# CS 744 Assignment 2

Siddhant Garg
*sgarg33@wisc.edu*

Varun Batra
*vbatra@wisc.edu*

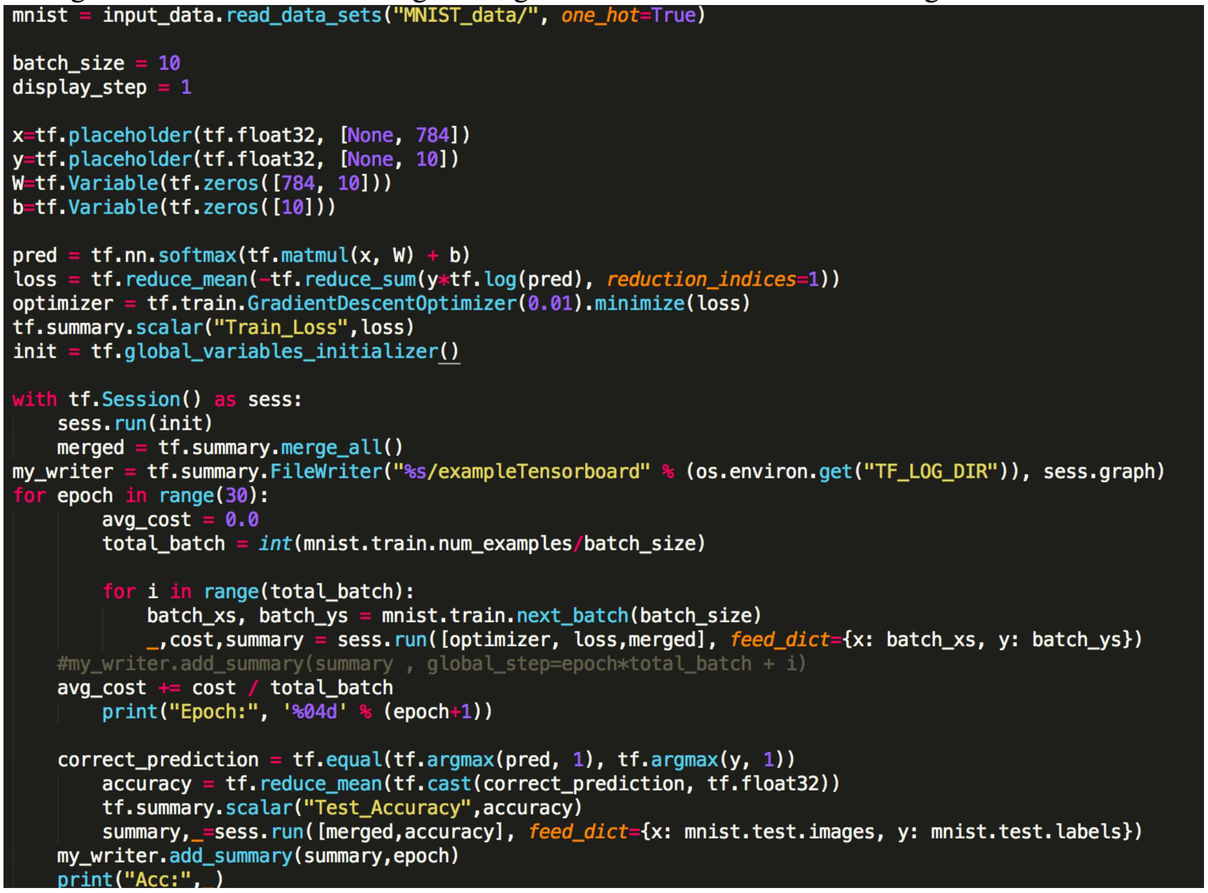
Rahul Jayan
*jayan@wisc.edu*

October 17, 2018

## 1 Part 1: Logistic Regression

### 1.1 Single Node

Following is the concise code for logistic regression in TensorFlow for a single node:



