CS744 - Big Data Systems
Assignment 2

Ushmal Ramesh uramesh2@wisc.edu

Varsha Pendyala varsha.pendyala@wisc.edu

Varun Thumbe thumbe@wisc.edu

1. *Part-1 - Logistic Regression with Tensorflow API*

1.1 Setup and Configuration

1) We followed the setup instructions posted on the assignment page and coded the implementation of Logistic regression using Tensorflow original APIs. While setting up the tensorboard we noticed that each time the code is run tensorflow summary writer overwrites the log directory and thus the graphs we obtain for different summary variables become useless. Thus we delete the log directory each time before the code is run to get the desired graphs.

2) We use MonitoredSession object to do the sync and async model updates. We can specify the summary writer while initiating the MonitoredTrainingSession object. Using this functionality, we observed that the summary variables were getting written each time sess.run() was getting executed, as a result summary for different workers were getting overwritten in the same graph. We overcame this problem by not passing the summary directory to the MonitoredSession object and used our own custom SummaryWriter for visualizing the graphs.

3) For monitoring the CPU, memory and network usage we used dstat command. We wrote a script to automatically run the logistic regression code with different parameters and cluster modes and store the output log and dstat logs separately for different parameter configurations. We later parse the saved logs to extract measurement data and generate plots using a custom python script.

1.2 Logistic Regression for Single worker

The main learning in this part of the assignment was understanding the tensorflow framework. The tensorflow code was organized as follows:

i. We initialized the input and the label as tensorflow placeholder objects and the weights and biases as variable objects and constructed the loss using these objects.

ii. We used the tensorflow's Gradientdescent optimizer object that takes loss as the input to minimize.

iii. Finally, after constructing this graph of execution, we take one batch of examples at a time and run the optimizer object at each iteration which updates the weights and biases variables. After every 100 iterations we print the test loss on to the console to monitor the convergence of the model parameters.

1.2.1 Issues Faced

We initially used the same number of iterations(6000) to compare the results for different cluster modes and batch sizes. However, we realized that in some cases the model would have already converged much before this and in some cases the model would not have converged at all. Hence, rather than comparing the results for a fixed number of iterations, we updated the model parameters until the loss reached a certain value. In this way we will be comparing the results for a fixed model state. We found that for the loss value of 0.35 the accuracy of the model was just around 90%. Hence we set loss<=0.35 as the *stopping criterion* to end the tensorflow session for every worker.

1.2.2 Observations:

i. Effect of increasing Batch size: Iterations and time to convergence are tabulated in table 1 below. It is to be noted here that since we calculate the test loss every 100 iterations, the total number of iterations in Table1 shows a constant value. The actual number of iterations will lie anywhere between 7001 and 7100. The time taken for reaching convergence also remained nearly constant.

Batch_Size	Iterations	Total time(seconds)	Accuracy(%)
1	7100	13.7441118	90.65
5	7100	13.9641016	90.67
10	7100	13.9424944	90.74
20	7100	14.008393	90.64
40	7100	13.8538275	90.63
100	7100	14.0354843	90.72
200	7100	13.8297627	90.62
600	7100	13.8365335	90.7

Table 1 - Different runs of the Logistic Regression Training on a single machine

ii. Memory and CPU usage: The CPU and memory usage(max) of different runs of LR on single machine with varying batch sizes is tabulated below:

Deploy_Mode	Batch_Size	cpu%	memory(KB)
single	1	34	701000
single	5	35	698000
single	10	35	677000
single	20	36	641000
single	40	35	671000
single	100	35	673000
single	200	35	647000
single	600	35	700000

Table 2 - CPU and memory Usage

We observe that the usage remains fairly constant throughout as expected.

1.3 Logistic regression in Async Mode:

For this part, the code is organized as follows:
i. The input, output and optimizer are defined as before in single machine case.
ii. For implementing Logistic regression in async mode, we used MonitoredSession session object in tensorflow that initializes session variables, handles asynchronous updates, performs checkpointing operations on the model as well as writing summary on to the tensorboard.
iii. We use the tensorflow get_or_create_global_step to obtain a global step tensor to keep track on global steps taken in training. Each time the weights are updated in ps, the global step is incremented.

1.3.1 Issues Faced:

1. *MNIST data download issue* : While running the experiment, we saw that some workers reported the error 'AlreadyExistsError: file already exists'. Internet search led us to discover that there was a similar github issue filed under Microsoft's nni repo. The issue thread explained that the issue might be due to multiple processes attempting to download the MNIST dataset at the same time(usually on the same machine). The fix provided involved using a function that retries the download after waiting for sometime instead of crashing. The function is given below:

```
def download_mnist_retry(seed=0, max_num_retries=20):
    for _ in range(max_num_retries):
        try:
            return input_data.read_data_sets ("MNIST_data",
            one_hot=True,seed=seed)
        except tf.errors.AlreadyExistsError:
            time.sleep(1)
    raise Exception("Failed to download MNIST.")
```

After applying the above fix, the error did not recur.

Some workers getting stuck: When we ran the model training, at first we started observing that some of the workers were stuck with the following message:

2. '*Waiting for model to be ready. Ready_for_local_init_op: None, ready: Variables not initialized: Variable, Variable_1, global_step'*

   After some troubleshooting and internet search, we discovered that this behaviour is the result of a 'race condition' between the worker initializing the variables and others checking for the initialized variables. If the workers checking for initialized variables win the race then they enter a 30 second wait state before checking again. However, since our tasks are so small this 30 second time is enough for other workers to start and finish and so when the worker wakes up to find there are no tasks and so gets blocked. To resolve this we added a device filter by asking tensorflow to ignore presence of other workers at session creation time. This is done using the tf.ConfigProto API. the exact config parameter added was the following:

   ```
   config = tf.ConfigProto(device_filters=
   ['/job:ps','/job:worker/task:%d'%FLAGS.task_index])
   ```

   With the above parameter passed in to MonitoredTrainingSession object, each worker now needed to communicate only with the PS task and the local worker task and the issue was resolved.

1.3.2 Observations

i. *Number of iterations to converge :* We see a general decrease in the number of iterations required to reach convergence as we increase the batch size. The results are tabulated in table 2 below. Again, the actual iteration numbers will be slightly different as we test for convergence every 100 iterations, however, the decreasing trend is evident from the table. Also, the 'single' cluster profile took larger number of iterations to converge than cluster and cluster 2.

ii. *Time taken to Converge* : The time taken for convergence was the least for 'single' cluster profile. This is due to the fact that this profile has no network overhead as the worker is local to the PS.

Also, across all the cluster profiles, the time to convergence varied within a 2-3 second interval. We believe that this is due to the fact that the effect of decreasing iterations with batch sizes could be canceled out by additional time required to compute the batch gradient with increasing batch sizes and therefore the overall time required would remain fairly constant.

