CS744 - Big Data Systems

Assignment 2

Group 26

Akash Navani,
Jonas Weile
Rahul Uday Chakwate

October 12, 2022

1 Overview

To get a deeper understanding of how a machine learning model is trained in a distributed environment, and how the training time and model accuracy is impacted by this, we train VGG-11 on the CIFAR-10 dataset on both a single node and in a number of distributed environments. To do this, we use the PyTorch machine learning framework to write the application and with Gloo as the communication framework. We then compare training time and model accuracy for the different approaches.

1.1 Setup

We have a simple 4-node cluster available to run the experiments. To set up the cluster for the experiments, we follow the instructions on the Course Assignment Page[1]. The setup consist of installing Miniconda, numpy and PyTorch for CPU on all four of the nodes, and then the cluster is ready to run the experiments.

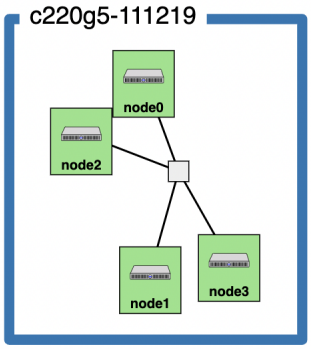

Figure 1: Cluster used for Distributed Data Parallel Training

2 Experiments

We train a VGG-11 model on CIFAR-10 dataset. Our training data comprises of 50,000 image samples. One epoch of training with a batch size of 256 gives us a total of 196 iterations. First, we train the network for one epoch on a single node and obtain a baseline for the model, see section 2.1. Next, in section 2.2, we explore the idea of Distributed Data Parallel Training. The training data is partitioned amongst the four nodes with a set seed to ensure consistent partitioning for all the runs. In the two sub-parts of this task, we first implement gradient aggregation using `gather()` and `scatter()` calls available in the Gloo backend framework, and then use the `all_reduce()` collective in Gloo to perform the same aggregation. Lastly, we use the built-in `DistributedDataParallel` module to perform the gradient synchronization, see section 2.3.

2.1 Task 1

As part of task 1, we were asked to complete the given base script for the training setup. The only missing part was to fill in the training loop, i.e. *forward pass, backward pass, loss computation* and *optimizer step*. The completed training loop is given by the code snippet in listing 1. Then, we simply run the scripts on node 0 in order to train the model on a single node.

2.2 Task 2

For task 2, we wish to *manually* distribute the training of the model over the 4 nodes in the cluster. For this assignment, we focus on **distributed data parallel training**. That is, each node will have it's own copy of the same model, and then the training data will be distributed among the nodes to perform training in parallel. To distribute the data among the nodes, we use the `DistributedSampler` module. Then, to guarantee that the distributed models are identical at all times, for all the nodes, we need the following two invariants:

```python
def train_model(model, train_loader, optimizer, criterion, epoch):
    ...

    # Training loop
    for batch_idx, (data, target) in enumerate(train_loader):
        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

        # Print statistics
        ...
```

Listing 1: The completed training loop as part of task 1.

1. All model parameters are initialized identically across all nodes.
2. All updates to model parameters are identical for all nodes.

To satisfy the first of these invariants, we make sure that all nodes use the same random seeds in the training setup; this will ensure that model weights are initialized to identical values. To satisfy the latter of the invariants, we must synchronize the gradients among all nodes after each backward pass - thus we must modify the training loop. The new training loop is shown in listing 2. Notice, that the only change that was made to the modified training loop compared to listing 1 is the addition of the call to `sync_gradients()` as highlighted in line 13.
In subsections 2.2.1 and 2.2.2, we synchronize the gradients using `gather() / scatter()` and `all_reduce()`, respectively, both using the *Gloo* communication framework.

```python
def train_model(model, train_loader, optimizer, criterion, epoch):
    ...

    # Training loop
    for batch_idx, (data, target) in enumerate(train_loader):
        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        sync_gradients(model, rank, world_size)
        optimizer.step()

        # Print statistics
        ...
```

Listing 2: The modified training loop as part of task 2.

