CS-744: Big Data Systems

Assignment 2

Adarsh Kumar, Arjun Balasubramanian, Sonu Agarwal

October 15, 2018

Logistic Regression

Introduction

The first part of the assignment is to implement a simple model using logistic regression. The loss function for logistic regression is given by the below equation -

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

We utilize TensorFlow's GradientDescentOptimizer to minimize the above loss function. We train the model over the MNIST handwritten digits database and in each case, try to measure the accuracy achieved by our model. From a system's perspective, we lay special emphasis on the performance of the model in running different batch sizes and training epochs as well as on the memory and network utilization of TensorFlow under these varied conditions.

Execution on a Single Node

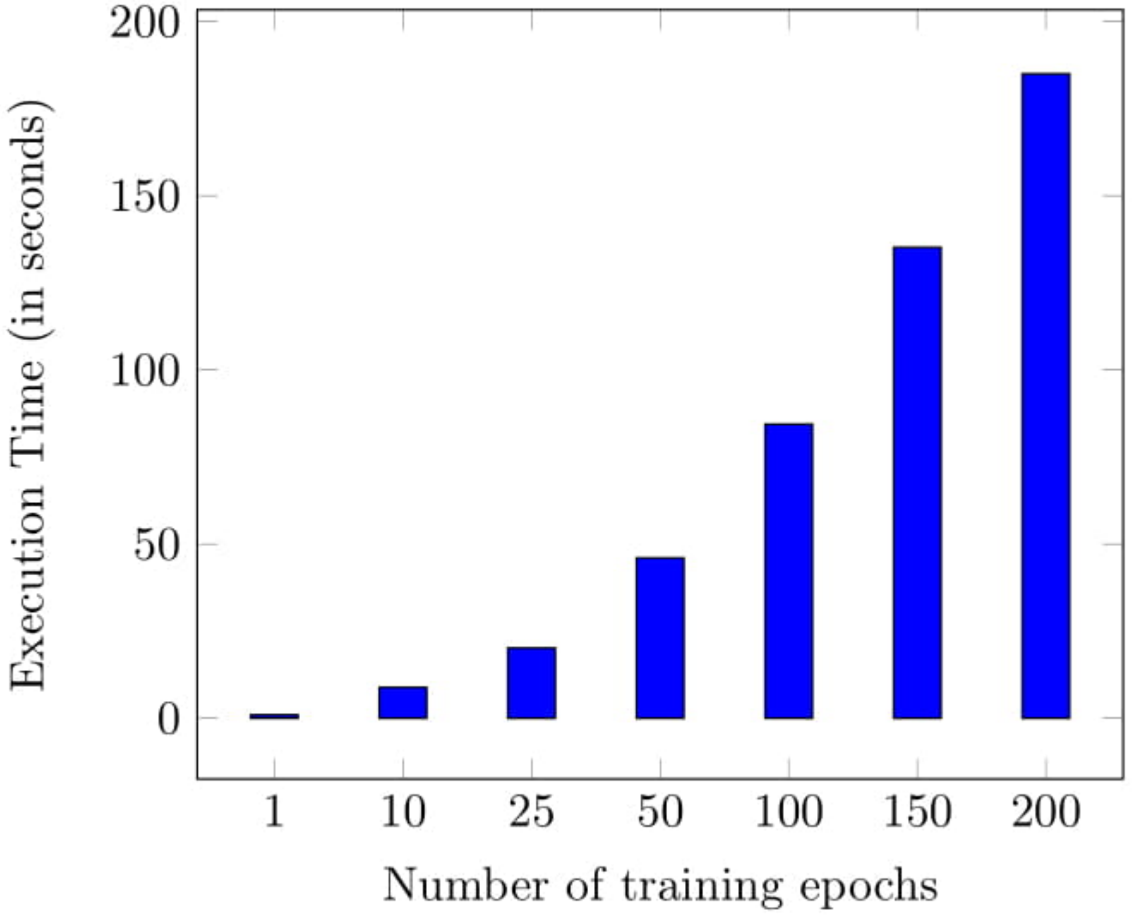
Executing the above required algorithm is straightforward on a single node. However, it is useful to understand one of TensorFlow's main feature, Deferred Execution. With deferred execution, a programmer needs to first set up a graph of how the computation should look like and can start the execution at a later point of time.

Below is a short summary of the various components of the program -

1. At every iteration, we wish to feed the training data x_i that has a label y_i to the model. TensorFlow provides an intuitive API tf.placeholder() for representing such data. For our model, we define two placeholders – one for the MNIST handwritten digit, which is a 28*28 image and one for the label, which can have 10 values ranging from 0-9. During each run, the value for placeholders is specified by the feed_dict argument when running the TensorFlow session.
2. Next, as in standard SGD implementations, we need to find W and B that minimize the loss function. Intuitively, W and B are global variables that are updated each time a new gradient is computed. TensorFlow provides an API named tf.Variable() for managing such items, which have values that can span across session runs.

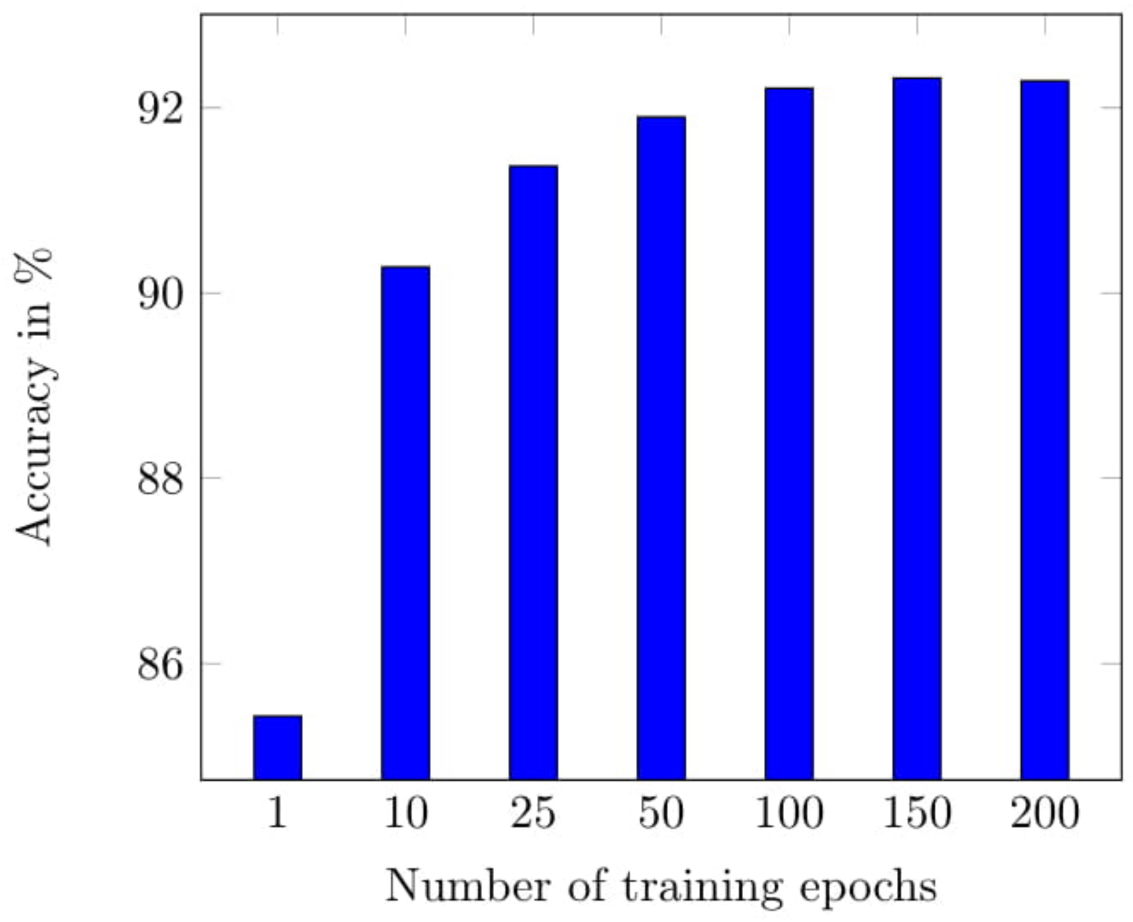
3. Next, we define how the prediction and loss are to be computed. We then specify the optimizer to run over the loss function. For the purpose of this assignment, we utilize the tf.train.GradientDescentOptimizer() API and supply the API with a learning rate and the loss function.
4. Finally, we create a TensorFlow session and run the above formulated model on the training data in batches and across multiple training epochs.