Deploy-Mode	Batch_Size	Iterations (Total across workers)	Total time (Max across workers), seconds	Accuracy (Max Across workers), %
cluster	1	10000	35.058	90.42
cluster	5	7500	31.26	90.55
cluster	10	7400	31.232	90.62
cluster	20	7100	31.273	90.66
cluster	40	7300	31.701	90.71
cluster	100	7200	34.499	90.68
cluster	200	7100	38.305	90.62
cluster	600	7100	38.082	90.7
cluster2	1	9400	32.866	90.51
cluster2	5	7300	31.195	90.57
cluster2	10	7600	31.226	90.63
cluster2	20	7100	31.252	90.71
cluster2	40	7300	10.81	90.65
cluster2	100	7200	13.207	90.69
cluster2	200	7100	16.616	90.69
cluster2	600	7100	36.796	90.7
single	1	10000	17.681	90.18
single	5	10200	18.11	90.28
single	10	9300	16.526	90.43
single	20	8500	14.984	90.43
single	40	9400	17.513	90.57
single	100	8800	16.742	90.34
single	200	8400	14.955	90.18
single	600	8900	15.541	90.34

Table 3: Measurement of Iterations, Time and Accuracy in Async mode

iii. Workload Distribution among workers: We made an interesting observation that the worker 0 always performs more iterations and hence more updates than any other worker. This is evident from the data tabulated in Table 4 below(worker 0 rows highlighted in bold). As is seen, for all batch sizes and cluster configs, worker 0 makes the largest number of iterations. For lower batch sizes, other workers are sometimes able to make no iterations as worker 0 converges before them. For higher batch sizes, we see the other workers performing some updates.

Batch_Size	Worker	Iterations	Total time	Accuracy
1	**0**	**8600**	**32.434**	**90.16**
1	1	400	32.866	90.51
1	2	400	32.791	90.51
5	**0**	**7300**	**28.494**	**90.54**
5	1	0	31.152	90.57
5	2	0	31.195	90.57
10	**0**	**7600**	**30.155**	**90.63**
10	1	0	31.226	90.61
10	2	0	31.223	90.61
20	**0**	**4100**	**12.704**	**90.67**
20	1	0	31.252	90.71
20	2	3000	12.855	90.69
40	**0**	**2900**	**10.625**	**90.65**
40	1	2200	10.81	90.63
40	2	2200	10.595	90.52
100	**0**	**2800**	**13.183**	**90.65**
100	1	2200	13.207	90.69
100	2	2200	12.982	90.66
200	**0**	**2700**	**16.135**	**90.69**
200	1	2200	16.616	90.65
200	2	2200	15.948	90.69
600	**0**	**3400**	**36.796**	**90.69**
600	1	500	36.691	90.7
600	2	3200	36.207	90.67

Table 4: Measurement for ‘cluster2’ config in async mode showing variation in updates made by different workers

We can attribute this observation to the fact that since worker0 and parameter server are located in the same node, worker 0 has the least latency in pushing updates and pulling the latest model, and hence is able to perform more number of model updates to the model before the model converges.

iv. *TensorBoard Analysis*

In order to log relevant variables like loss, accuracy and weights, we added code to write the variable summaries for later viewing in tensorboard. The summaries are written every 100 iterations to reduce the overhead of this operation. Following are tensorboard plots of accuracy for each worker in cluster2 configuration with a batch size of 50

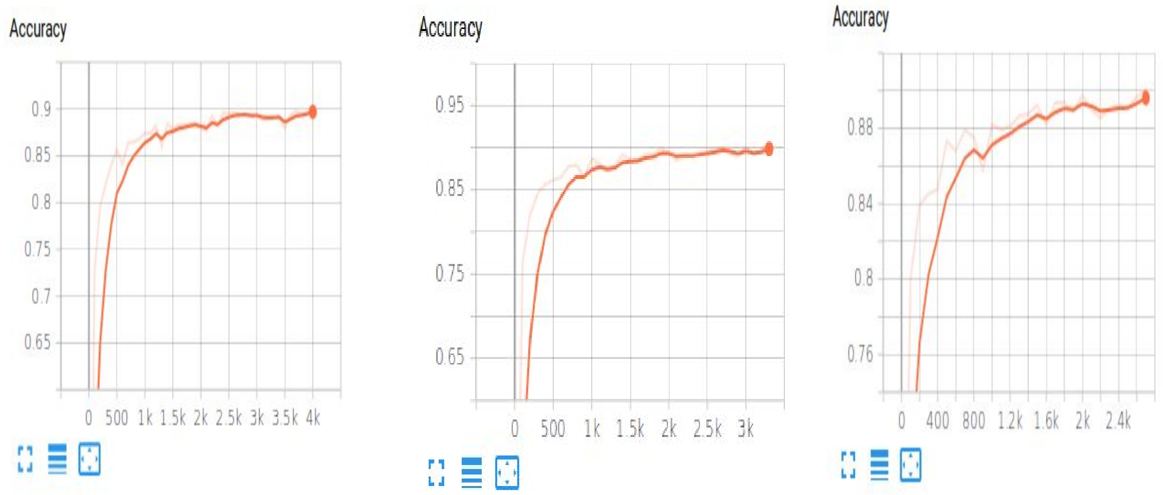

Figure 1 Accuracy Graphs in Tensorboard for worker 0 , 1 & 2 in cluster2 deploy mode

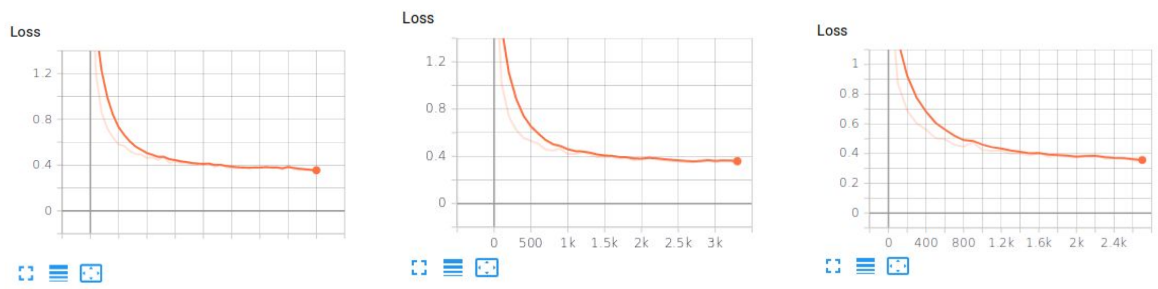

Figure 2 Loss Curves in Tensorboard for worker 0 , 1 & 2 in cluster2 deploy mode

Two main takeaways from the graph are as follows:

1) As we discussed before, worker-0 does most of the updates and completes 4k iterations before the loss converges to the optimal value while worker1 completes around 3.2k and worker-2 around 2.4k.
2) The graph is not as smooth as the single node case or even the async case (discussed in the next sub-section). Since the model updates are asynchronous, some model updates from the worker may not even change the model state, and as a result the accuracy may down across iterations(since different workers may be affecting different aspects of the model during their updates)

1.4 Logistic regression in Sync Mode

For performing training in sync mode we made use of the SyncReplicasOptimizer object whose function is to synchronize, aggregate gradients and pass them to the optimizer. This object needs the optimizer, the number of replicas and the number of workers to aggregate. All the other components of the code remain unchanged.

