CS744 - Big Data Systems
Assignment 2

Akshata Bhat
akshatabhat@cs.wisc.edu

Kyle Klassy
klassy@cs.wisc.edu

Rohit Kumar Sharma
rsharma@cs.wisc.edu

October 11, 2019

1 Goals

- Deploy and configure TensorFlow in a distributed setting with 3 VMs.
- Explore TensorFlow performance on a cluster.
- Write TensorFlow applications using Keras in single and distributed modes.

2 Environment

The environment for this assignment consisted of a 3-node cluster with Ubuntu 16 installed on each machine. TensorFlow is an end-to-end open source platform for machine learning used in part 1 to build a logistic regression model. Keras is a high-level API for TensorFlow used in part 2 of this assignment to implement LeNet.

3 Logistic Regression

In part 1 of the assignment, a simple logistic regression machine learning model was implemented in TensorFlow and trained it on the MNIST handwriting digits dataset. After building a simple TensorFlow application which runs on a single node, TensorFlow's replicated training ability was then used to do between-graph replication in order to run the logistic regression application in distributed mode.

3.1 Single Node Mode

To first create our logistic regression model, it was necessary to choose a loss function to minimize over the training epochs. The following equation for cross-entropy was provided to us in the assignment, which was then provided as the loss function to the gradient descent optimizer:

$$L(D_{tr}) = \sum_{y,x \in D_{tr}} -y \log softmax(w^T x)$$

A standard hyperparameter value of 0.01 was chosen for the learning rate, which defines the model's step size in weight space. In single node mode, there is no setup necessary for running the code on the cluster– running the TensorFlow application on node0 of the experiment cluster requires no communication or work from other nodes. The TensorFlow computational graph corresponding to our model is shown in Figure 1.

