Shawn Zhong, Suyan Qu, Sulong Zhou

CS 744: Big Data System

Introduction

First, we write simple Tensorflow applications, and run them on the cluster for the following purposes:

- Compare workloads running on a single machine with running in distributed systems.
- Compare Synchronous SGD with Asynchronous SGD
- Compare different APIs such as Tensorflow Core and Keras.

Second, test the performance of usage of CPU/MEM/Network for these applications

Dataset

The MNIST database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that consists of 60,000 training images and 10,000 testing images. Each image is normalized to 28x28 pixel in grayscale levels.

Part I: Logistic Regression

We Implement a simple *Logistic Regression* application with Tensorflow 1.14. Logistic Regression (LR) is a statistical model for classification, and we implement LR based on the given formula:

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

In this LR implementation, random initialization is used, epoch number is set to 50, and GradientDescentOptimizer is set with a learning rate of 0.01. The methodology comprises two tasks: First, we implement LR and train the model in a single machine with original Tensorflow APIs. Second, we implement LR in clusters using Sync-SGD and Async-SGD, and evaluate the performance and test error for both of them while monitoring the CPU/Memory/Network usage and using dstat to figure out the bottleneck.

Task 1: Single worker

We train the model with 50 epochs and batch size of 50 on a single node, and visualize the results with TensorBoard.

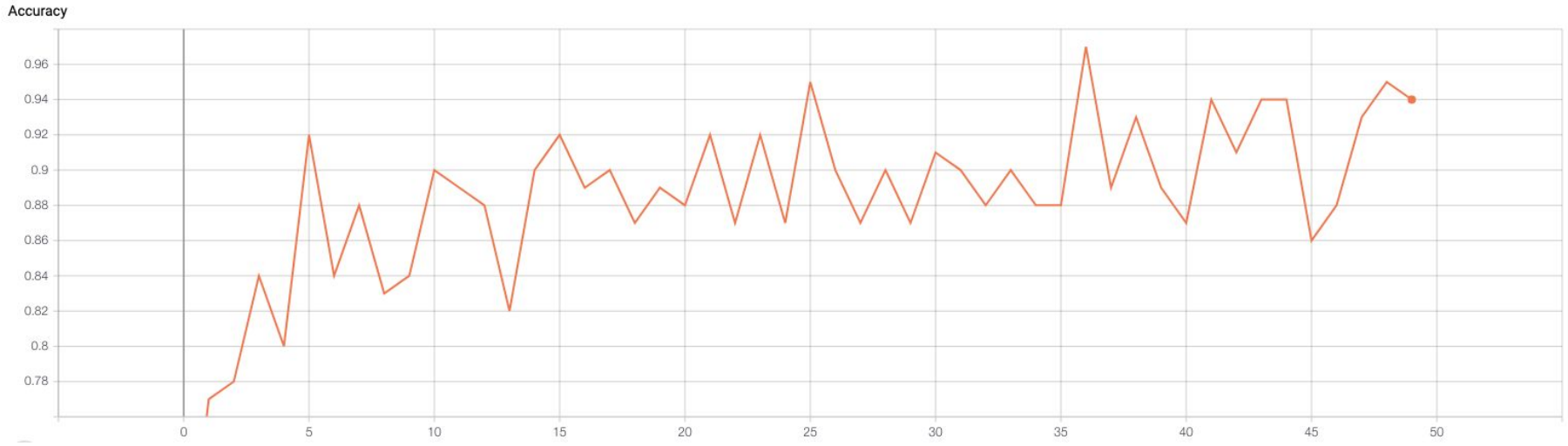


Figure 1.1.1. Single node mode validation accuracy across time in TensorBoard

(Final accuracy of 94% at 1min 22s. Maximum accuracy of 97% at 1min 0s)

Figure 1.1.1 demonstrates how the accuracy of our training model fluctuates over time, and the time is denoted by how many epochs that have been completed. As a whole, the accuracy twists but tends to constantly increase, reaches the maximum (97%) at the time when the 36th epoch is completed, and finally ends up with 94%. In total, the running time is 1 minute and 22 seconds.