1.4.1 Observations

i. *Time taken to converge*: After running experiments in sync mode we find that to converge to the same loss value sync mode takes a lot more time to converge. The sync mode takes a lot more time compared to the single mode case which can be attributed to the fact that the workers have to wait if the queue for computations is full.
ii. *Number of iterations to converge :* We observe that each worker in sync mode makes the same number of iterations as that made by a worker in single mode, but since the updates are synchronized, the iterations take longer to finish. These observations are tabulated in table 5 below.

Deploy-Mode	**Batch_Size**	**Iterations(Sum across workers)**	**Total time(Max across workers)**	**Accuracy(Max Across workers)**
cluster	1	15400	34.728	90.4
cluster	5	15000	62.946	90.6
cluster	10	14600	34.02	90.68
cluster	20	14400	34.342	90.64
cluster	40	14200	64.791	90.69
cluster	100	14200	42.995	90.66
cluster	200	14200	51.435	90.73
cluster	600	14200	112.628	90.71
cluster2	1	23100	38.226	90.59

cluster2	5	21600	67.6	90.55
cluster2	10	21900	67.019	90.73
cluster2	20	21600	67.95	90.62
cluster2	40	21300	69.272	90.62
cluster2	100	21300	76.554	90.68
cluster2	200	21300	56.213	90.72
cluster2	600	21300	122.622	90.69
single	1	7100	18.799	90.62
single	5	7100	18.667	90.62
single	10	7100	18.71	90.62
single	20	7100	18.81	90.62
single	40	7100	19.017	90.62
single	100	7100	18.69	90.62
single	200	7100	18.845	90.62
single	600	7100	18.862	90.62

Table 5 : Measurement of Iterations, Time and Accuracy in Async mode

iii. Workload Distribution among workers: We observe that irrespective of the batch size and cluster delpy confis, every worker in sync mode takes approximately the same number of iterations. This is confirmed in the data tabulated in Table 6 below for cluster2 deploy mode. Other cluster modes are omitted for brevity but show the same trend.

Batch_Size	**Worker**	**Iterations**	**Total time**	**Accuracy**
1	0	7700	38.226	90.59
1	1	7700	38.186	90.59
1	2	7700	38.081	90.59
5	0	7200	67.285	90.55
5	1	7200	67.6	90.55
5	2	7200	67.179	90.55
10	0	7300	66.655	90.73
10	1	7300	67.019	90.73
10	2	7300	66.705	90.73

20	0	7200	67.741	90.62
20	1	7200	67.95	90.62
20	2	7200	67.637	90.62
40	0	7100	69.149	90.62
40	1	7100	69.272	90.62
40	2	7100	68.844	90.62
100	0	7100	76.495	90.68
100	1	7100	76.554	90.68
100	2	7100	76.392	90.68
200	0	7100	56.213	90.72
200	1	7100	56.164	90.72
200	2	7100	56.107	90.72
600	0	7100	122.503	90.69
600	1	7100	122.622	90.69
600	2	7100	122.374	90.69

Table 6 : Measurement for 'cluster2' config in sync mode showing workload distribution among workers

1.4.3 Our Inference

From the observations in 1.4.2 we infer that these can be attributed to the way SyncReplicasOptimizer sets up the sync training. The SyncReplicasOptimizer functions can be summarized as follows:

1. The object first creates gradient accumulator for each variable to be trained. The chief worker waits for all gradients to be accumulated before applying the averaged gradient. The accumulator also drops stale gradients. The global step gets updated after the averaged gradients have been applied.
2. Then a token queue is set up where the optimizer pushes the updated global_step value.
3. Meanwhile the workers wait for the queue to be updated with a fresh global step token and fetch the token from the token queue when available and start their local step and compute gradients.

This means that running a synchronous training on N workers is *equivalent to running N single worker jobs* as each worker takes approximately the same number of iterations. However, due to the added overhead of workers having to wait for the global step to be pushed to the token queue, the sync job takes approximately *N times longer* than the single worker job.

1.4.4 TensorBoard analysis:

Just like in the async mode we write the accuracy logs onto the tensorboard after every 100 iterations. Following are tensorboard plots of accuracy for each worker in cluster2 configuration with a batch size of 50

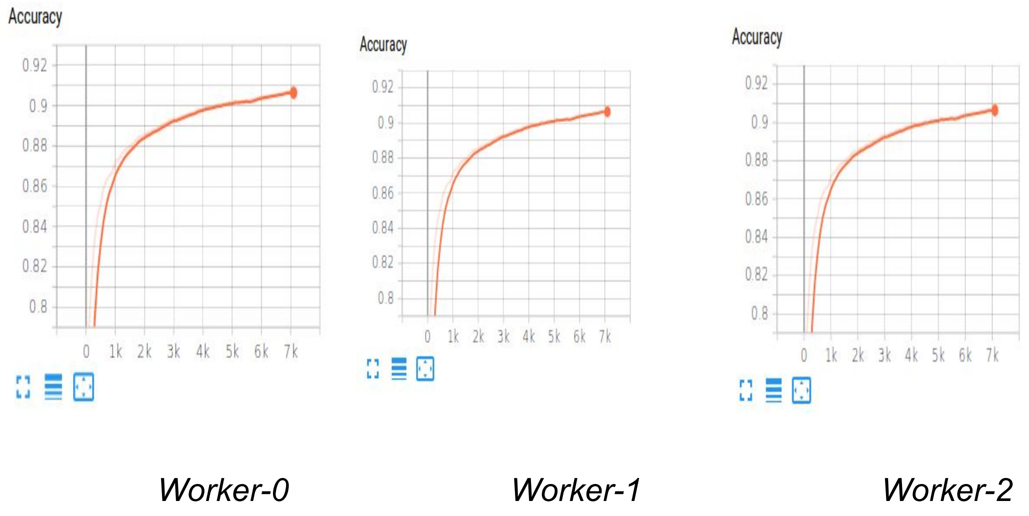

Figure 3- Tensorflow logs showing Accuracy plots for all workers

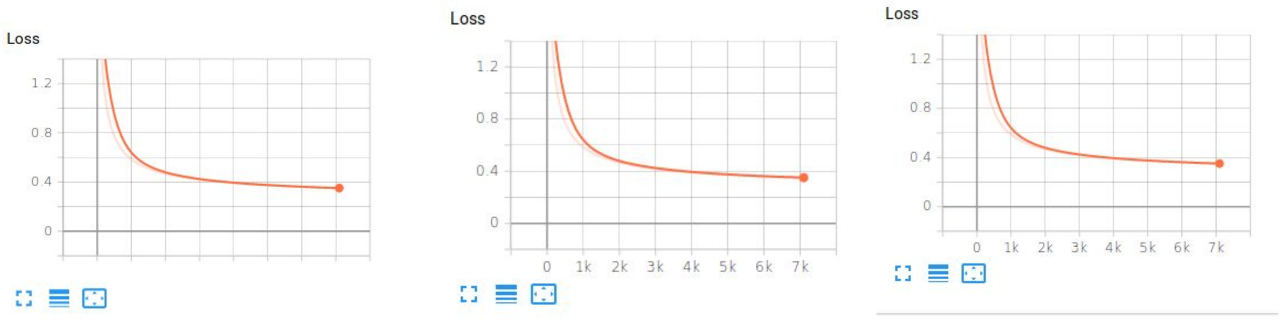

Figure 4- Tensorflow logs showing Loss curves for all workers

Following are the takeaways from the graph:

1. Here we see that all the workers perform almost the same number of updates to the model as we can see that all the workers roughly take around 7.2k iterations to reach the desired loss value(0.35) as was observed in Table 5.
2. We observe that the graph is much more smoother than the async mode case. This can be attributed to the way the updates are applied. We believe that since weight updates from all workers are averaged out before applying the update and since all subsequent updates from the workers are based off of the updated weight value, this results in a much smoother curves.

