Homework 3
CSC 277 / 477
End-to-end Deep Learning
Fall 2024

Tianyi Zhou - tzhou25@ur.rochester.edu

Oct. 22 2024

Deadline: See Blackboard

Instructions

Your homework solution must be typed and prepared in LaTeX. It must be output to PDF format. To use LaTeX, we suggest using `http://overleaf.com`, which is free.

Your submission must cite any references used (including articles, books, code, websites, and personal communications). All solutions must be written in your own words, and you must program the algorithms yourself. **If you do work with others, you must list the people you worked with.** Submit your solutions as a PDF to Blackboard.

Your programs must be written in Python. The relevant code should be in the PDF you turn in. If a problem involves programming, then the code should be shown as part of the solution. One easy way to do this in LaTeX is to use the verbatim environment, i.e., \begin{verbatim} YOUR CODE \end{verbatim}.

Problem 1: Distributed Model Training and Optimization Techniques (30 Points)

Part(a): Benefits and Challenges of Distributed Model Training (10 points)

Distributed model training enables faster training times and allows scaling to larger datasets and models, but it also presents several challenges. Write a brief essay (around 200 words) addressing the following points:

- **Benefits**: Discuss the advantages of distributed training, such as reduced training time, scalability, and the ability to handle large models and datasets that don't fit on a single GPU. Include real-world examples (e.g., training models like GPT-3, BERT).
- **Challenges**: Explore the difficulties, including communication overhead, model synchronization, and potential bottlenecks like straggler nodes. Use real-world scenarios where distributed training is essential (e.g., cloud-based environments, large-scale NLP models).

Answer:

Benefits:
The advantages of distributed model training greatly depends on *parallelization*. By handling different chunks of data on multiple GPUs using *Data Parallel*, we can reduce training time and allows for larger dataset to be trained using same time as we using a single GPU. By fitting one model across different GPUs, one can train model that won't suit a single GPU, thus allowing scalability. For example, since industry models are too large, one always have to use sharding to enable larger model and dataset size for training.

Challenges:
Distributed model training requires communication between GPUs, and the communication overhead is sometimes horrifying. When deploying nodes with different GPUs, synchronization can be challenging due to different reliabilities: some might finished task and forced to wait till those less reliable GPUs to finish their tasks, causing longer wait time (the Straggler problem). Distributed training is essential that it is prone to hardware failure by having multiple nodes.

Part(b): Mixed-Precision Training and Activation Checkpointing (10 points)

Distributed training often requires efficient memory and computational resource management. In this section, briefly discuss:

- How **mixed-precision training** reduces memory usage and increases computational efficiency. Include a mathematical justification of how reducing the precision of floating-point operations can speed up training and lower memory requirements.

- How **activation checkpointing** trades off memory for additional computation. Use examples to illustrate how recomputing activations can reduce memory usage but increase computation time.

Answer:

Mixed-precision training:
Take FP32 and FP16 as an example, the memory requirement is proportional to the size of the data types. Therefore, by switching from FP32 to FP16, we halve the memory requirement, Mathematics justification:

$$Mem_{FP32} = N \times 32bits = N \times 4bytes \tag{1}$$

$$Mem_{FP16} = N \times 16bits = N \times 2bytes \tag{2}$$

$$\therefore \frac{Mem_{FP16}}{Mem_{FP32}} = 0.5 \tag{3}$$

Activation checkpointing:
Instead of storing all intermediate activations in the forward pass, only a subset of activations is saved. During the backward pass, activations are recalculated on-the-fly.

Memory Saving: Since we saved less activations, that means we are using less memory.

Extra Computation: During backward pass, any activation not stored need to be recomputed, adding computation overhead to the whole training process. For example, Let n be the total number of layers and c be the number of checkpoints (where $c < n$). The memory required for activations reduced from $O(n)$ to $O(c)$. However, the computational overhead increases because you need to recompute $O(n - c)$ activations during the backward pass.

