CS 494 - Cloud Data Center Systems

Homework 2

University of Illinois at Chicago

Montanari Claudio

Porter David

Shinde Komal

1 Logistic Regression

1.1 Implementation

The implementation of the logistic regression model over the MNIST dataset was quite straightforward. First we downloaded the dataset through the appropriate tensorflow API:

```
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)
```

Then, we provided the model description on the workers node:

```
with tf.device(tf.train.replica_device_setter(
            worker_device="/job:worker/task:%d" % FLAGS.task_index,
            cluster=clusterSpec[FLAGS.deploy_mode])):

global_step = tf.get_variable('global_step', [],
                initializer=tf.constant_initializer(0),
                trainable=False)

X = tf.placeholder(tf.float32, shape=[None, featureCount], name="x-input")
Y = tf.placeholder(tf.float32, shape=[None, classCount], name="y-input")

with tf.name_scope('train'):

    W = tf.Variable(tf.random_normal(shape=[featureCount, classCount]))
    b = tf.Variable(tf.random_normal(shape=[1, classCount]))

# setup graph, cost, optimizer
y_pred = tf.nn.softmax(tf.add(tf.matmul(X, W), b))
cost = tf.reduce_mean(-tf.reduce_sum(Y*tf.log(y_pred), reduction_indices=1))
opt = tf.train.GradientDescentOptimizer(learning_rate)

train_step = opt.minimize(cost, global_step=global_step)

correct_pred = tf.equal(tf.argmax(y_pred, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

init_op = tf.global_variables_initializer()
```

Finally, we set up a session where we run the training and testing of our model:

```
sv = tf.train.Supervisor(is_chief=(FLAGS.task_index == 0),
                logdir=LOG_DIR, global_step=global_step,
                init_op=init_op, recovery_wait_secs=1)

with sv.managed_session(server.target) as sess:
    step = 0
    while not sv.should_stop() and step <= TRAINING_STEPS:
        batch_x, batch_y = mnist.train.next_batch(BATCH_SIZE)
        _, acc, step = sess.run([train_step, accuracy, global_step],
                        feed_dict={X: batch_x, Y: batch_y})
```

```
                    if step % PRINT_EVERY == 0:
                        print ("Worker : {}, Step : {}, Accuracy (batch): {}"
                        .format(FLAGS.task_index, step, acc))

            test_acc = sess.run(accuracy,
                        feed_dict={X: mnist.test.images, Y: mnist.test.labels})
            print ("Test-Accuracy: {}".format(test_acc))
    sv.stop()
```

The code above implements the logistic regression model using the *Asynchronous* methodology; for the implementation of the *Synchronous* one instead it's necessary to wrap the model optimizer inside another class like this:

```
    opt = tf.train.GradientDescentOptimizer(learning_rate)

# In a typical asynchronous training environment, it's common to have some
#  stale gradients. For example, with a N-replica asynchronous training,
# gradients will be applied to the variables N times independently.
# Depending on each replica's training speed, some gradients might be
# calculated from copies of the variable from several steps back
# (N-1 steps on average). This optimizer avoids stale gradients
# by collecting gradients from all replicas, averaging them,
# then applying them to the variables in one shot,
# after which replicas can fetch the new variables and continue.

    sync_opt=tf.train.SyncReplicasOptimizer(opt, replicas_to_aggregate=2,
                                            total_num_replicas=2)
    optimizer = sync_opt.minimize(cost, global_step=global_step)
```

1.2 Performance Evaluation

The performance of the regression model are indeed dependent on the batch size and on the training technique utilized. In particular using a bigger batch size affects the accuracy of the model, as expected. We can't make any statement about the dependency of training mode and accuracy solely based on the experiments done with the model above. This because the model used is quite simple and not very stable; but in any case the expectations are that when training in synchronous mode the accuracy of the model is higher with respect to asynchronous training. In the figures below we can better quantify the loss in accuracy with respect to the batch size (for both synchronous and asynchronous training).

