ECE194N HW 1

Classification Report

February 7, 2019

Student: Erik Rosten
Perm Number: 7143571
Email: erosten@ucsb.edu

Department of Electrical and Computer Engineering, UCSB

Original Code

The original code is shown below as a reference for mentioned changes.

```
import tensorflow as tf

# Import MINST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)

# Parameters
learning_rate = 0.01
training_epochs = 25
batch_size = 100
display_step = 1

# tf Graph Input
x = tf.placeholder(tf.float32, [None, 784]) # mnist data image of shape 28*28=784
y = tf.placeholder(tf.float32, [None, 10]) # 0-9 digits recognition => 10 classes

# Set model weights
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))

# Construct model
pred = tf.nn.softmax(tf.matmul(x, W) + b) # Softmax

# Minimize error using cross entropy
cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(pred), reduction_indices=1))
# Gradient Descent
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()

# Start training
with tf.Session() as sess:
    sess.run(init)

    # Training cycle
    for epoch in range(training_epochs):
        avg_cost = 0.
        total_batch = int(mnist.train.num_examples/batch_size)
        # Loop over all batches
        for i in range(total_batch):
            batch_xs, batch_ys = mnist.train.next_batch(batch_size)
            # Fit training using batch data
            _, c = sess.run([optimizer, cost], feed_dict={x: batch_xs,
                                                          y: batch_ys})
            # Compute average loss
            avg_cost += c / total_batch
        # Display logs per epoch step
        if (epoch+1) % display_step == 0:
            print("Epoch:", '%04d' % (epoch+1), "cost=", "{:.9f}".format(avg_cost))
```

```
51
52      print("Optimization Finished!")
53
54      # Test model
55      correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
56      # Calculate accuracy for 3000 examples
57      accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
58      print("Accuracy:", accuracy.eval({x: mnist.test.images[:3000], y:
            mnist.test.labels[:3000]}))
```

Running this code produces the following output (copied without tensorflow warnings)

```
1  Epoch: 0001 cost= 1.184108747
2  Epoch: 0002 cost= 0.665375831
3  Epoch: 0003 cost= 0.552803518
4  Epoch: 0004 cost= 0.498656733
5  Epoch: 0005 cost= 0.465463099
6  Epoch: 0006 cost= 0.442625841
7  Epoch: 0007 cost= 0.425523379
8  Epoch: 0008 cost= 0.412180205
9  Epoch: 0009 cost= 0.401422062
10 Epoch: 0010 cost= 0.392397343
11 Epoch: 0011 cost= 0.384789766
12 Epoch: 0012 cost= 0.378213962
13 Epoch: 0013 cost= 0.372432504
14 Epoch: 0014 cost= 0.367323240
15 Epoch: 0015 cost= 0.362747280
16 Epoch: 0016 cost= 0.358631084
17 Epoch: 0017 cost= 0.354876307
18 Epoch: 0018 cost= 0.351478168
19 Epoch: 0019 cost= 0.348360910
20 Epoch: 0020 cost= 0.345446592
21 Epoch: 0021 cost= 0.342724433
22 Epoch: 0022 cost= 0.340284539
23 Epoch: 0023 cost= 0.337962102
24 Epoch: 0024 cost= 0.335773917
25 Epoch: 0025 cost= 0.333726952
26 Optimization Finished!
27 Accuracy: 0.888
```

Part a: Displaying Training and Test Accuracy per Epoch

The modified original code for part a is below.

```
1  import tensorflow as tf
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5
6
7  # Import MINST data
8  from tensorflow.examples.tutorials.mnist import input_data
9  mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
```

```
# Parameters
learning_rate = 0.01
training_epochs = 25
batch_size = 100
display_step = 1

# tf Graph Input
x = tf.placeholder(tf.float32, [None, 784]) # mnist data image of shape 28*28=784
y = tf.placeholder(tf.float32, [None, 10]) # 0-9 digits recognition => 10 classes

# Set model weights
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))

# Construct model
pred = tf.nn.softmax(tf.matmul(x, W) + b) # Softmax

# Minimize error using cross entropy
cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(pred), reduction_indices=1))
# Gradient Descent
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()
# Start training
test_accs = np.zeros((training_epochs,))
train_accs = np.zeros((training_epochs,))
with tf.Session() as sess:
    sess.run(init)
    # Training cycle
    for epoch in range(training_epochs):
        avg_cost = 0.
        total_batch = int(mnist.train.num_examples/batch_size)
        # Loop over all batches
        for i in range(total_batch):
            batch_xs, batch_ys = mnist.train.next_batch(batch_size)
            # Fit training using batch data
            _, c = sess.run([optimizer, cost], feed_dict={x: batch_xs,
                                                          y: batch_ys})
            # Compute average loss
            avg_cost += c / total_batch
        # Display logs per epoch step
        correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
        accuracy = tf.reduce_mean(tf.cast(correct_prediction,tf.float32))

        if (epoch+1) % display_step == 0:
            print("Epoch:", '%04d' % (epoch+1), "cost=", "{:.9f}".format(avg_cost))
            test_acc = accuracy.eval({x: mnist.test.images[:3000], y:
                mnist.test.labels[:3000]})
            train_acc = accuracy.eval({x:mnist.train.images, y: mnist.train.labels})
            test_accs[epoch] = test_acc
            train_accs[epoch] = train_acc
            print ("\tTest Accuracy:", test_acc)
            print("\tTrain Accuracy:", train_acc)
```

```
63
64      print("Optimization Finished!")
65      print ("Final Test Accuracy:", accuracy.eval({x: mnist.test.images, y:
            mnist.test.labels}))
66
67  fig, axes = plt.subplots(nrows=2)
68  x = range(training_epochs)
69  axes[0].scatter(x, train_accs, marker = "o", color = "r")
70  axes[1].scatter(x, test_accs, marker = "x", color = "k")
71  axes[0].set(xlabel = "Epoch", ylabel = "Training Accuracy (%)")
72  axes[1].set(xlabel = "Epoch", ylabel = "Test Accuracy (%)")
73  fig.tight_layout()
74  plt.show()
```