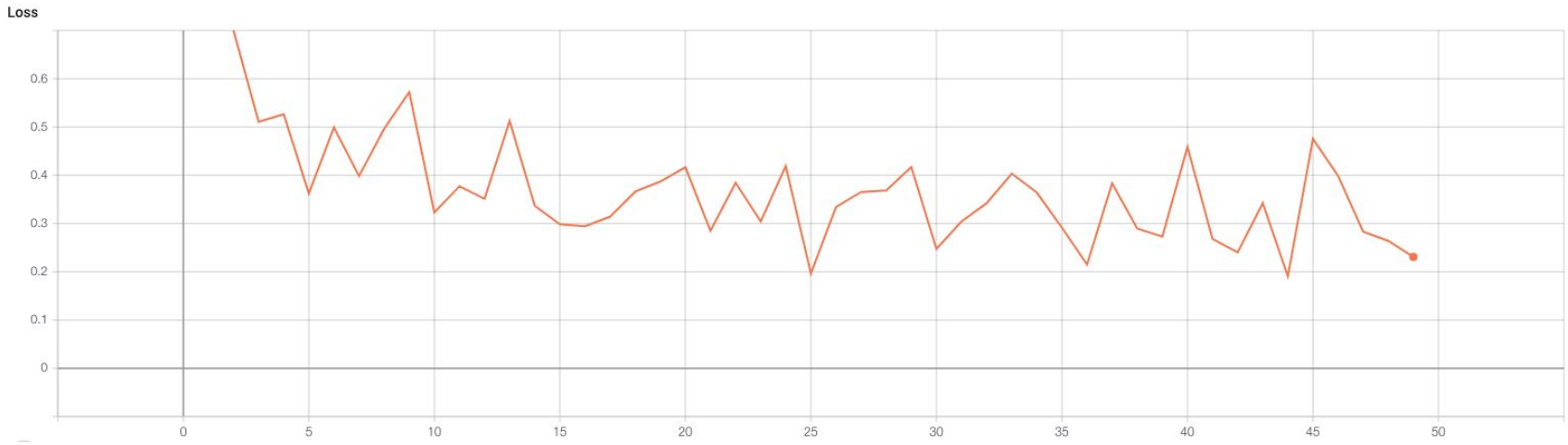


Figure 1.1.2: Single node mode loss across time in TensorBoard
(Final loss of 0.2308 at 1min 22s. Minimum loss of 0.1909 at 1min 13s)

Figure 1.1.2 demonstrates how the loss of our training model fluctuates over time. As a whole, the loss twists but tends to constantly decrease, reaches the minimum value (0.2308) at the time when the 44th epoch is completed, and finally ends up with 0.1909. In total, the running time is 1 minute and 13 seconds.

Figure 1.1.3: Resources used during training on a single node.

Figure 1.1.3 shows the resources used for training on a single machine, recorded on Ubuntu desktop. We notice that the CPU utilization rate is around 50%~60%, in contrast with 20%~25% utilization rate before and after training. However, we do not see significant changes in network usage. This is mainly because we are using only one machine for training, and everything happens on the same node, so no data transmission is required. We also did not notice any huge change in memory, which maybe because of the small size of the linear regression model (7850 parameters) and input data size (1×28×28 per input image) relative to total memory available (31.4 GB).

Task 2: Synchronous and Asynchronous Distributed SGD

We conducted this experiment with 3 workers and 50 batch size, and using asynchronous SGD and synchronous SGD respectively.

Async

Asynchronous Stochastic Gradient Descent (Async-SGD) means that each worker communicates with the parameter servers independently of the others in three steps. First, the most up-to-date parameters are fetched. Second, the gradients of the loss with respect to these parameters are computed. Third, these gradients are sent back to the parameter servers, which then updates the model accordingly.

Sync

Synchronous Stochastic Gradient Descent(Sync-SGD), where the parameter servers wait for all workers to send their gradients, aggregate them, and send the updated parameters to all workers afterward, making sure that the actual algorithm is a true mini-batch stochastic gradient descent, where the actual batch size is the sum of all the mini-batch sizes of the workers.