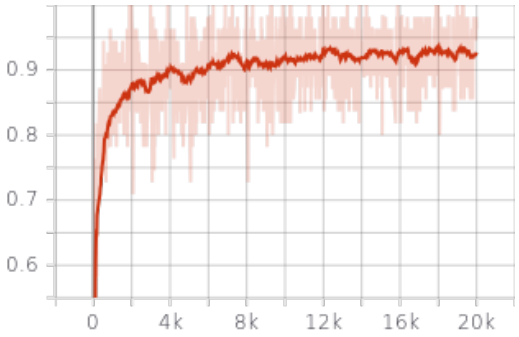

Figure 1: Accuracy during training with batch size of 50

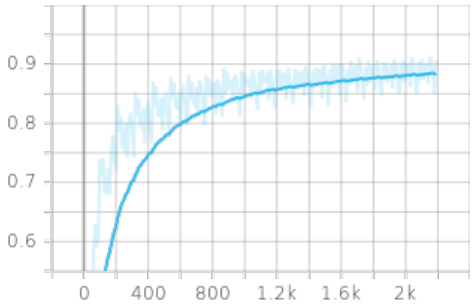

Figure 2: Accuracy during training with batch size of 500

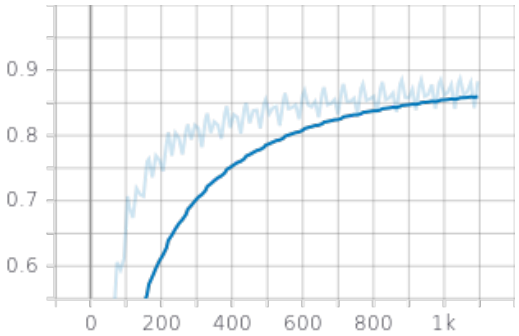

Figure 3: Accuracy during training with batch size of 1000

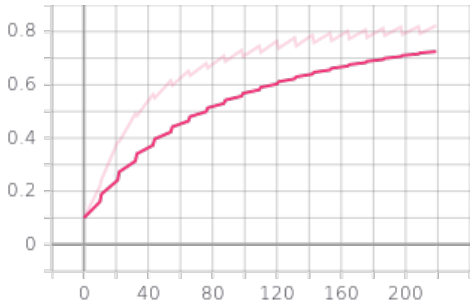

Figure 4: Accuracy during training with batch size of 5000

For what concern system performance we didn't observed big differences between synchronous and asynchronous mode. What we expected was that asynchronous mode had a lower completion time and higher CPU utilization due to the fact that each worker can work independently with respect to the others without any kind of synchronization. At the same time we expected the synchronous training to have an higher network utilization since all the workers have to synchronize after the execution of each epoch.

In terms of completion time they are almost the same, again this is probably due to the simplicity of the model and the relatively small size of the dataset. For what concern system utilization we used **dstat** to measure CPU utilization, disk accesses and network traffic. While we found an online visualization tool for comparing the results found. Also in this case, as it's possible to see from the plots in Figures 5 and 6 there aren't remarkable differences between these two training options in terms of CPU utilization and the reason is the same as before. While if we look at the network traffic we can see how the asynchronous training tends to have values around 27MB that are quite higher with respect to the 3MB of the synchronous. This is against our initial hypothesis; the explanation can be that either something went wrong when configuring the parameter server or that in the asynchronous mode the parameter server keeps updating the workers every time some of the weights variables are updated.