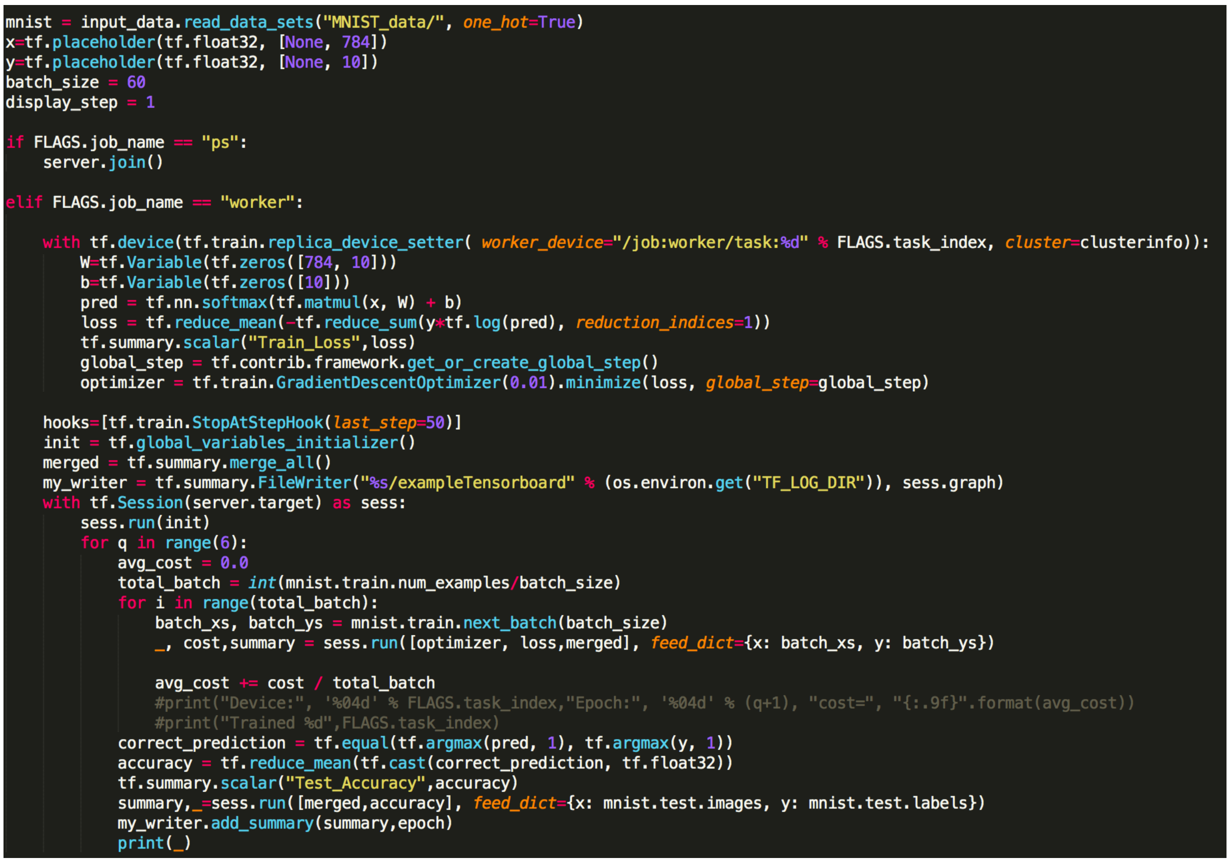
```
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

batch_size = 10
display_step = 1

x=tf.placeholder(tf.float32, [None, 784])
y=tf.placeholder(tf.float32, [None, 10])
W=tf.Variable(tf.zeros([784, 10]))
b=tf.Variable(tf.zeros([10]))

pred = tf.nn.softmax(tf.matmul(x, W) + b)
loss = tf.reduce_mean(-tf.reduce_sum(y*tf.log(pred), reduction_indices=1))
optimizer = tf.train.GradientDescentOptimizer(0.01).minimize(loss)
tf.summary.scalar("Train_Loss",loss)
init = tf.global_variables_initializer()

with tf.Session() as sess:
    sess.run(init)
    merged = tf.summary.merge_all()
my_writer = tf.summary.FileWriter("%s/exampleTensorboard" % (os.environ.get("TF_LOG_DIR")), sess.graph)
for epoch in range(30):
        avg_cost = 0.0
        total_batch = int(mnist.train.num_examples/batch_size)

        for i in range(total_batch):
            batch_xs, batch_ys = mnist.train.next_batch(batch_size)
            _,cost,summary = sess.run([optimizer, loss,merged], feed_dict={x: batch_xs, y: batch_ys})
    #my_writer.add_summary(summary , global_step=epoch*total_batch + i)
    avg_cost += cost / total_batch
        print("Epoch:", '%04d' % (epoch+1))

    correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
        accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
        tf.summary.scalar("Test_Accuracy",accuracy)
        summary,_=sess.run([merged,accuracy], feed_dict={x: mnist.test.images, y: mnist.test.labels})
    my_writer.add_summary(summary,epoch)
    print("Acc:",_)
```

We created a single tf session and trained the graph using training data from MNIST in a round robin manner by choosing the next batch after one batch was processed. We also incorporated summary writing by initializing File Writer and then adding scalars to the summaries after each training step. This helped us visualize how the training loss decreased and test accuracy increased with training iterations using Tensorboard.

## 1.2 Distributed TensorFlow

Following is the concise code for logistic regression in TensorFlow for a a distributed setting:

### 1.2.1 ASYNC MODE



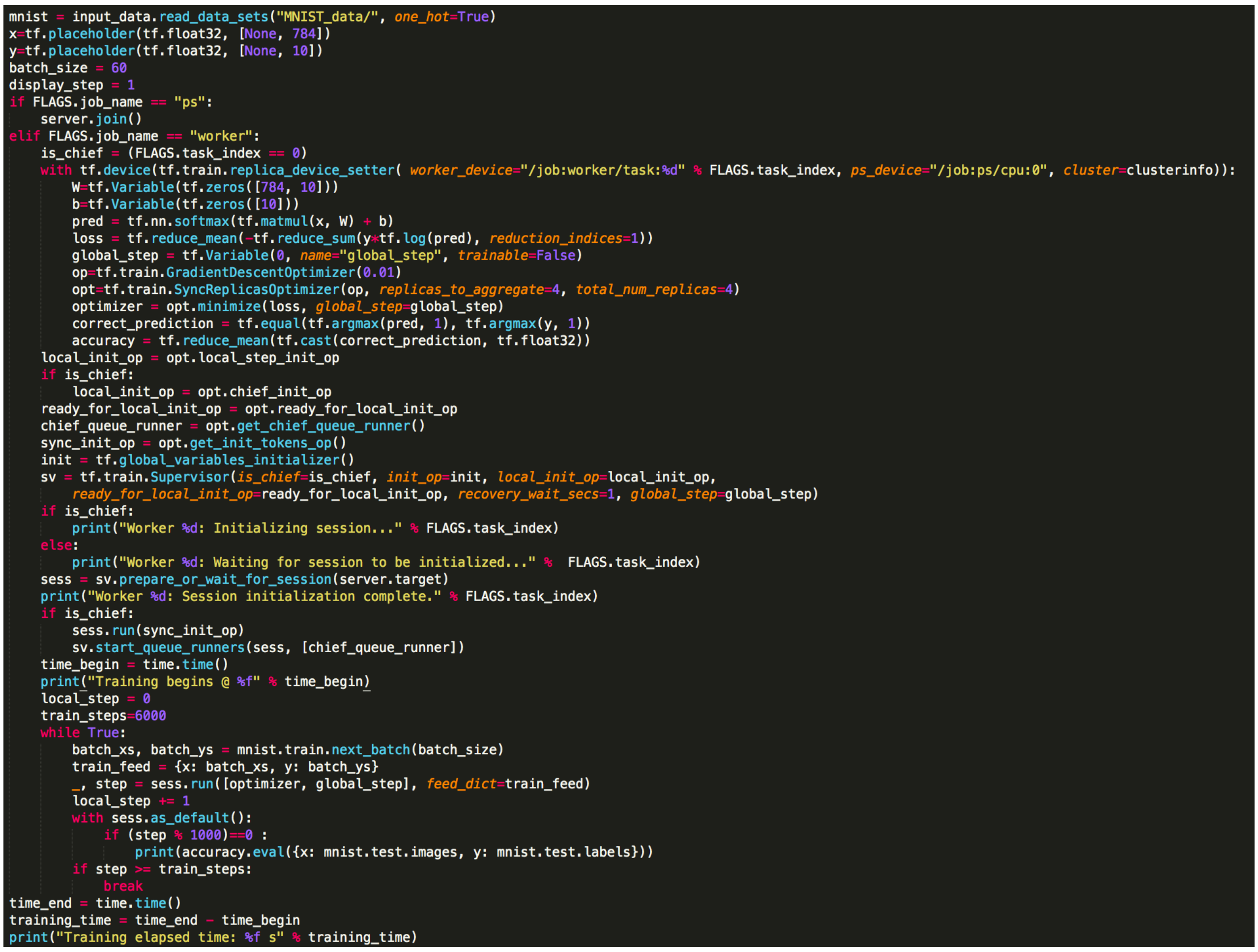
```python
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
x=tf.placeholder(tf.float32, [None, 784])
y=tf.placeholder(tf.float32, [None, 10])
batch_size = 60
display_step = 1

if FLAGS.job_name == "ps":
    server.join()

elif FLAGS.job_name == "worker":

    with tf.device(tf.train.replica_device_setter( worker_device="/job:worker/task:%d" % FLAGS.task_index, cluster=clusterinfo)):
        W=tf.Variable(tf.zeros([784, 10]))
        b=tf.Variable(tf.zeros([10]))
        pred = tf.nn.softmax(tf.matmul(x, W) + b)
        loss = tf.reduce_mean(-tf.reduce_sum(y*tf.log(pred), reduction_indices=1))
        tf.summary.scalar("Train_Loss",loss)
        global_step = tf.contrib.framework.get_or_create_global_step()
        optimizer = tf.train.GradientDescentOptimizer(0.01).minimize(loss, global_step=global_step)

    hooks=[tf.train.StopAtStepHook(last_step=50)]
    init = tf.global_variables_initializer()
    merged = tf.summary.merge_all()
    my_writer = tf.summary.FileWriter("%s/exampleTensorboard" % (os.environ.get("TF_LOG_DIR")), sess.graph)
    with tf.Session(server.target) as sess:
        sess.run(init)
        for q in range(6):
            avg_cost = 0.0
            total_batch = int(mnist.train.num_examples/batch_size)
            for i in range(total_batch):
                batch_xs, batch_ys = mnist.train.next_batch(batch_size)
                _, cost,summary = sess.run([optimizer, loss,merged], feed_dict={x: batch_xs, y: batch_ys})

                avg_cost += cost / total_batch
                #print("Device:", '%04d' % FLAGS.task_index,"Epoch:", '%04d' % (q+1), "cost=", "{:.9f}".format(avg_cost))
                #print("Trained %d",FLAGS.task_index)
            correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
            accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
            tf.summary.scalar("Test_Accuracy",accuracy)
            summary,_=sess.run([merged,accuracy], feed_dict={x: mnist.test.images, y: mnist.test.labels})
            my_writer.add_summary(summary,epoch)
            print(_)
```