Part(c): Comparing DataParallel and DistributedDataParallel in PyTorch (10 Points)

In this section, you'll explore two methods for multi-GPU training in PyTorch: `DataParallel` and `DistributedDataParallel`.

- Read online tutorials on PyTorch's DataParallel and DistributedDataParallel methods. Starting with a single-GPU script, describe how to modify the code to enable DataParallel and to enable DistributedDataParallel. Focus on the changes needed in the **model definition** and **dataloader** (if required) and provide a **high-level description** of these changes.
- Compare **DataParallel** and **DistributedDataParallel** based on: (1) The workload needed to modify the code; (2) Typical run time for training; (3) Which method is more flexible and can be used in more situations and why?

Answer:

Implementation:

- **DataParallel:**

```
# model = Model()
model = torch.nn.DataParallel(model)  # Enable dataparallel
```

- **DistributedDataParallel:**

```
torch.distributed.init_process_group(backend='nccl')
model = Model()
model = torch.nn.parallel.DistributedDataParallel(model)
train_sampler = torch.utils.data.distributed.DistributedSampler(train_dataset)
train_loader = torch.utils.data.Dataloader(dataset=train_dataset,
sampler=train_sampler)
```

Comparison: Workload: DataParallel requires less workload to modify the codes. It just need to wrap the model with *torch.nn.DataParallel*() is enough. On the other hand, DistributedDataParallel requires more work, as well as utilize sampler on the original dataloader. **Training time:** DDP is faster than DP because it only distribute model to multiple GPU once. DP distributes model to all nodes at each step, causing drastic communication overhead. **Flexibility:** DDP is more flexible and scalable since it's more efficient. And more workload to adapt it means more options can be added to the training process.

Problem 2: Programming Task (30 Points)

In this section, you will build upon the code from Homework 1, Problem 1.

Part(a) Effect of `num_workers` in DataLoader: (10 Points)

One important hyperparameter that affects training time in HW1 is `num_workers`, found in train.py under `loader_args`. In this task, you'll explore how this parameter impacts data loading speed.

Instructions:

- Measure the total run time for iterating through all batches in the **training set** as you increase `num_workers` from 1 to 10.

- Since no model training is required, create a script with only the necessary components for data loading.

Questions:

- What is the default value of `num_workers` in `torch.utils.data.DataLoader`? What does this default setting mean?
- Plot the run time as `num_workers` increases. What do you observe? Is the default setting optimal?

Answer: The default value of *num_workers* is 0, meaning the script is only using the main process to load data.