SGD Mode		Average User CPU Usage	Average Memory Usage (in Bytes)	Average Network Receive (in Bytes)	Average Network Send (in Bytes)	Final Validation Loss	Final Validation Accuracy	Time
Async	1st node	30.25	2,485,566,063	47,216	53,654	0.1309	96%	2m 12s
	2nd node	21.52	814,936,915	29,598	167	0.4010	86%	2m 9s
	3rd node	20.78	799,894,613	29,815	227	0.2003	96%	2m 17s
Sync	1st node	39.33	2,996,801,310	39,543,803	39,574,471	0.1721	96%	8m 46s
	2nd node	11.96	1,252,024,517	19,770,832	19,775,391			
	3rd node	12.36	1,230,581,440	19,794,942	19,801,780			

Table 1.2.1: Usage of CPU/Memory/Network, Loss/Accuracy/Convergence Time

Evaluation

Sync-SGD converges slower but the rate is very close to compared to Async-SGD. Sync takes 8 minutes and 46 seconds to achieve an accuracy of 96% while Async takes around 2 minutes to achieve a maximum accuracy of 96%. In practice, Async-SGD means that while a worker computes gradients of the loss with respect to its parameters on a given mini-batch, other workers also interact with the parameter servers and thus potentially update its parameters; hence it reduces the overhead that multiple workers are waiting for each other in Sync-SGD. Moreover, the actual update time in Sync-SGD depends on the slowest worker because we don't have backup workers.

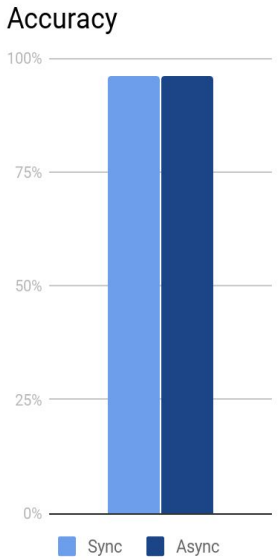
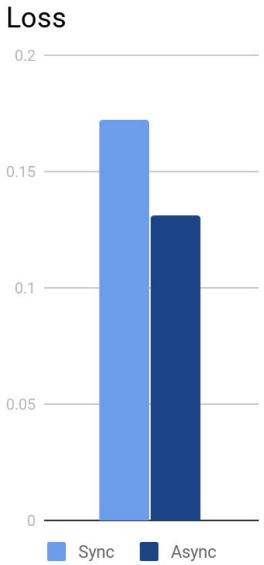
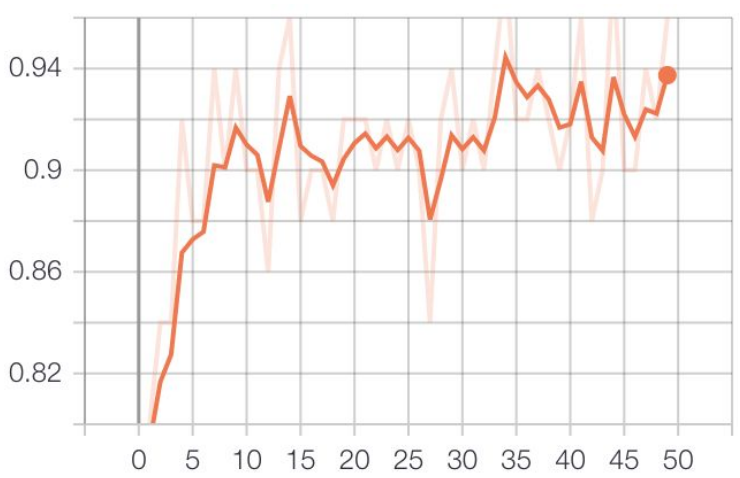
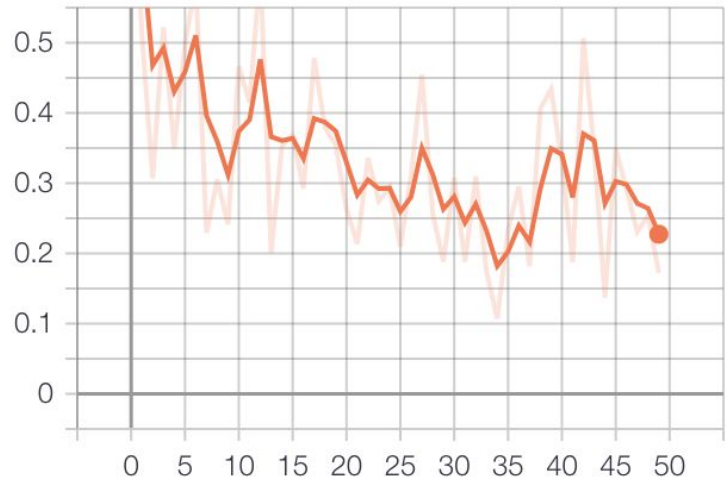