For each device corresponding to a worker, we initialize the computation of the loss and the optimizer (Since tf variables are globally shared, there is no need of adding them to the ps task). Since the default parameter sharing implemented in TensorFlow is asynchronous, we just create a single session and let each worker run one fourth of the total epochs that a single node would have run (to pass the training data the same number of times as it would have in the single node case).

We observed on running the async mode that the node0(which also has the ps task running) finishes its allocated execution in the prescribed amount of time (one fourth of the single node case), but the other nodes take more time to complete the execution and end the task (on exploring this more using tools like dstat, we observed that the unexpected overhead in the execution time is due to the GRPC protocol of communicating parameters between nodes as used by TensorFlow).

### 1.2.2 SYNC MODE

For each device corresponding to a worker, we initialize the computation of the loss and the SyncReplicasOptimizer (Since tf variables are globally shared, there is no need of adding them to the ps task). We define the chief worker as the worker node which is also executing the ps task (node-0 in our case). We initialize several TensorFlow ops and create a global step. Here on we run the session in a while loop (on the global step) so that all the workers execute synchronously to reach the total global steps passes over the train data.



```
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
x=tf.placeholder(tf.float32, [None, 784])
y=tf.placeholder(tf.float32, [None, 10])
batch_size = 60
display_step = 1
if FLAGS.job_name == "ps":
    server.join()
elif FLAGS.job_name == "worker":
    is_chief = (FLAGS.task_index == 0)
    with tf.device(tf.train.replica_device_setter( worker_device="/job:worker/task:%d" % FLAGS.task_index, ps_device="/job:ps/cpu:0", cluster=clusterinfo)):
        W=tf.Variable(tf.zeros([784, 10]))
        b=tf.Variable(tf.zeros([10]))
        pred = tf.nn.softmax(tf.matmul(x, W) + b)
        loss = tf.reduce_mean(-tf.reduce_sum(y*tf.log(pred), reduction_indices=1))
        global_step = tf.Variable(0, name="global_step", trainable=False)
        op=tf.train.GradientDescentOptimizer(0.01)
        opt=tf.train.SyncReplicasOptimizer(op, replicas_to_aggregate=4, total_num_replicas=4)
        optimizer = opt.minimize(loss, global_step=global_step)
        correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
        accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
    local_init_op = opt.local_step_init_op
    if is_chief:
        local_init_op = opt.chief_init_op
    ready_for_local_init_op = opt.ready_for_local_init_op
    chief_queue_runner = opt.get_chief_queue_runner()
    sync_init_op = opt.get_init_tokens_op()
    init = tf.global_variables_initializer()
    sv = tf.train.Supervisor(is_chief=is_chief, init_op=init, local_init_op=local_init_op,
        ready_for_local_init_op=ready_for_local_init_op, recovery_wait_secs=1, global_step=global_step)
    if is_chief:
        print("Worker %d: Initializing session..." % FLAGS.task_index)
    else:
        print("Worker %d: Waiting for session to be initialized..." %  FLAGS.task_index)
    sess = sv.prepare_or_wait_for_session(server.target)
    print("Worker %d: Session initialization complete." % FLAGS.task_index)
    if is_chief:
        sess.run(sync_init_op)
        sv.start_queue_runners(sess, [chief_queue_runner])
    time_begin = time.time()
    print("Training begins @ %f" % time_begin)
    local_step = 0
    train_steps=6000
    while True:
        batch_xs, batch_ys = mnist.train.next_batch(batch_size)
        train_feed = {x: batch_xs, y: batch_ys}
        _, step = sess.run([optimizer, global_step], feed_dict=train_feed)
        local_step += 1
        with sess.as_default():
            if (step % 1000)==0 :
                print(accuracy.eval({x: mnist.test.images, y: mnist.test.labels}))
        if step >= train_steps:
            break
time_end = time.time()
training_time = time_end - time_begin
print("Training elapsed time: %f s" % training_time)
```