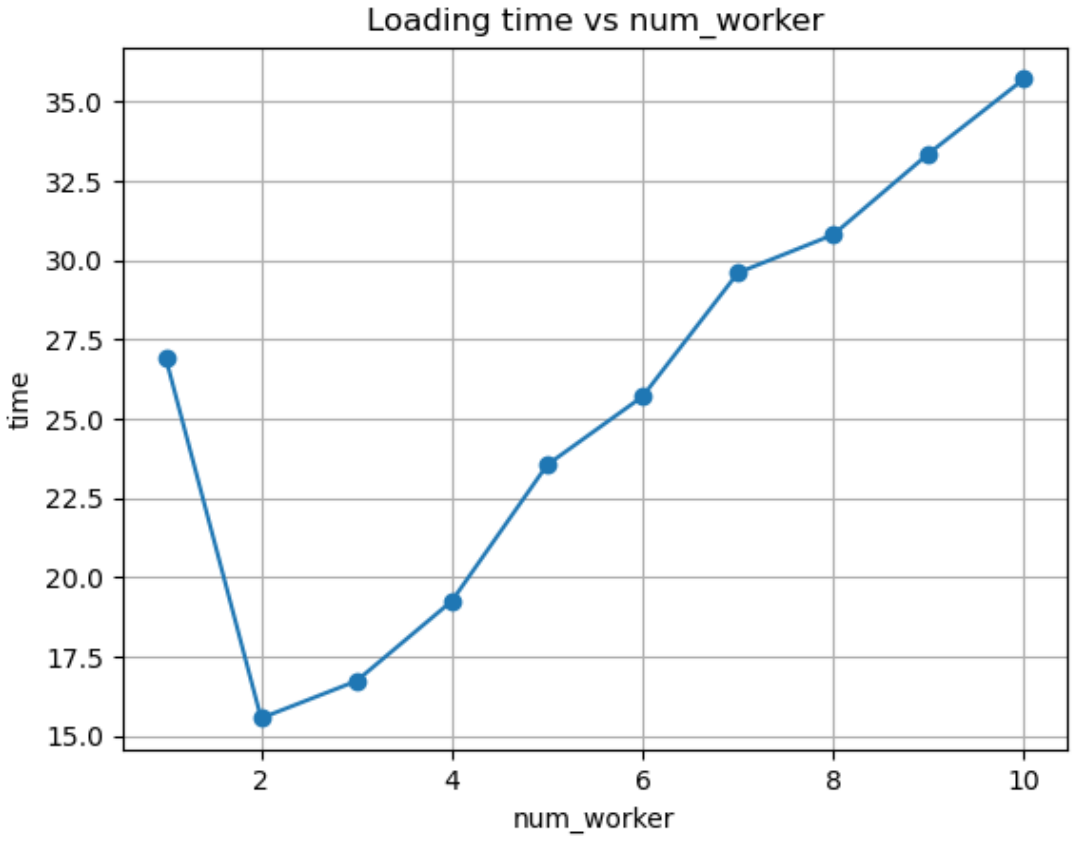

Figure 1: Number worker vs. Loading time

Observation:
As observed from graph, we can see a dramatic decrease (also the global minimal) when number of worker increases from 1 to 2, and it gradually increases back. When number of worker reaches 7, the loading time takes longer than the default setting. This might get explained by increasing synchronization overhead required for having more processes transferring the data.

Part(b) Code Profiling: (10 Points)

Profiling your code is crucial for optimizing deep learning models. In this task, you'll analyze the runtime of different components during model development.

Instructions:

- Modify train.py to record the **total time** spent in the following stages: (1) Data loading from the training Data Loader; (2) Model forward pass; (3) Loss calculation and backward pass; (4) Evaluation on the validation and test sets; (5)Other parts (i.e., overall runtime minus the above four parts).
- You can simply use `time.time()` or any profiling tool of your choice.

Questions:

- Briefly explain your method for recording the time taken for loading training data.
- Create a pie chart showing the proportion of time spent on the five components. Also, present the results in a LaTeX table. Analyze the pattern, and propose one way to optimize training efficiency.

Answer:

I'm using *Pytorch Profiler* for block labeling for futural inspection. But for recording time spent for loading training data (and other time spent), I used *time.time()* function.

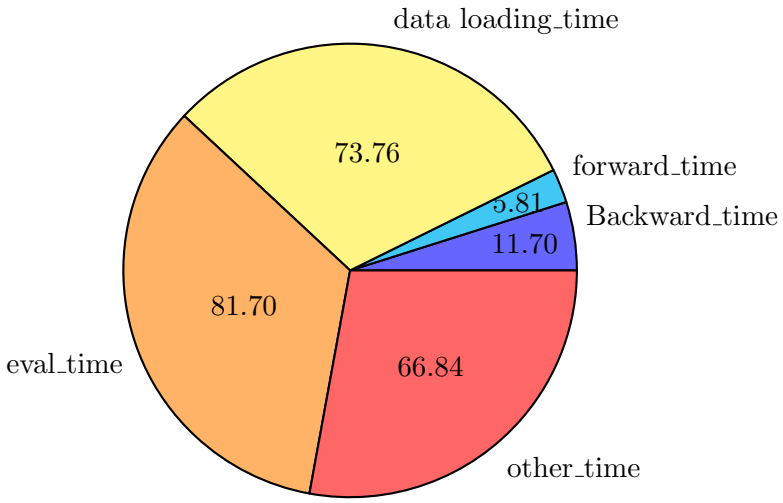

Table 1: Summary on RoBERTa-base wihtout lora

Category	Time in seconds
Eval Time	81.7
Data Loading Time	73.76
Forward Time	5.81
Backprop Time	11.7
Other Time	66.84