In this edited code, I've evaluated, displayed and saved the test/training accuracy via the adjusted code on lines 57-62 above, using the *eval*() function to get the values, print statements to display, and preallocated arrays (lines 35-36) to save. The plots are generated by lines 67-74 using matplotlib. The output of running the code is below.

```
1   Epoch: 0001 cost= 1.183271719
2      Test Accuracy: 0.81733334
3      Train Accuracy: 0.84278184
4   Epoch: 0002 cost= 0.664949193
5      Test Accuracy: 0.83933336
6      Train Accuracy: 0.8628
7   Epoch: 0003 cost= 0.552618324
8      Test Accuracy: 0.852
9      Train Accuracy: 0.87183636
10  Epoch: 0004 cost= 0.498570159
11     Test Accuracy: 0.85366666
12     Train Accuracy: 0.8782909
13  Epoch: 0005 cost= 0.465545682
14     Test Accuracy: 0.85966665
15     Train Accuracy: 0.8834364
16  Epoch: 0006 cost= 0.442550419
17     Test Accuracy: 0.864
18     Train Accuracy: 0.88616365
19  Epoch: 0007 cost= 0.425508814
20     Test Accuracy: 0.87
21     Train Accuracy: 0.8895636
22  Epoch: 0008 cost= 0.412153942
23     Test Accuracy: 0.871
24     Train Accuracy: 0.8918909
25  Epoch: 0009 cost= 0.401348825
26     Test Accuracy: 0.87133336
27     Train Accuracy: 0.8934
28  Epoch: 0010 cost= 0.392338373
29     Test Accuracy: 0.8743333
30     Train Accuracy: 0.8956182
31  Epoch: 0011 cost= 0.384791222
32     Test Accuracy: 0.876
33     Train Accuracy: 0.89723635
34  Epoch: 0012 cost= 0.378189985
35     Test Accuracy: 0.87666667
```

```
   Train Accuracy: 0.89803636
Epoch: 0013 cost= 0.372434046
   Test Accuracy: 0.8793333
   Train Accuracy: 0.8993273
Epoch: 0014 cost= 0.367282976
   Test Accuracy: 0.8796667
   Train Accuracy: 0.8997273
Epoch: 0015 cost= 0.362728022
   Test Accuracy: 0.8806667
   Train Accuracy: 0.90156364
Epoch: 0016 cost= 0.358599785
   Test Accuracy: 0.88233334
   Train Accuracy: 0.9022545
Epoch: 0017 cost= 0.354887340
   Test Accuracy: 0.88166666
   Train Accuracy: 0.9027454
Epoch: 0018 cost= 0.351456176
   Test Accuracy: 0.883
   Train Accuracy: 0.9034909
Epoch: 0019 cost= 0.348322004
   Test Accuracy: 0.8843333
   Train Accuracy: 0.9044909
Epoch: 0020 cost= 0.345424564
   Test Accuracy: 0.8833333
   Train Accuracy: 0.90507275
Epoch: 0021 cost= 0.342756367
   Test Accuracy: 0.886
   Train Accuracy: 0.9055273
Epoch: 0022 cost= 0.340203486
   Test Accuracy: 0.88666666
   Train Accuracy: 0.90663636
Epoch: 0023 cost= 0.337950682
   Test Accuracy: 0.88766664
   Train Accuracy: 0.90687275
Epoch: 0024 cost= 0.335766400
   Test Accuracy: 0.88766664
   Train Accuracy: 0.90705454
Epoch: 0025 cost= 0.333709573
   Test Accuracy: 0.889
   Train Accuracy: 0.9078909
Optimization Finished!
Final Test Accuracy: 0.889
```

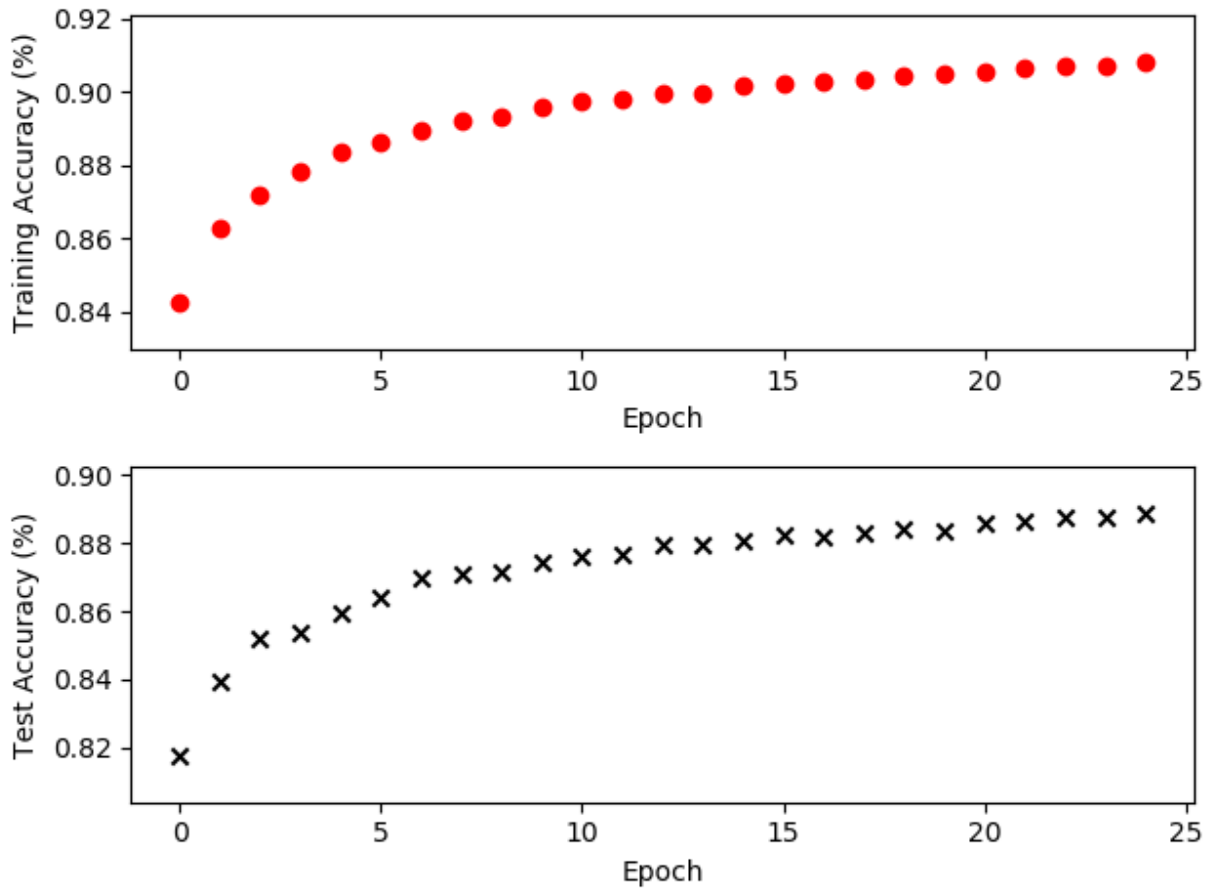

Figure 1: Test and Train Accuracies vs Iterations

Note that the final test accuracy differs between part a's iteration and the original code. This is due to the randomization of the images that *mnist.train.next_batch*(*batch_size*) returns.