In our attempt to analyze the performance of TensorFlow from a system's perspective, we first look the impact of running time by increasing the number of training epochs. Below is a graph that captures our observations -

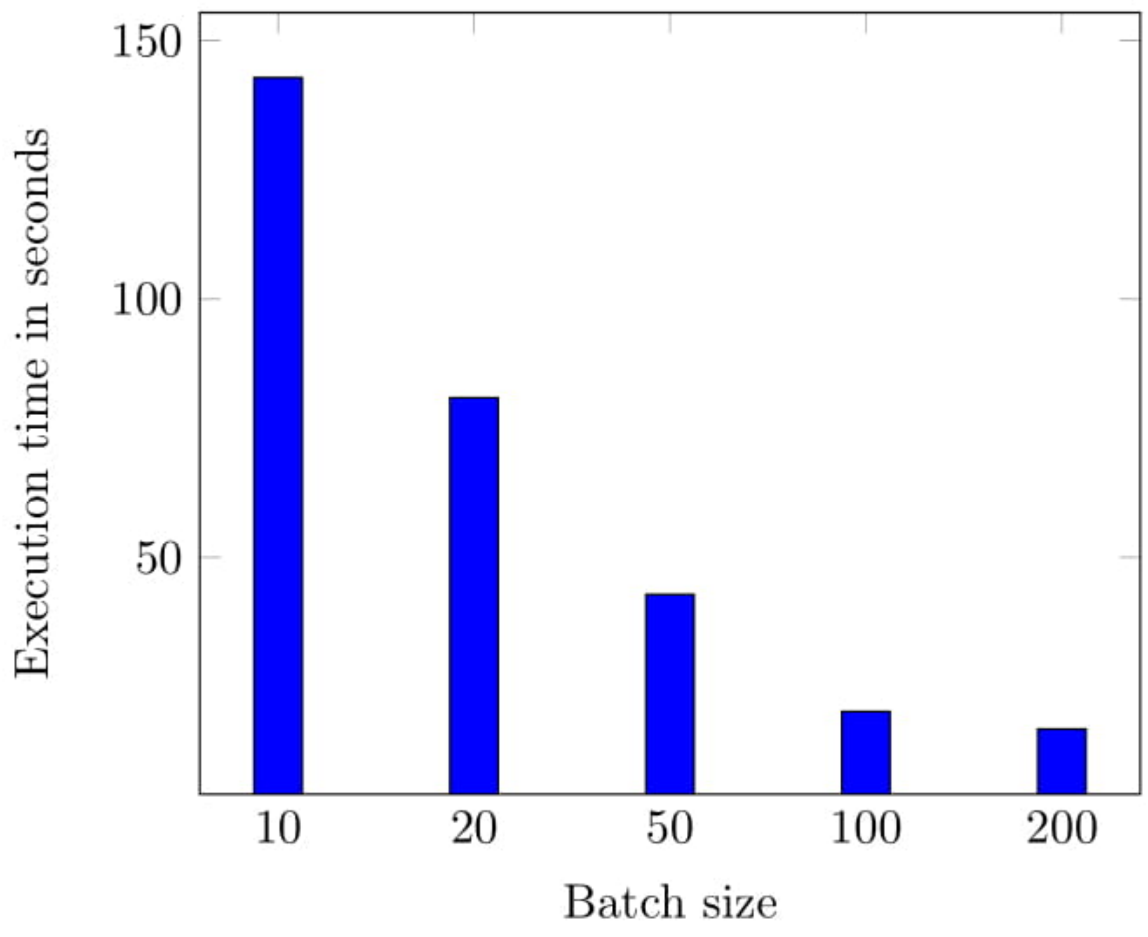


As evident from the graph, we see an almost linear increase in runtime with an increase in training epochs. This is because TensorFlow on by default executes on a single thread because of the Global Interpreter Lock in Python. To execute parallelly on all available threads, the programmer needs to explicitly place the operations on different threads. The way to do so would be like distributed TensorFlow, which we would be looking at later.