Since training data for MNIST is 60000, for a batch size of 60, the number of batches in one epoch is 1000. So we can ideally compare the async and sync modes to have trained over the train data equal number of times as follows: 1 epoch of async mode = 1000 global steps of sync mode [ We ran our experiments using 6 epochs of async mode and 6000 global steps of sync mode so that both the sync and async modes achieves convergence in test accuracy.

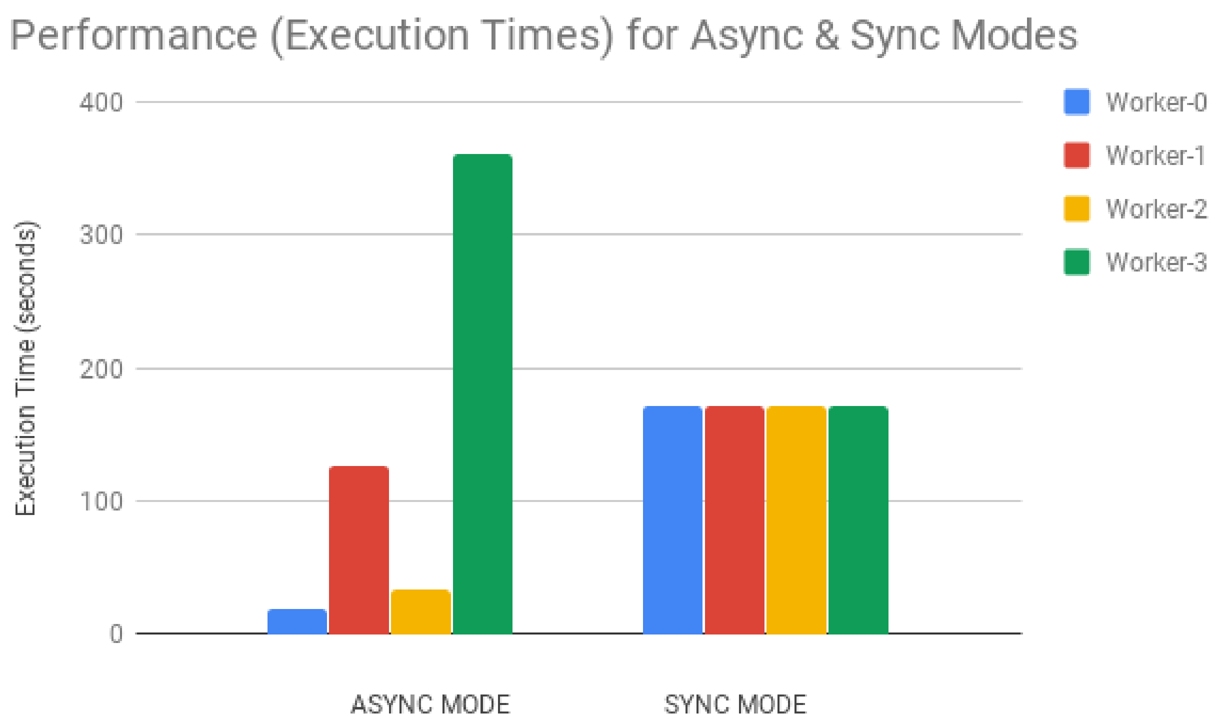
## 1.3 Task-1

We have provided the code for the single-node logistic regression, async and sync distributed logistic regression in the submission folder.

## 1.4 Task-2

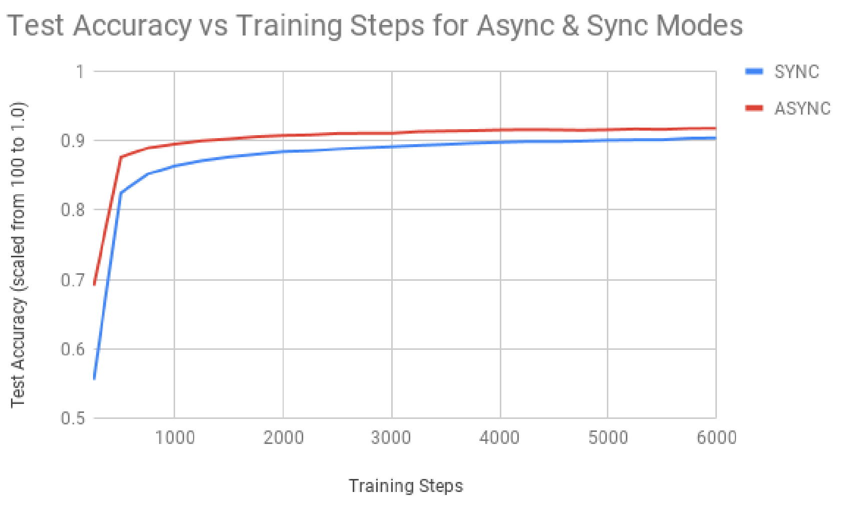
For these experiments, we fixed the batch size to 60 which we observed was giving the maximum test accuracy in all the different cluster scenarios.

For analysis of performance, we ran the code for the same number of training data passes (6 epochs for Async and 6000 global steps for Sync mode) and recorded the execution times for all the processes [The execution time we captured is the time between creating the graph and exiting from the program and closing the session].



We observe from this experiment that different worker machines in the Aync mode have different execution times even though each performs exactly the same computation over the training data whereas the execution times for all machines in the Sync mode is the same (This is in accordance with what is theoretically expected). We also observe that there is a large disparity in the execution times within the different worker nodes in the Async node where node-0 (which also has the ps process) finishes execution the fastest (due to no network overheads of exchanging the parameters) while other nodes require more time to finsih execution. Running the experiment multiple times lead us to observe that every time different machines (out of node1, node2 and node3) take the longest to finish and the large delay is due to overhead of GRPC protocol.