Part b: Identifying Misclassed Test Samples

The modified code for part b is below.

```
1  import tensorflow as tf
2  import numpy as np
3  import matplotlib.pyplot as plt
4
5
6
7  # Import MINST data
8  from tensorflow.examples.tutorials.mnist import input_data
9  mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
10 # Parameters
11 learning_rate = 0.01
12 training_epochs = 25
13 batch_size = 100
14 display_step = 1
```

```
15
16 # tf Graph Input
17 x = tf.placeholder(tf.float32, [None, 784]) # mnist data image of shape 28*28=784
18 y = tf.placeholder(tf.float32, [None, 10]) # 0-9 digits recognition => 10 classes
19
20 # Set model weights
21 W = tf.Variable(tf.zeros([784, 10]))
22 b = tf.Variable(tf.zeros([10]))
23
24 # Construct model
25 pred = tf.nn.softmax(tf.matmul(x, W) + b) # Softmax
26
27 # Minimize error using cross entropy
28 cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(pred), reduction_indices=1))
29 # Gradient Descent
30 optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
31
32 # Initialize the variables (i.e. assign their default value)
33 init = tf.global_variables_initializer()
34 # Start training
35 with tf.Session() as sess:
36     sess.run(init)
37
38     # Training cycle
39     for epoch in range(training_epochs):
40         avg_cost = 0.
41         total_batch = int(mnist.train.num_examples/batch_size)
42         # Loop over all batches
43         for i in range(total_batch):
44             batch_xs, batch_ys = mnist.train.next_batch(batch_size)
45             # Fit training using batch data
46             _, c = sess.run([optimizer, cost], feed_dict={x: batch_xs,
47                                                           y: batch_ys})
48             # Compute average loss
49             avg_cost += c / total_batch
50         # Display logs per epoch step
51         my_prediction = tf.argmax(pred,1)
52         correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
53         accuracy = tf.reduce_mean(tf.cast(correct_prediction,tf.float32))
54
55         if (epoch+1) % display_step == 0:
56             print("Epoch:", '%04d' % (epoch+1), "cost=", "{:.9f}".format(avg_cost))
57
58
59     print("Optimization Finished!")
60     print ("Final Accuracy:", accuracy.eval({x: mnist.test.images, y:
           mnist.test.labels}))
61     is_correct = correct_prediction.eval({x: mnist.test.images, y: mnist.test.labels})
62     false_indices = np.where(is_correct == False)[0]
63     bad_images = mnist.test.images[false_indices]
64     bad_image_labels = mnist.test.labels[false_indices]
65
66     for i in range(0,10):
67         bad_image_counter = 0
```

```
68          fig = plt.figure(1)
69          for j in range(0, false_indices.shape[0]):
70              false_index = false_indices[j]
71              img = bad_images[j].reshape(28,28)
72              correct_label = np.nonzero(bad_image_labels[j,:])[0]
73              if (correct_label == i and bad_image_counter < 10):
74                  ax = plt.subplot(2,5,bad_image_counter + 1)
75                  ax.axis('off')
76                  imgplot = plt.imshow(img, cmap='gray')
77                  predicted_label = my_prediction.eval({x: bad_images[j].reshape(1,784)})
78                  fig.suptitle('Correct Label: {}'.format(correct_label))
79                  ax.set_title('Predicted Label: {}'.format(predicted_label), size=6)
80                  bad_image_counter += 1
81          fig.tight_layout()
82          plt.show()
```

The code for part b is exactly the same as the original, save for the *my_prediction* variable on line 51 and the code from line 61 and down. My strategy for detecting false images evaluates the $is_correct$ variable using tensorflow, which returns whether a test image prediction was the same as it's true label. I use numpy to grab the indices where this array is false, then find the images and labels using those indices. I use the *my_prediction* variable to determine which class the network thought the image was. One final note is that I opted to use the entire test set to find these images, since using the original subset of 3000 did not return 10 incorrect predictions for the 0 class. Below are the misclassified images, as well as theories as to why they were incorrectly predicted.