2.2.1 Task 2A

The first way of performing the synchronization/aggregation of gradients is by using `gather()` and `scatter()` calls. We arbitrarily choose the worker with rank 0 to orchestrate. Thus, the worker with rank 0 gathers all gradients, computes the mean, and then scatters copies of this mean tensor to all workers. The resulting `sync_gradients()` method is shown in listing 3.

```
def sync_gradients(model, rank, world_size):
    ...
    for param in model.parameters():

        if rank == 0:
            gradients = [torch.zeros_like(param.grad.data) for _ in range(world_size)]

            # Gather and scatter
            dist.gather(param.grad.data, gradients, dst=0)
            mean = torch.mean(torch.stack(gradients), dim=0)
            gradients = [mean for _ in range(world_size)]
            dist.scatter(param.grad.data, gradients, src=0)

        else:
            # Gather and scatter
            dist.gather(param.grad.data, dst=0)
            dist.scatter(param.grad.data, src=0)
```

Listing 3: The `sync_gradients()` method using `gather()` and `scatter()`.

2.2.2 Task 2B

For task 2B, we use the builtin `all_reduce` collective to perform gradient aggregation using ring reduce, instead of gather and scatter. This reduction collective is simpler than manually gathering and scattering values, and the resulting `sync_gradients()` method, as shown in listing 4, is therefore significantly shorter than listing 3.

```
def sync_gradients(model, world_size):
    ...
    for param in model.parameters():
        dist.all_reduce(param.grad.data, op=dist.ReduceOp.SUM)
        param.grad.data /= world_size
```

.

Listing 4: The `sync_gradients()` method using `all_reduce()`

2.3 Task 3

For task 3, we let PyTorch do all the hard work for us. We use the `DistributedDataParallel` module to perform the distributed data parallel training across the 4 nodes, with an absolute minimum of changes to the code from task 1. In fact, the training loop is identical to the one in listing 1. All we have to do is to distribute the data as described for task 2, and then initialize the `DDP` container by calling `DistributedDataParallel()` with our model as input, using the default bucket size of 25MB. Then, we use this new container instead of our model throughout the code.

3 Results and Discussion

We run each task for 1 epoch, using a total batch size of 256 in each experiment, and in case of a distributed setup, we use all 4 nodes. Further, we time the first 40 iterations for each task (that is, the first 40 mini-batches of data), discard the timings of the first iteration, and then report the average time per iteration for the remaining 39 iterations for each task. For all distributed training experiments, we then average then time for each worker as well. The average time per iteration for each task is then shown in table 1, along with the test loss and test accuracy for each model as measured after the full epoch of training.

Task Number	Avg. Time per Iteration (s)	Loss	Accuracy (%)	Speedup over single machine
1	1.855	2.179	16.33	1
2a	1.304	2.107	18.18	1.4
2b	0.932	2.120	18.20	2.0
3	0.664	2.088	18.75	2.8

Table 1: This table presents the data that was collected from each of the experiments. The average time per iteration is based on the running time of iteration 1 through 40, whereas the loss and accuracy are computed for the full epoch. For tasks with multiple machines, the average time is further averaged over all the nodes.

3.1 Speedup and Scalability

As one can read from table 1, there are clear improvements in the average time per iteration as we progress through the different parts of the assignment. All of the distributed setups improve the average time per iteration compared to the single machine setup, but the differences among the different distributed setups is also vast. The first point demonstrates that distributed data parallel training can speedup the training of neural networks even in its' most simple and naive form, and therefore shows how it can be advantageous for systems to distribute work among different machines. However, the latter point emphasizes that one can achieve much greater speedups with more complex distributed setups. We discuss the reason for the speedup in each task in the following subsections.

3.1.1 Gather and Scatter

The simplest distributed data parallel setup using `gather()` and `scatter` calls to synchronize gradients achieves a speedup of 1.4 compared to the single machine case. While this is definitely a clear improvement, one would intuitively expect a greater speedup considering we are using 4 machines instead of 1, as this greatly reduces the amount of computation each worker is responsible for. For a batch size of 256, each worker now trains a network with a mini-batch size of just 64 samples; a quarter of the workload of the single-node setup. However, the distributed setup adds severe communication overhead to the training, due to the need to aggregate gradients among all nodes