For analysis of test accuracy, we ran the code for the same number of training data passes (6 epochs for Async and 6000 global steps for Sync mode) and printed the test accuracy periodically for both the Async and Sync modes.



From the above graph, we can infer that Async mode reaches a slightly better test accuracy ( 1%) than the Sync mode (even after convergence) [We observed this even on varying batch sizes]. Convergence in test accuracy is similar for both the modes with convergence achieved after around 600-800 training steps.

We also used dstat to monitor the CPU usage, memory and network congestion between the nodes and how it changes for Async and Sync modes. We present the results in the table below (Using dstat only on node-0 was sufficient for our experiment as for cpu and memory usage it would be the one having the highest values due to both the ps and worker task running on it. Also the network bandwidth for node0 was three times that each of node1, node2 and node3 as was observed in our experiments).

Table 1: CPU and Memory usage for Async and Sync modes

| Mode | Max CPU Usage | Avg CPU Usage | Memory Usage |
|---|---|---|---|
| ASYNC | 4.212 | 7 | 1916MB |
| SYNC | 1.27 | 4 | 2146MB |

Table 2: Network utilization for Async and Sync modes

| Mode | Total Read | Total Write | Avg Read Throughput | Avg Write Throughput |
|---|---|---|---|---|
| ASYNC | 585 MB | 537 MB | 17.72 MBps | 16.27 MBps |
| SYNC | 617 MB | 614 MB | 3.65 MBps | 3.63 MBps |

Based on these observations, we conclude that for the async mode, the bottleneck is due to the network read and writes (This is also in accordance with what we were expecting to observe since in async mode, there is frequent fetching and updating the weight parameters since all the workers are operating independently). Further we observed for async that for the particular worker node which was behaving as a straggler (taking the longest time to finish), the cpu and memory usage was much less than the maximum allowed and hence the bottleneck was due to the network read and writes. For the sync mode as well, the network reads and writes were the bottleneck (Though the average network read and writes were higher for the async mode than the sync mode).

## 1.5 Task-3

We used different batch sizes and observed the convergence of test accuracies for the Async and Sync modes. We have compiled the results in the following tables and graphs. For Async mode, as the batch size increases, the time taken to complete the same number of epochs decreases linearly (since now less number of batches of train data need to be processed). For Sync mode, we present times taken to execute the same number of global steps (which remain the same across different batch sizes and the model has to run tf.session this many number of times in all the batch sizes)

Table 3: Execution times for 20 epochs for Async Mode across batch sizes

| Batch Size | Execution Time (seconds) |
|---|---|
| 10 | 291.9 |
| 30 | 115.9 |
| 60 | 66.9 |
| 120 | 44.1 |

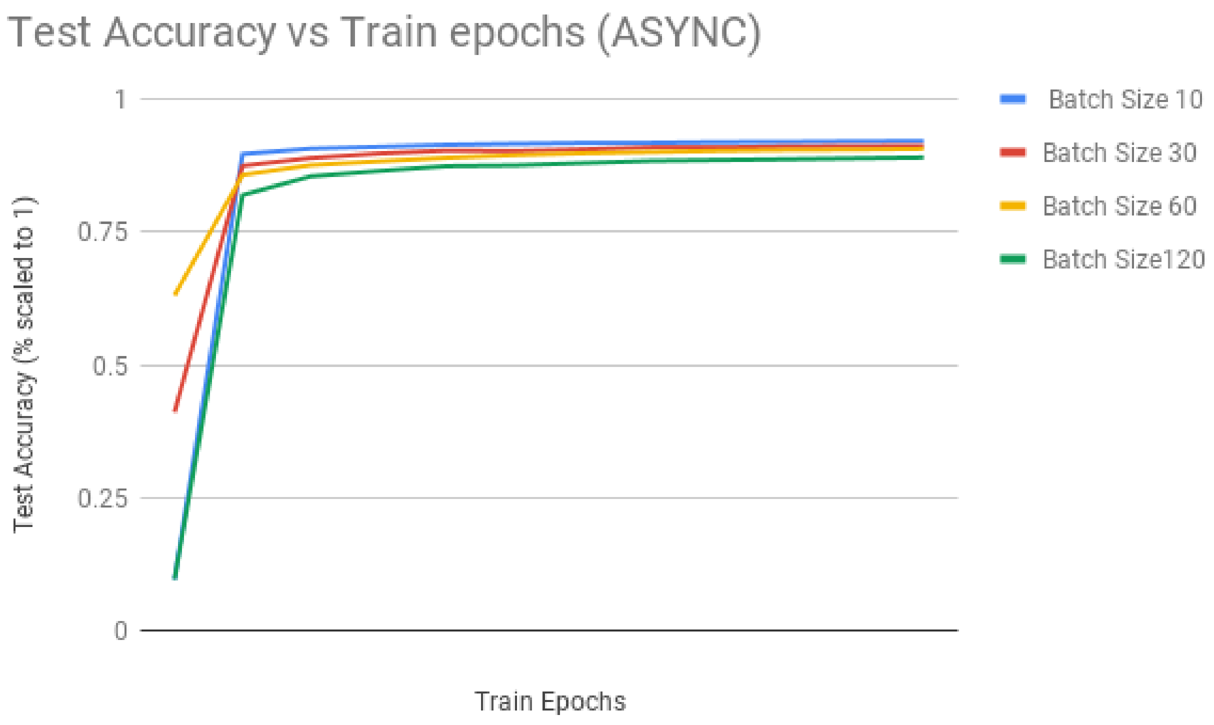
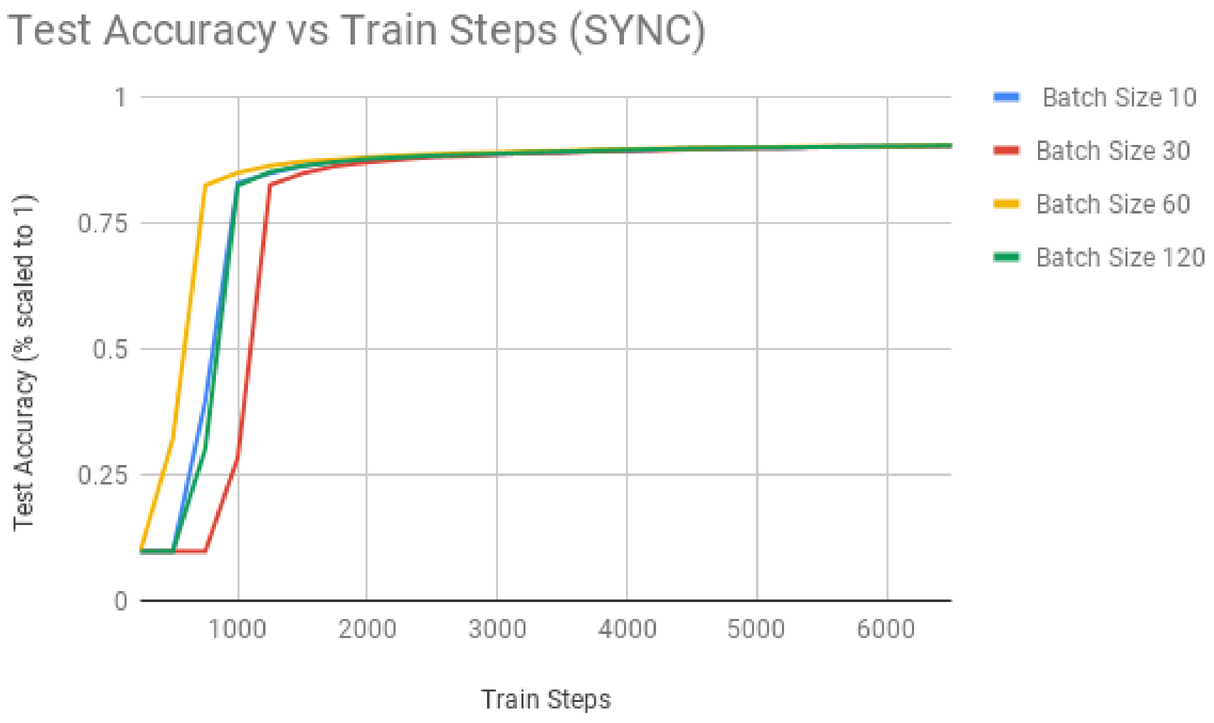