Class 0

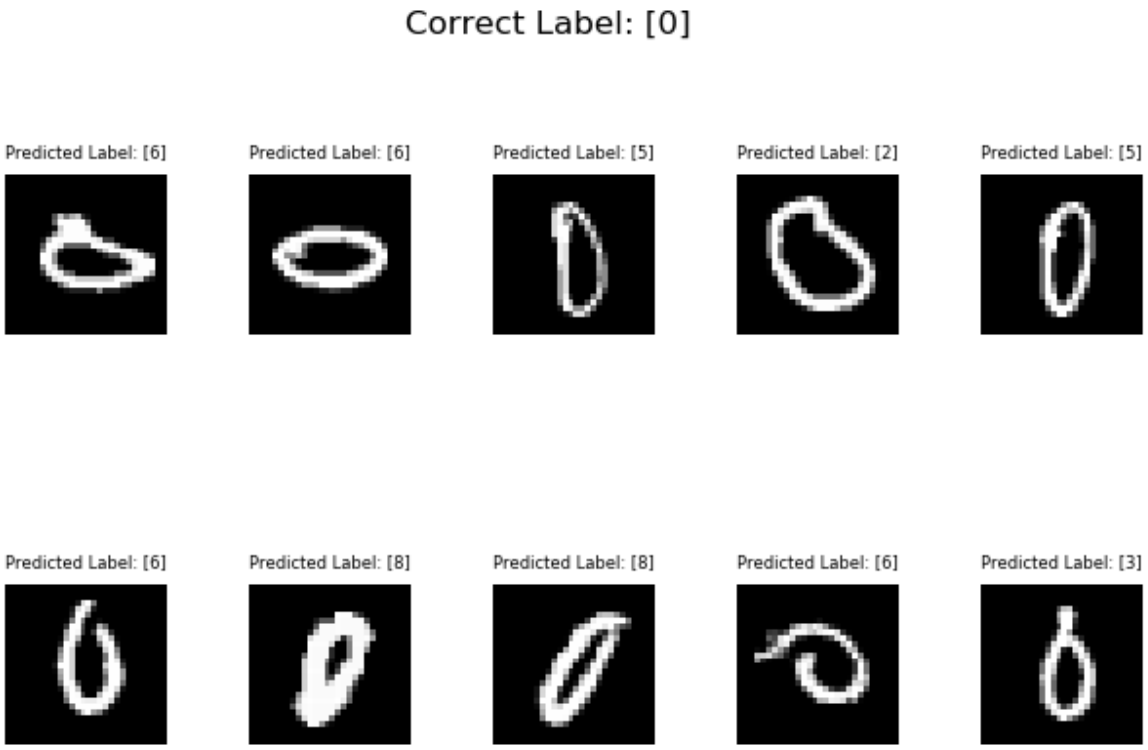

Figure 2: Misclassified Images for Class 0

Some of those misclassified as 6 were 0's that weren't closed, which makes sense since often handwritten 6's are not fully closed. The others that were misclassified as 6 had protrusions out of the loop, which could have been interpreted as the tail of the 6. Those misclassified as 8 were more oval-y than the rest, which is often characterized as 8's when the two loops are badly distinguished when written.

0's were most misclassified as: 6

Class 1

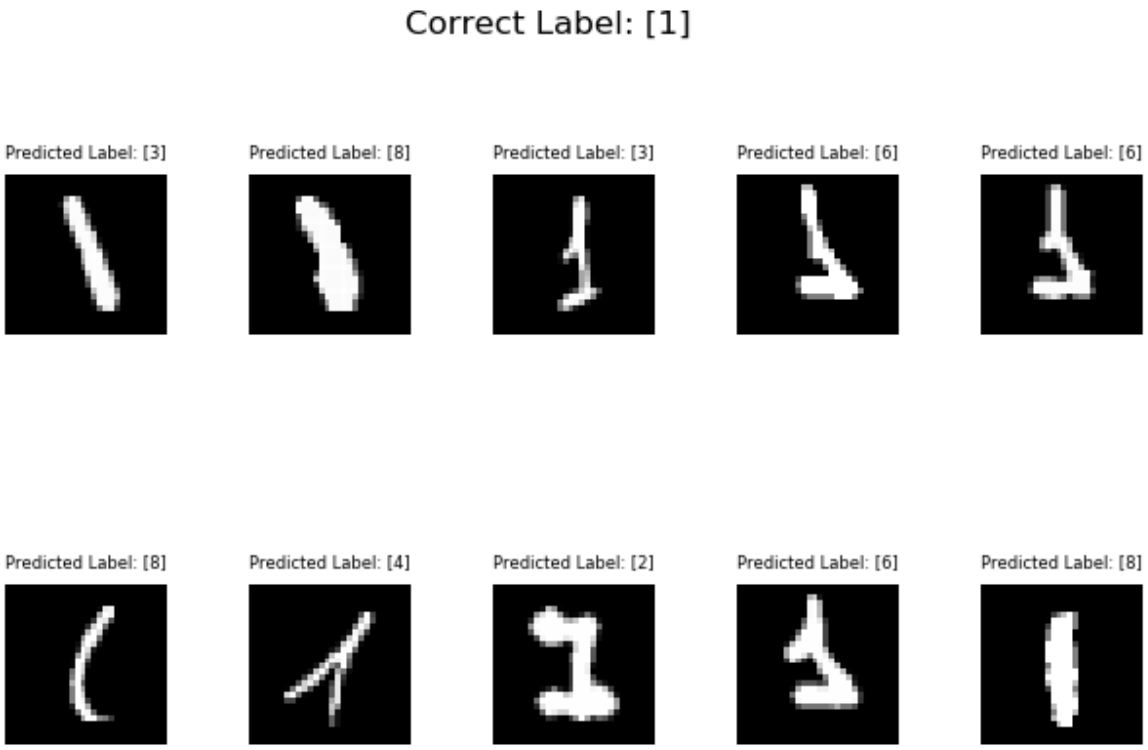

Figure 3: Misclassified Images for Class 1

The most egregious of these misclassified images are the ones on the far left. Other than that, thoe misclassified as 8's are bulky fat 1's and could be badly written 8's. The one predicted as 3 has interesting protrusions on the left, which could be interpreted as the top and bottom of a 3 curling in. Those misclassified as 6 look like badly drawn 6's that do not have the loop fully closed. The one misclassified as 2 looks more like a 2 than a 1 even to me.