Table 1.2.1 records the resource usage and convergence time when training with synchronous and asynchronous gradient descent. Since our programming development and communication with the whole cluster are exclusively through the 1st node, it has significantly larger resource usage compared with other 2 nodes in all cases, we therefore focus on the 2nd and 3rd node for analysis. We notice that asynchronous training actually achieves higher CPU utilization rate compared with synchronous training and therefore takes a shorter time to converge. This matches the fact that in synchronous training, each worker should wait for all other workers to finish the current batch before continuing with the next batch. Meanwhile, since synchronous training will send the gradient to the master node for aggregation and then fetch the updated parameters, it takes much more network read and write during training. In contrast, we did not see any improvement in final loss and accuracy when using synchronous training because the model has already converge in 50 epochs. We also notice that synchronous training takes more memory than asynchronous training, which may be due to the cached information sent from other nodes.



Compared with single node execution, both asynchronous training and synchronous training uses longer training time. In asynchronous training, we did not notice a clear bottleneck. The training time it takes is also not significantly larger than single node situation, indicating that this may have been some extra calculation in code implementation, such as passing code and data to workers for training at the beginning. However, in synchronous training, the training time is much longer compared with single node situation. The significant difference in network usage indicates that bandwidth may have been a bottleneck for synchronous training. Another possible bottleneck is the stragglers, the nodes that takes longer to

proceed compared, since all other nodes need to wait for them in synchronous training. This is a possible cause, especially since we communicate with the entire cluster through node 0, which may take a lot network.

SGD Mode	Accuracy	Loss
Synchronous		

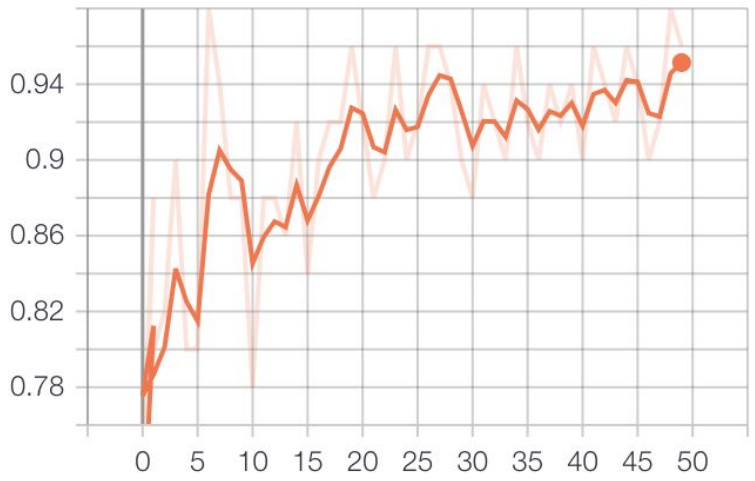
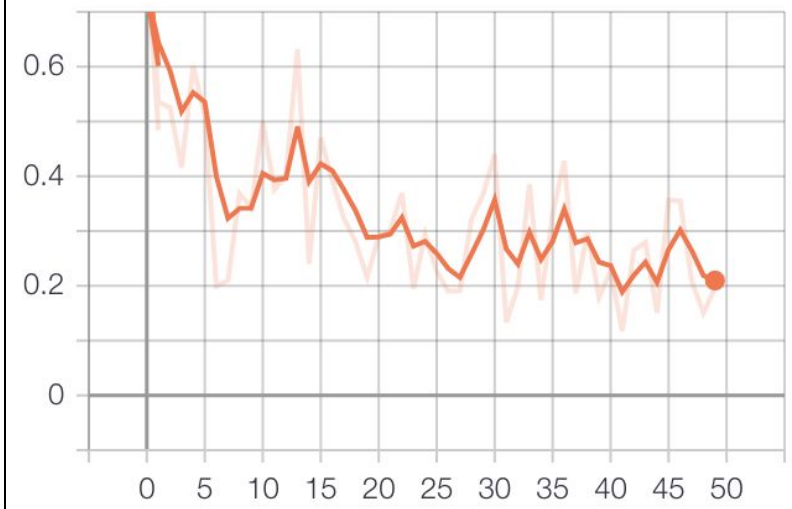
Asynchronous		