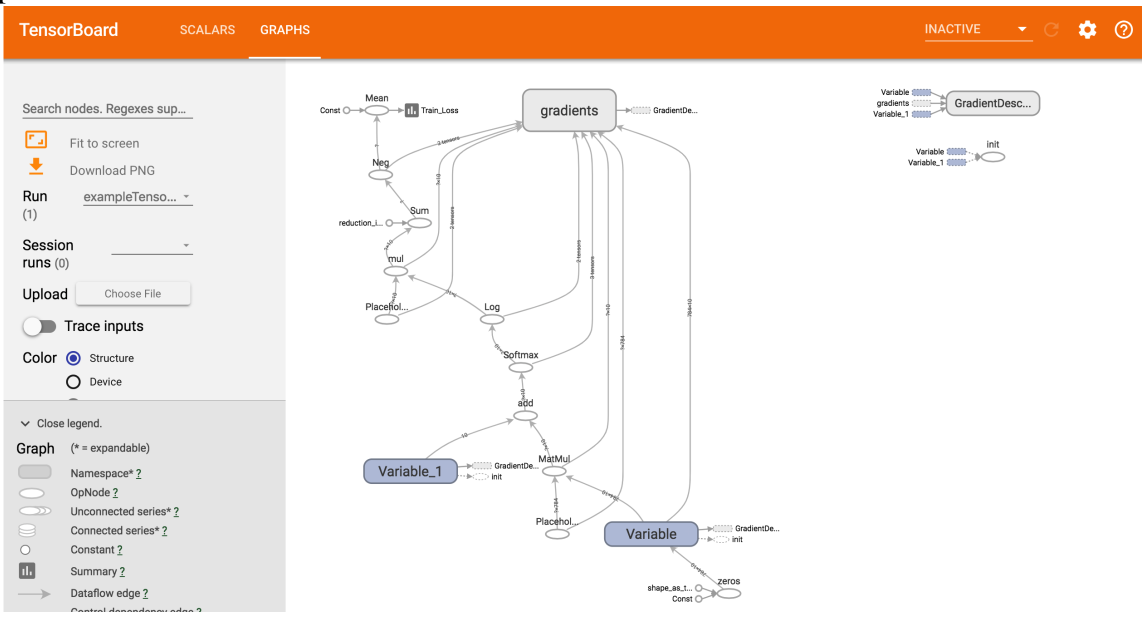
Table 4: Execution times for 6000 global steps for Sync Mode across batch sizes

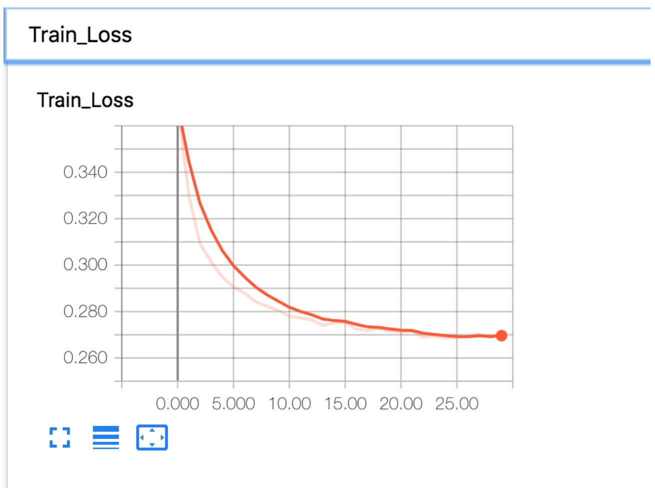
| Batch Size | Execution Time (seconds) |
|---|---|
| 10 | 163.8 |
| 30 | 177.4 |
| 60 | 172.5 |
| 120 | 181.2 |

