CS 744 - Distributed ML

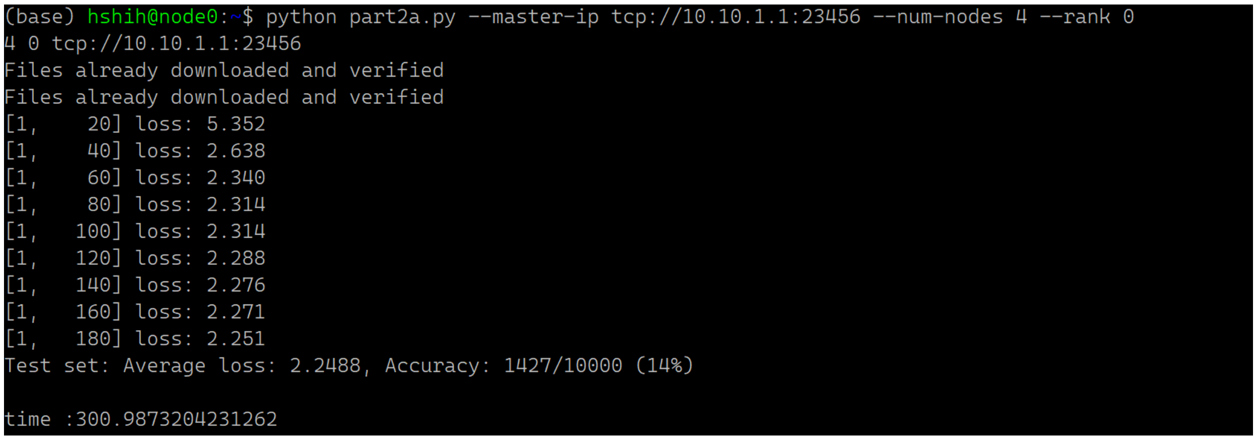
Winfred Li, Noah Cohen Kalafut, Hao-Yu Shih

Part 1

For part one, training was run over 40 mini batches of total size 256. Over three separate runs, the average per iteration time was 1.88 seconds, not including the first iteration. After training, the average loss was 2.1 with a test accuracy of 19%.

Part 2

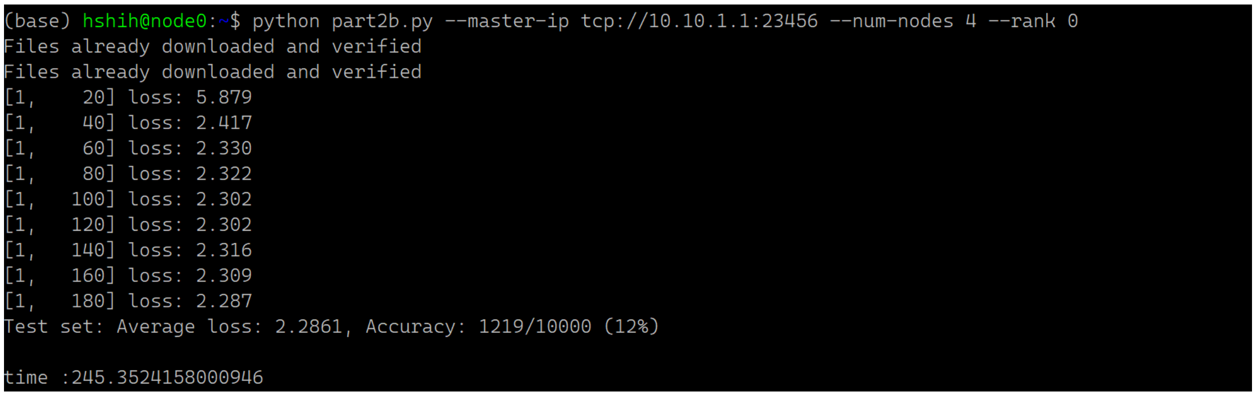
The biggest difference between part 2 and part 1 is that in part 2, we used Pytorch's DistributedDataParallel. In this part, we have to partition the data into four workers without overlapping and to set up the random seed for each worker. In addition, part 2a used gather and scatter and part 2b used all-reduce to train. To make the training distributed to four workers, we use distribued_sampler and torch.manual_seed calls to implement.



```
(base) hshih@node0:~$ python part2a.py --master-ip tcp://10.10.1.1:23456 --num-nodes 4 --rank 0
4 0 tcp://10.10.1.1:23456
Files already downloaded and verified
Files already downloaded and verified
[1,    20] loss: 5.352
[1,    40] loss: 2.638
[1,    60] loss: 2.340
[1,    80] loss: 2.314
[1,   100] loss: 2.314
[1,   120] loss: 2.288
[1,   140] loss: 2.276
[1,   160] loss: 2.271
[1,   180] loss: 2.251
Test set: Average loss: 2.2488, Accuracy: 1427/10000 (14%)

time :300.9873204231262
```

In part 2a, we used gather/scatter call and gloo as the backend to implement the training. The average iteration time was 1.53 seconds, and average loss was 2.2488 with accuracy of 14%.



```
(base) hshih@node0:~$ python part2b.py --master-ip tcp://10.10.1.1:23456 --num-nodes 4 --rank 0
Files already downloaded and verified
Files already downloaded and verified
[1,    20] loss: 5.879
[1,    40] loss: 2.417
[1,    60] loss: 2.330
[1,    80] loss: 2.322
[1,   100] loss: 2.302
[1,   120] loss: 2.302
[1,   140] loss: 2.316
[1,   160] loss: 2.309
[1,   180] loss: 2.287
Test set: Average loss: 2.2861, Accuracy: 1219/10000 (12%)

time :245.3524158000946
```

In part 2b, we used all-reduce call and gloo as the backend to implement the training. The average iteration time was 1.25 seconds, and average loss was 2.2861 with accuracy of 12 %.