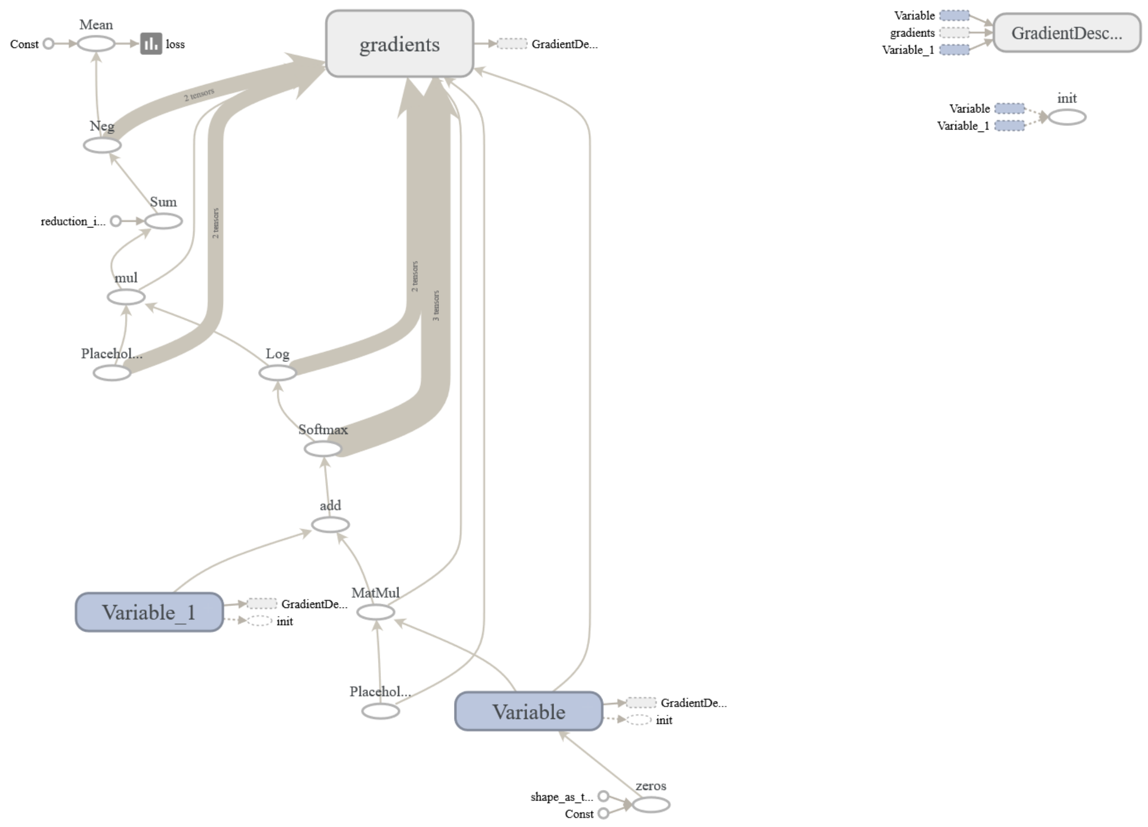


Figure 1: TensorFlow data flow graph

After 10 iterations and a batch size of 75, the cross entropy loss was down to around 0.33, which was determined to be when the model converged after running the model for more training epochs. Since these parameters were suitable for convergence, we reused these same values for the asynchronous and synchronous distributed training tasks in sections 3.2 and 3.3 to keep our comparisons accurate. After training, the model was evaluated on a set of unseen test data which resulted in a classification accuracy of 90.13%. Measurements of the cross entropy loss were taken after each training batch, and the resulting plot in Figure 2 was produced, showing convergence of the model.

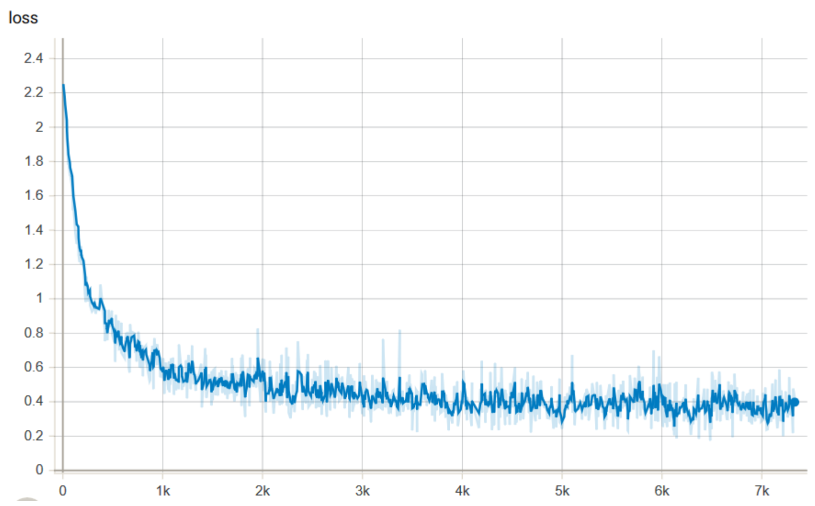


Figure 2: Single node logistic regression loss over number of training samples

3.2 Distributed Mode with Asynchronous SGD

Using distributed TensorFlow, we were able to run our logistic regression application on multiple worker machines in the cluster. The setup of nodes and processes made it such that node0 had the parameter server process, and node0, node1, and node2 all had worker tasks. We first trained the model in asynchronous mode using two workers (excluding node2 as a worker process) and then trained the model on all three workers. Running the distributed application results in the TensorFlow computational graph being distributed across nodes in the cluster.

In implementing asynchronous training, between-graph replication was used, where each client builds a similar graph containing the model parameters and also has a single copy of the compute-intensive part of the model. The workers each perform computations independently and push their gradient updates to the parameter server, which get applied to the shared state. Since asynchronous training means that no worker device has to wait for any other, the workers can immediately pull the new shared state in the next iteration and begin computation.