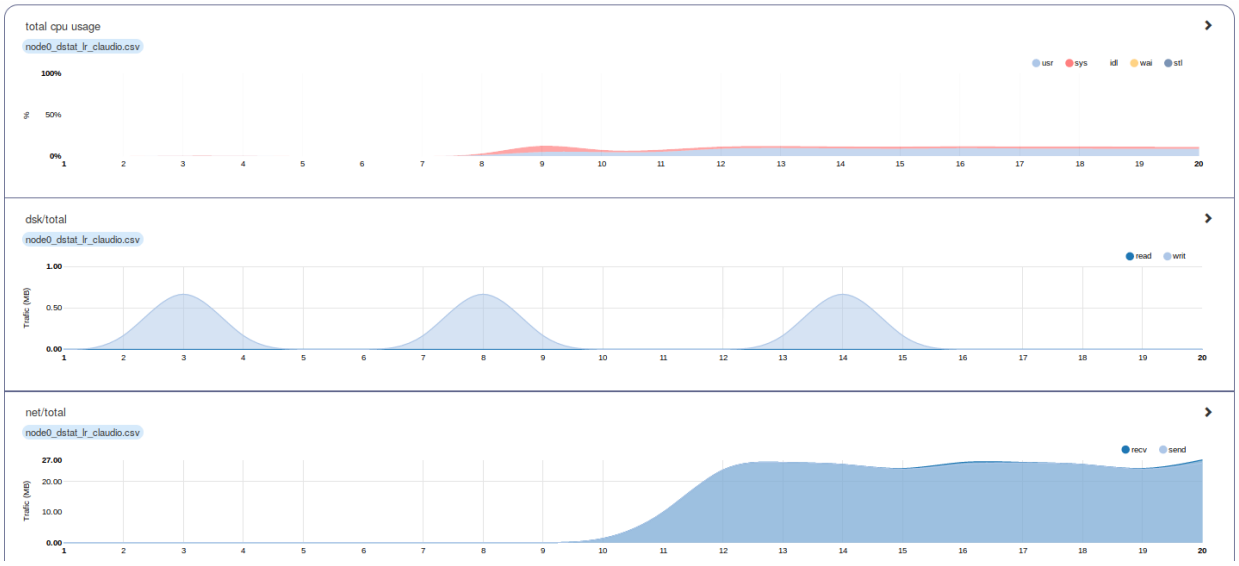

Figure 5: Starting from the upper plot: total CPU usage(%), disk read/write accesses (MB) and network send/received data (MB) for asynchronous training.

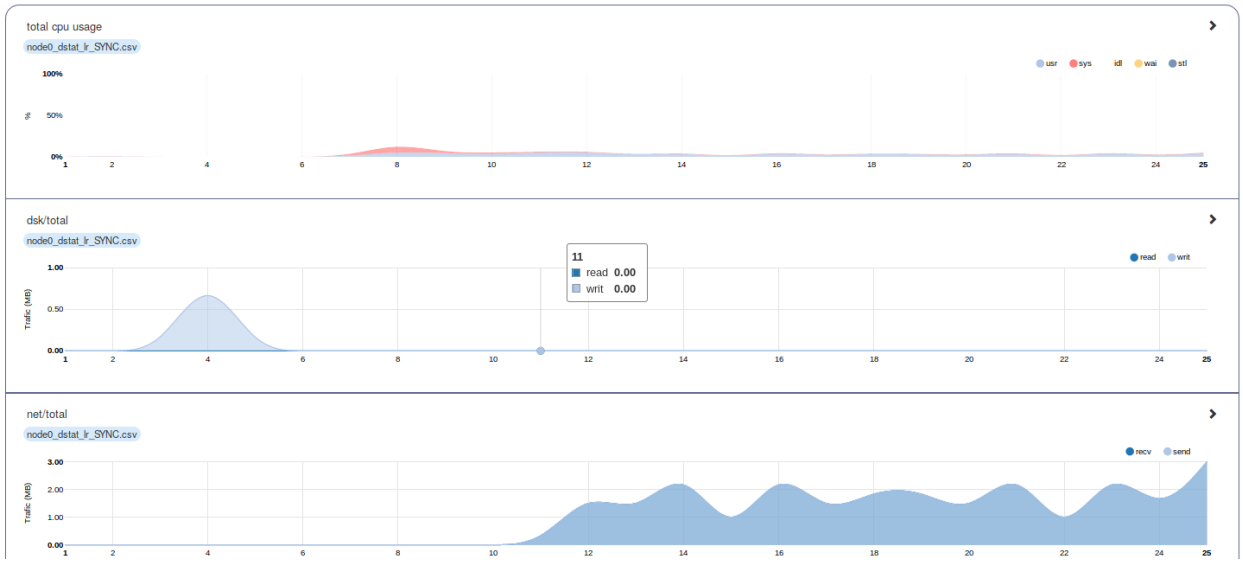

Figure 6: Starting from the upper plot: total CPU usage(%), disk read/write accesses (MB) and network send/received data (MB) for synchronous training.

