# CS 229S Final Project

Aditya Agrawal
adityaag@stanford.edu

Soumya Chatterjee
soumyac@stanford.edu

Simon Kim
ksimon12@stanford.edu

## 1 Parallelism

For this part, we were able to procure 4 TITAN V GPUs. As per the handout, we used the PyTorch FSDP library for FSDP, and implemented our own version of Tensor Parallelism in line with the Megatron paper: `https://arxiv.org/pdf/1909.08053.pdf`.

### 1.1 Fully Sharded Data Parallelism (FSDP)

Fully Sharded Data Parallelism (FSDP) is a method for parallelizing neural network training across multiple GPUs or nodes. It is particularly effective for training large models that cannot fit into the memory of a single GPU. FSDP addresses the limitations of traditional data parallelism which maintains a per-GPU copy of a model's parameters, gradients and optimizer states, by sharding all of these states across data-parallel workers and even optionally offload the sharded model parameters to CPUs.

The parameters (and gradients and optimizer state) of the model are divided into FSDP units which are distributed across GPUs and the required unit is fetched into a particular GPU when it needs that unit (for example when doing a forward pass on a layer which it does not have the weights for). Since at any point a GPU will only store a subset of units, the memory requirement is reduced from traditional data parallel but at a added cost of communicating FSDP units.

#### 1.1.1 Design Choices

There are a variety of adjustable configurations in FSDP, e.g. how to shard the parameters, what the size of each shard will be and how the sharded data is synchronized and aggregated across devices during the training process. We tune the exposed configurations in PyTorch for efficiency and faster convergence.

**Wrapping Policy** FSDP provides an `auto_wrap_policy` feature that is used to decide which parameters and modules will be sharded into distinct FSDP units. For Transformer architectures where there can be weight sharing (e.g. input embeddings and LM head), the shared weights need to be placed in the outer FSDP unit. In addition, having each Transformer block in a single FSDP results in more efficient communication. As can be seen in the code (Appendix A, we utilize these features in our implementation.

**Sharding Strategy** The sharding strategy decides if parameters, gradients, optimizer states or some combination of them are sharded across devices. As seen in class, these correspond to various stages of the ZeRO technique. For our implementation, we use the `FULL_SHARD` policy which corresponds to ZeRO Stage 3 - or sharding parameters, gradients and optimizer state. We also tried the other sharding strategies.

#### 1.1.2 Deliverable 1: Throughput vs Batch Size

The plot for throughput vs batch size for FSDP is given in Figure 1. Given a batch size $B$, a block size $L$, gradient accumulation steps $m$, iterations $i$ and execution time $t$, the throughput is given by $\frac{iBLm}{t}$. As we can see from the figure, the throughput increases with batch size. However, the training throughput gains with batch size is sub-linear. The maximum batch size that we able to get to run on our 12GB GPUs was 64 and the training throughput at this batch size was about 14k tokens/sec.

#### 1.1.3 Deliverable 2: Memory Usage vs Batch Size

The plot for memory usage vs batch size for FSDP is in Figure 1. Observe that as we increase the batch size, the peak GPU memory usage during training increases linearly. For the highest batch size of 64, the reported maximum allocated

GPU memory is about 7.6GB. Note that we obtain all of these values by using the `torch.cuda.max_memory_allocated` function.

**Measurement Discrepancies** However, we would like to note that upon manual inspection via `nvidia-smi` (even with a very high refresh rate) it seemed that the reported GPU memory usage stays stable at about 11.5GB for each GPU at batch size 64. Upon further investigation, we observed that using the `PYTORCH_NO_CUDA_MEMORY_CACHING=1` flag with our experiment makes the memory usage values reported by `nvidia-smi` align more closely with the GPU memory usage numbers reported in the plot, albeit at the cost of making the experiment significantly slower. To investigate why this discrepancy exists we used the `torch.cuda.memory_summary` function during our experiments and observed that torch reserves some of the GPU for caching tensors and this component of the GPU utilization is omitted by the `torch.cuda.max_memory_allocated` function. Therefore, it seems that `torch.cuda.max_memory_allocated` is at best a rough estimate for the actual GPU memory usage and thus the consequent inferences must not be taken at face value.