Table 1.2.2: Accuracy and Loss During Training

Table 1.2.2 shows the change in accuracy and loss during training. We notice that although synchronous training does take less epochs to converge, the difference is very slight and almost negligible.

Execution Graph

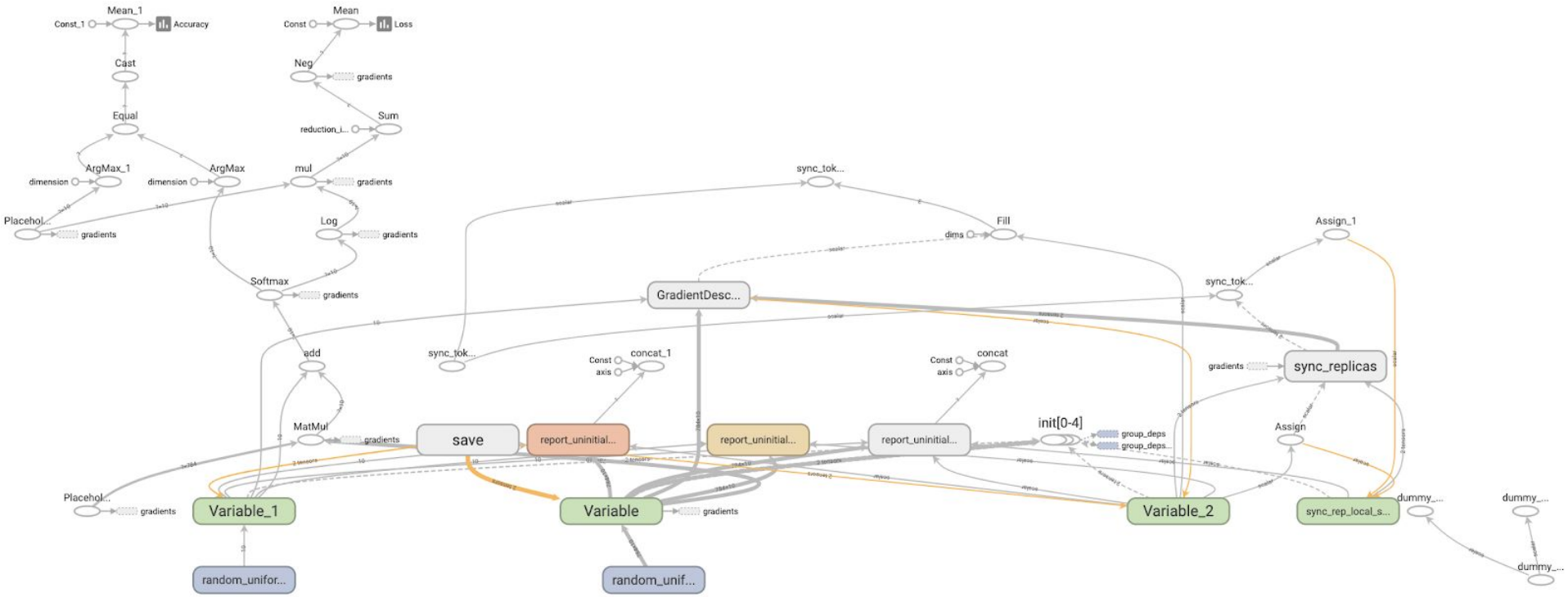


Figure 1.2.1 Execution Graph for Sync-SGD Logistic Regression

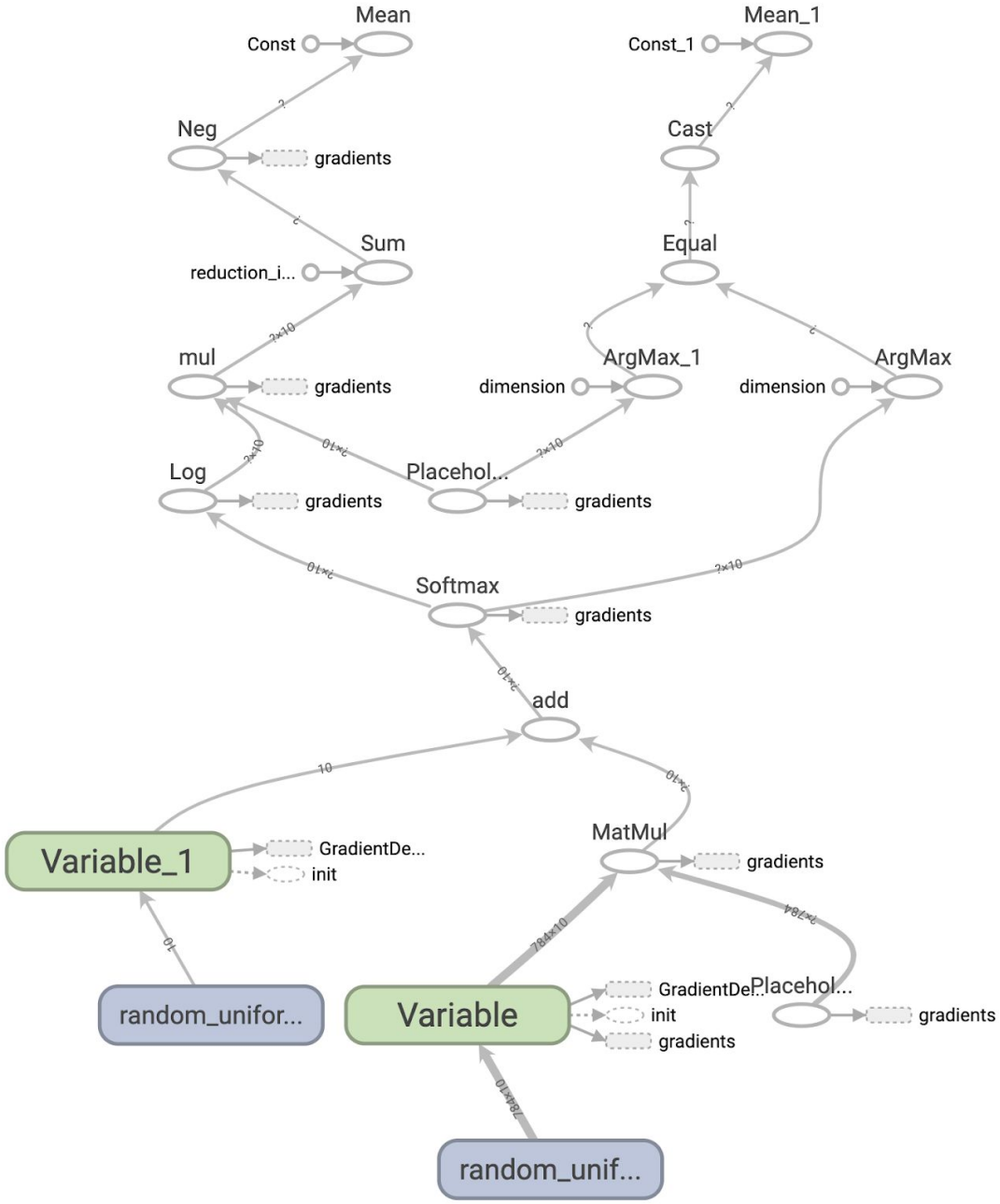


Figure 1.2.2 Execution Graph for Async-SGD Logistic Regression

Part II: LeNet

LeNet is one of the first convolutional neural networks. Its architecture consists of 7 layers including 3 convolutional layers, 2 sub-sampling layers, 1 fully connected layer, and 1 output layer. First, we implement a simple LeNet application with Keras API based on Tensorflow 2.0. We use cross-entropy loss and stochastic gradient descent with the learning rate set to 0.001. Second, we train the MNIST for 30 epochs and experiment with different modes ranging from a single node to three nodes, and different batch sizes of 50, 100, 200 respectively. Third, we present the summary of loss and accuracy using Tensorboard and monitor the usage of CPU/MEM/Network using dstat for each combination of mode and batch size.

