precision, in particular, T4 has an optimal theoretical optimal performance of 65 TFLOPS for half-precision. The inference latency of GPT2 is recorded to be 427.8 tokens per second in half precision, but only 89.2 tokens per second in full precision when running with batch size 12.

REFERENCES

[1] Tao Ge, Heming Xia, Xin Sun, Si-Qing Chen, and Furu Wei. 2022. Lossless Acceleration for Seq2seq Generation with Aggressive Decoding. arXiv:2205.10350 [cs.CL]

[2] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2021. LoRA: Low-Rank Adaptation of Large Language Models. arXiv:2106.09685 [cs.CL]

[3] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2017. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. arXiv:1712.05877 [cs.LG]

[4] Yaniv Leviathan, Matan Kalman, and Yossi Matias. 2023. Fast Inference from Transformers via Speculative Decoding. arXiv:2211.17192 [cs.LG]

[5] Tailin Liang, John Glossner, Lei Wang, Shaobo Shi, and Xiaotong Zhang. 2021. Pruning and quantization for deep neural network acceleration: A survey. *Neurocomputing* 461 (2021), 370–403.

[6] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism. arXiv:1909.08053 [cs.CL]

4 APPENDIX

Total Finetuning Iterations	Pruning Magnitude r_k	Percent Pruned	Validation Loss
100	0.0	0.0	3.0115
200	0.015	0.121411	2.9465
300	0.025	0.200278	2.9221
400	0.0385	0.302579	2.9228
500	0.055	0.419007	2.9726
600	0.068	0.502855	3.0223
700	0.085	0.600713	3.1691
800	0.11	0.719068	3.5823
900	0.135	0.808705	4.2020
1000	0.175	0.902288	6.1618

Table 5. Number of total finetuning iterations, pruning magnitudes r_k, percentage pruned, and validation loss for each successive single-element-pruned model.

Total Finetuning Iterations	Pruning Magnitude r_k	Percent Pruned	Validation Loss	Training Iteration Time (seconds)
100	0.0	0.0	3.0115	31.62078
150	0.0004475	0.050147	3.1267	30.30906
200	0.000502	0.100862	3.5571	30.25804
250	0.0014	0.150033	9.4494	29.51662
300	0.0016176	0.200024	10.7443	28.65535
350	0.0076	0.250145	10.5797	28.29412
400	0.0155	0.301626	10.5547	28.54077
450	0.0235	0.352815	10.5269	28.61473
500	0.165	0.907922	10.8089	26.80073

Table 6. Number of total finetuning iterations, pruning magnitudes r_k, percentage pruned, validation loss, and seconds per training iteration for each successive row-wise-pruned model.