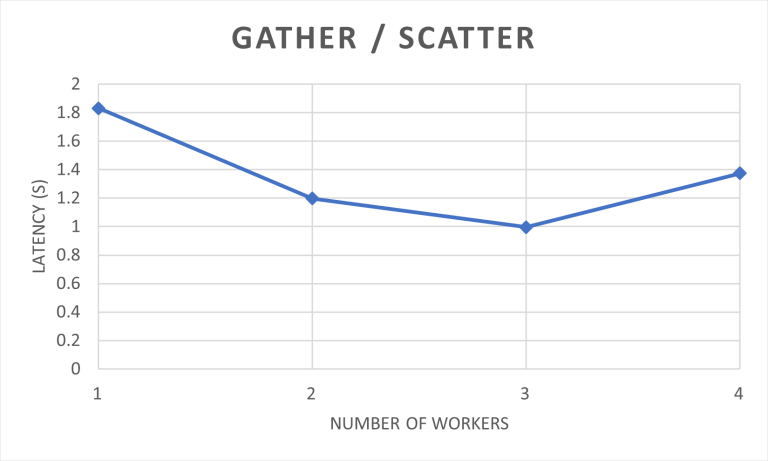

Figure 2: Average Time per Iteration (s) vs number of nodes using the `gather` and `scatter` collectives.

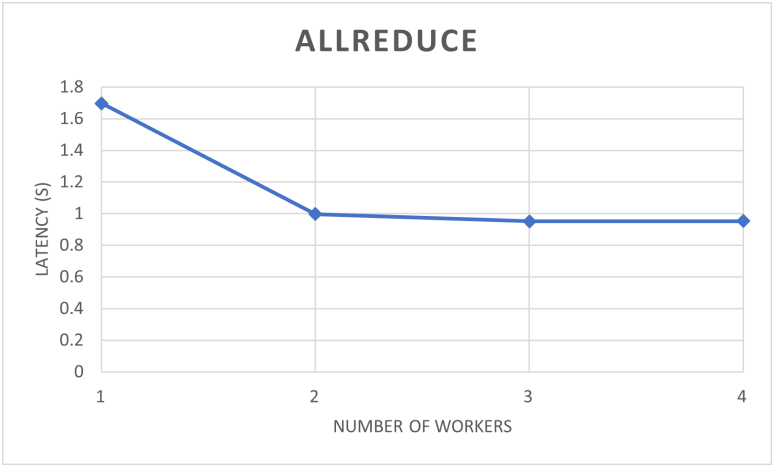

Figure 3: Average Time per Iteration (s) vs number of nodes using the `allreduce` collective.

after each backward pass. As the gather and scatter calls both goes through the master process (node 0 in our case), the master process can quickly become a performance bottleneck as the number of workers increase. This is because the master has to both communicate with all processes as well as perform the entire aggregation; and as the number of workers increase, so does the cost for communication and aggregation for the master. Figure 2 shows the latency plotted against the number of workers. It is clearly seen, just as we have argued in this section, that there is a clear performance bottleneck, and the latency increases as the number of workers surpasses 3. In fact, the per iteration latency for 4 workers is higher than for both 2 and 3 workers. Thus, the gather and scatter approach is far from scalable.

3.1.2 All_reduce

By switching to the `all_reduce` collective for aggregating the gradients, we observe a speedup of 2.0 compared to 1.4 for `gather / scatter`. Thus, simply by changing the communication algorithm, we see a much greater speedup. This is because `all_reduce` performs a `Ring-Reduce`. Compared to the simple gather and scatter approach, ring-reduce distributes computation and communication evenly over all workers instead of having the master doing all the work, thereby eliminating the performance bottleneck that we recognized for task 2A. Now, because of the topology of the ring, each worker will always communicate with exactly two other workers, and thus this reduction approach should be more scalable. However, the communication does still add some overhead as the computation must halt whenever the workers are performing the reduction, and thus the speedup is still not quite as good as one might hope to achieve, considering we are using 4 workers. Further, looking at figure 3, we see that the per iteration latency actually stops improving once we have three workers. Thus, although more scalable than the gather and scatter approach, the per iteration latency stagnates pretty quickly.