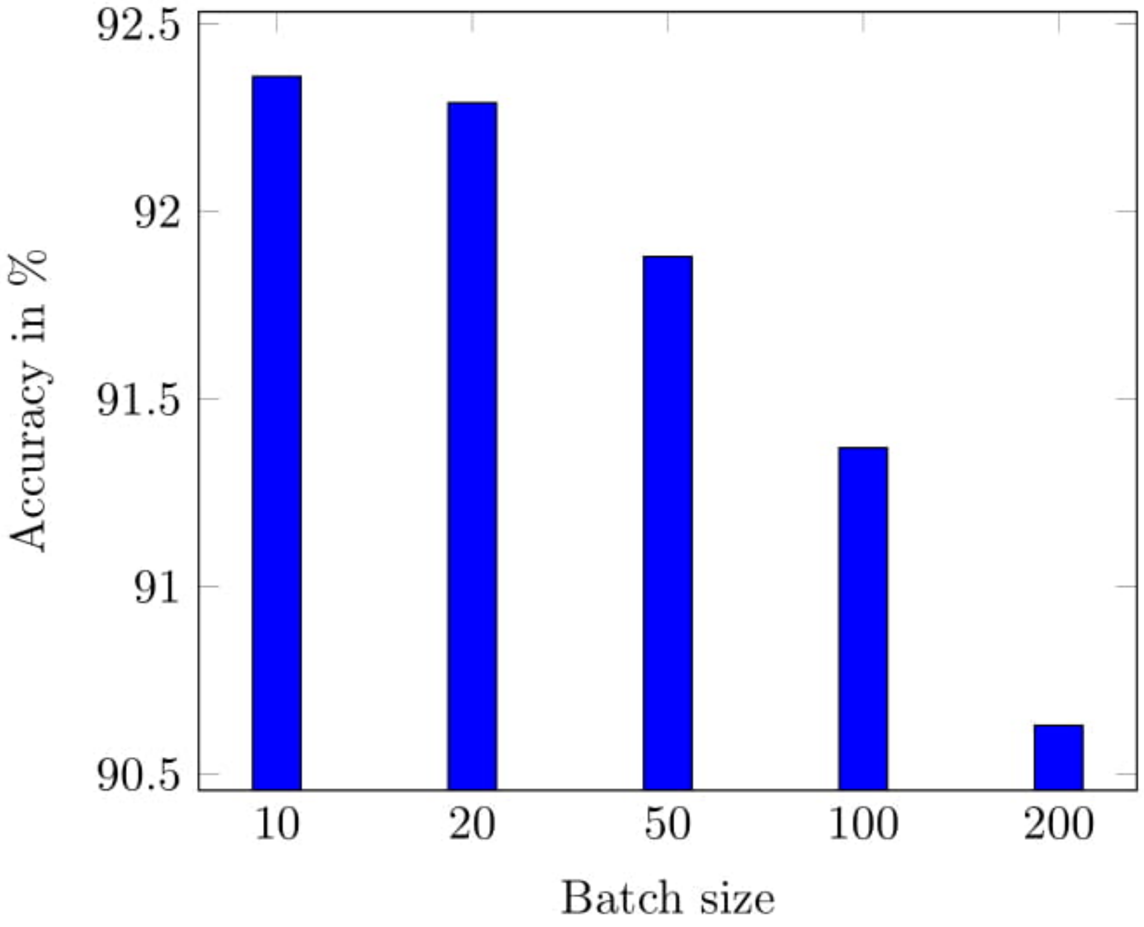
Next, we try to measure the variation of the accuracy of the model as the number of training epochs increases. Our expectation is that at first, the accuracy would increase on increasing the number of training epochs. Post that, we should see an inflection point where the accuracy starts decreasing due to over-fitting on the training data. Our predictions match the empirical results as presented below -



Next, while keeping the number of training epochs same, we try to analyze the impact of different batch sizes on the runtime. We expect smaller batch sizes to have greater runtimes since the optimizer would need to run a greater number of times on a single thread. The below graph corroborates our observations -



With respect to accuracy, the relationship between the observed accuracy and the batch size would largely depending on the input data. With random distributions of training data, we would expect “moderately” sized batch sizes to give best results in terms of accuracy. Small batch sizes would not capture enough characteristics of the training data, while larger batch sizes would tend to overfit. Below is a graph that captures our empirical observations -



For the MNIST dataset, it looks like smaller batch sizes are the way to go with respect to accuracy. This means that there is an interesting trade-off between runtime and accuracy for choosing the batch size. Per our intuition, moderate sized batches seem to be the way to go!

With respect to memory, we see that the TensorFlow allocates a maximum of about 500 pages during the lifetime of the program irrespective of the batch size or the number of training epochs. This corresponds to about 2MB of RAM usage. With respect to CPU, we observe a constant 5% CPU usage by the TensorFlow program. This can be directly correlated to the fact that TensorFlow on Python inherently executes on a single thread.

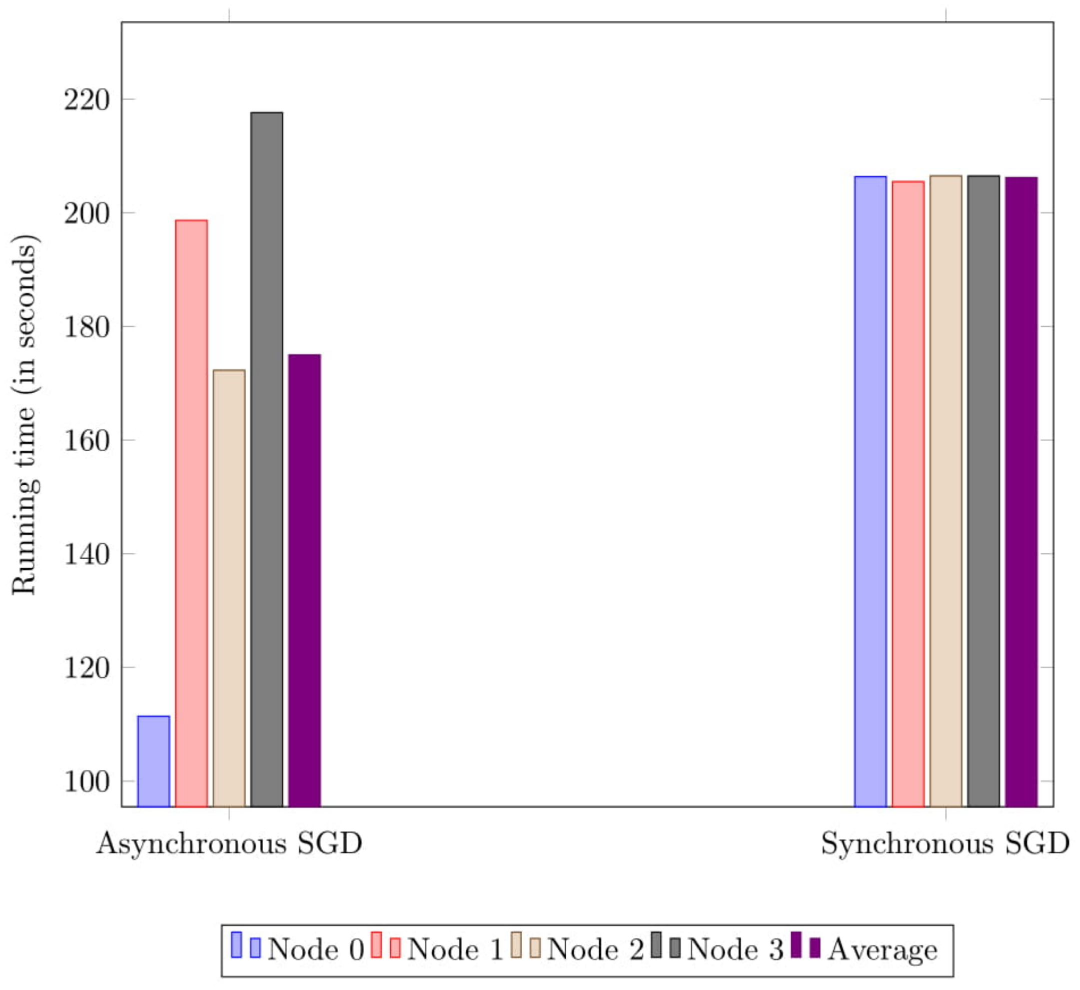
Synchronous vs Asynchronous SGD

We use Distributed TensorFlow to implement Asynchronous SGD. In our setup, we have one parameter server hosted at node-0 and four workers at node-0, node-1, node-2, and node-3. TensorFlow allows programmers to explicitly place tasks on different workers using the tf.device() API. In other respects, the implementation of Asynchronous SGD is pretty much similar to SGD on a single node.