Part 3

For part three, training was implemented using PyTorch's DistributedDataParallel. Training was run for 40 mini batches using the DistributedSampler with a batch size of 64. The average iteration time was 0.82 seconds over three runs, and the average loss was 2.0 with an accuracy of 23%.

Comparison

Part	**1**	**2a**	**2b**	**3**
Method	**Single Node**	**Gather/Scatter**	**Ring Reduce**	**DDP**
Batch Time* Three Each	1.918 1.956 1.769	1.316 1.352 1.315	1.061 1.064 1.029	0.838 0.804 0.821
Average	1.881	1.328	1.051	0.821
Time Pct	100%	70.6%	55.9%	43.6%
*Average batch time taken over first 40 batches with the initial batch removed				

The accuracy (~20%) and loss (~2.0) of all three solutions were comparable once training was completed. This is to be expected of parts 1, 2a, and 2b, as their training methodologies are equivalent, just distributed among varying numbers of nodes in different ways. Part 3 also had similar accuracy/loss, however, which is a testament to the DDP method.

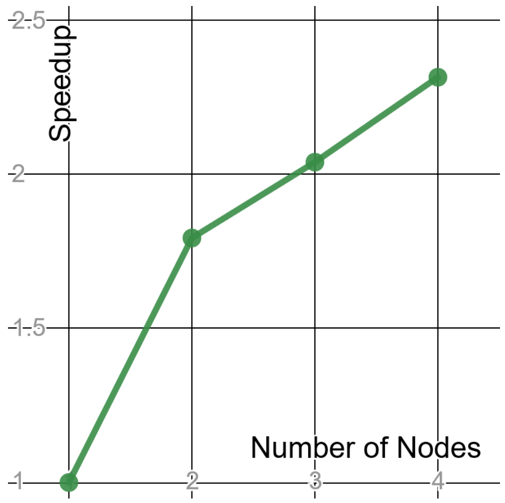
As expected, the results get faster the more integrated the solution. In terms of speed, part 1 serves as a baseline to show that distributed calculation is, in fact, faster than computation on only one node. However, even in the optimal case (part 1 vs 3), the performance benefit of 4x the nodes is 56%, when the expectation would be 75%. What gives? The overhead of communication and gradient aggregation plus syncing introduces additional work and bottlenecks not present in a single-node solution. The worst-case scenario time-wise also happens more frequently, given that, for solutions such as Gather/Scatter or Ring Reduce, all nodes have to finish computing their gradients before sending them off for further aggregation.

For comparison between multi-node solutions, we can first look at the difference in computation time between 2a and 2b (21% reduction). Gather/Scatter directs a lot of network traffic to one node. This can result in congestion and a non-negligible increase in computation time. In contrast, Ring Reduce (AllReduce) spreads out network usage by having each node send gradients only to its neighbor, spreading out the network and aggregation overheads.

DDP has the most significant time reduction of all methods, taking only 44% of the time per batch of the single-node method. The primary difference between DDP and other methods is that DDP parallelizes communication and computation. Rather than waiting for a backpropagation step from other nodes, nodes will continue to run forward propagations until receiving a weight update. The gradients calculated will be applied to stored copies of previous model weights. Then, the weights are averaged and the next model iteration is calculated. This approach allows more batches to be run faster with similar model performance.

Scalability

Distributed Data Parallel on Multiple Nodes				
Nodes	**1**	**2**	**3**	**4**
Batch Time* Three Each	1.972 1.804 1.930	1.085 1.012 1.084	0.924 0.922 0.949	0.838 0.804 0.821
Average	1.902	1.060	0.932	0.821
Time Pct	100%	55.7%	49.0%	43.2%
*Average batch time taken over first 40 batches with the initial batch removed				



Due to the increased overhead, it appears that returns begin to diminish when multiple nodes are involved. There appears to be a linear speedup when more nodes are added beyond 2. However, the speedup is not as significant as the initial shift from 1 to 2 nodes, which almost results in perfect efficiency (a 44.3% reduction in time per batch). This trend of diminishing returns would likely continue as overhead increases and network limitations are exceeded. However, with only 4 nodes, it is difficult to come to a solid conclusion.

Contributions

Winfred set up the CloudLab instance. Noah set up the GitHub and documentation. Each member individually completed parts 1-3 of the assignment before splitting up the writing responsibilities. Hao-Yu wrote on parts 2a and 2b and wrote their code submissions. Winfred wrote on parts 1 and 3. Noah wrote on and ran data for the comparison between methods and scalability and wrote the code submissions for parts 1 and 3.