3.1.3 DDP

Finally, using the built-in module `DistributedDataParallel` module from PyTorch, we see a speedup of 2.8 compared to the single machine setup, a speedup of 2 compared to the gather and scatter approach, and a speedup of 1.5 compared to using allreduce. The DDP module achieves this speedup by overlapping communication with computation, thereby reducing the impact of the communication overhead incurred by aggregating gradients among the workers. This is achieved by *gradient bucketing*[2]. When looking at figure 4, we see that this setup is the only one for which the per iteration latency decreases for every increment in the number of workers. From the figure it is also clear, however, that the largest improvement is seen when increasing the number of workers from 1 to 2; from this point on the slope is not nearly as steep. Thus, this approach is resulting in the, by far, best speedup, and at the same time seems to be the most scalable of the 3 approaches.
It would also have been interesting to consider how the speedup might have changed if we had considered different bucketsizes, and further how skipping gradient synchronizations could have impacted the scalability.

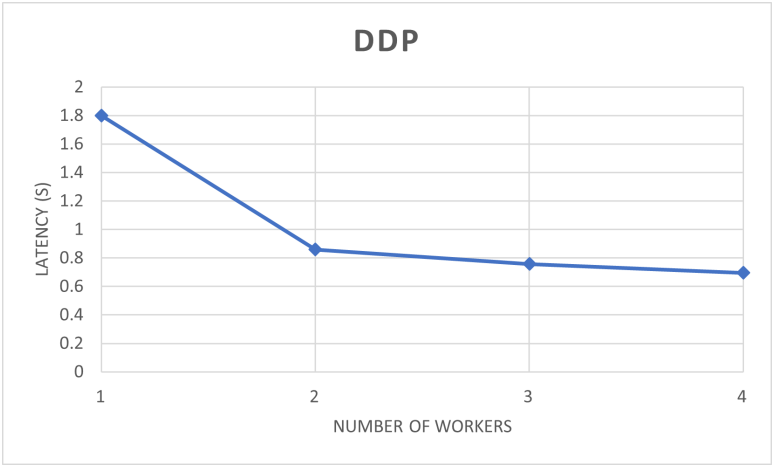

Figure 4: Average Time per Iteration (s) vs number of nodes using `DistributedDataParallel` module.

3.2 Test Loss and Accuracy

We see that the loss and accuracy of the model after 1 epoch does not have significant differences. This, again, is in-line with our expectations. One expects to have a distributed setup which does not alter the performance of the network and gets you consistent results. The minor differences, however, stem from the underlying implementation of the network. We were able to identify one such reason while investigating the possible causes for this discrepancy, namely the `BatchNormalization` layer in the model. The BatchNorm computes the running_mean and running_var independently on each node. Since this depends on the input data, it has differently values which brings in some inconsistency in the model performance. We disabled the running_mean and running_var which started giving us better results. In DDP, PyTorch just broadcasts these values from node-0 and disregards the running_mean and running_var from the other nodes. This is the reason we did not see different loss and accuracy values on different nodes in Part 3 but did so in Part 2. The values above for Part 2 were taken without track_running_stats. Further, we are also cognizant of the randomness that creeps in due the floating point computations in network training and evaluation.

4 Contributions

4.1 Akash Navani

Akash was in charge of writing and structuring the report. He added the logic for recording times for each task and was responsible for compiling all our findings into a report. He led the discussions for reasoning the observations recorded and presented them with the experimental evidence gathered.

4.2 Jonas Weile

Jonas was instrumental in writing the implementation of gradient synchronization in the second task of the assignment. In terms of the final submission, Jonas structured all the code in the respective folders and made it fit for submission. He also wrote large parts of the report explaining some key implementation details, as well as the discussion on speedup and scalability.

4.3 Rahul Chakwate

Rahul spearheaded the complete setup of the nodes. He was the designated Machine Learning expert on the team and guided the other team members with respect to the different ways the tasks could be achieved. Rahul wrote the logic for the forward and backward pass which was used in all the tasks of the assignment, and also helped with the ddp implementation in third task.

References

[1] *CS744 assignment 2*. URL: `https://pages.cs.wisc.edu/~shivaram/cs744-fa22/assignment2.html`.

[2] Shen Li et al. *PyTorch Distributed: Experiences on Accelerating Data Parallel Training*. arXiv:2006.15704 [cs]. June 2020. URL: `http://arxiv.org/abs/2006.15704` (visited on 10/12/2022).