1's were most misclassified as: 6 or 8

Class 2

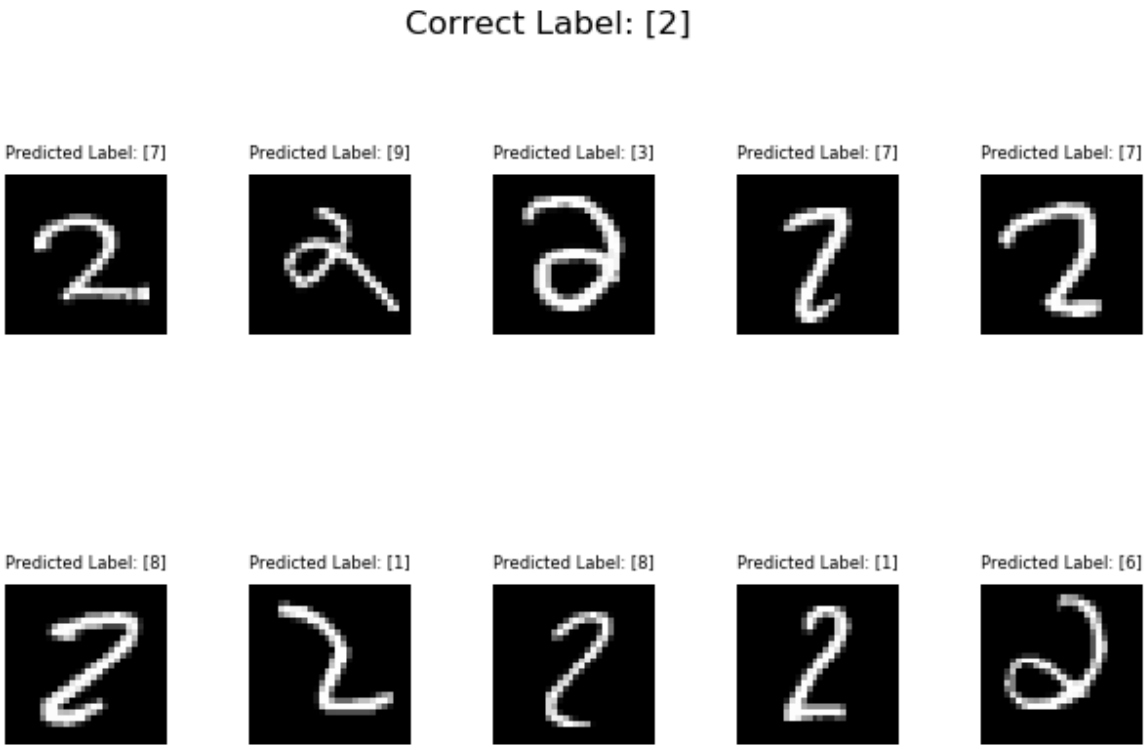

Figure 4: Misclassified Images for Class 2

Those misclassified as 2 often have a straighter top than the others, which makes sense since if the tail of the 2 is short, it would look more like a 7 than a 2. The one predicted as a 9 looks like a rotated 9 with a long tail after the loop. The one predicted as a 3 looks like a badly drawn 3 who's bottom curve loops a bit too far back in. The ones predicted as 8's are particularly curvy on both the top and bottom, which hints towards an unclosed 8. The ones predicted as 1 look as though they might have the tick at the top of the 1 that some people often do.

2's were most misclassified as: 7

Class 3

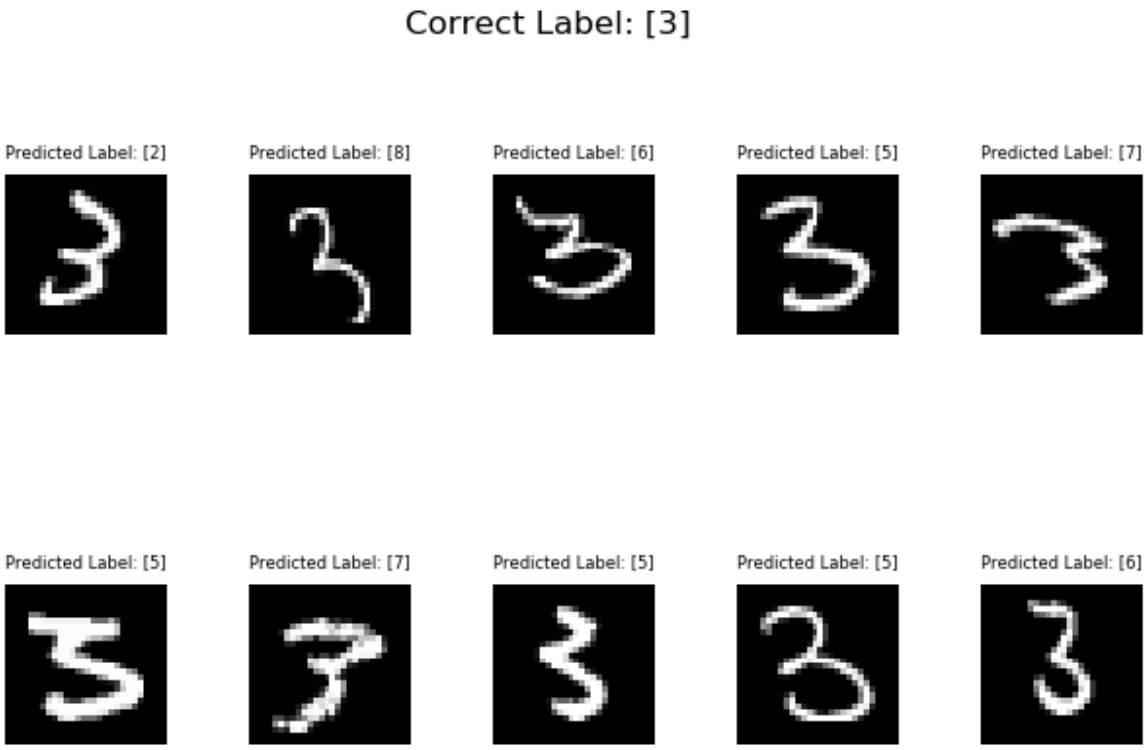

Figure 5: Misclassified Images for Class 3

The one misclassified as 2 could be a badly drawn 2 with it's bottom curved out. The one misclassified as 8 looks exactly like half an 8. The two misclassified as 6 have bottom halves that are nearly closed, which is indicative of a 6. Those misclassified as 5 are mostly those with flat tops, which makes them nearly indistinguishable from 5's. The two misclassified as 7 have flat tops, but a small jut out in the middle, which makes it hard to tell, since some people write 7's with a line through the middle.