3.3 Distributed Mode with Synchronous SGD

Similar to asynchronous SGD, we were able to run our logistic regression application on the cluster first with two workers, and then later using three workers. For the synchronous SGD running in distributed mode, all of the workers read the same values for the shared model state, compute their gradients in parallel, and then all updates get aggregated together before being applied to the shared state. This is done with the tf.train.SyncReplicasOptimizer(), which is a TensorFlow Class specifically used to avoid stale gradients by collecting gradients from all workers, averaging them, then applying them to the shared state in a single shot. It is only after the gradients from all workers get applied to the shared state that the workers can then get the new variables and continue computation.

The chart in Figure 3 shows the difference in training times across worker nodes when using asynchronous SGD and synchronous SGD with 3 worker nodes. The nodes in synchronous mode took slightly longer to finish training, which was expected due to the individual worker nodes having to wait for each other to finish individually before pulling the shared state variables again and performing computation. Additionally, the chart shows that the asynchronous nodes all took around the same time to finish their specified number of training epochs, which was mostly likely due to our VMs having very similar resources. If this same experiment were re-run on machines with different specifications, we would expect the times for asynchronous mode to vary much more.

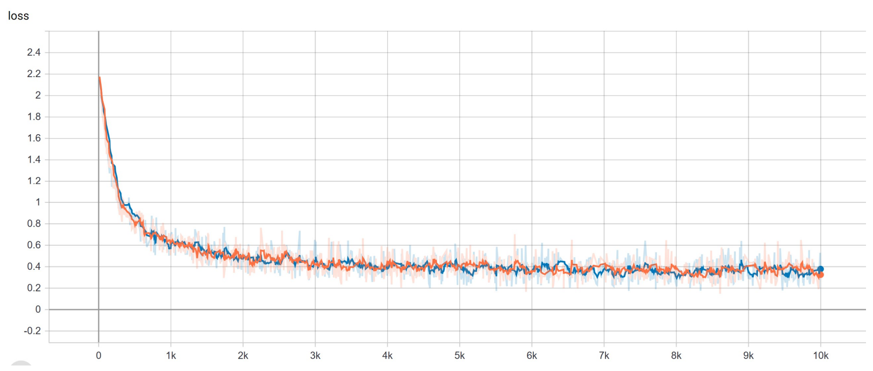


Figure 4: Comparison of synchronous and asynchronous convergence. Blue line is for asynchronous replication and orange is for synchronous. The convergence in both synchronous and asynchronous replication is very close and it is hard to differentiate between them.

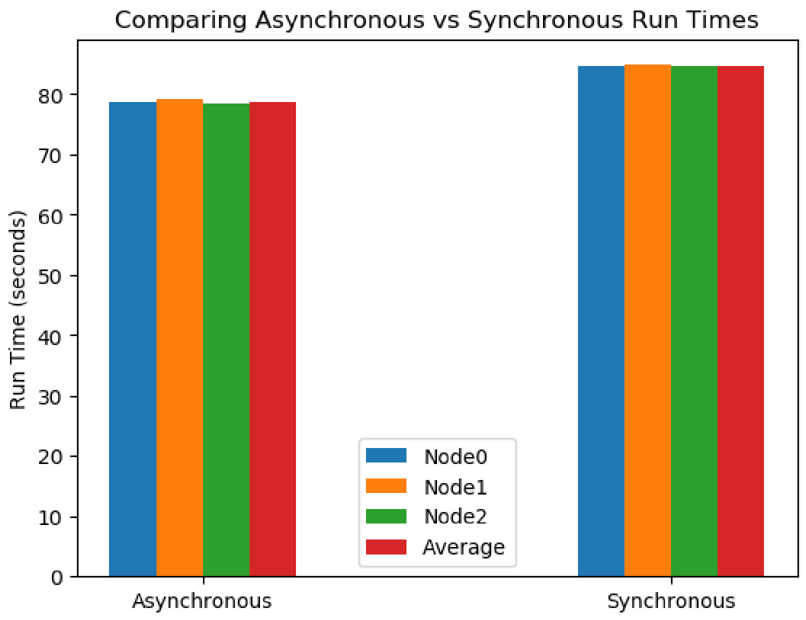


Figure 3: Comparison of synchronous and asynchronous training times. Both the experiments were run on 3 nodes for 10000 steps.