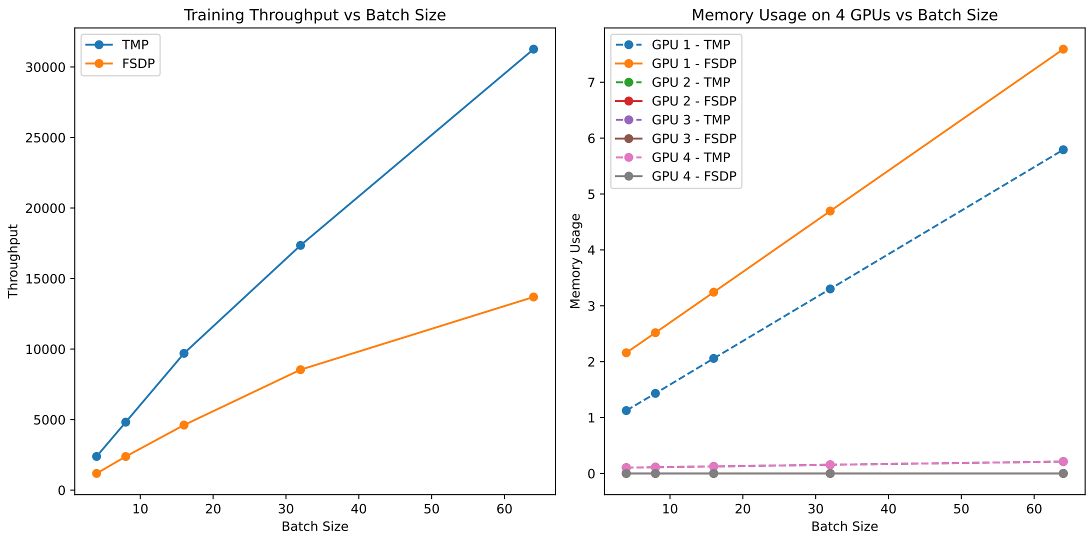


Figure 1: Plot of training throughput and memory usage per GPU as a function of batch size when training gpt2-medium using FSDP and Tensor Model Parallelism (TMP) on 4 TITAN V GPUs with a block size of 128 and 40 gradient accumulation steps. As suggested, the reported training throughput are averages over training iterations 20 to 40. As expected training throughput increases almost linearly with increasing batch size as does memory usage.

## 1.2 Tensor Model Parallelism

Tensor model parallelism involves decomposing large models into smaller tensors and distributing them across multiple devices. Unlike other parallelism strategies that keep individual weights intact, tensor model parallelism splits the parameters of layer across devices and computes parts of the layer in parallel across devices. Aggregation of the full tensor is only done when the full tensor is required. The derivation for the operations used in the tensor model parallel implementation of the transformer MLP is given in Appendix B and the code in Appendix C.

### 1.2.1 Design Choices

We implement tensor parallel versions of the Transformer MLP and Causal Self Attention layers. All other layers like the embedding, layers norm, etc are significantly less computationally expensive and are all performed on the master device (GPU 0 in our case). For the two layer MLP, the weights of the first layer are sharded along the output dimension across 4 GPUs while the weights of the second layer are sharded along the input dimension. As can be seen in Appendix B, this sidesteps the need for communication between the layers. We only need a broadcast at the start and an all reduce at the end (vice versa during backward pass) both of which have efficient implementations.

For the self attention layer, we note that the computation across heads is independent and that the output projection layer can be sharded along the input dimension. These observations lead to a very efficient tensor parallel attention implementation (Appendix D) where the communications happen only at the start and end of the attention block. Further we note that since the model had 12 heads while we had access to 4 GPUs only, each GPU computed a 3 head MHA instead of a single head attention.

#### 1.2.2 Deliverable 1: Throughput vs Batch Size

The plot for the throughput vs batch size for tensor model parallelism is given in Figure 1. Observe that, similar to FSDP, the throughput increases as we increase batch size. Moreover, observe that, similar to FSDP, the training throughput gains with batch size is sub-linear. The maximum batch size that we able to get to run on our 12GB GPUs was 64 and the training throughput at this batch size was about 14k tokens per second. We suspect that the additional data communication overhead for larger batch sizes (as the data may not be loaded completely in parallel) is what causes the sub-linear nature of the change in throughput with the change in batch size as opposed to the expected theoretical linear scaling.

#### 1.2.3 Deliverable 2: Memory Usage vs Batch Size

The plot for memory usage vs batch size for tensor model parallelism is in Figure 1. Observe that as we increase the batch size, the peak GPU memory usage during training increases linearly (for both the master GPU and the others). For the highest batch size of 64, the reported maximum allocated GPU memory is about 5.5GB. Note that we obtain all of these values by using the `torch.cuda.max_memory_allocated` function and the measurements discrepancies discussed in Section 1.1.3 still apply here.