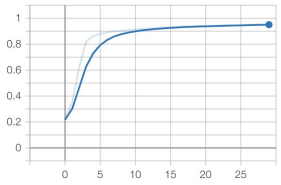
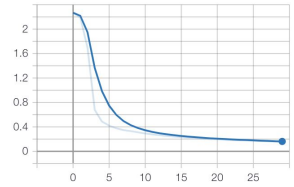
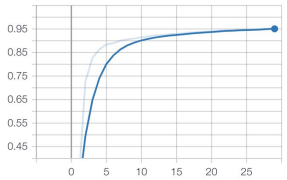
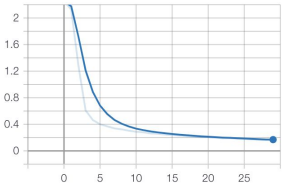
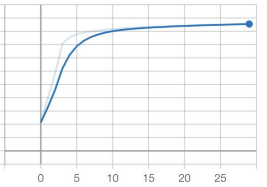
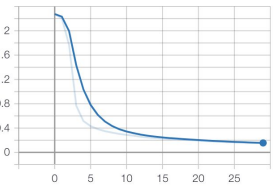
Number of workers	Batch Size	Average User CPU Usage	Average Memory Usage (in Bytes)	Average Network Receive (in Bytes)	Average Network Send (in Bytes)	Final Validation Loss	Final Validation Accuracy	Time
1 (Single Node)	50	70.62	2,886,289,387	56,494	38,754	0.1563	95.18%	9m 48s
	100	71.38	2,879,134,472	57,997	23,030	0.1674	95.27%	8m 32s
	200	70.77	2,875,044,159	52,452	24,094	0.1469	95.71%	8m 32s
2 (Cluster Mode)	50	47.98	3,318,783,850	9,691,353	9,738,355	0.1491	95.48%	19m 29s
	100	51.76	3,628,522,197	7,211,978	7,218,599	0.2406	92.93%	9m 7s
	200	50.46	3,407,835,171	4,320,017	4,306,106	0.3467	89.97%	7m 44s
3 (Cluster Mode)	50	44.95	3,072,682,977	11,666,126	11,655,754	0.1636	95.06%	18m 33s
	100	47.66	3,734,689,071	8,036,325	8,038,431	0.2529	92.49%	10m 55s
	200	46.01	3,343,109,245	5,071,573	5,077,728	0.3558	98.79%	9m 0s
Idle	-	13.95	2,285,073,944	39,454	7,969	-	-	-

Table 2.1.1 Usage of CPU/Memory/Network, Loss/Accuracy/Convergence Time

Table 2.1.1 records the resource usage during training, and the final loss/accuracy and convergence time. We notice that there is not a significant difference in model validation loss and accuracy when experimenting with different number of workers and batch sizes, indicating that our model converges within 30 epoches.

We notice that, however, working with different number of nodes, CPU and memory usage drop significantly when we move from working with a single worker to working with multiple workers, and that average network read and write increases significantly as the number of workers increases. This is because when working with synchronous training, the workload is split between multiple nodes, in which case the calculation performed by a single worker would decrease, and the resources needed for a worker would thereby decrease significantly. However, to split the work among multiple workers in synchronous training, we need more network resources to transmit data and the calculated gradient among workers, which is why the network read and write increases as the number of workers increases.

For the same reason, the time a worker takes to complete its split of work is much smaller when working with multiple nodes, yet passing data across workers creates a significant overhead after completing each batch. When working with smaller batch sizes, the overhead caused by data transmission is much greater than the improvement in calculation time, so the training time is actually shorter for single worker when batch size of 50 is used. In contrast, when working with large batch sizes, the overhead is actually less than the improvement in calculation time, so the training time is smaller for more number of workers when batch size of 200 is used instead.

