Assignment 4

The goal of this assignment is to implement distributed training methods, including data parallelism and pipeline parallelism.

Setting up the code

The starting code base is provided in `https://github.com/llmsystem/llmsys_s24_hw4.git`. You will need to first install the requirements:

```
pip install -r requirements.txt
```

Setting up the environment

We strongly suggest you using machines in PSC to complete this homework. This is the link to the guide of using PSC. The command to require an interactive node with multiple GPUs is as follows.

```
# use GPU-shared flag for n <= 4
# use GPU flag for n = 8 or 16
# an interactive job is at most 8 hours (1 hour in this example)
interact -p GPU-shared --gres=gpu:v100-32:n -t 1:00:00
```

Requesting machines needs time, which will be much longer if there are lots of people requesting resources at the same time. Please plan your time to finish this assignment accordingly.

Problem 1: Data Parallel (50)

In this part, you are going to implement the data parallel training for GPT2. You are only allowed to use the following two packages to implement GPU communication. All the packages you need are defined in the starter codes. You must not write new `import` codes.

```
import torch.distributed as dist
from torch.multiprocessing import Process
```

Problem 1.1

1. Implement the helper function `partition_dataset` in `data_parallel/dataset.py`:

```
def partition_dataset(
    dataset, batch_size=128, collate_fn=None
) -> Tuple[DataLoader, int]:
    ...
```

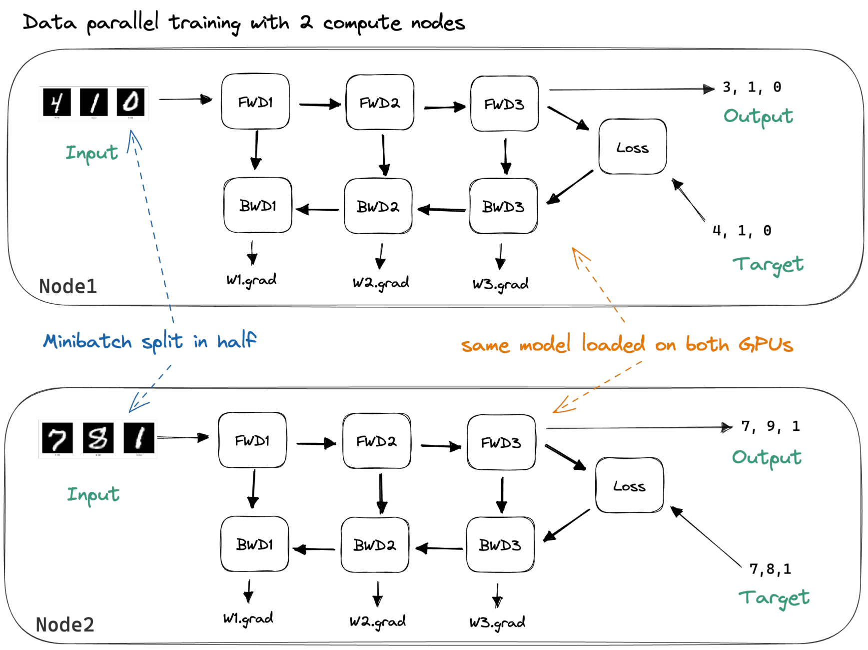


Figure 1: Data Parallel

Hint:

1. Calculate the partitioned batch size. We define `partitioned batch size` as the batch size on every individual device. The `partitioned batch size` for each GPU, for instance, would be $128//4 = 32$ if we have four GPUs and the total batch size is 128.
2. Create a partitioner class `DataPartitioner` with dataset and a list of partitioned sizes. The dataset's fraction that will be distributed among the parallel devices is represented by the list of partitioned sizes, which is a list of float values. For instance, if the dataset is divided equally over 4 GPUs, the list of `sizes` should look like this: $[0.25, 0.25, 0.25, 0.25]$.
3. Get the current partition dataset given `rank` by using the `use` function in DataPartitioner.
4. Wrap the dataset with `torch.utils.data.DataLoader`, remember to customize the `collate_fn` from our customized function defined in `project/utils.py`.