Finally, in comparing the convergence of our logistic regression model between the synchronous and asynchronous training with three worker nodes, Figure 4 shows very similar performance of loss vs number of training samples. Before running the experiment, we would have expected the synchronous SGD to converge with fewer training samples due to not operating on stale model parameters, but empirically there is no difference between the two distributed methods. This could be due to the nature of the MNIST dataset or the variance in selecting the initial weight values when performing logistic regression.

Table 1 shows the CPU, memory, and network usage for two of the worker nodes (node0 and node1) and compares the usages between asynchronous and synchronous execution. Since Node2 was running the same processes as node1, we've excluded it to avoid redundant data. Before running this experiment, we expected node0 to have more resources used than the other nodes because the parameter server process was running on node0. Additionally, we expected that the synchronous SGD to consume more memory than the asynchronous because of the overhead of creating queues to store worker gradients. From these values in Table 1, we can see that the peak memory on node0 in synchronous mode is significantly more than in asynchronous mode, supporting our claim that the gradient queues would increase memory usage. Additionally, the CPU and network usage in asynchronous mode is more than in synchronous mode, which is logical because of the worker

	Asynchronous	**Synchronous**
Average CPU usage on node 0 (%)	42	38
Average CPU usage on node 1 (%)	33	30
Peak Memory on node 0 (MB)	634	772
Peak Memory on node 1 (MB)	456	466
Average Network usage on node 0 (KB received)	3315	3082
Average Network usage on node 0 (KB sent)	6251	5700
Average Network usage on node 1 (KB received)	3173	2930
Average Network usage on node 1 (KB sent)	1698	1560

Table 1: Comparison of synchronous and asynchronous resource usages. The experiments were run on 3 nodes for 10000 steps. Results for node 2 are omitted as they are similar to node 1 for brevity. The CPU usage on node 0 is higher than node 1 for both Async and Sync. This is because node 0 runs both PS and worker task. Similarly, the Memory usage and Network usage for node 0 is more than node 1. The CPU usage for async is higher than sync. This is because sync needs to wait to synchronize updates in each step. The network usage is slightly more in async compared to sync because of similar reason.

nodes not having to wait for other worker nodes to finish, which can therefore increase the number of calculations per second that can be performed.

4 LeNet

4.1 Implementing LeNet using Keras

In part 2 of the assignment, the Keras API for TensorFlow was utilized to implement LeNet, a convolutional neural network with a 7-layer architecture. Once again, the network was trained on the MNIST dataset. Using the distributed API for Keras, the LeNet model was able to be trained on a single node, as well as clusters of 2 and 3 workers using synchronous SGD, described in section 3.3. To implement distributed tensorflow using Keras, we used tf.distribute.MirroredStrategy(), which performs in-graph replication on all shared variables. A batch size of 64 and 25 epochs were chosen as training parameters. Figure 5 shows the loss of LeNet on sets of one, two, and three worker nodes and the number of epochs taken to reach convergence. Somewhat counter-intuitively, LeNet seems to converge more quickly when only running on a single machine, and takes longer to converge when running in distributed mode. This may be because for a fixed batch size, in each epoch the amount of data being processed in each step is three times compared to the one in single node mode. As a result, it is more robust to the noise in the data. Table 2 shows how the resource usage of node0 varies when running on a single machine as well as how node1 consumes resources when LeNet is running in distributed mode.