For Synchronous SGD, the important API to use is tf.train.SyncReplicasOptimizer(), which allows for provisions to supply the number of workers and the number of replicas to accumulate before updating each variable on the model.

From a theoretical standpoint, Synchronous SGD would ensure that workers do not operate on a stale set of parameters and thus we expect faster convergence and greater accuracy in the model. However, the overhead of synchronization should negatively affect performance. The interplay between these two trade-offs is subtle and hence we gather empirical results to compare the two approaches.

We first compare the runtimes of Asynchronous vs Synchronous SGD by using a batch size of 25 and 20 training epochs in the graph below -



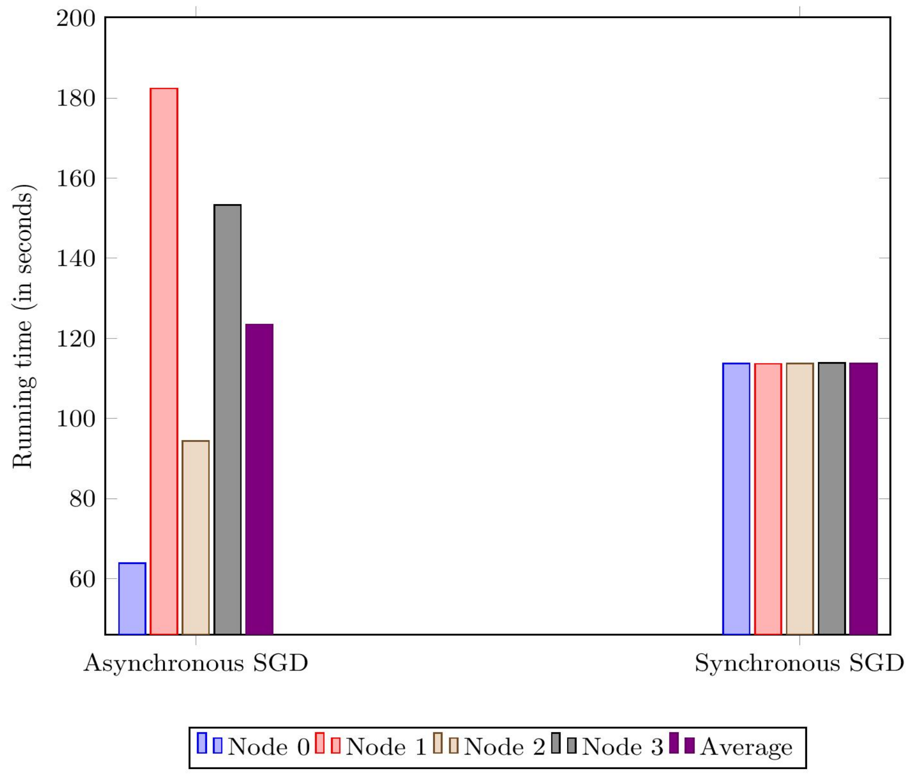
Below are some of the key takeaways from the above graph -

- We see that on an average Asynchronous SGD runs the same number of iterations faster than Synchronous SGD. This is expected because Synchronous SGD introduces the overhead of having to wait for enough gradients to be updated to the parameter server before a worker can pull parameters and start the next iteration.

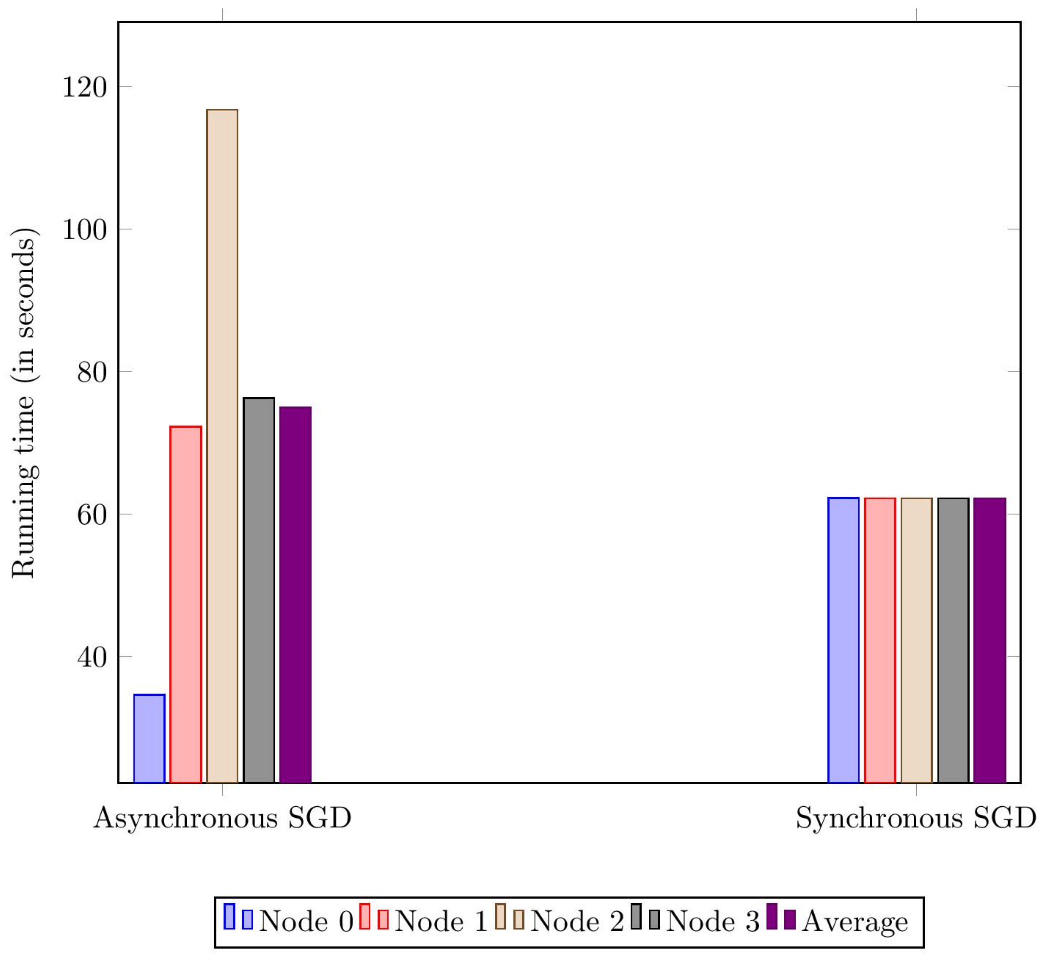
- It is surprising to see a large skew in the runtime at different workers in the case of Asynchronous SGD. We can observe that the worker on node-0 finishes its execution very quickly in comparison to the other workers. We went deeper into this problem and observed the networking stack wrapped under gRPC to be a potential bottleneck that results in such skews. We used the netstat to monitor the network traffic between a worker node and parameter server node. Now, the TCP networking stack at the parameter server node hosts three TCP connections to the workers. In theory, we expect fair multiplexing of the available bandwidth among the connections. However, in practice, we do not observe a uniform distribution of bandwidth among the workers, which causes large skews in running times!
- In Synchronous SGD, we see all workers having nearly the same runtimes. Our belief is that the wait for synchronizing the parameters indirectly induces fair multiplexing at the networking stack.