1.5 Inferences from Comparing Sync vs Async Mode of Training

1. From the experiments conducted on Sync and Async mode, we conclude that running jobs in *sync mode is a major bottleneck* as the time required to complete the job increases linearly with the number of workers. This is explained in our observation s in section 1.4.2.
2. Further, we see that in Sync mode every worker performs the same number of iterations but in Async mode, the worker which is closest to the Parameter server, i.e., worker 0 carries out most number of interactions. This distribution of workload results in almost the same time required for achieving convergence irrespective of the number of workers. Though the overall time was longer than single deploy mode due to added network overhead.
3. CPU, Network and Memory Usage : For monitoring the CPU and memory usage we used the dstat tool. Following plots the CPU, Network and Memory usage patterns of Sync and Async modes of training. Figures 5 - 8 show our measurements across workers 0 and 2 in cluster2 deploy config. We do not show measurements for worker 1 as it is symmetrical.

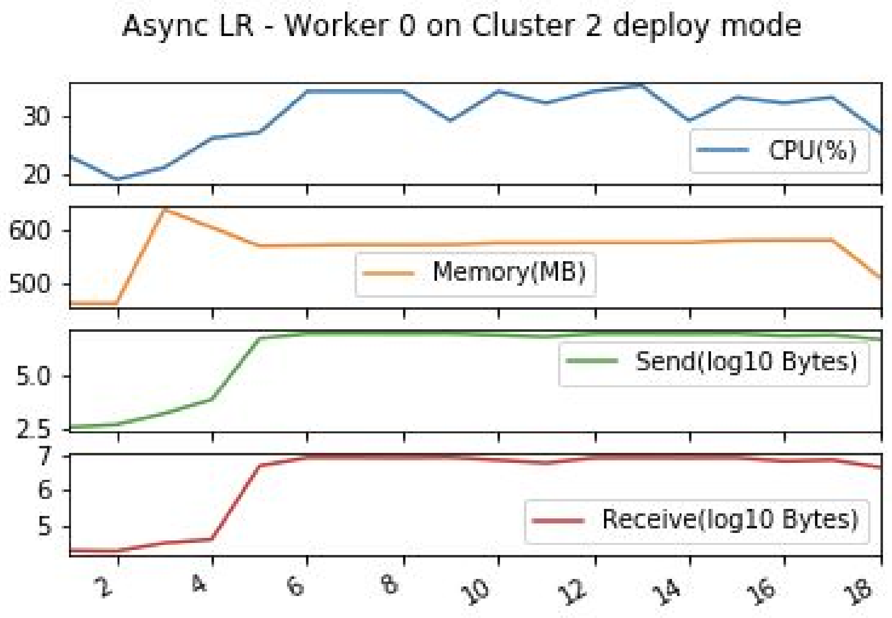

Figure 5 - Async LR training on Worker 0 in Cluster 2

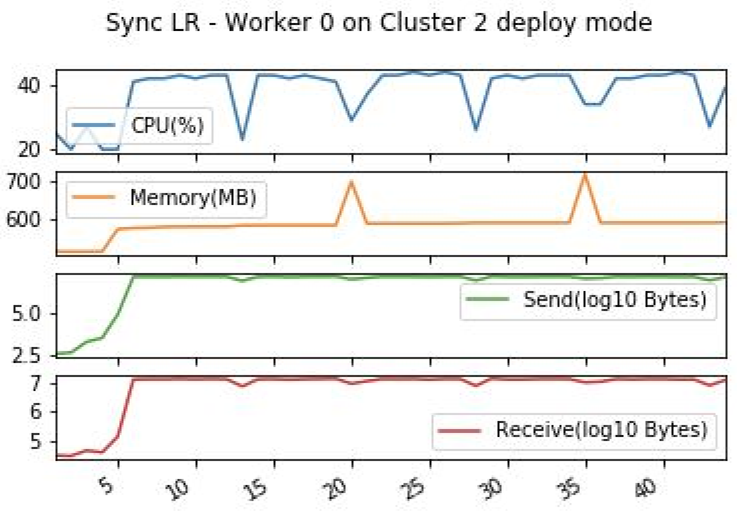

Figure 6 - Sync LR training on Worker 0 in Cluster2

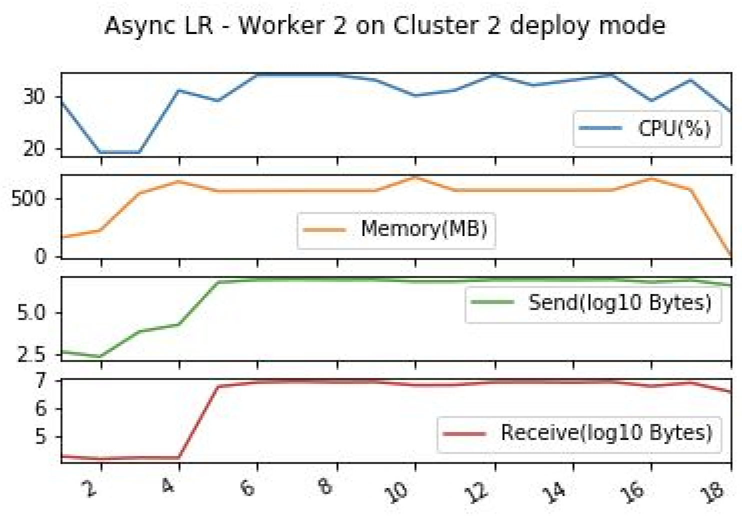

Figure 7 - Async LR training on Worker 2 in Cluster 2

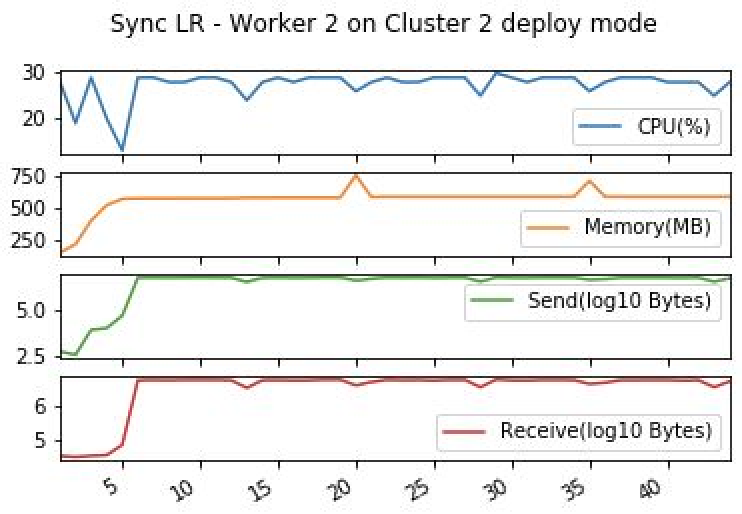

Figure 8 - Sync LR training on Worker 2 in Cluster 2

From the measurements we make the following inferences:

1. In async mode the workers apply their gradients independently and do not have to wait for other workers and therefore their CPU usage profile looks similar across workers.
2. In sync training mode we are observing dips in CPU usage of the workers which might correspond to the worker waiting for the global step token to be pushed to the queue.