3's were most misclassified as: 5

Class 4

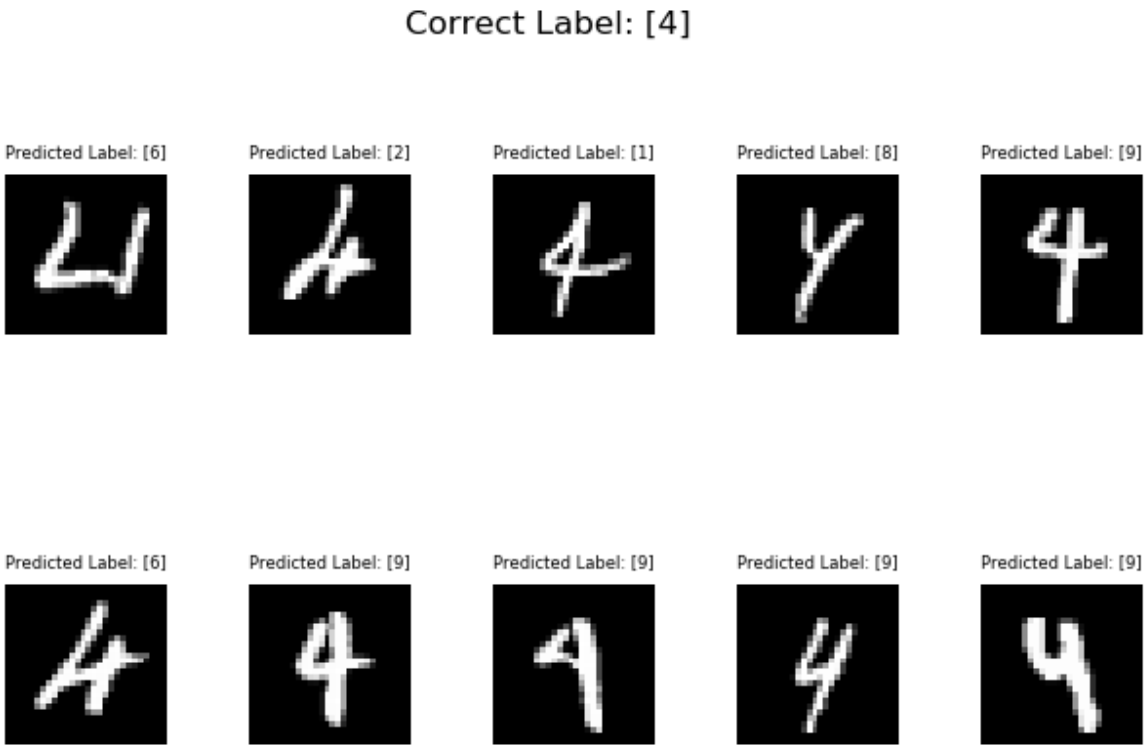

Figure 6: Misclassified Images for Class 4

The majority of 4's were misclassified as 9's. This makes sense since if handwritten 9's did not close the top loop properly they would look exactly like 4's. Those misclassified as something else are quite atrociously drawn 4's. The one exception might be the one misclassified as 1, since it is particularly skinny and is drawn with a closed loop compared to the others.

4's were most misclassified as: 9

Class 5

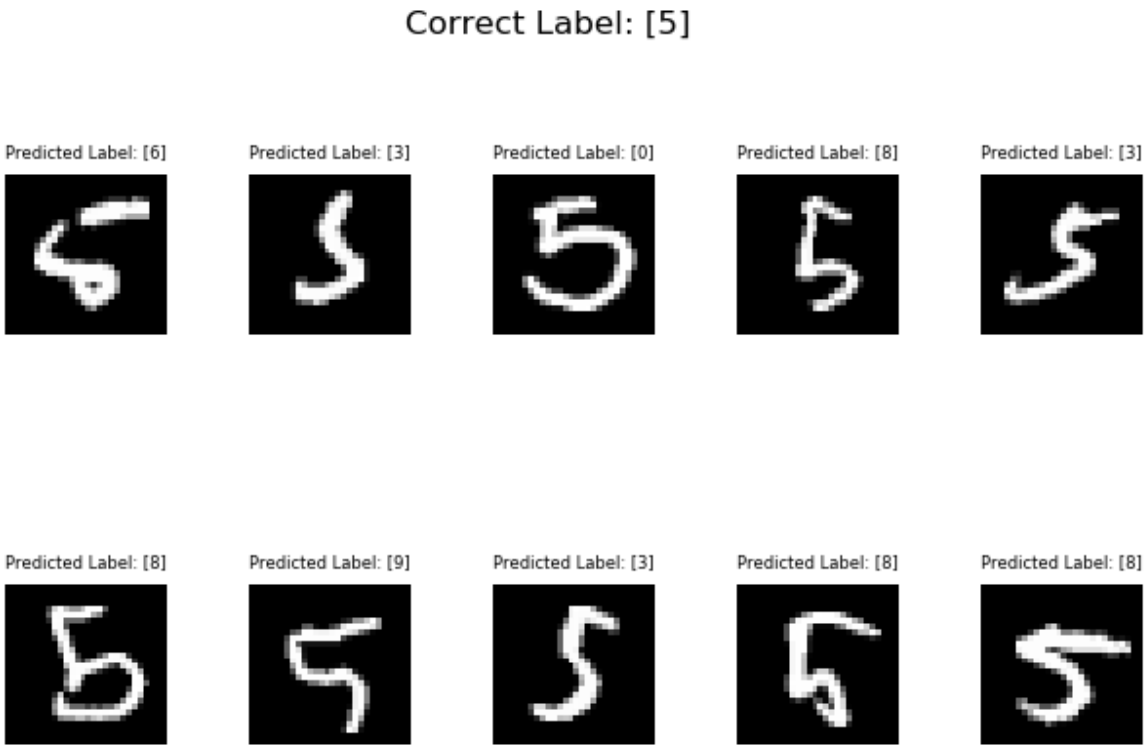

Figure 7: Misclassified Images for Class 5

The 5 misclassified as a 6 makes sense since it is such a badly drawn 5, that the top half looks curved and the bottom loop is closed. The two misclassified as 3 have particularly short curvy bottoms, which is representative of 3's. The one misclassified as a 0 has an extraordinarily large curvy bottom, which is nearly closed. The one misclassified as 9 has almost no bottom curvy part at all, but is straight and could be interpreted as an unclosed 9. The rest were misclassified as 8's, which is understandable since if the top half of the 5 were closed, it would be indistinguishable from an 8.