Next, it is also useful to look at how changing the batch size affects the runtimes of both Asynchronous and Synchronous SGD. Our observations are presented in the below two graphs -

Batch size of 50



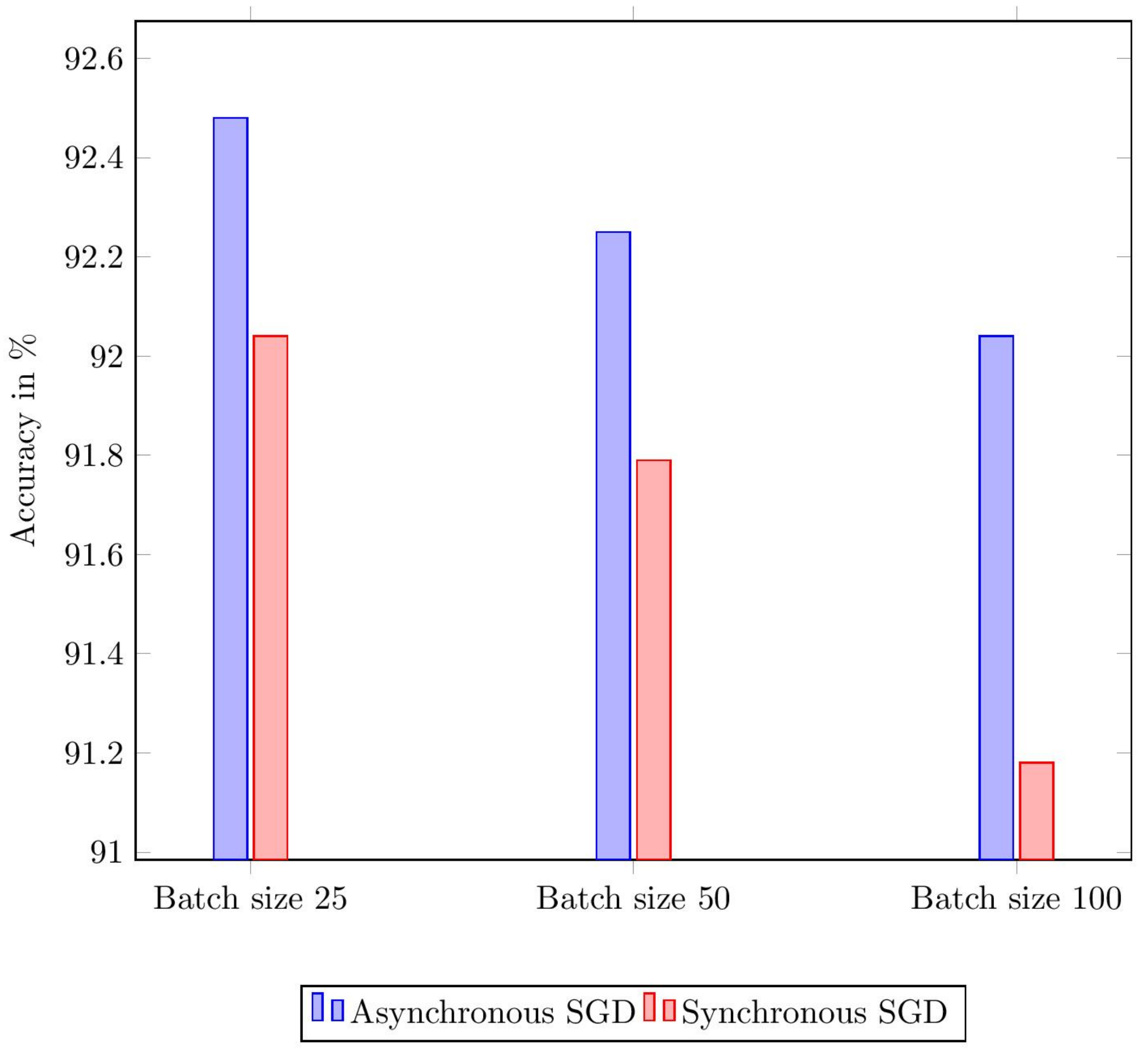
Batch size of 100



From the above two graphs, we can infer -

- As expected and as observed even in the single node case, both Synchronous and Asynchronous SGD have lesser average running times with a larger batch size.
- Contrary to the observations while using a batch size of 25, we observe that the average performance of Synchronous SGD is better than Asynchronous SGD. Though, the reason for this observation is not intuitive at first, we can understand this trend by looking at the breakdown of individual worker runtimes. In Asynchronous SGD, we notice that the worker which resides on the same machine as the parameter server finishes the quickest among all workers, while all other workers in the Asynchronous have very high runtimes. Once again, the lack of fair multiplexing on the networking seems to be the chink in the armor for Asynchronous SGD with respect to running time.

The next important aspect to consider while comparing Asynchronous and Synchronous SGD is the accuracy of the models generated by them. Because Asynchronous SGD ends up operating on stale parameters, we logically expect Synchronous SGD to have better convergence properties and consequently better accuracy. The findings from our experiments are presented in the below graph -



The data from the graph above clearly contradicts our expected results. Most likely, we believe that this is to do with the characteristics of the MNIST input data. Perhaps, the input data overfits the model and asynchronous updates mask some of this skew.

The finally aspect we compare is the memory, CPU usage, and network bandwidth of Synchronous and Asynchronous SGD. We execute Synchronous and Asynchronous SGD on a cluster with 4 nodes. For sake of brevity, we present the data only the data from node-0, which hold one worker and the parameter server, and node-1, which holds one worker. Due to symmetry, we believe that the numbers from node-2 and node-3 would be similar to node-1.

First, we consider the memory usage. We monitored the number of pages allocated by all TensorFlow processes on each node. The ballpark numbers are presented below -

Memory - Asynchronous SGD

Batch Size	Memory – Node 0 (in pages)	Memory – Node 1 (in pages)
25	704	403
50	701	402
100	708	402

Memory – Synchronous SGD

Batch Size	Memory – Node 0 (in pages)	Memory – Node 1 (in pages)
25	715	404
50	741	405
100	702	401

From the above graphs, we notice that the memory footprints are pretty much the same in both Synchronous SGD and Asynchronous SGD. This is expected since there is practically no difference in memory allocations between Asynchronous and Synchronous SGD. Also, as expected the memory footprint of TensorFlow on node-0 is higher since it has 2 processes – the parameter server process and the worker process.