2. *Part-2 - MNIST Classification with LeNet using Keras API*

2.1 Task 1 - Implementation of LeNet Architecture Using Keras API

2.1.1 Model architecture:

We implemented the LeNet neural network architecture using Keras APIs based on the description presented in the original paper. While we have tried to maintain the network layers as close as possible to the original paper, following tweaks have made been based on our experimental observations and implementation convenience.

1. In the original paper, the given architecture assumes the input to be of size 32x32. As we have 28x28 inputs in MNIST dataset, we have slightly reduced the convolution filter kernel sizes instead of doing zero padding to the inputs.
2. We chose to use ‘relu’ activation function instead of ‘tanh’ since it lead to better model convergence. With the choice of our other hyperparameters, ‘relu’ gave above 90% accuracy in about 20 epochs whereas ‘tanh’ would take nearly 50 epochs to achieve similar accuracy.
3. In the paper, some custom layer design was adopted in certain sub-sampling layers. However, we have only used the AveragePooling2D layer of Keras due to the ease of model implementation.

2.1.2 Distributed training

We used the *MultiWorkerMirroredStrategy* for distributed training. We do that by building and compiling our Keras model within the scope of our chosen strategy and then using the `model.fit()` API with the constructed `tf.dataset` to start the training. In order to enable fault tolerance to the training mechanism, we provide a *callback* function to `fit()`, that will periodically checkpoint the model state to the disk. The callback is as given below:

```
callbacks=
[tf.keras.callbacks.ModelCheckpoint(filepath='./lenet_logs/keras-ckpt')]

with strategy.scope():
  multi_worker_model = build_model()
multi_worker_model.fit(x=train_datasets,epochs=FLAGS.epochs,
steps_per_epoch=FLAGS.steps_per_epoch, callbacks=callbacks)
```

We have relied on the default auto data sharding policy of the above strategy. To enable the training to take place on multiple workers for sufficiently longer epochs, we used the `repeat()` transform on the MNIST train data. `repeat()` will enable the workers to loop through the same shard of data that is allocated to them in a repeated fashion thus enabling us to perform distributed training for any number of epochs.

```
train_datasets_unbatched=datasets['train'].map(scale).cache().repeat().shuffle(
BUFFER_SIZE)
```

2.1.3 TF_CONFIG Setup:

We have defined three deploy modes that is given as an argument to our python script. Each deploy mode specifies how many machines are being used for the distributed training. For automation of our experiments, we included the TF_CONFIG relevant to each of our deploy modes inside our code and choose the appropriate one based on input argument.
Following is snippet of relevant code:

```
TF_CONFIG_ONE=
{'cluster':{'worker':['10.10.1.1:2223']},'task':{'index':FLAGS.task_index,'type
':'worker'}}
TF_CONFIG_TWO=
{'cluster':{'worker':['10.10.1.1:2223','10.10.1.2:2222']},'task':{'index':FLAGS
.task_index,'type':'worker'}}
TF_CONFIG_THREE=
{'cluster':{'worker':['10.10.1.1:2223','10.10.1.2:2222','10.10.1.3:2222']},'tas
k':{'index':FLAGS.task_index,'type':'worker'}}

Config_list = [TF_CONFIG_ONE, TF_CONFIG_TWO, TF_CONFIG_THREE]
os.environ['TF_CONFIG'] = json.dumps(Config_list[FLAGS.deploy_mode - 1])
```

2.1.4 Hyperparameters:

As any neural network performance is highly influenced by the choice of various tunables, we present our hyperparameters and our observations.

1. *Buffer size for shuffle*: 10000
2. *Loss function:* sparse_categorical_crossentropy
3. *Optimizer*: SGD
4. *Activation function:* ‘relu’, ‘softmax’
   ‘relu’ is used to implement the non-linearity after every layer except the output layer. Since it is a classification problem we have ‘softmax’ nonlinearity at the output layer to make label prediction.
5. *Epochs:* 50 - For this part of the assignment, we have fixed our Epochs value to 50. We observed that similar to the Logistic regression training, the accuracy

keeps increasing with increased number of epochs. From the default logging mode: `tf.logging.DEBUG`, we were able to check the model accuracy at the end of every epoch. We observed an accuracy of about 93% after 20 epochs and it converged to approximately 96% at the end of 50 epochs.

6. *steps_per_epoch*: 1000 - We observed that the choice of this parameter largely influences the model performance, particularly in the setting of distributed training. When we set this parameter to 375, the performance of the model was very poor that the accuracy saturated at nearly 85% even after 50 epochs. Increasing this one to 1000 resulted in the accuracies we mentioned above.

   One possible reason for this behavior is that when only 375 steps are given per epoch, the dataset received by all the workers combined is missing to include many training points in the training loop. We think that this parameter has to be approximately close to `num_training_examples/batch_size` so that with higher confidence we can assume that every point in the training data is included in the training loop.

2.1.5 System monitoring:

As in part 1, to monitor the CPU, Memory, Network usage in each of the deploy modes for various batch sizes, we used dstat tool to monitor the CPU and memory usage. We used the following command to get the relevant stats at every 5 seconds. As long as our python process is executing, we observed that it is the top memory and cpu consuming process. Hence, following command does a good job in posting the stats to .csv file which we processed later to get more insights.

```
dstat --time -c --top-cpu -dn --top-mem --output dstat-0-deploy-$2-batch-$3.csv
5
```

As mentioned previously we have conducted all our experiments for this task with 50 epochs. So, the time axis in the below plots represents the duration for these 50 epochs. The labels in the plots clearly express the batch size and deploy mode for each of the figures. Read any plot label as follows:

dstat-<node id>-deploy-<deploy mode>-batch-<batch size>.csv

Where:

- node id $\in \{0, 1, 2\}$
- deploy mode $\in \{1, 2, 3\}$
- batch size $\in \{32, 64, 128, 256\}$

2.1.6 Impact of number of workers on cpu usage of each worker:

We have observed that, as we increase the number of workers for distributed training, the cpu usage on each worker reduces. This could be because of lesser percentage of compute cycles being utilized by each node as some cpu cycles are wasted in the waiting that is involved in a synchronous model update. This is clearly evident from the following plots showing percentage of cpu consumed by our tensorflow application with time.