2. Implement dataset class `class Partition()` in `data_parallel/dataset.py`:

```python
class Partition():
    def __init__(self, data, index):
        ...

    def __len__(self):
        ...
```

```
    def __getitem__(self, index):
        ...
```

and partition class `class DataPartitioner()` in `data_parallel/dataset.py`:

```
class DataPartitioner():
    def __init__(self, data, sizes=[0.7, 0.2, 0.1], seed=1234):
        ...

    def use(self, partition):
        ...
```

Hint:

1. Create a list of indices for every data point and use `rng` to shuffle the indices. The indices should be integers. A simple way can be a list like this $[0, 1, ..., len(data) - 1]$.
2. Create different partitions of indices according to `sizes` and store in `self.partitions`. Let's suppose that we have 8 data points, and 2 partitions for the data. One way of `self.partitions` can be $[[0, 3, 6, 7], [1, 2, 4, 5]]$.

To summarize, the `partition_dataset` function is to help create a training dataset partition you need for a single device within a cluster of devices. The `Partition` class is used to define a dataset class to return the data according to partitioned indices. The `DataPartitioner` class is used to partition any datasets according to different workload defined as `sizes`.

Pass the test

```
python -m pytest -l -v -k "a4_1_1"
```

We simply test whether you distribute the data into different partitions without overlapping.

Problem 1.2

1. Implement function `setup` for data parallel training in `project/run_data_parallel.py`:

```
def setup(rank, world_size, backend):
    '''Setup Process Group'''
    ...
```

and the code in `main` section.

```
if __name__ == '__main__':
    ...
    processes = []
```

```
    '''Create Process to start distributed training'''
```

This part is to help you understand how to setup the process group to manage the distributed work on different devices. We create processes to complete the training work individually.

Hint:

1. For the setup function, please set the environment variables `MASTER_ADDR` as `localhost` or `127.0.0.1` and `MASTER_PORT` as `11868`. Then use `init_process_group` function in `torch.distributed` to init the process group
2. In the `main` section, you can use `Process` from `torch.distributed` to define the process
3. We define the number of processes as `world_size`
4. You should create processes, start the processes to work and terminate resources properly

2. Implement communication function `average_gradients` to aggregate gradients:

```
def average_gradients(model):
    '''Aggregate the gradients from different GPUs'''
    ...
```

Every device only trains on a portion of the data, but we still need the global gradients. This function is important for gradient communication and aggregation. You need to walk through the parameters of the model and call function in `torch.distributed` to aggregate the gradients.

After implementing the function `average_gradients`, please call the function after backward propagation in the `train` function in `project/utils.py`.

To pass the test here, you should first run the following command to store the gradients of one batch. Please set the `world_size` as 2.

```
python project/run_data_parallel.py --pytest True --n_epochs 1
```

Then check whether there are weight files `model{rank}_gradients.pth` in your `tests` directory. Run the following command to compare the gradients on different devices.

```
python -m pytest -l -v -k "a4_1_2"
```

We test whether you successfully accumulate the gradients from all the devices and broadcast the reduced gradients across all the devices.

Problem 1.3

Compare the performance between training on single device and on multiple devices. To evaluate performance, we encourage you to drop the results from the first epoch or at least have one warmup run before collecting the metrics.

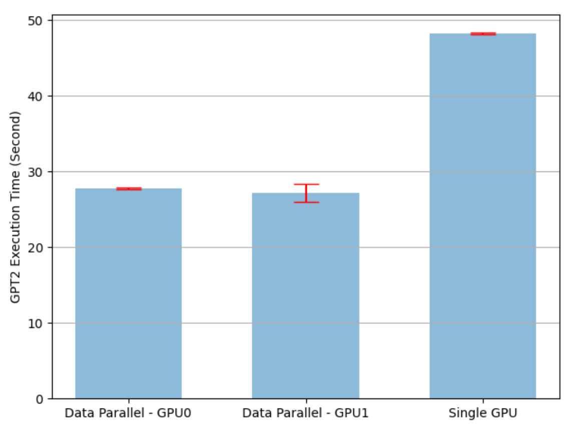
```
# single node
python project/run_data_parallel.py --world_size 1 --batch_size 64

# double nodes
python project/run_data_parallel.py --world_size 2 --batch_size 128
```

We use two metrics to measure performance `training time` and `tokens_per_second`. We provide the code to print out the average `training time` and average `tokens_per_second` over several epochs and also store them in the json files.