Second, we consider the amount of CPU consumed by TensorFlow processes in the table below -

CPU - Asynchronous SGD

Batch Size	CPU usage– Node 0 (in %)	CPU usage– Node 1 (in %)
25	12	5
50	12	5
100	12	5

CPU – Synchronous SGD

Batch Size	CPU usage– Node 0 (in %)	CPU usage– Node 1 (in %)
25	10	5
50	10	5
100	10	5

Below are some observations from the above table -

- CPU usage is same on workers in both Synchronous and Asynchronous SGD. In theory, we expect workers in Synchronous SGD to have lesser CPU consumption due to some periods of wait for synchronization, but perhaps this is pretty negligible.
- CPU usage is more on node-0 in both Synchronous and Asynchronous SGD due to the fact that it holds 2 processes.
- CPU usage on node-0 for Synchronous SGD is lesser than Asynchronous SGD. This is because in Asynchronous SGD, the parameter server ends up performing more gradient computations which are generally computationally intensive.

Last, we consider the amount of network bandwidth consumed at node-0 and node-1. For sake of brevity, we consider the amount of data received over the network interface of node-0 and the amount of data sent over the network interface of node-1 -

Network Utilization - Asynchronous SGD

Batch Size	Received on Node 0 (in MB)	Sent from Node 1 (in MB)
25	4524	1500
50	2050	747
100	1081	407

Network Utilization – Synchronous SGD

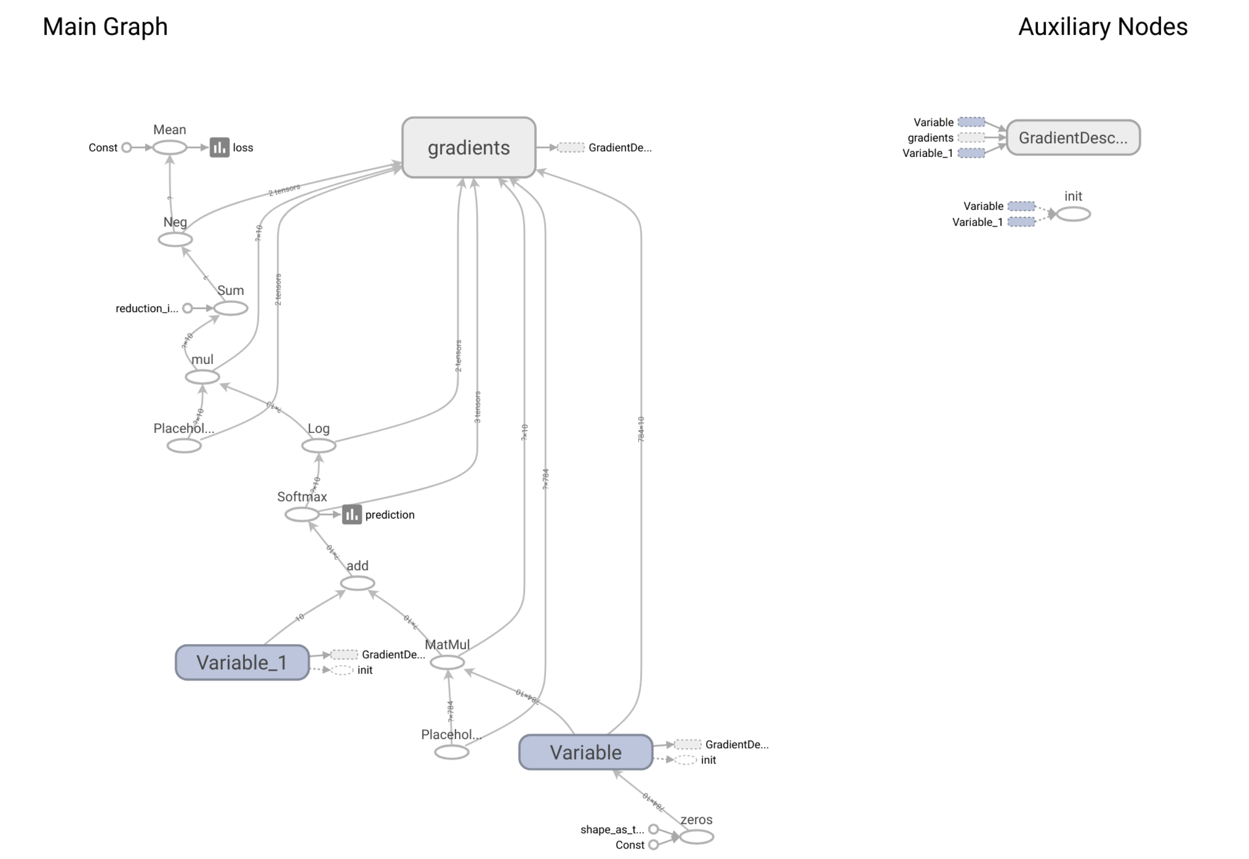
Batch Size	Received on Node 0 (in MB)	Sent from Node 1 (in MB)
25	4950	1680
50	2596	885
100	1344	448

Below are some of the key take-aways from the above two graphs -

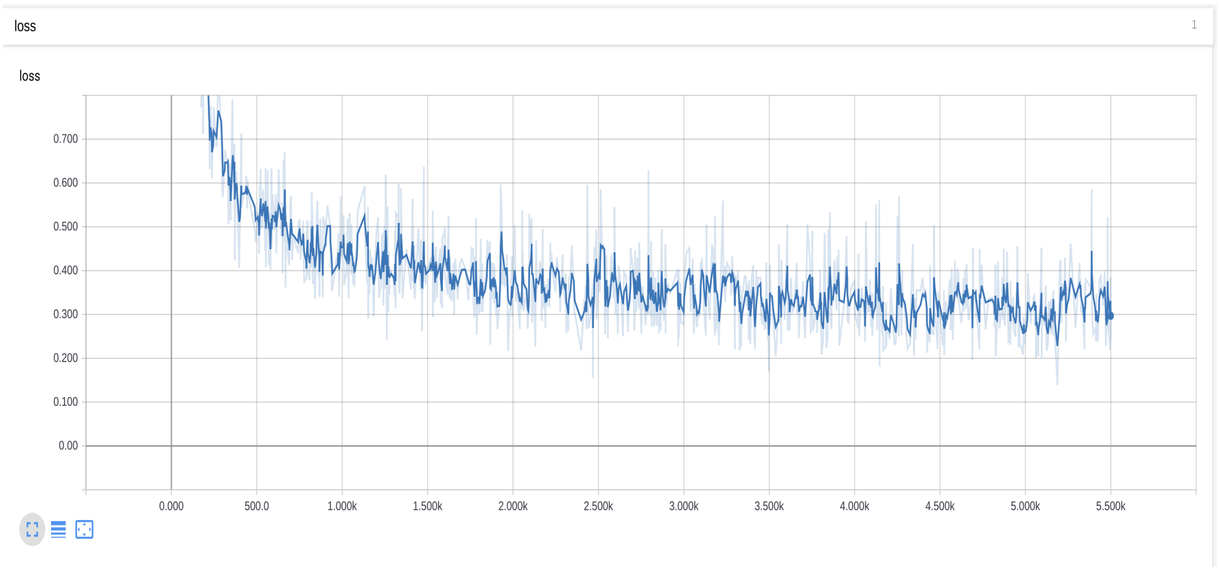
- In both Synchronous and Asynchronous SGD, the data received by node-0 is approximately three times the data sent from node-1. This is obvious because the amount of data received by the parameter server will be the sum of the data sent from all remote worker nodes.
- As batch size increases, the network utilization proportionately decreases. This is because a larger batch size results in lesser computations being propagated by workers to the parameter server.
- Data sent/received is slightly higher in Synchronous SGD. This can be accounted to be due to synchronization primitives between the parameter server and the workers.