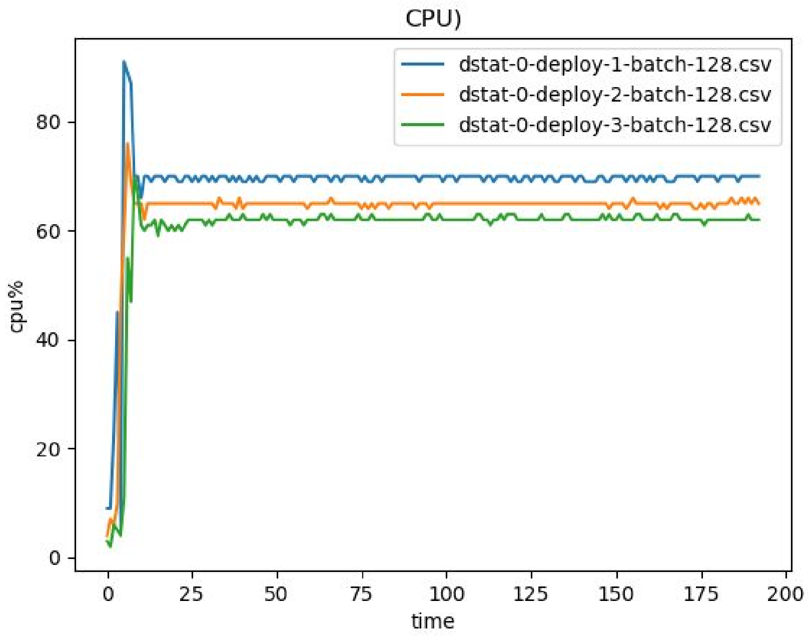

Figure 9 CPU usage on node0 with batch size 128 and three deploy modes

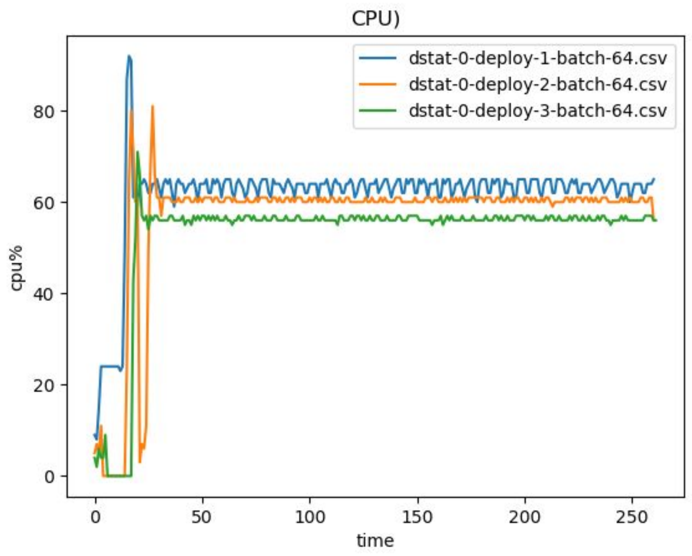

Figure 10 CPU usage on node0 with batch size 64 and three deploy modes

*2.1.7 Impact of number of workers on memory usage***:**

Similar to the CPU usage, memory usage also reduces as we increase the number of workers of distributed training. This could be attributed to the lesser amount of cache that the process will use as each worker would need to operate on lesser amount of the data when more workers are available in total. The figure is shown below

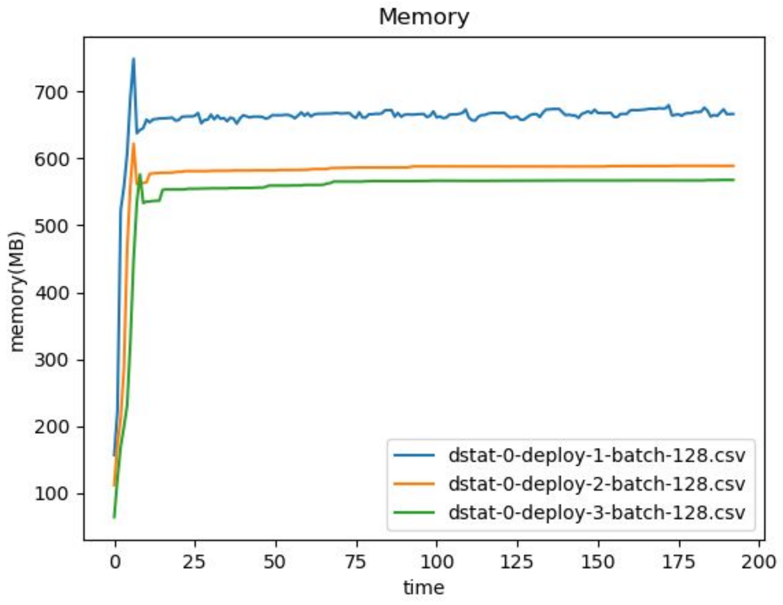

Figure 13 Memory usage on node 0 with batch size 128

Network usage:
We plot the network usage data in logarithmic scale for better visualization of the results. Plots in Figure 15 below shows the data sent and received over the network by each worker node (*the given numbers represent whole of the data sent or received by all the processes running on the worker*)

2.1.8 Impact of number of workers on network usage:

Adding more workers to the distributed training is very helpful when the application needs to be trained on very large training datasets that are stored in some global file system. This is because a single machine may not be able to store all the data and training over vast data by a single machine takes a lot of time.

However, we can only derive those benefits provided we have very good network bandwidth. This is because as the number of workers increases more parameters as well as data needs to be sent over the network for model synchronization among the workers. We have observed a large increment in network usage from single to multiple workers training mode even with a model as simple as LeNet.

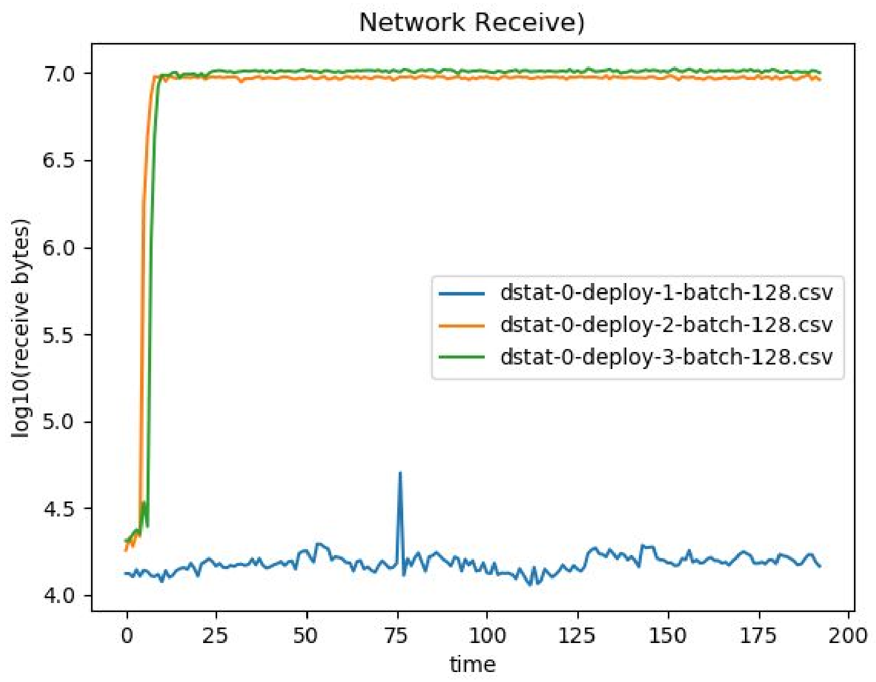

Figure 15 network receive bytes on node 0 with batch size 128

2.2 Task 2 - Impact of Varying Batch Sizes

We conducted several experiments with varying batch sizes and collected CPU, Network and memory usage statistics. These measurements are presented below.