### 1.3 Conclusion & Analysis

#### 1.3.1 Deliverable 1: Comparison of FSDP vs TMP

Both Fully Sharded Data Parallelism (FSDP) and Tensor Model Parallelism (TMP) shard parameters across multiple devices to reduce the per-device memory footprint. However, they approach this memory reduction in very different ways.

FSDP is similar to vanilla data parallel but instead of each device storing the entire set of parameters leading to a huge redudancy, in FSDP each device stores a subset of parameters (FSDP units) and fetches units from other devices as and when needed. Typically, the weights of a single layer would not be split across FSDP units. So, similar to data parallel, the entire computation on a single portion of a batch is completed on each device. However, the batch is split across devices.

In TMP on the other hand, the weights of a layer are split across devices and each device holds a portion of the weights of a particular layer. In contrast to FSDP, these weights are never moved around and each device is responsible for doing its own computation using the parts of the weight that it holds.

So in FSDP, model weights are communicated but activations are not while in TMP model weights are never communicated but activations are. FSDP is quite general and can work for a wide range of models but TMP requires careful analysis of the computation to find avenues for parallelism without adding too much synchronization overhead.

#### 1.3.2 Deliverable 2: Throughput & Memory Usage of FSDP and TMP

Figure 1 compares the throughput and memory usage of FSDP and tensor model parallelism. We can see that tensor model parallelism achieves higher throughput than FSDP and has better scaling properties as well (a larger slope). This could be because of the fact that we have activation checkpointing in FSDP which increases the amount of computation needed in the forward + backward pass of the model leading to a higher running time or a lower throughput.

Initially, we hypothesized that FSDP would lead to a higher throughput as opposed to TMP. TMP typically requires much more frequent communication, and given that the communication bandwidth is the same for both settings (single node, inter-device communication), it would make sense that FSDP leads to higher throughput. Surprisingly, as shown in Figure 1, TMP had a higher throughput. We offer the following explanation. The per-device memory requirement for FSDP was higher than TMP, as shown in Figure 1 (right). To support larger batch sizes, we performed activation checkpointing. This step is explained in greater detail in later sections. We found that this extra computation step is probably responsible for the decrease in throughput for FSDP. As shown in a recent NVIDIA report `https://people.eecs.berkeley.edu/~matei/papers/2021/sc_megatron_lm.pdf`, activation checkpointing can lead to a sublinear scaling of throughput as a function of batch size (Figure 18).

The memory usage for both FSDP and tensor model parallelism both increase with increasing batch size. But the memory usage of tensor model parallelism is less than that of FSDP for all batch sizes. This could be for a variety of reasons: FSDP uses more memory as there are multiple shards (which are large due to the number of GPUS and our wrapping policy) that need to be moved onto a single GPU during the forward and backward passes.

## 2 Pruning

For this part, we were able to procure a single instance of A40 GPUs. We performed iterative magnitude-based pruning for both L1 Unstructured and L2 Structured approaches. As expected, unstructured pruning has greater advantages in memory usage and inference time at the sake of model quality. For unstructured pruning, we were able to get down to 10% of the original model size without noticeable loss in quality. For structured pruning, however, we were not able to prune as drastically, due to reasons explained below.

### 2.1 L1 Unstructured Pruning

#### 2.1.1 (Protocol A) Deliverable 1: Model Size vs Quality

For this part, we implemented the "fixed-number-finetuning" approach outlined in Algorithm 1. We calculate the fixed pruning rate $\gamma$ as follows:

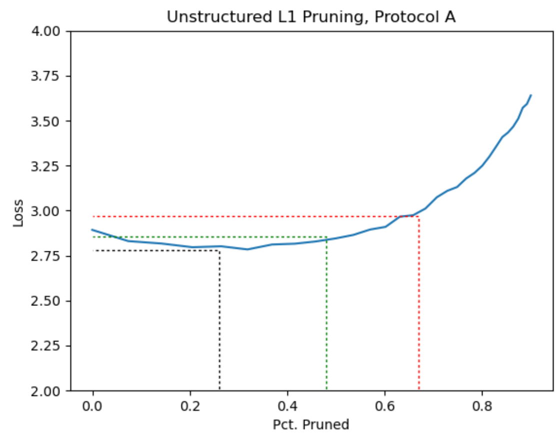
$$\gamma = 1 - \left(\frac{1}{10}\right)^{1/\theta} \tag{1}$$

where $\theta$ is the number of pruning steps we want to perform. This guarantees that by the end of $\theta$ pruning iterations, we reach the desired size of 10% of the original model parameters. Note that, formulated this way, $\theta$ effectively sets how *aggressively* we want to prune at each step.