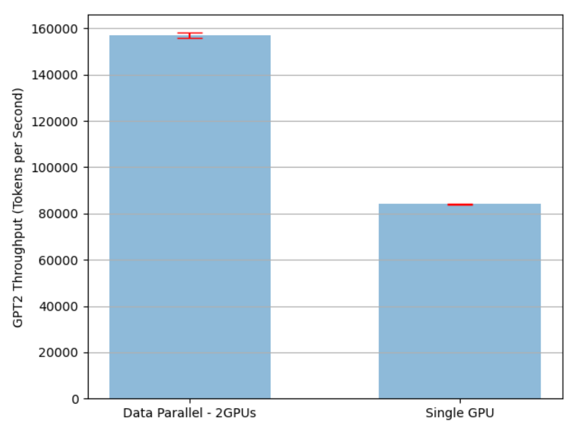
We **average** the `training time` over `epochs` from multiple devices and compare them with the `training time` with a single device. That is to say, if you have two devices, you calculate the training time separately for different devices.

For to, we **sum up** the `tokens_per_second` from multiple devices as the throughput and compare the throughput with a single device. The throughput should also be **an average number over epochs** because we usually repeat the experiments to avoid outliers when collecting metrics.

To visualize the scaling improvement, please plot **2 figures** of the 2 performance metrics separately. A figure plot helper file can be found in `project/plot.py`. A sample figure is shown in Figure 2. Please save the figures in the directory `submit_figures`.



(a) Training time



(b) Tokens per second

Figure 2: Data Parallel Performance Sample Figure

You are encouraged to increase the size of dataset or increase the devices to explore the scalability.

Problem 2: Pipeline Parallel (50)

In this part, you are going to implement pipeline parallel for GPT2. You should only use packages already imported in the codebase.

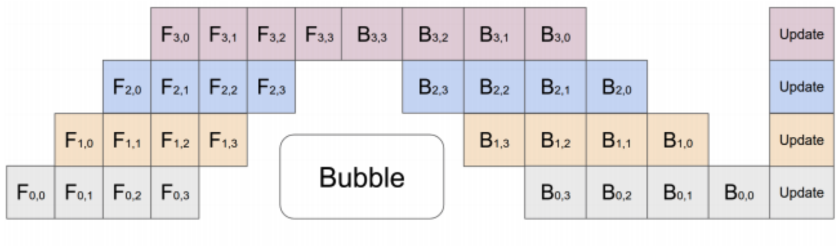


Figure 3: Pipeline Parallel

There are three sub-modules for this problem:

Problem 2.1

Implement the helper function `_split_module` in `pipeline/partition.py`. We are doing layer-wise split here: `_split_module` takes in an nn.Sequential module, and splits the module into multiple partitions. Each partition resides in a different GPU (each colored row in Figure 4).

```
def _split_module(
    modules: nn.Sequential
) -> Tuple[List[nn.Sequential], List[torch.device]]:
    ...
```

Implement the helper function `_clock_cycles` in `pipeline/pipe.py`: This produces a schedule for each timestep, based on the number of minibatches and the number of partitions. This corresponds to each vertical step in Figure 4.

```
def _clock_cycles(
    num_batches: int,
    num_partitions: int
) -> Iterable[List[Tuple[int, int]]]:
    ...
```

Pass the tests:

```
python -m pytest -l -v -k "a4_2_1"
```

Problem 2.2

Understand the code in `worker.py` and implement `Pipe.forward` and `Pipe.compute` in `pipeline/pipe.py`.

The `Pipe` module is a generic wrapper over any `nn.Sequential` modules to convert them into a pipelined module. `Pipe` moves the input through the pipeline, by splitting the input into multiple microbatches (each item in Figure 4), and computing the microbatches in parallel.

```
class Pipe(nn.Module):
    def forward(self, x):
        ...

    def compute(self,
        batches,
        schedule: List[Tuple[int, int]]
    ) -> None:
        ...
```

Hint: In `create_workers`, we spawn a worker thread per device. Each worker takes tasks from `in_queue` and puts the results in `out_queue`.