*2.2.1 Impact of batch size on CPU Usage**:***

As the batch size is increased, more CPU computations need to be performed over the available data to complete the batch. Thus CPU usage goes up which is evident in Figure 11 shown below

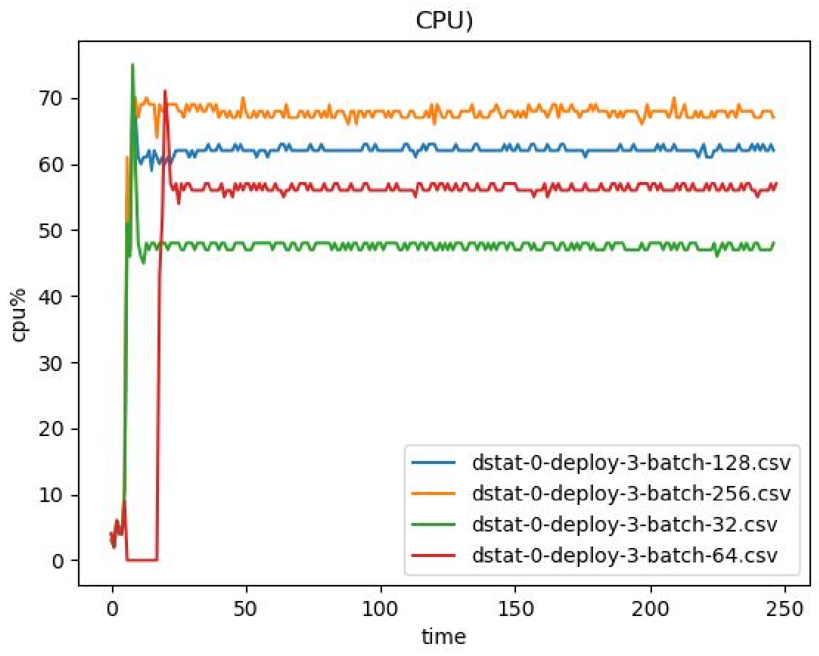

Figure 11 CPU usage for node 0 on 3 machine cluster

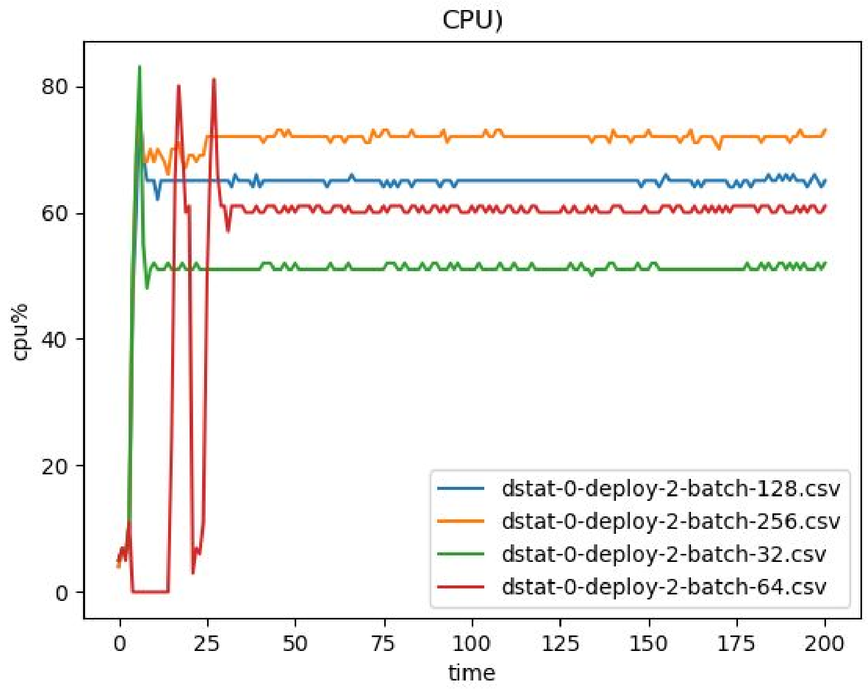

Figure 12 CPU usage on node 0 for 2 machine cluster

2.2.2 Impact of batch size on memory usage:

As expected, memory usage will increase as the batch size is increased since whole batch of data is loaded into the memory in order to perform gradient updates in every iteration of every epoch. This is evident from the figure 14 shown below

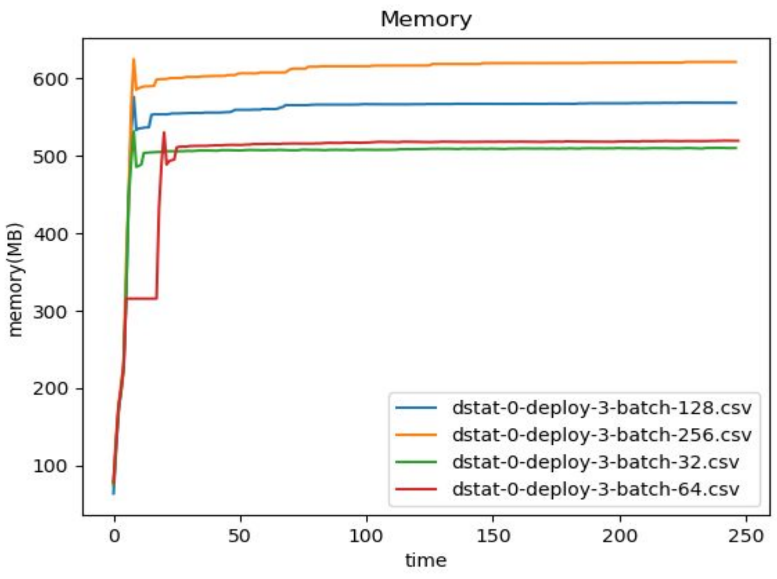

Figure 14 memory usage on node 0 for a 3 machine cluster

2.2.3 *Impact of batch size on network usage:*

We have observed that in the distributed training at every iteration, model updates are exchanged over the network. So, we might believe that batch size will not have any impact on the network usage since the process of batching of a worker's shard of the training data happens within the worker itself.

However this is not the case in practice; network bandwidth usage has an interesting dependency on the batch size. We have observed in the CPU Usage section that as the batch size increases, the number of compute cycles the application takes on each batch of data also increases. This implies, every iteration takes a longer time to finish and so the number of model updates sent over the network would decrease when we increase the batch size. This phenomenon is evident from the figure 16 shown below as LeNet and with a dataset as small and not so complex as MNIST.