2 AlexNet

2.1 Task1

Without a clear picture of the dataset for the fake-data example, it was hard to verify why we were getting our precision and loss values with the model which happened to converge very quickly. We split the data returned from the input_data() method on the fake_data set into train and test data, then split the train again between the two worker nodes for synchronized optimization. We modified the source code to perform an evaluation on the test data set after the final epoch was run. Per the assignment, we developed the distributed Alexnet model and performed some black box experiment to see how modifying the batch size and batch numbers affected the loss and accuracy of the model. We conducted the black box experiment many times, but the precision was consistent over the range of batch sizes, so we only provide results from a high and low experiment below. We saw that as we increased the batch numbers we were able to get a higher accuracy. There was a threshold for accuracy discovered: less than 5 epochs, there was 0 precision on the test samples, and greater than 6 epochs there was a 100% precision. However, regardlesss of how many epochs we ran, our loss always remained the same after the second iteration 3.91. Therefore, convergence is quite fast with our model on the given dataset.

Here is the performance for a large batch and small batch training:

batchsize=600
batchnum=6

correct_pred: [600]
precision @ 600.0 examples = 1.000
correct_pred: [600]
precision @ 1200.0 examples = 1.000
correct_pred: [600]
precision @ 1800.0 examples = 1.000
correct_pred: [600]
precision @ 2400.0 examples = 1.000
correct_pred: [600]
precision @ 3000.0 examples = 1.000
correct_pred: [600]
precision @ 3600.0 examples = 1.000

batchsize=600
batchnum=5

correct_pred: [2]
precision @ 600.0 examples = 0.003
correct_pred: [0]
precision @ 1200.0 examples = 0.002
correct_pred: [0]
precision @ 1800.0 examples = 0.001
correct_pred: [0]
precision @ 2400.0 examples = 0.001
correct_pred: [0]
precision @ 3000.0 examples = 0.001

batchsize=600
batchnum=4
    correct_pred: [0]
    precision @ 600.0 examples = 0.000
    correct_pred: [0]
    precision @ 1200.0 examples = 0.000
    correct_pred: [0]
    precision @ 1800.0 examples = 0.000
    correct_pred: [0]
    precision @ 2400.0 examples = 0.000

batchsize=6
batchnum=5
    correct_pred: [0]
    precision @ 6.0 examples = 0.000
    correct_pred: [0]
    precision @ 12.0 examples = 0.000
    correct_pred: [0]
    precision @ 18.0 examples = 0.000
    correct_pred: [0]
    precision @ 24.0 examples = 0.000
    correct_pred: [0]
    precision @ 30.0 examples = 0.000

batchsize=6
batchnum=6
    correct_pred: [5]
    precision @ 6.0 examples = 0.833
    correct_pred: [2]
    precision @ 12.0 examples = 0.583
    correct_pred: [3]
    precision @ 18.0 examples = 0.556
    correct_pred: [4]
    precision @ 24.0 examples = 0.583
    correct_pred: [5]
    precision @ 30.0 examples = 0.633
    correct_pred: [2]
    precision @ 36.0 examples = 0.583

batchsize=6
batchnum=7
    correct_pred: [6]
    precision @ 6.0 examples = 1.000
    correct_pred: [6]
    precision @ 12.0 examples = 1.000
    correct_pred: [6]
    precision @ 18.0 examples = 1.000

correct_pred: [6]
precision @ 24.0 examples = 1.000
correct_pred: [6]
precision @ 30.0 examples = 1.000
correct_pred: [6]
precision @ 36.0 examples = 1.000
correct_pred: [6]
precision @ 42.0 examples = 1.000

2.2 Task2

In order to compare performance between the two machines and the four machines scenario we used **dstat** and the online visualization tool, as before. In particular, we show the plots just for one machine since it's more readable and the other machines showed a similar behavior.