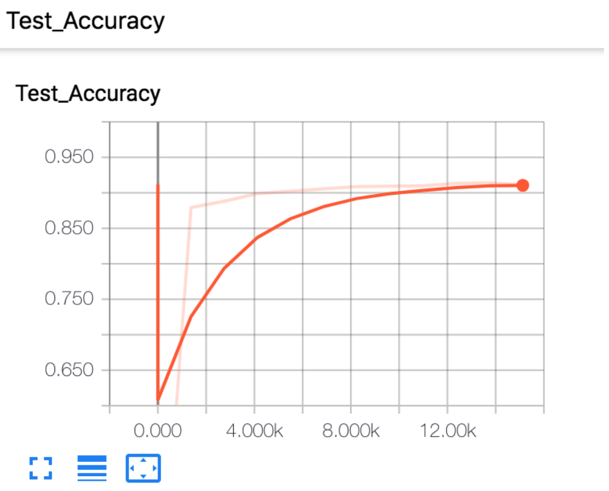


## 1.6 Task-4

We used Tensorboard to visualize the TensorFlow graph and metric plots. We present some of the graphs below:





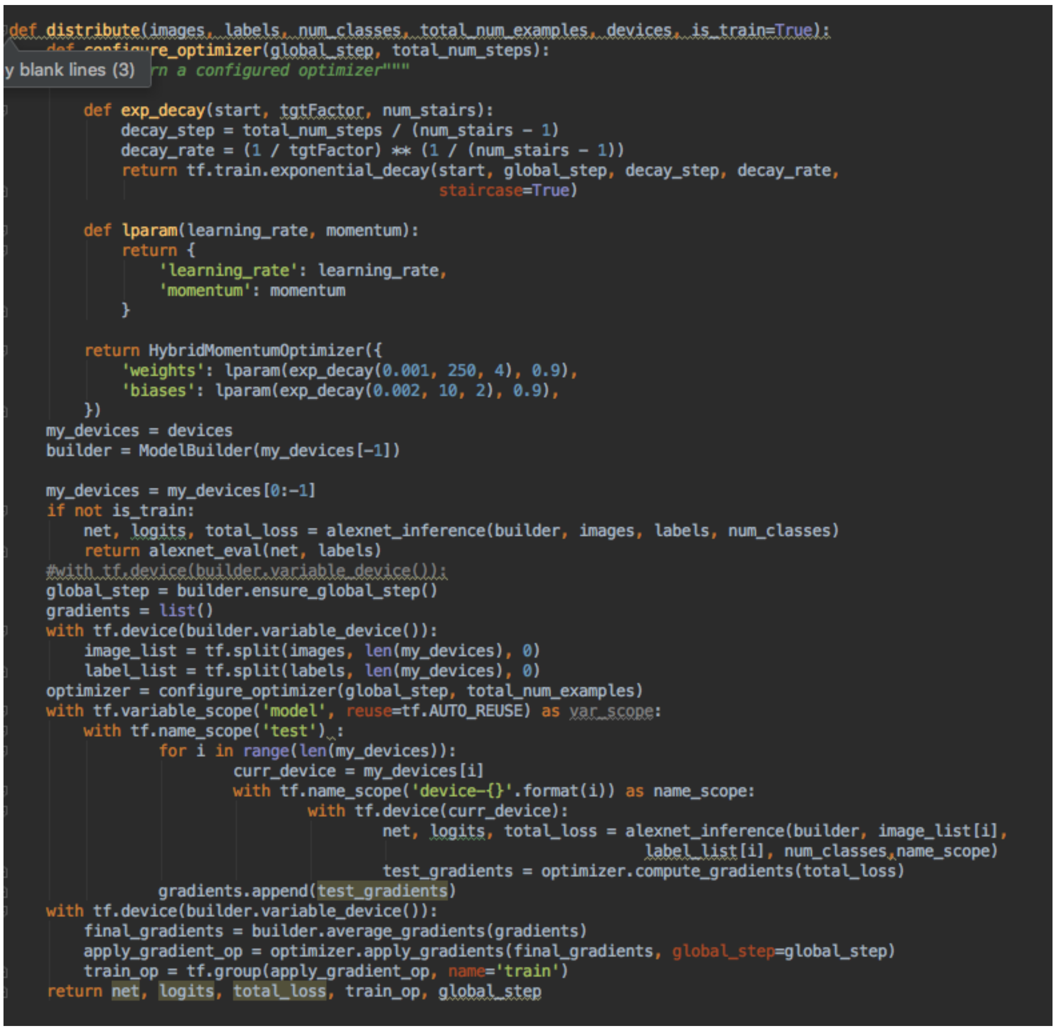


# 2 Part 2: AlexNet

## 2.1 Introduction

This is the part where it was asked to get a feel of how Alexnet works. We had to write a custom distribute function to distribute the code that was working correctly for a single node. This required us to get familiar with the code base and correspondingly implement the distribute function.