	Single Node	**2 Node Cluster**	**3 Node Cluster**
Average CPU usage (%)	60	58, 54	57, 56
Peak Memory usage (MB)	1098	1052, 710	1034, 680
Average Network usage (KB received)	21	4800, 4600	5826, 5828
Average Network usage (KB sent)	18	4600, 4550	5802, 5830

Table 2: LeNet with 64 batch size resource usage comparison with various cluster sizes. The values reported for 2 node and 3 node clusters are for node0 and node1 respectively. For 3 node cluster, node2 values were omitted as they were similar to node1 for brevity.

	Batch Size											
	32		64		128		256		512		1024	
	Node 0	Node 1	Node 0	Node 1	Node 0	Node 1	Node 0	Node 1	Node 0	Node 1	Node 0	Node 1
Average CPU usage(%)	47	47	56	56	63	63	72	72	79	79	83	83
Peak Memory usage (MB)	885	675	895	685	1004	769	1066	785	1195	894	1203	948

Table 3: Comparison of average CPU usage and peak memory usage as the batch size increases from 32 to 1024. The experiments were run on 3 nodes with a fixed number of epoch=25. For each batch, we observe that peak memory usage is higher in Node 0 when compared to Node 1. This is because Node 0 also performs gradient aggregations in each step. Average CPU and peak memory usage increases as the batch size increases as expected, because it has to process more data in each step.

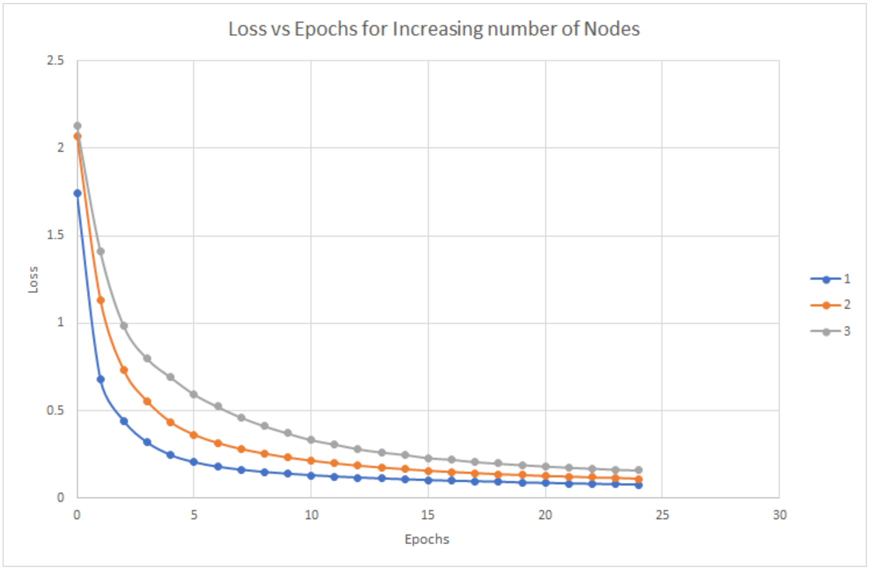


Figure 5: Comparison of convergence of LeNet on single node, and distributed setting with a cluster of two and three nodes for a fixed number of epochs. The experiments were carried out with a batch size of 64 epochs=25.

4.2 Comparing Training Batch Sizes

The final test performed on our LeNet implementation was modifying the batch size to see how the training accuracy was affected. Keeping the number of epochs constant at 25, we expected that our model would be more accurate with a smaller batch size since the model would converge after fewer epochs. Figure 6 shows the convergence over time for various batch sizes, confirming our hypothesis that because of taking smaller steps in weight space when using a small batch size, the steps would be more fine-grained with a smaller batch, resulting in quicker convergence.

Figure 7 shows how the same batch sizes affected test-set accuracy and shows that our model was indeed more accurate with a smaller batch size. However, if using an even smaller batch size in our experiments, we would expect our model to potentially overfit to its training data, resulting in a worse test set accuracy.