Visualization on TensorBoard

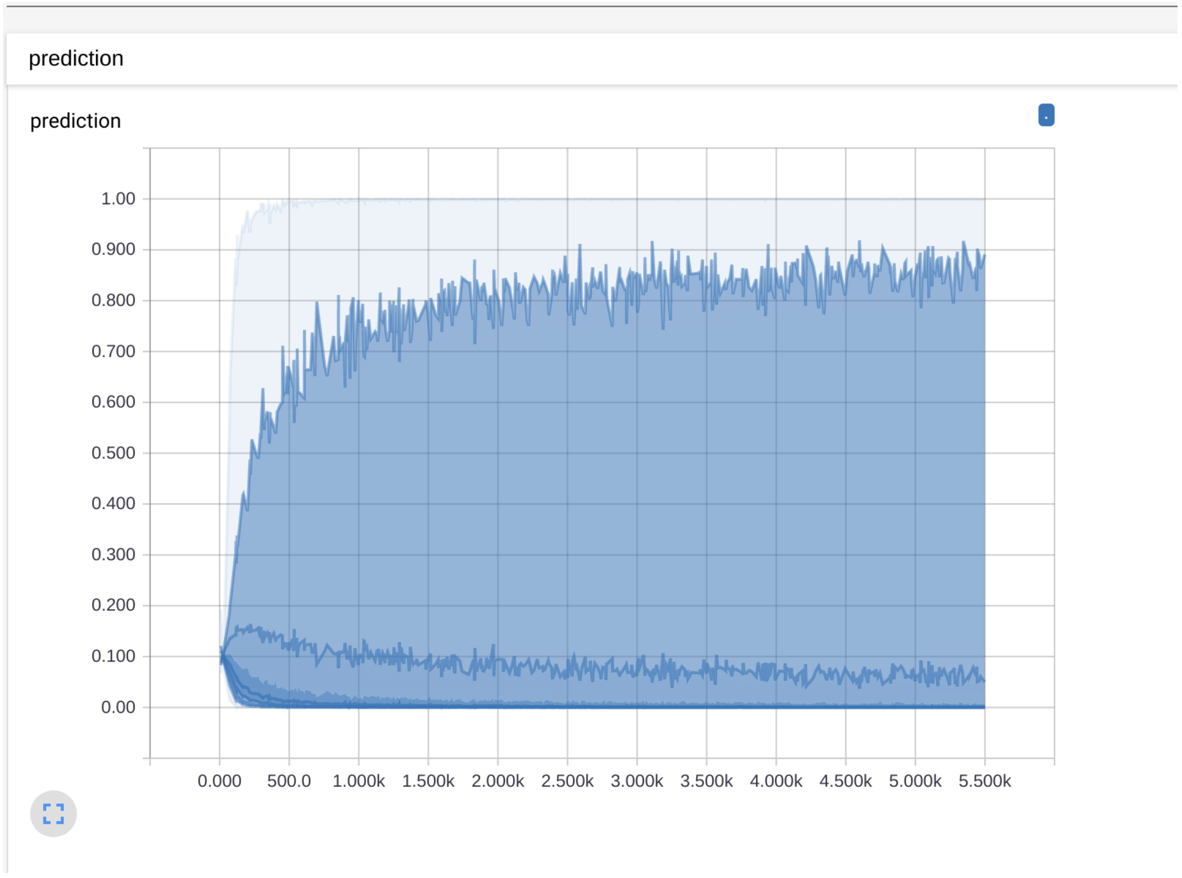
We first utilize TensorBoard to visualize the computation graph of our model. This makes debugging easy since it allows developers to see the flow of computation. As visible in the below diagram, the flow is exactly what we described in the first section of this report -



TensorFlow also allows developers to dump the values of tensors/variables during each session run. In our case, this was particularly useful to check how the convergence of the model varies with time. As a sample, we should below a plot of how the loss convergences with time for Asynchronous SGD -



Similarly, we can see how the accuracy of prediction increases with time. This would be particularly important in order to know whether the training is overfitting.



From both the loss and prediction graphs, we can see the convergence is well formed and gradual and we believe that the model does not overfit.

AlexNet

In this part, we do performance analysis of AlexNet architecture, which is designed to classify images.

We train AlexNet using widely used Tensorflow (TF) framework on single machine and distributed setting.

The architecture of AlexNet is as follows : Five convolutional layers followed by three fully connected layers. To achieve non-linearity it uses ReLU instead of Tanh.

Our goal is to analyze the performance of training AlexNet model in distributed and single machine setting.

We study the performance by changing the batch-size and number of workers for training. All experiments are done on Flowers Data mentioned in the repo.

Experiments on a single machine (Task 1):

In this section we focus on single machine setting. We vary the batch-size from 32 to 512 linearly and study its effect on CPU, Memory and Network usage.

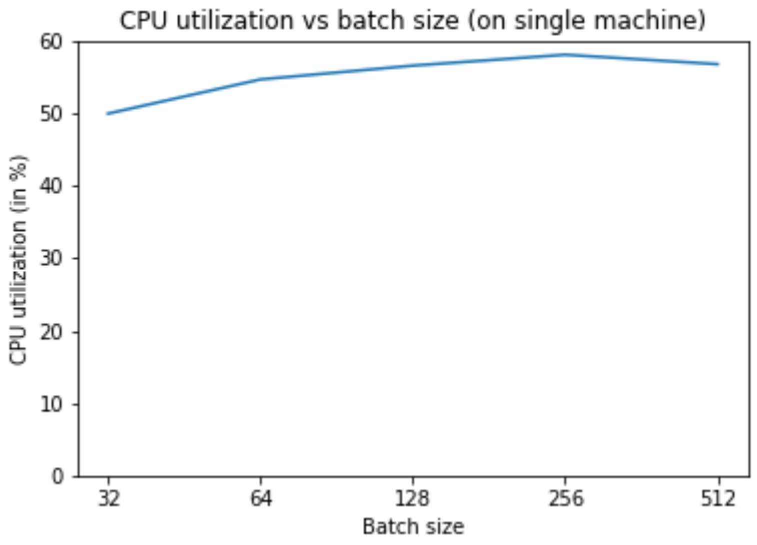
Performance

Batch Size	Examples/sec	Time (in sec)/batch
32	15	2.13
64	26	2.29
128	31	4.13
256	33	7.75
512	33	15.51

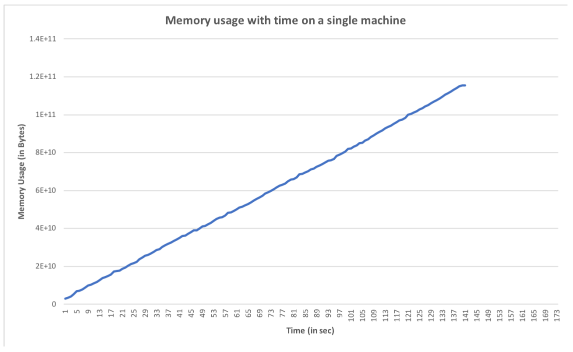
On increasing the batch size from 32 to 64 we see significant increase in the processing speed, while we don’t observe similar increase when increasing the batch size further. As per our understanding, at batch size 32, CPU is underutilized and increasing the batch size to 64 gives the gain. While after 64 batch-size, CPU utilization is almost constant for greater batch-sizes, thus showing minor gains and eventually it gets flattened.