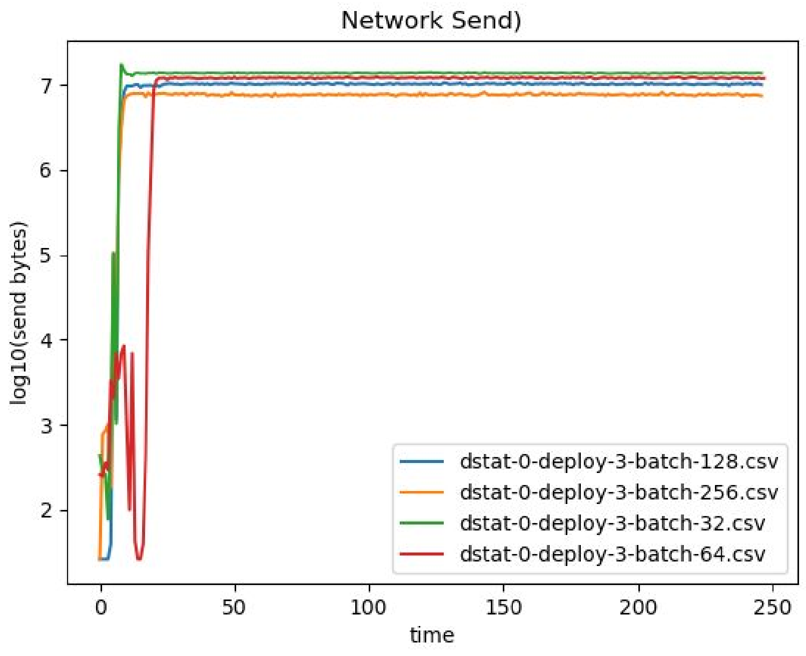

Figure 16 network send bytes on node0 with 3 machine cluster

2.2.4 Inferences from an Interesting Memory, Disk Read and CPU usage Event

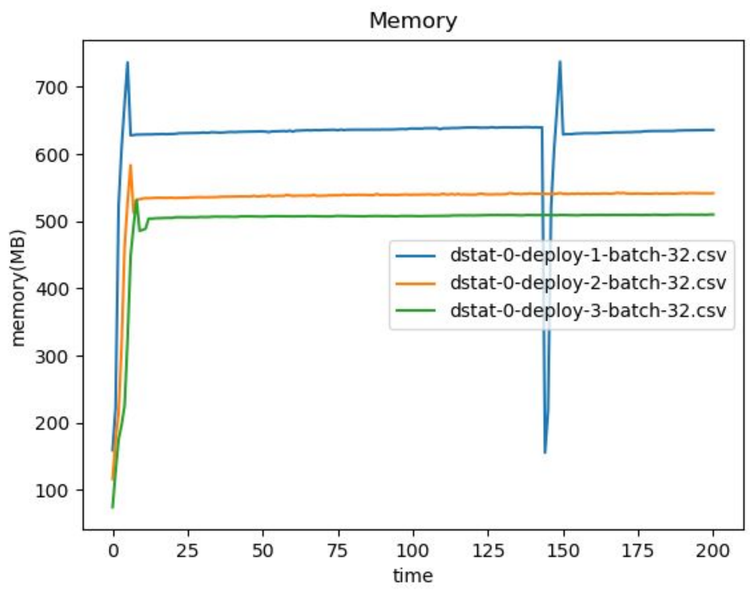

Figure 17 - Memory usage plot showing the event at time 150

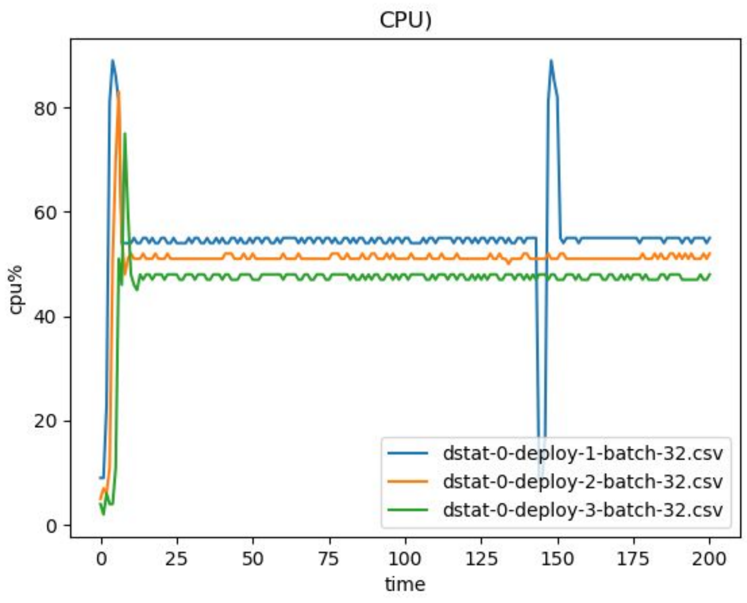

Figure 18 - CPU usage plot showing the event at time 150

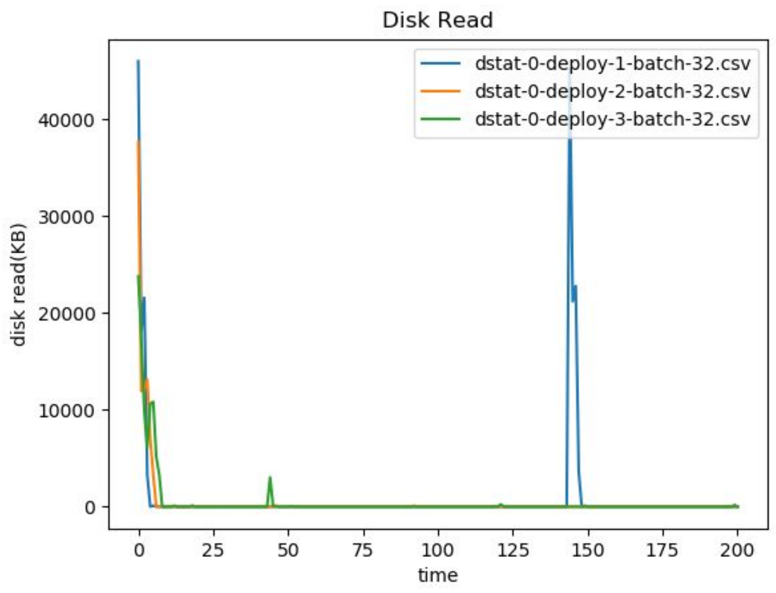

Figure 19 - Disk read plot showing the event at time 150

Around some time indexed by 150 in the above plots, we see a sharp fall and rise in the CPU and memory usage by tensorflow application on node0. In addition, on the same node, the rise in memory usage after it's steep fall is associated with a simultaneous peak in the disk read data. We hypothesize that the following sequence of steps could be at play on node0 around time index 150, that could have lead to the observed effect on various system stats:

i. As we see that process memory is being deallocated, we believe there could be some memory corruption issue or some other fault that enforces the driver program to wipe out the memory corresponding to our tensorflow application. So, there is a fall in the 'Memory' plot.

ii. This implies, our application cannot make use of the cpu compute cycles until the memory corresponding to it is restored. So, we observe a dip in the cpu usage.

iii. Since we have provided a callback to the `model.fit()` API to periodically checkpoint the model state, driver program can restore the state from disk and that could have led to the sharp peak that we observe in the below 'Disk read' plot.

(iv) And once the memory is restored from the disk, cpu usage peaks to compute the operations that are required to restore any lost objects and later it stabilizes to the original percentage.