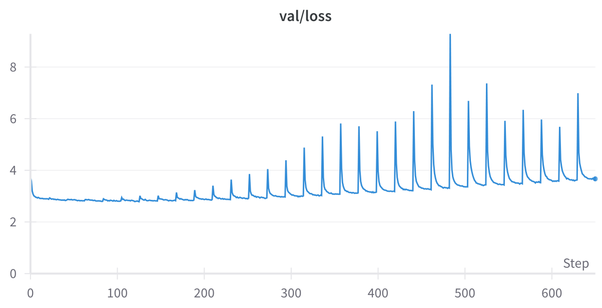
**Algorithm 1** Fixed Number Finetuning Model Pruning

```
1: while model_size > orig_model_size × 10% do
2:     Finetune for 100 iterations
3:     Prune $\gamma$% of model's entire paramters (i.e. prune globally, not per layer)
4: end while
```

For each parameter in the model, we define a mask that is initially set to ones. At each pruning step, we find the top-k smallest parameters globally and zero out their mask. Then, for all subsequent forwards, we do an element-wise multiplication of the paramter and its mask. Notice that this approach does not lead to improved memory footprint (in fact, memory usage would actually increase due to the extra storage requirement of the masks).



(a) Resulting Model Quality



(b) Val Loss While Pruning

Figure 2: Unstructured L1 Pruning, Fixed-iteration

For the following experiments, we set $\theta = 30$, resulting in a $\gamma \simeq 7\%$. The results are shown in Figure 2. The dashed lines in Figure 2a are reference losses for magnitude-based pruning obtained in Table 1 of the following paper: `https://aclanthology.org/2023.findings-acl.692.pdf`. Note that the reference paper used GPT-2 mini ($\simeq$ 120 params), whereas we used GPT-2 medium ($\simeq$ 350M params). Figure 2b graphs out the validation loss throughout pruning.

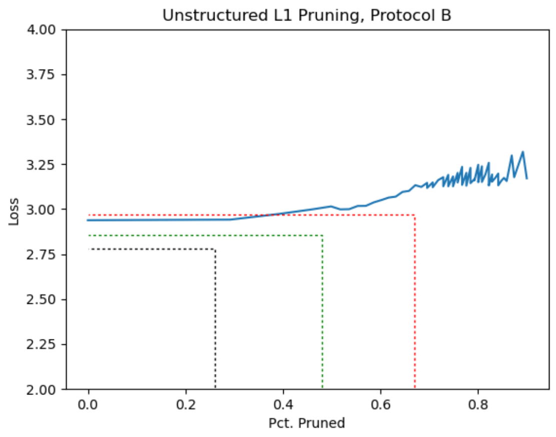
As expected, quality diminishes as the model size decreases. However, contrary to what we had initially hypothesized, the loss *does not* increase monotonically with decreasing model size. We believe that this is because, at the earlier stages of pruning, the graph is overlapped with the *fine-tuning* curve. We were using GPT-2 out of the box without any fine-tuning, and therefore, the decreasing loss in the earlier stages is an effect of both the fine-tuning curve (going down) and the pruning curve (going up), with a stronger emphasis on the former when pruning is not as present. The trend observed during training/pruning is clear by looking at Figure 2b. At the start of each pruning iteration, validation loss spikes, indicating a significant degradation of quality. This is expected, as the current weights are trained on the assumption of certain parameters existing. During the 100 step finetuning, however, model quality is quickly restored as the model learns to do its task with the remaining parameters. This confirms that LLMs are over-parameterized and amenable to pruning. Finally, as expected, we observed that decreasing $\theta$ leads to a more noticeable degradation of quality. The final quality of the model at 10% is also higher, as the spikes in loss are harder to recover from at each step.