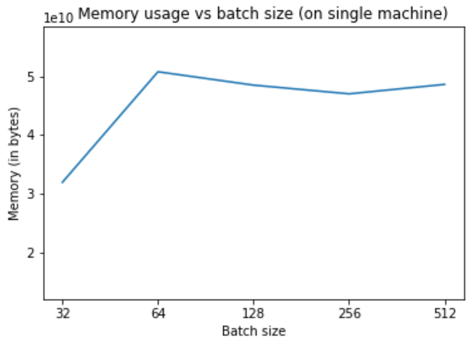
The increase in batch processing speed is not linear as computation increases at a faster rate than the increase in the batch size because of larger intermediate tensor computation.

As we increase the batch size to 256, computation time overcomes the advantage of processing greater number of instances at a time, and number of examples processed per second decreases.



Above plot shows the average CPU utilization for different batch-sizes. The CPU utilization reading presented above corroborates our analysis for processing speed. We can observe that there is significant increase in CPU utilization by changing batch size 32 to 64, but post that the increase is not that much and it also eventually flattens.





The left plot above shows the memory usage of the model as the time varies. As the time goes, the training data keeps on accumulating in the memory, thus monotonic increase in the plot. We observe same monotonic increase for each batch-size.

Plot Y on the right compares the memory usage at a particular instance of time for different batch-sizes. Intuitively, for larger batch sizes at a particular instance of time the memory usage would be higher. But for very large batch-sizes(128-512), the processing speed flattens leading to flattening in memory usage also. Our plot exhibits similar behavior.

Network usage for this setup was constant for all batch-sizes as everything is done on same machine.

Experiments on cluster (Task 2):

In this task we change our configuration from single machine to distributed machines to study its effect on performance.

In first configuration (a.k.a. **cluster1**), there is **one parameter server and two workers**. Parameter server and one of the workers are on **node0** and another worker is on **node1**.

In second configuration (a.k.a. **cluster** 2), there is **one parameter server** (on node0) and **4 workers**, one each on node0, node1, node 2 and node3.

Table x presents the performance speed of both clusters for different batch sizes.

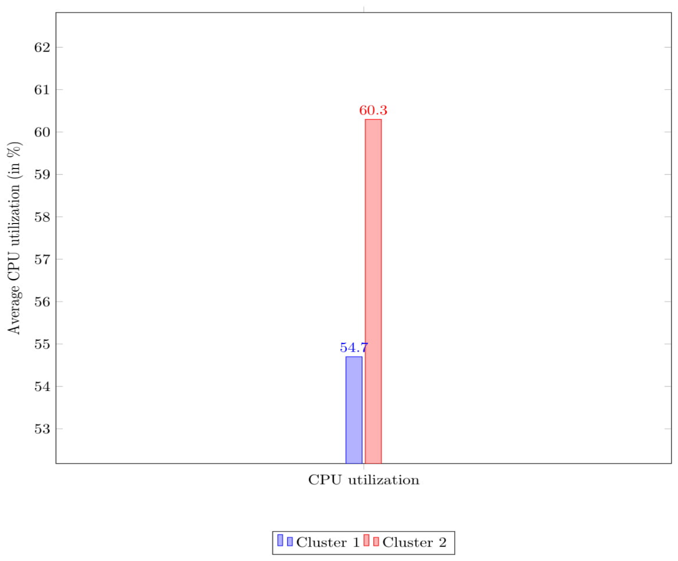
Performance

Batch Size	Cluster1	Cluster1	Cluster2	Cluster2
	Examples/sec	Time (in sec)/batch	Examples/sec	Time (in sec)/batch
32	28	2.3	54	2.4
64	52	2.5	80	3.2
128	48	5.3	95	5.4
256	46	11.1	96	10.7

Intuitively, on increasing the number of workers for a particular batch size, the processing speed, I.e. examples processed per second should increase linearly. We can indeed observe this from Table X. For ex.: for batch-size 32, ratio of processing speed of cluster 2 to cluster 1 is almost 2:1 and this ratio is almost similar for further batch-sizes also.

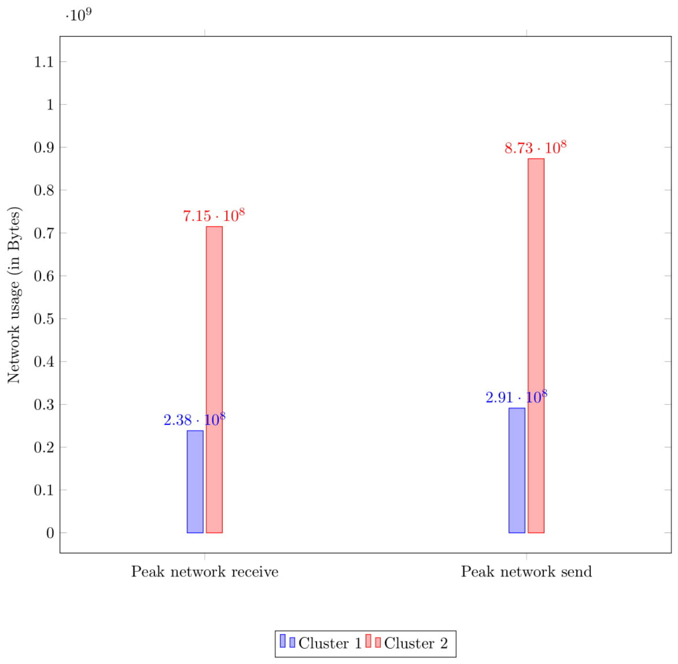
CPU/Memory/Network Usage

We study Network and CPU usage for a fixed batch size (64).



Above plot shows the CPU utilization comparison of Parameter Server node for Cluster1 and Cluster2. Although not significant, there is slight increase in the CPU utilization for Cluster2, as the Parameter Server has to do little more processing in terms of accumulating gradients.

For Network usage we present the peak data transfer at Parameter Server for both setup.



Ideally as we increase the num of workers, network traffic should increase at the Parameter Server node, as the model parameters/data has to be sent/receive from all workers. Above plot supports this intuition as the network traffic increased on increasing the worker.

We also see that the ratio of peak network receive data is around 3:1 for cluster2 and cluster1. As per our logic, this is because of the 3 workers are on different machines in Cluster2 while 1 in Cluster1.

Also, the data received at Parameter Server should be less than the Data Send, as Parameter Server will only receive the parameter updates, but will send parameters along with data to the workers. For both the setup, we observe this pattern in the above plot.

--- END ---