Analysis:
The result shows the three dominating categories are evaluation, data loading, and other time (disk IO, etc.); There are multiple ways to optimize my training process. First, since I'm using default data loader arg, which uses number of worker of 0. I can change it, according to my observation in part a, to 2 to reduce data loading time. Secondly, I can do some sampling to reduce the size of my validation set, and thus reduce evaluation time. Lastly, I can always do hardware upgrade to ensure not too many times are wasted in other categories.

Part(c) Automatic Mixed Precision Training: (10 Points)

In this task, you'll explore mixed precision training using `torch.amp`. Follow this tutorial, focusing on the "Typical Mixed Precision Training" section. Modify your code to enable mixed precision training.

Instructions:

- Compare the following aspects using your WandB logs: (1) Training loss dynamics;

(2) Final model performance; (3) System metrics, including GPU memory usage, total run time, and other relevant logs you found interesting under the "System" section in Wandb (these are automatically logged by Wandb during the experiments; x-axis in these plots indicates runtime).

Questions:

- What did you observe in terms of training loss, final performance, and system metrics? What conclusions can you draw from these experiments?

Answer: Performance Comparison:

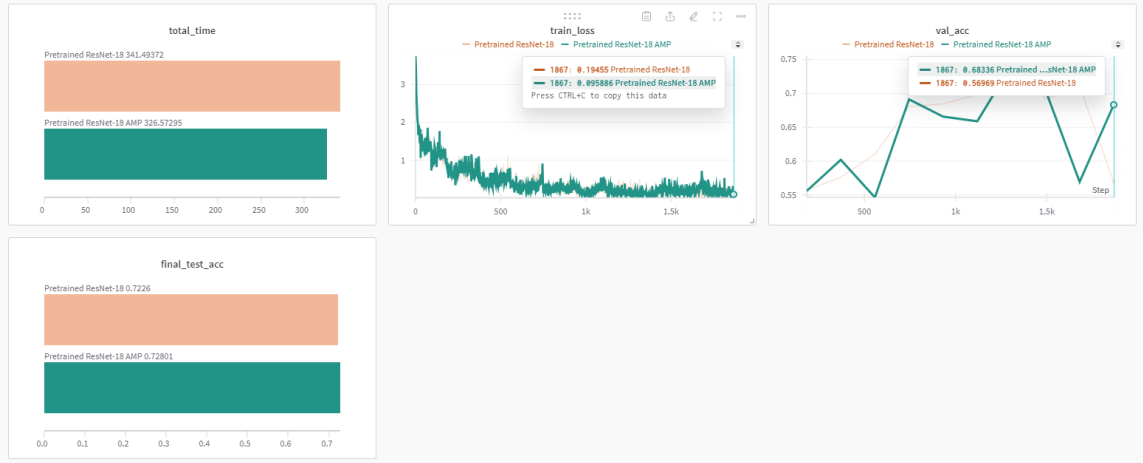

Figure 2: AMP Model vs. Normal model performance

System Comparison:

Figure 3: System Resources Usage AMP vs. Normal

Observation:
By comparing the two model's performance and system resources usage, we can observe that the two models obtained very similar final test accuracy; plus, they share a similar decrease in training loss.

Specifically, AMP model takes 14 seconds less training time. However, when it comes to final test accuracy, I find AMP model performs similar, even slightly better than the original model (by 0.6%).

Regarding system resources usage, I found that AMP model tends to use less GPU utilization (when batch size controlled to be the same), and less GPU RAM is allocated for the AMP model. Suggesting AMP model to be more system resources efficient.

1 Reference:

- pytorch profiler