#### 2.1.2 (Protocol B, OPTIONAL) Deliverable 2: Model Size vs Quality

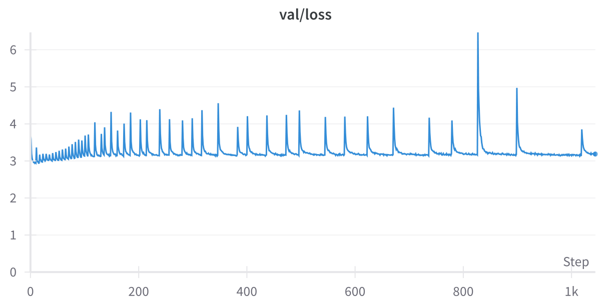
For this part, we implemented the "loss-guided" approach outlined in Algorithm 2. For the large part, we observe the same trend as we did using Protocol A. One thing to note from Figure 3 is that the loss is relatively stable. This is expected, because we essentially fixed the model to have a certain loss. Another interesting thing to note is the pruning interval, readily observable in Figure 3b. In the earlier stages, pruning occurs much more frequently than in Protocol A. This is because we are able to prune more aggressively without losing too much quality in the early stages when the model is severely over-paramaterized. However, once the model is pruned to a reasonable degree, we are not able to prune as much at each step. In conclusion, protocol B would be suitable if pruning the model to a conservative degree, as it is faster. However, if pruning to 10% of the original size, using fixed finetuning would yield faster results.

**Algorithm 2** Loss-Guided Model Pruning

```
1: Finetune the model for 50 iterations
2: while model_size > orig_model_size × 10% do
3:     Prune as much as possible until validation loss ≤ 3.2
4:     Finetune for 25 iterations
5: end while
```



(a) Resulting Model Quality



(b) Val Loss While Pruning

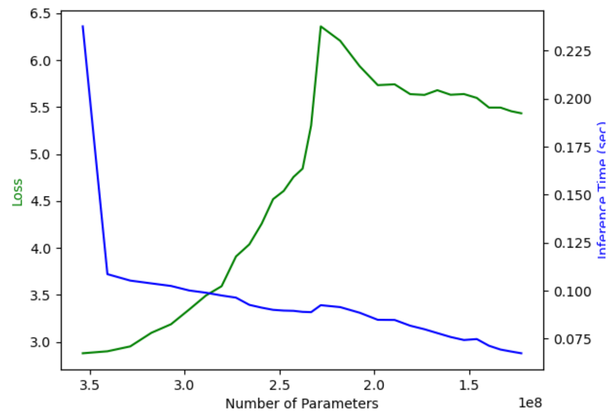
Figure 3: Unstructured L1 Pruning, Loss-guided

## 2.2 L2 Structured Pruning

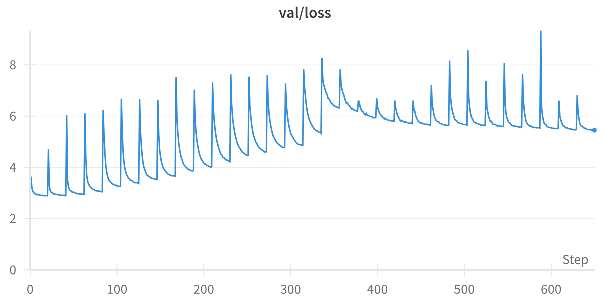
#### 2.2.1 (Protocol A) Deliverable 1: Model Size vs Quality

For this part, we implemented the "fixed-number-finetuning" approach outlined in part A, this time with structured L2 pruning. Instead of assessing the magnitude of each individual weight, we consider the L2 norm of the entire row. We used the same masking approach as above, this time masking entire rows. Then, at checkpoint save, we modified the checkpoint such that all zeroed out weights are actually removed before getting saved. Along with it, we saved a single row mask indicating the indices of rows retained for each weight. Once the checkpoint is loaded, we are able to remove the dimensions from the *input* on the fly, using the saved mask, to get the right dimensions.

**NOTE:** We were running our experiments on a shared GPU instance. When doing L2 pruning, the memory usage of the device increased, and we were unable to have all the masks in memory. Therefore, we were only able to reach around 35% of the original model size.



(a) Resulting Model Quality (b - time, g - loss)



(b) Val Loss While Pruning

Figure 4: Structured L2 Pruning, Fixed-iteration

In Figure 4a, we observe a clear, general trend of decreasing inference time (blue, right) and increasing loss (green, left) with decreasing model parameters. This is expected, because structured pruning directly translates to reduced memory and faster inference, but loses quality due to a more aggressive pruning. An interesting anomaly happens when there is a sudden spike in the loss, observed both in Figure 4a and 4b, then a *decrease*. This result was contrary to our expectation that loss would be monotonically increasing. We hypothesize that this may be due to significant over-paramterization in the original model. In the beginning, removing channels may expectantly introduce noise to the model resulting in the increasing loss. After a certain number of channels are pruned, the model may be learning to better adapt to the task with the channels that remain. We also observe in recent literature such as `https://arxiv.org/pdf/2204.00408.pdf` that there are instances where accuracy does not monotonically increase with model size (e.g. certain models in figure 2 of the referenced paper).