5's were most misclassified as: 8

Class 6

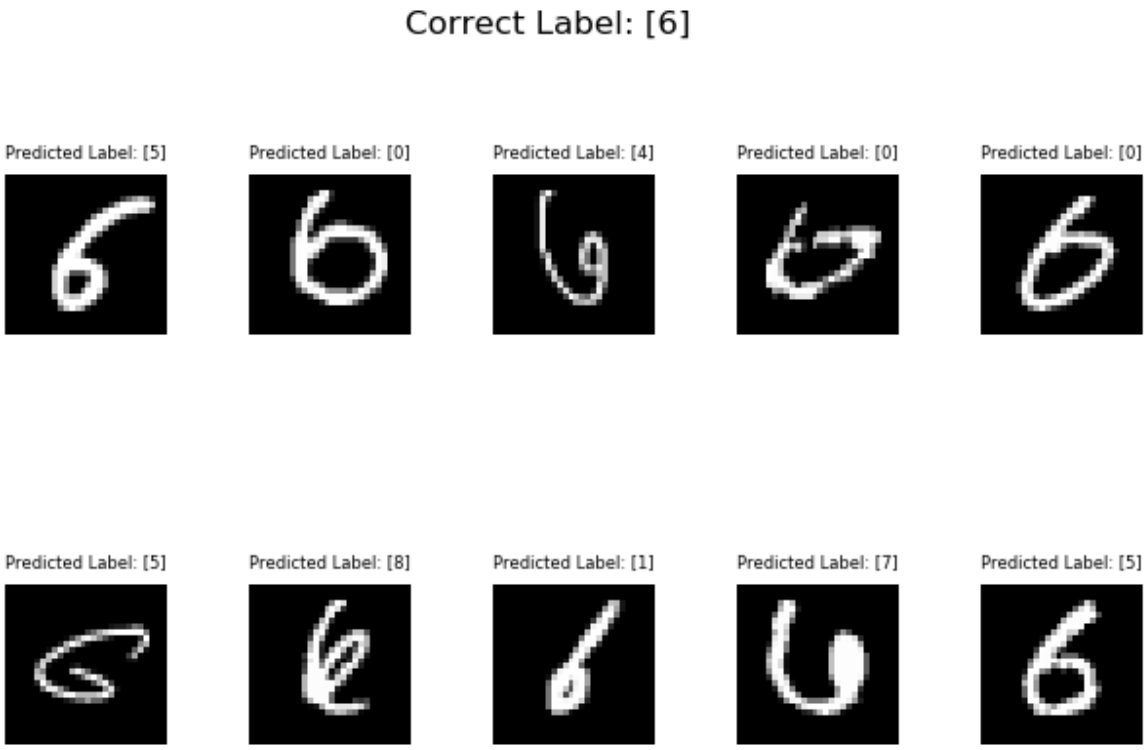

Figure 8: Misclassified Images for Class 6

Most of the 6's that were misclassified as 5's have relatively flat tops and nearly closed loops, indicating a poorly drawn 5. Those misclassified as 0 have small tops and large, closed circles. The one misclassified as 1 has a straight, vertical top and a very small hole. The one predicted as a 4 looks like a 4 without a bottom. The one predicted as 8 has multiple curved protrusions, looking like an extremely poorly drawn 8. The one predicted as 7, I have no explanation for.

6's were most misclassified as: 0 or 5

Class 7

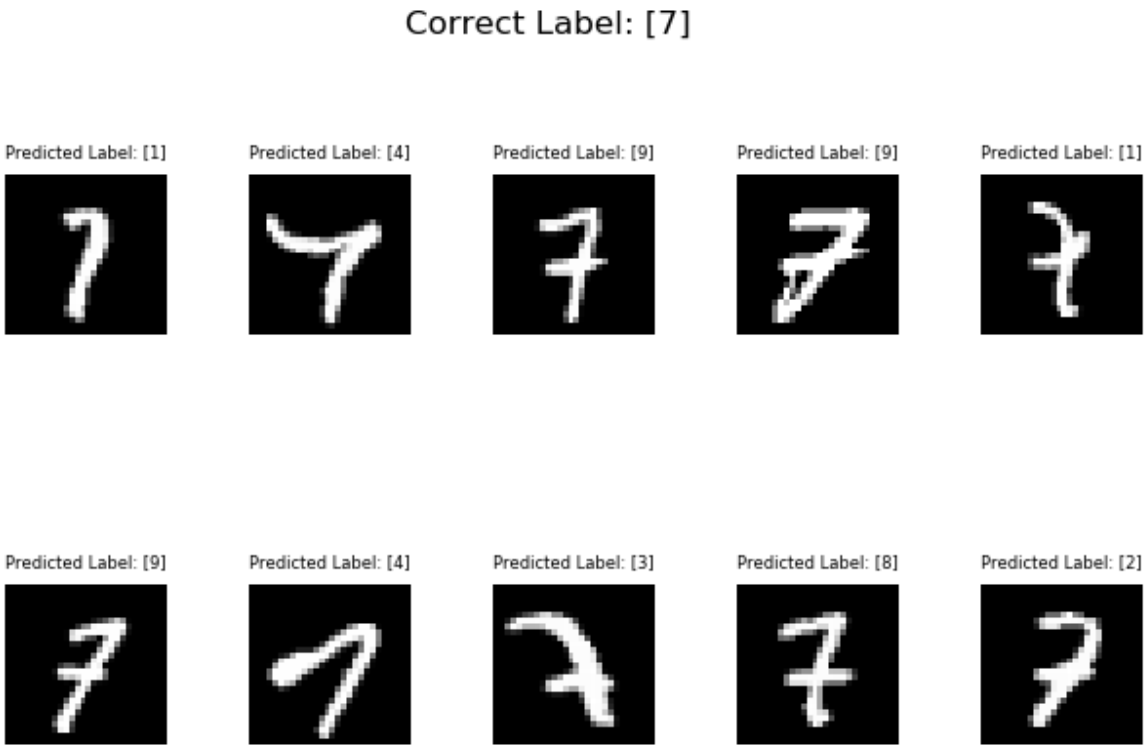

Figure 9: Misclassified Images for Class 7

The 7's predicted as 1's have small tops, and thus look like 1's. The top row 7 predicted as a 4 looks like it has two peaks at the top, indicating a 4. Those predicted as 9's have dashes in the middle, and a close flat top, which indicates a 9 that has not been closed. The bottom row 7 predicted as a 4 looks terrible. The 7 predicted as a 3 has slight curve to it, indicating a possibly badly drawn 3. The one predicted as an 8 has a consistent curve throughout it's vertical segment as well as a line through it, indicating a possible 8. The one predicted as a 2 has a curve in the top part, and remains consistent throughout it's vertical segment, looking like a 2 without it's bottom tail.