To send a task to a worker, you need to (1) wrap the computation in a function and create a `Task` object, (2) put the task in the corresponding `in_queue` for the device, and (3) retrieve the results from `out_queue`.

Pass the tests:

```
python -m pytest -l -v -k "a4_2_2"
```

Problem 2.3

Implement `_prepare_pipeline_parallel` in `pipeline/model_parallel.py`.

This part prepares a GPT-2 model for pipeline parallelism. Note that different blocks in GPT-2 are already moved to different GPUs in `GPT2ModelCustom.parallelize`. This function extracts the transformer blocks (stored in `self.h`) in the model and packages them into an `nn.Sequential` module for `Pipe`.

```
class GPT2ModelParallel(GPT2ModelCustom):
    def _prepare_pipeline_parallel(self, split_size=1):
        ...
```

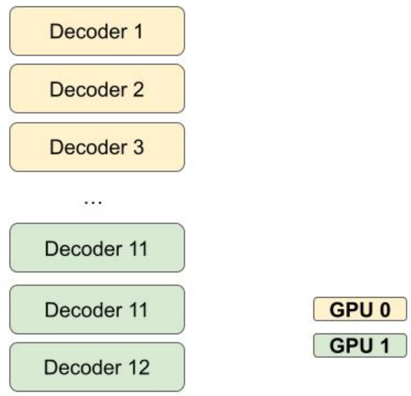
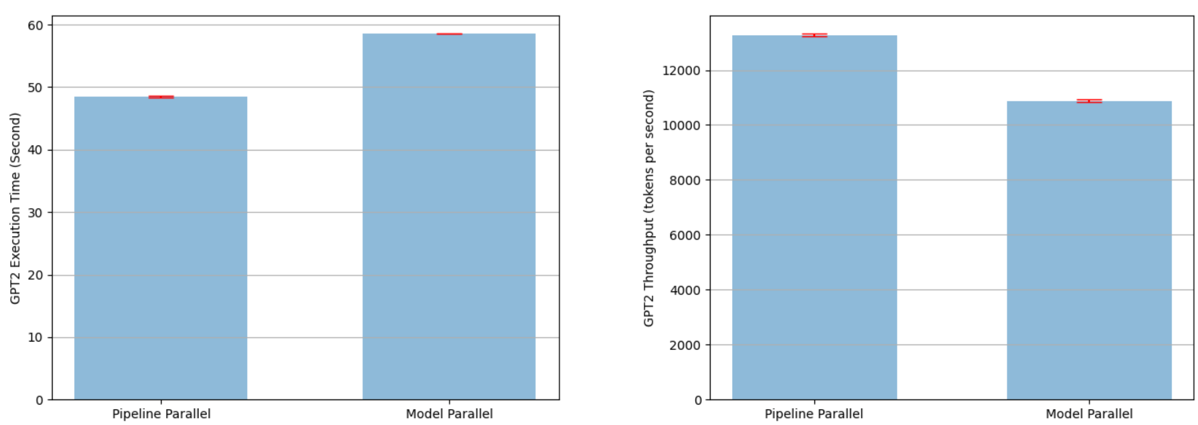


Figure 4: Example 2-GPU Partitioning for GPT-2 (12 decoder layers). Note that we are doing layer-wise splits only for the decoder blocks.

Train the GPT2 model on two GPUs. Observe and compute speedups from pipelining.

```
python project/run_pipeline.py --model_parallel_mode='model_parallel'
python project/run_pipeline.py --model_parallel_mode='pipeline_parallel'
```

To visualize the scaling improvement, please plot **2 figures** of the 2 performance metrics separately. A sample figure is shown in Figure 5. Please save the figures in the directory `submit_figures`.



(a) Training time

(b) Tokens per second

Figure 5: Pipeline Parallel Performance Sample Figure

Submission

Please submit the whole `llmsys_s24_hw4` as a zip on canvas. We will inspect your codes manually to make sure that you follow the implementation restrictions on specific packages. We will also compile and run your codes to make sure they are runnable and check your figures of the performance metrics.