Overall, we discovered that pruning entire rows results in a more apparent decrease in memory usage and inference latency at the cost of model quality compared to pruning individual weights. This is expected. Unless using very specialized hardware, it is difficult to fully utilize an unstructured sparse matrix. However, reducing entire dimensions can result in memory and latency gains.

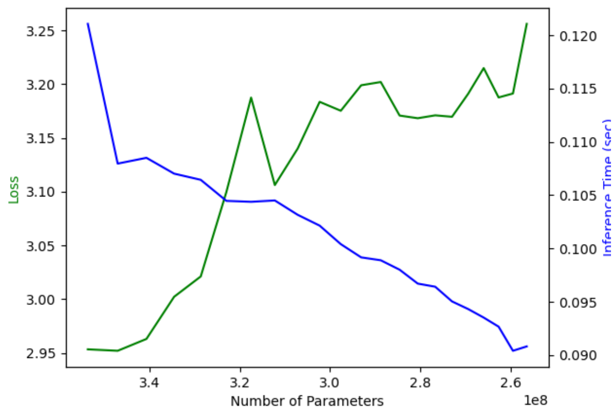
### 2.2.2 (Protocol B, OPTIONAL)

We implemented L2 pruning using the loss-guided pruning approach, the results of which are shown in Figure 5. As the general trend seems to be for L2 pruning, we have a rather unstable model quality (green). though, as expected, the inference time is decreasing with a smaller model size.

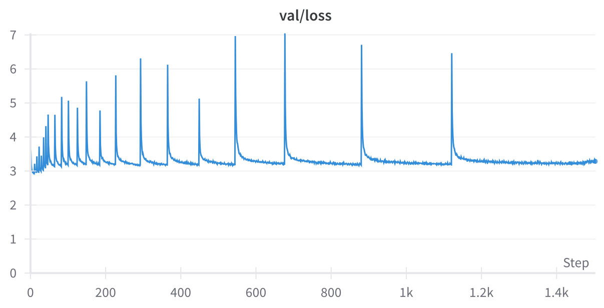
## 3 Leaderboard

In addition to trying out various configuration setting for FSDP and Tensor Model Parallelism for achieving the best loss and highest throughput numbers, we tried adding in activation checkpointing in FSDP to support ever larger batch sizes by reducing the amount of intermediate activations stored. This led to better loss values due to the model having seen more examples in the same number of interations. The activation checkpointing method is detailed below:

**FSDP with activation checkpointing** Though not a part of FSDP, we used activation checkpointing for our Transformer blocks. In activation checkpointing, the intermediate activations of operations inside a block are not stored during the forward pass to reduce memory requirements. During the backward pass, when these intermediate activations are required, the are recomputed at a block level. This reduces the memory utilization but increases the training time due to the additional computation needed during the backward pass. To reduce memory utilization for allowing training



(a) Resulting Model Quality (b - time, g - loss



(b) Val Loss While Pruning

Figure 5: Structured L2 Pruning, Loss-guided

on larger batch sizes, we used activation checkpointing. Our hypothesis was that the model quality improvements arising from larger batch size training would outweight the increase in training time.

## 4 Bonus Opportunity 1, Error Analysis

We perform error analysis on our pruned models. It is intuitive that quality degrades as model size decreases; however, we are interested in understanding the *types* of texts in which this discrepancy is most noticeable. Here are a few notable examples:

**Input:** Following his only season at Western Washington, McCarty declared...

| **1.00x original parameters** | **0.68x original parameters** | **0.10x original parameters** |
|---|---|---|
| ...for the 2009 NBA draft . Following the draft , McCarty was waived by the Colorado Rapids on July. | ...for the United States National Football League 's ( NFL ) Green Bay Packers in 1946 . He played | ...that " things are not acceptable in here , but as a result , nobody should not feel that they. |

"Western Washington" is an ambiguous term–it could mean the location (west-side of Washington) or the university. From context (e.g. the word "season"), the latter makes more sense. The first two "reasonably" pruned models properly recognized the connection to sports; however, the latter model did not, leading to bad results.

**Input:** The Federal Deposit Insurance Corporation (FDIC) is a...

| **1.00x original parameters** | **0.68x original parameters** | **0.10x original parameters** |
|---|---|---|
| ...federal agency that manages the federal savings accounts. Its primary mission is to manage and finance the savings and | ...federal agency serving as the primary body responsible for insured deposits and financial losses. It is a division of | ...reform of the state's fiscal system). As a result, MDOT issued a $3 |

We also hypothesized that an entity comprising of multiple tokens would add complexity to the task. We see that the first two models perform reasonably well at this task, whereas the last model fails dismally.