From Figures 7, 8, 9 it's possible to see how CPU utilization and disk accesses were quite similar in all the scenarios; what really changed was the network traffic. In particular, we can see how the parameter server, shown in Figure 8, which is in charge of synchronize all the four machines, have an average network traffic around 700MB; way more higher than the worker which is around 250MB as in the case of two machines. Such results were expected and are coherent with what previously stated.

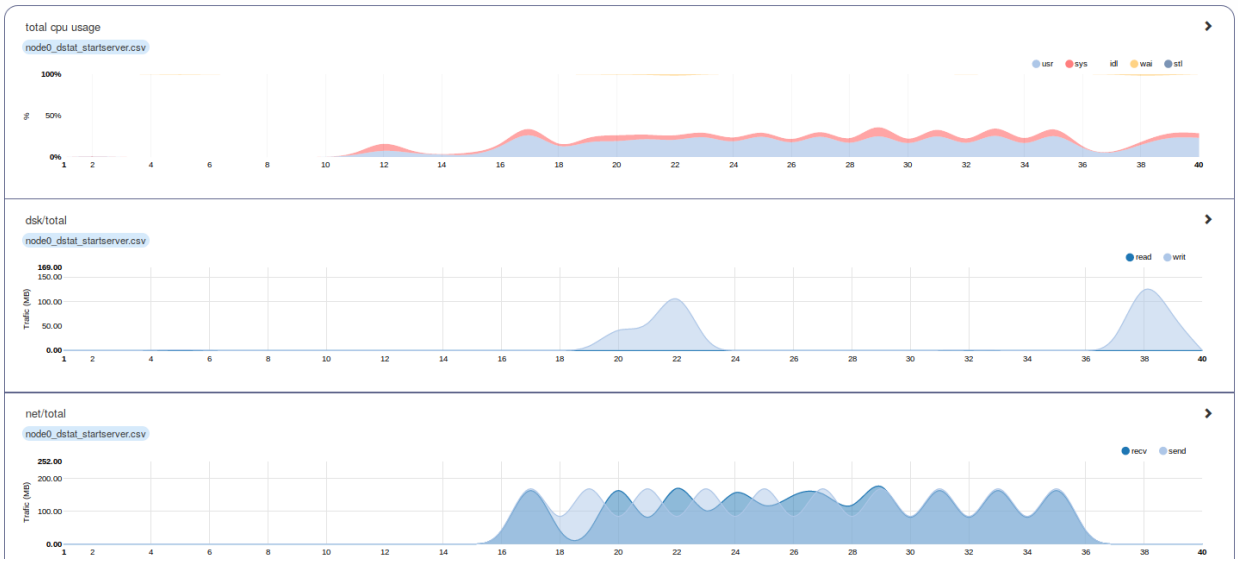

Figure 7: Starting from the upper plot: total CPU usage(%), disk read/write accesses (MB) and network send/received data (MB) for synchronous training on two machines.