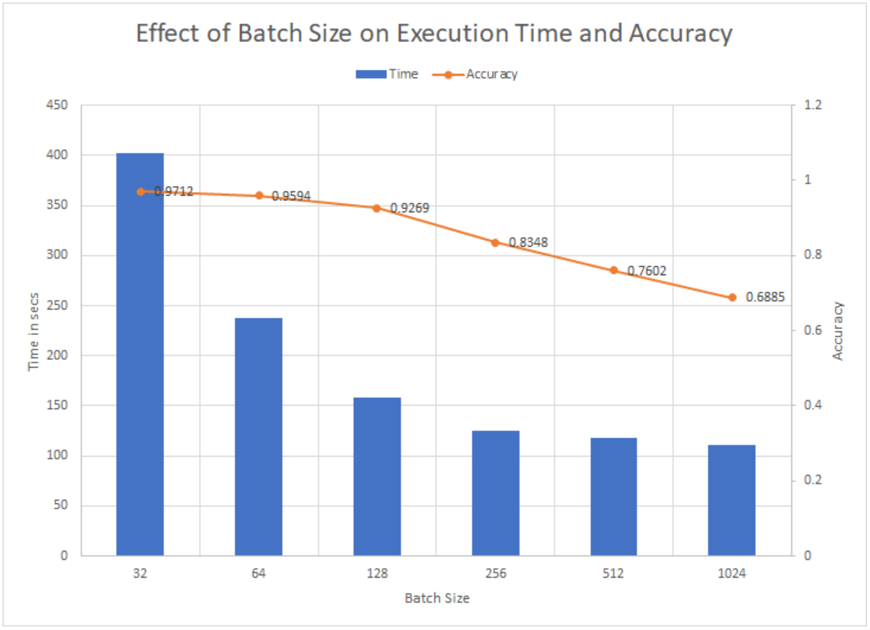


Figure 7: Comparison of time taken and accuracy achieved for various batch sizes for a fixed number of epochs of 25

Additionally, converging after fewer samples also means that because the gradient needs to be calculated after every batch, the real run time taken for the model to converge is longer, which is also shown in figure 7.

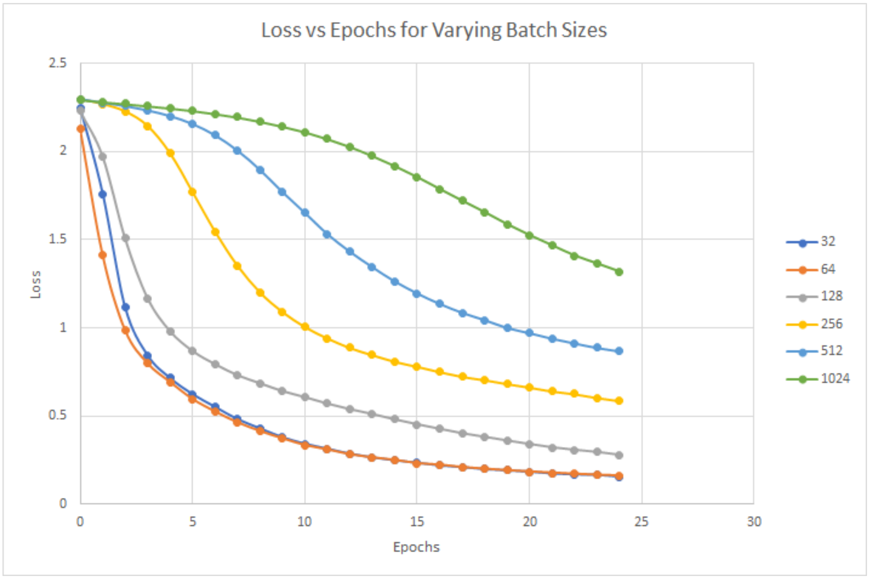


Figure 6: Comparison of convergence of LeNet for various batch sizes. The experiments were run on a cluster of 3 nodes.