**Input:** The capital of France is Paris, the capital of Korea is Seoul, and the capital of Italy is...

| **1.00x original parameters** | **0.68x original parameters** | **0.10x original parameters** |
|---|---|---|
| ...Rome . | ...Rome . | ...Buenos Aires . |

For a more structured prompt like the above, we see that the quality degradation is not noticeable (except for the very obviously incorrect answer). Overall, we have found that the effects of pruning are more noticeable in tasks that involve ambiguous terms or complex entities. On more structured tasks such as the last example, which is essentially pattern-matching, we see that the degradation of quality is less noticeable. We use the checkpoints from L1 unstructured pruning. One interesting point to note is that the model's validation loss itself is within a reasonable bound ($< 4$). The pruned model's failure in certain tasks may be due to the inherent nature of the task (i.e. the types of examples) itself rather than the number of parameters. It would be interesting to further research whether different pruning approaches (iterative, adaptive, etc.) are more or less receptive to the above categories.

## A FSDP Code

```
gpt_auto_wrap_policy = functools.partial(
    transformer_auto_wrap_policy,
    transformer_layer_cls={Block,},
)
model = FSDP(model,
             auto_wrap_policy=gpt_auto_wrap_policy,
             mixed_precision=None,
             sharding_strategy=torch.distributed.fsdp.ShardingStrategy.FULL_SHARD,
             device_id=torch.cuda.current_device(),
             limit_all_gathers=True)

non_reentrant_wrapper = functools.partial(
    checkpoint_wrapper,
    offload_to_cpu=False,
    checkpoint_impl=CheckpointImpl.NO_REENTRANT,
)
check_fn = lambda submodule: isinstance(submodule, Block)
apply_activation_checkpointing(
    model, checkpoint_wrapper_fn=non_reentrant_wrapper, check_fn=check_fn
)
```

## B Tensor Parallel MLP

We give a worked out example of the Tensor Parallel MLP forward pass used in our implementation.

$$
\begin{aligned}
X &= \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \\ x_{31} & x_{32} \end{bmatrix}, \quad A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \\
Y &= \mathrm{GeLU}(XA) = \mathrm{GeLU}\left( \begin{bmatrix} x_{11}a_{11} + x_{12}a_{21} & x_{11}a_{12} + x_{12}a_{22} \\ x_{21}a_{11} + x_{22}a_{21} & x_{21}a_{12} + x_{22}a_{22} \\ x_{31}a_{11} + x_{32}a_{21} & x_{31}a_{12} + x_{32}a_{22} \end{bmatrix} \right) \\
A &= \begin{bmatrix} A_1 & A_2 \end{bmatrix}, \quad A_1 = \begin{bmatrix} a_{11} \\ a_{21} \end{bmatrix}, \quad A_2 = \begin{bmatrix} a_{12} \\ a_{22} \end{bmatrix} \\
XA_1 &= \begin{bmatrix} x_{11}a_{11} + x_{12}a_{21} \\ x_{21}a_{11} + x_{22}a_{21} \\ x_{31}a_{11} + x_{32}a_{21} \end{bmatrix}, \quad XA_2 = \begin{bmatrix} x_{11}a_{12} + x_{12}a_{22} \\ x_{21}a_{12} + x_{22}a_{22} \\ x_{31}a_{12} + x_{32}a_{22} \end{bmatrix} \\
Y_1 &= \mathrm{GeLU}\left(XA_1\right) = \begin{bmatrix} y_{11} \\ y_{21} \\ y_{31} \end{bmatrix}, \quad Y_2 = \mathrm{GeLU}\left(XA_2\right) = \begin{bmatrix} y_{12} \\ y_{22} \\ y_{32} \end{bmatrix} \\
Y &= \begin{bmatrix} Y_1 & Y_2 \end{bmatrix} \\
B &= \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} B_1 \\ B_2 \end{bmatrix}, \quad B_1 = \begin{bmatrix} b_{11} & b_{12} \end{bmatrix}, \quad B_2 = \begin{bmatrix} b_{21} & b_{22} \end{bmatrix} \\
Y_1B_1 &= \begin{bmatrix} y_{11}b_{11} & y_{11}b_{12} \\ y_{21}b_{11} & y_{21}b_{12} \\ y_{31}b_{11} & y_{31}b_{12} \end{bmatrix}, \quad Y_2B_2 = \begin{bmatrix} y_{12}b_{21} & y_{12}b_{22} \\ y_{22}b_{21} & y_{22}b_{22} \\ y_{32}b_{21} & y_{32}b_{22} \end{bmatrix} \\
g\left(Y_1B_1, Y_2B_2\right) = Y_1B_1 + Y_2B_2 &= \begin{bmatrix} y_{11}b_{11} + y_{12}b_{21} & y_{11}b_{12} + y_{12}b_{22} \\ y_{21}b_{11} + y_{22}b_{21} & y_{21}b_{12} + y_{22}b_{22} \\ y_{31}b_{11} + y_{32}b_{21} & y_{31}b_{12} + y_{32}b_{22} \end{bmatrix} = YB
\end{aligned}
$$