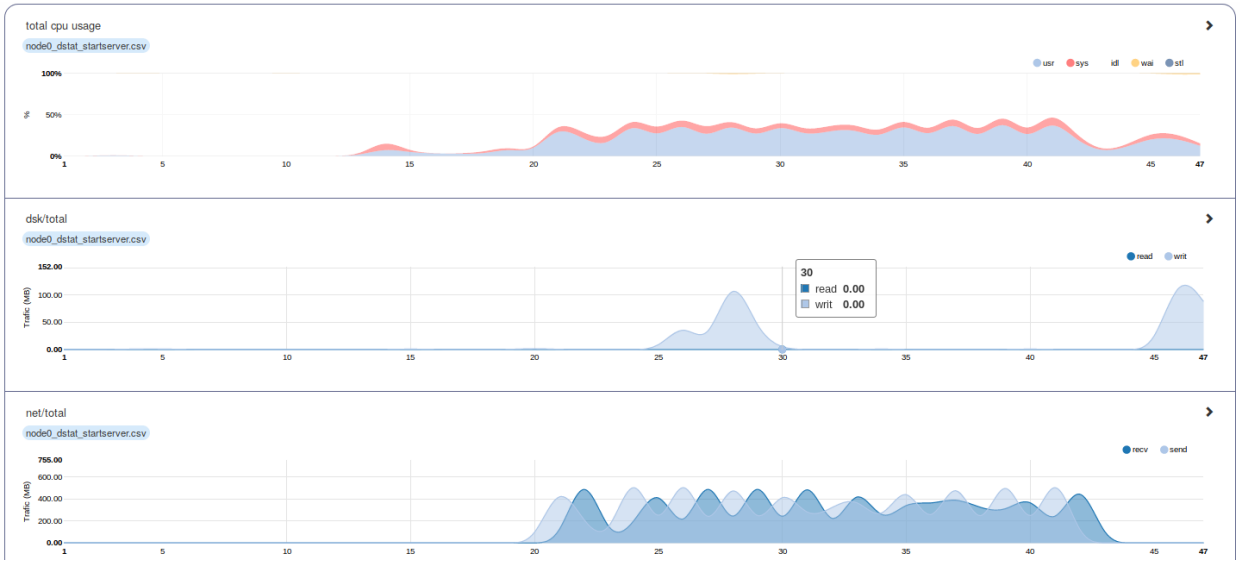

Figure 8: Starting from the upper plot: total CPU usage(%), disk read/write accesses (MB) and network send/received data (MB) for synchronous training on four machines - parameter server.

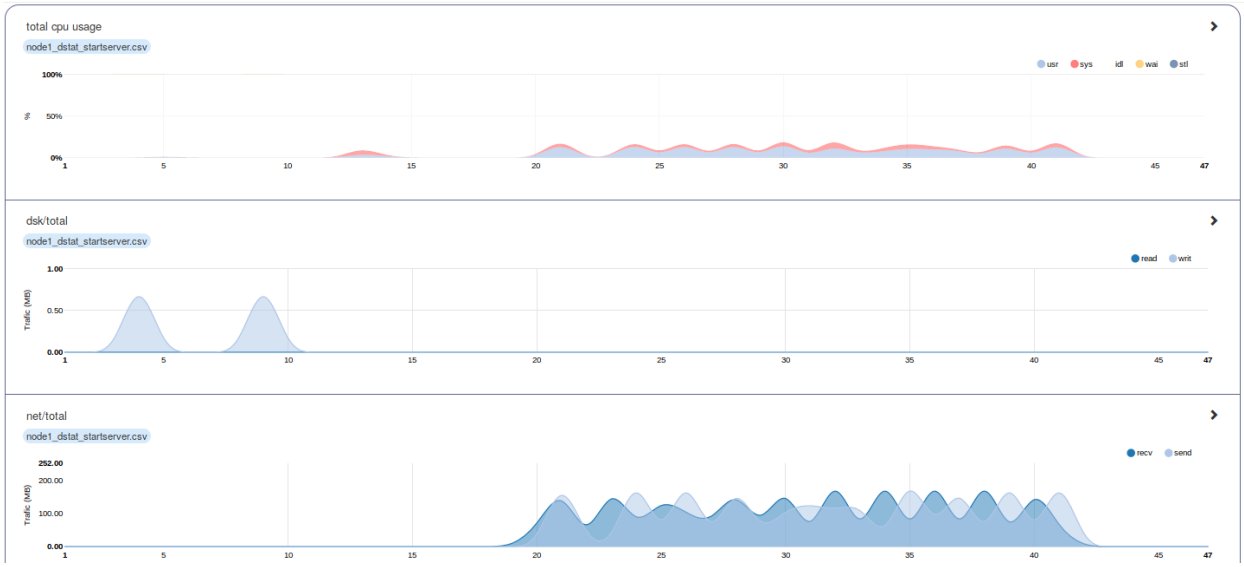

Figure 9: Starting from the upper plot: total CPU usage(%), disk read/write accesses (MB) and network send/received data (MB) for synchronous training on four machines - worker machine.