We started by changing the code to make it work correctly for a single machine by adjusting the minor glitches in the code base. Next, we had to understand how to provide the distributed configuration information like number of machines to run on, the address of the machines, batch size, number of batches, etc. We found the ”train.py” file in the scripts folder to contain the majority of configurational and implementation information. It had variables like batch size, labels and image information, the set of available devices and their config, etc. We did not modify the optimizer and focused on dividing our code on the devices available. The last device was set to have worker as well as PS tasks, and the other 3 as just workers. Our model builder takes in the PS device. Next, we split our IMAGES and LABELS vectors based on the number of devices. We also use variable and name scope functionality of TensorFlow. We compute the inference and the gradients for a single split on a specific device and we do this for all the devices in a distributed manner. Finally, we aggregate the gradients and apply them to the PS task device.



```
def distribute(images, labels, num_classes, total_num_examples, devices, is_train=True):
    def configure_optimizer(global_step, total_num_steps):
y blank lines (3) rn a configured optimizer"""

        def exp_decay(start, tgtFactor, num_stairs):
            decay_step = total_num_steps / (num_stairs - 1)
            decay_rate = (1 / tgtFactor) ** (1 / (num_stairs - 1))
            return tf.train.exponential_decay(start, global_step, decay_step, decay_rate,
                                              staircase=True)

        def lparam(learning_rate, momentum):
            return {
                'learning_rate': learning_rate,
                'momentum': momentum
            }

        return HybridMomentumOptimizer({
            'weights': lparam(exp_decay(0.001, 250, 4), 0.9),
            'biases': lparam(exp_decay(0.002, 10, 2), 0.9),
        })
    my_devices = devices
    builder = ModelBuilder(my_devices[-1])

    my_devices = my_devices[0:-1]
    if not is_train:
        net, logits, total_loss = alexnet_inference(builder, images, labels, num_classes)
        return alexnet_eval(net, labels)
    #with tf.device(builder.variable_device()):
    global_step = builder.ensure_global_step()
    gradients = list()
    with tf.device(builder.variable_device()):
        image_list = tf.split(images, len(my_devices), 0)
        label_list = tf.split(labels, len(my_devices), 0)
    optimizer = configure_optimizer(global_step, total_num_examples)
    with tf.variable_scope('model', reuse=tf.AUTO_REUSE) as var_scope:
        with tf.name_scope('test') :
            for i in range(len(my_devices)):
                curr_device = my_devices[i]
                with tf.name_scope('device-{}'.format(i)) as name_scope:
                    with tf.device(curr_device):
                        net, logits, total_loss = alexnet_inference(builder, image_list[i],
                                                   label_list[i], num_classes,name_scope)
                        test_gradients = optimizer.compute_gradients(total_loss)
            gradients.append(test_gradients)
    with tf.device(builder.variable_device()):
        final_gradients = builder.average_gradients(gradients)
        apply_gradient_op = optimizer.apply_gradients(final_gradients, global_step=global_step)
        train_op = tf.group(apply_gradient_op, name='train')
    return net, logits, total_loss, train_op, global_step
```

## 2.2 Single Node Operation

Here, we have performed the analysis for a Single Node using different batch sizes and report our findings.

Table 5: Alexnet Analysis for Single Node Configuration (Fake Dataset)

| Batch Size | Peak Mem (B) | Peak CPU | Network R | Networks W | Examples/s | s/Batch |
|---|---|---|---|---|---|---|
| 32 | 3515719680 | 26.19 | 22857 | 2538 | 14.3 | 2.22 |
| 64 | 3910672384 | 42.574 | 10296 | 2709 | 27.3 | 2.36 |
| 128 | 3806588928 | 63.755 | 13866 | 2881 | 41.1 | 3.1 |
| 256 | 4413964288 | 78.005 | 32256 | 3140 | 44.1 | 5.78 |

It is evident from the above table that as the batch size increases, the Peak Network, CPU and Memory usage increase. Also, the train data examples processed per second and seconds taken to process per batch both increase. The reason for the former is that we are able to process a higher number of examples per iteration. This increase is non linear as computation overheads increase non linearly because of large intermediate tensor computations.

We had also performed analysis on Flower dataset. It showed similar trends, apart from the fact the the Steps it took to reach a 3% loss varied a lot but in Fake Dataset it took just 2-3 steps. As the batch size increased, the number of steps for convergence decreased.

Table 6: Alexnet Analysis for Single Node Configuration (Flower Dataset)

| Batch Size | Steps | Peak Mem (B) | Peak CPU | Network R | Networks W | Examples/s | s/Batch |
|---|---|---|---|---|---|---|---|
| 1 | 1712 | 3891945472 | 8.306 | 30961 | 650385 | 0.5 | 2.15 |
| 32 | 350 | 34017820672 | 50.425 | 29820 | 17220.5 | 14.1 | 2.27 |
| 64 | 278 | 64997298176 | 57.26 | 48660 | 8044 | 27.5 | 2.35 |
| 128 | 254 | 1.29E+11 | 66.054 | 21708.5 | 504537 | 40 | 3.05 |

## 2.3 Multiple Node Operation

To run on multiple nodes, we ran on 2 types of configurations :

- 2 Node Cluster - 1 PS and 2 Worker Tasks
- 4 Node Cluster - 1 PS and 4 Worker Tasks

We present our findings below in tabular form, for running on the Fake Dataset.

Table 7: Alexnet Analysis for 4 Node Cluster Configuration ( For Fake Dataset )

| Batch Size | Peak Mem (B) | Peak CPU | Network R | Networks W | Examples/s | s/Batch |
|---|---|---|---|---|---|---|
| 32 | 2979549184 | 27.448 | 128463071 | 422947097 | 57.5 | 2.21 |
| 64 | 3701927936 | 35.409 | 128534735 | 462345028 | 107.5 | 2.41 |
| 128 | 4427182080 | 41.79 | 128101423 | 541209639 | 120.7 | 4.24 |
| 256 | 5402697728 | 59.098 | 125763538 | 467515239 | 126.8 | 8.13 |

Table 8: Alexnet Analysis for 2 Node Cluster Configuration ( For Fake Dataset )

| Batch Size | Peak Mem (B) | Peak CPU | Network R | Networks W | Examples/s | s/Batch |
|---|---|---|---|---|---|---|
| 32 | 2701582336 | 18.88 | 141082960 | 127840850 | 29.7 | 2.17 |
| 64 | 3276800000 | 26.453 | 127846150 | 154131082 | 57.6 | 2.31 |
| 128 | 3250630656 | 38.667 | 127757468 | 180467965 | 69.4 | 3.79 |
| 256 | 3498917888 | 57.425 | 125720457 | 233128414 | 75.6 | 6.98 |

We can see that due to distributed mode of operations, the number of examples handled per second is more than the corresponding value for a particular batch size in a single node model.

Table 9: Alexnet Analysis for 2 and 4 Node Cluster Configuration using 32 Batch Size ( For Flower Dataset )

| Nodes | Peak Mem (B) | Peak CPU | Network R | Networks W |
|---|---|---|---|---|
| 2 | 6.48E+10 | 60.26 | 42066 | 103149 |
| 4 | 1.28E+11 | 66.506 | 74011 | 8061 |

We also performed a small experiment on the Flower data set to showcase the Memory/ CPU and Network usage of the same. We refrained from further testing on the Flower data set as it was not required as mentioned on Piazza, and also, it required custom modifications on the batch_num and batch_size parameters to execute properly without crashing.

We have tried to train Alexnet on GPU using flowers dataset as well.But it was crashing for all batch sizes after some 35 iterations and was showing memory error. When we checked the GPU memory usage , we could see it running on 100%.