## C Code for Tensor Parallel MLP

```python
class TensorParallelMLP(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.devices = [torch.device(f'cuda:{i}') for i in range(config.num_shards)]

        self.c_fc_shards = nn.ModuleList()
        self.gelu_shards = nn.ModuleList([nn.GELU() for _ in range(config.num_shards)])
        self.c_proj_shards = nn.ModuleList()
        self.dropout = nn.Dropout(config.dropout)

        split_dim = 4 * config.n_embd // config.num_shards

        for i in range(config.num_shards):
            self.c_fc_shards.append(nn.Linear(
                config.n_embd, split_dim, bias=config.bias, device=self.devices[i]))
            self.c_proj_shards.append(nn.Linear(
                split_dim, config.n_embd, bias=config.bias, device=self.devices[i]))

    def forward(self, x):
        x = f.apply(x, self.devices)
        x = parallel_apply(self.c_fc_shards, x, None, self.devices)
        x = parallel_apply(self.gelu_shards, x, None, self.devices)
        x = parallel_apply(self.c_proj_shards, x, None, self.devices)
        x = g.apply(x, self.devices)
        x = self.dropout(x)
        return x

class f(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x, devices, out_device=None):
        ctx.devices = devices
        ctx.out_device = out_device
        x = Broadcast.apply(devices, x)
        return x

    @staticmethod
    def backward(ctx, gradient):
        gradient = comm.reduce_add(gradient, ctx.out_device)
        return gradient

class g(torch.autograd.Function):
    @staticmethod
    def forward(ctx, x, devices, out_device=None):
        ctx.devices = devices
        ctx.out_device = out_device
        x = comm.reduce_add(x, ctx.out_device)
        return x

    @staticmethod
    def backward(ctx, gradient):
        gradient = Broadcast.apply(ctx.devices, gradient)
        return gradient
```

## D Code for Tensor Parallel Attention

```
class TensorParallelAttention(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.devices = [torch.device(f'cuda:{i}') for i in range(config.num_shards)]

        self.k_shards = nn.ModuleList()
        self.q_shards = nn.ModuleList()
        self.v_shards = nn.ModuleList()
        self.attn_shards = nn.ModuleList()
        self.c_proj_shards = nn.ModuleList()
        self.resid_dropout = nn.Dropout(config.dropout)
        self.n_head = config.n_head

        num_shards = config.num_shards
        assert self.n_head % num_shards == 0

        attn_split_dim = config.n_embd // num_shards

        for i in range(num_shards):
            self.k_shards.append(nn.Linear(config.n_embd, attn_split_dim, device=self.devices[i]))
            self.q_shards.append(nn.Linear(config.n_embd, attn_split_dim, device=self.devices[i]))
            self.v_shards.append(nn.Linear(config.n_embd, attn_split_dim, device=self.devices[i]))

            self.attn_shards.append(
                Attention(config, dropout=nn.Dropout(config.dropout)).to(self.devices[i]))
            self.c_proj_shards.append(nn.Linear(
                config.n_embd // num_shards, config.n_embd, device=self.devices[i]))


    def forward(self, x):
        x = f.apply(x, self.devices)
        ks = parallel_apply(self.k_shards, x, None, self.devices)
        qs = parallel_apply(self.q_shards, x, None, self.devices)
        vs = parallel_apply(self.v_shards, x, None, self.devices)

        inputs = []
        for i in range(len(ks)):
            inputs.append((ks[i], qs[i], vs[i]))

        ys = parallel_apply(self.attn_shards, inputs, None, self.devices)
        out = parallel_apply(self.c_proj_shards, ys, None, self.devices)
        z = g.apply(out, self.devices)
        z = self.resid_dropout(z)
        return z
```