# wk	Batch Size = 50		Batch Size = 100		Batch Size = 200	
	Validation Accuracy	Validation Loss	Validation Accuracy	Validation Loss	Validation Accuracy	Validation Loss
1						

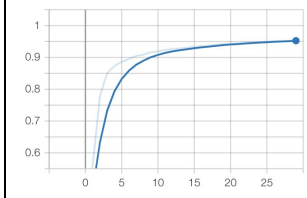
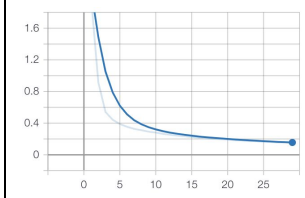
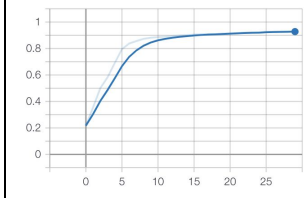
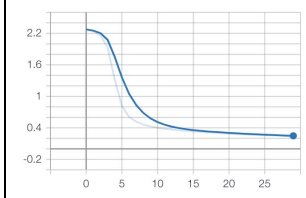
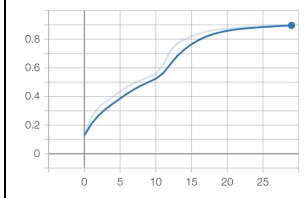
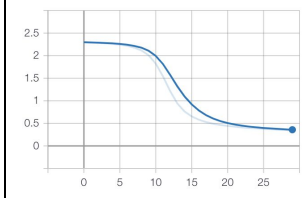
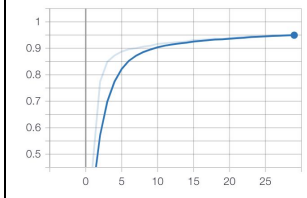
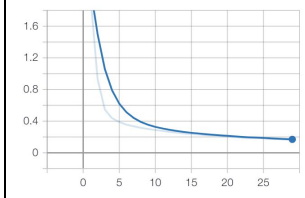
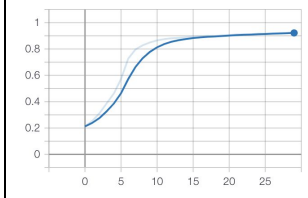
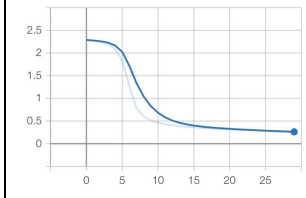
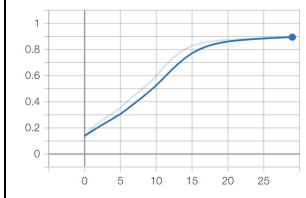
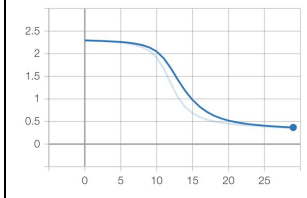
2						
3						

Table 2.1.2: Loss and accuracy graphs when training with different number of nodes and batch sizes.

Table 2.1.2 shows the accuracy and loss during training. The graphs look all very similar general, but we surprisingly notice that as the number of workers increases, the model converges slower. This difference is especially significant when we are working with larger batch sizes. This may be caused by inconsistency when training on multiple devices, even when training synchronously. And when using larger batch sizes, there are less updates to the variables during each epoch, and thereby requiring even more epochs to converge.

Conclusion

In conclusion, our experiment has achieved the expected results and goals for this assignment. In the first part, we implement simple Logistic Regression applications with Tensorflow 1.14 based on both Async-SGD and Sync-SGD models, discover the similarities and differences between these two models, and report the resource utilization. The Async-SGD training model costs less time to converge compared to Sync-SGD model whereas the difference in terms of rate of convergence is negligible. On the other side, the Async-SGD utilizes the CPU more efficiently and utilizes less network. In the second part, we implement a simple LeNet application with Keras API based on Tensorflow 2.0, and examine the resource

utilization, accuracy, and convergence time across combinations of different batch sizes and numbers of workers. The number of workers affects the resource utilization, especially for CPU and network while batch size affects the convergence time.

Execution Graph