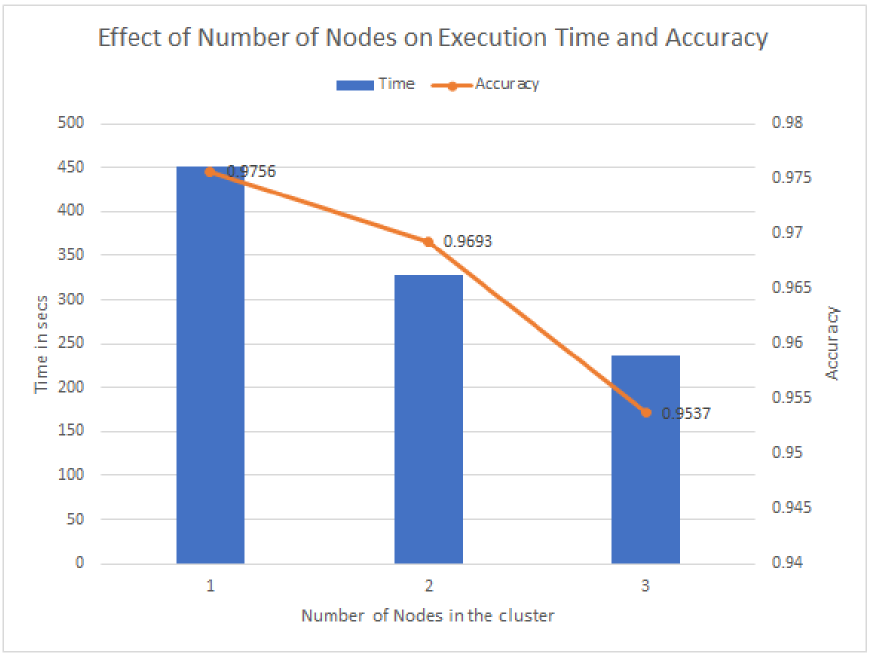
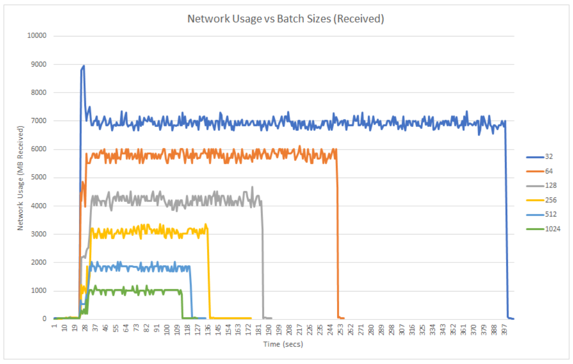
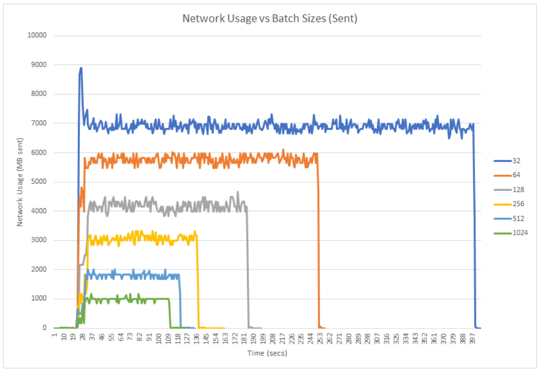


Figure 8: Comparison of time taken and accuracy achieved when run on various cluster sizes for a fixed number of epochs=25. The experiments were run on a cluster of 3 nodes.



(a) Received in MB



(b) Sent in MB

Figure 9: Comparison of Network usage as the batch size increases from 32 to 1024 when run with 25 epochs in a cluster of three nodes. The values shown are for node 0 and similar results were observed for other nodes with network usage for node 0 being more than other nodes. As the batch size increases, it takes less time to complete the process. The network usage is almost consistent across each run. The amount of data sent is almost the same as received as expected. As the batch size increases, the number of gradient updates that happen in each iteration decrease. Therefore, the overall network usage reduces as the batch size increases as expected.