7's were most misclassified as: 9

Class 8

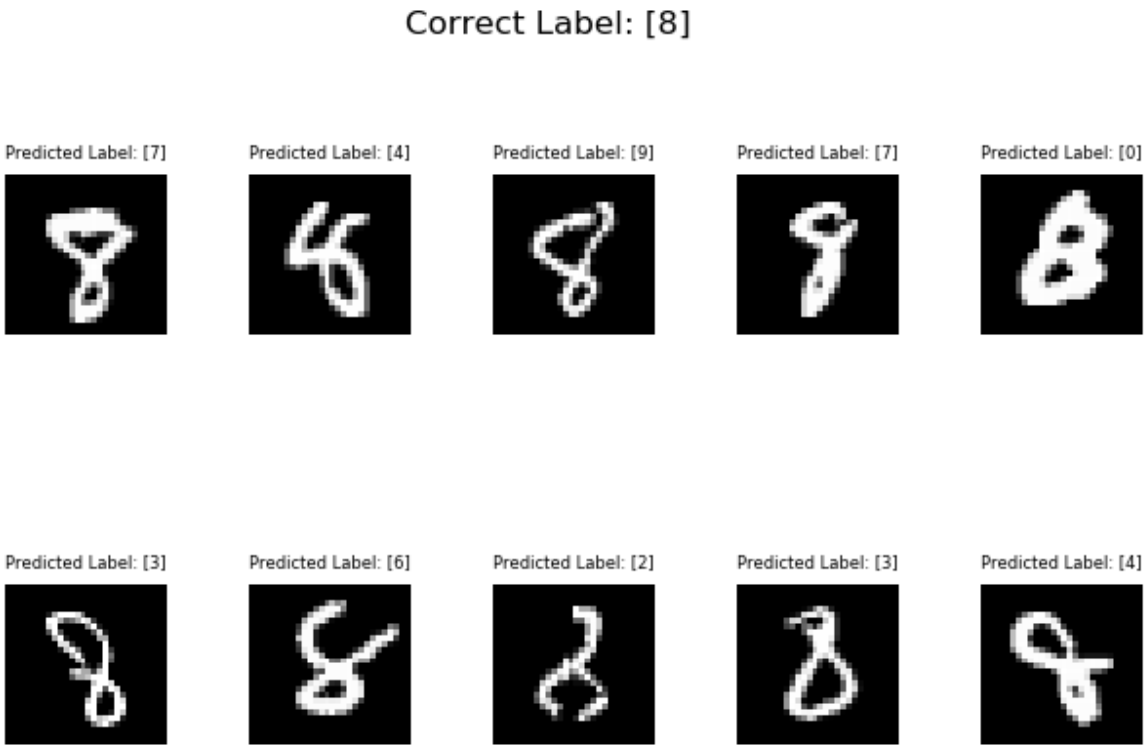

Figure 10: Misclassified Images for Class 8

My first observation is that these are some badly drawn 8's. Those misclassified as 7 have skinny bottoms, and the top loop may be interpreted as a flat top and line through the middle. The top row 4 has an unclosed top and relatively skinny bottom, indicating a 4. The bottom row 4 does not look like any recognizable digit. The one predicted as a 9 has a skinny bottom, and could be a badly drawn 9. The one predicted as a 0 has thick lines, indicating that the top circle may have been a mistake in a badly drawn 0. Those predicted as a 3 have unclosed top loops, but no other signs that it may be a 3. The one predicted as a 2 has a half loop in the top, and unclosed bottom. 8's were most misclassified as: 3 or 4 or 7

Class 9

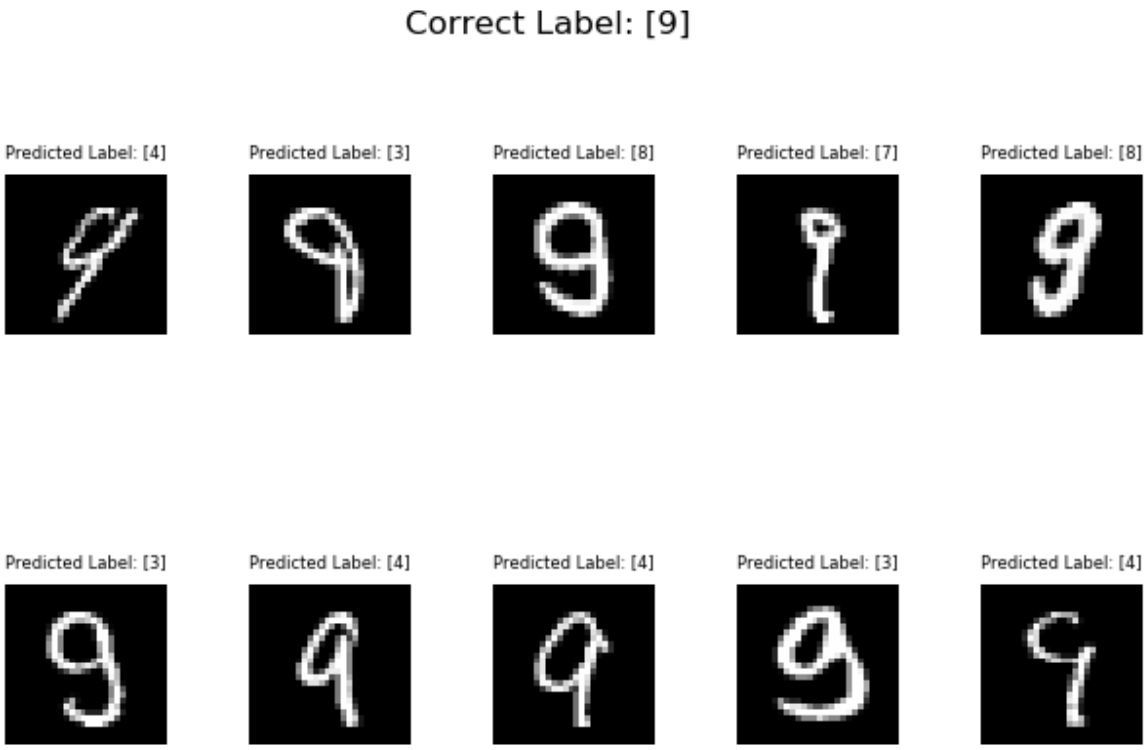

Figure 11: Misclassified Images for Class 9

9's predicted as 4's have mostly unclosed top loops and are relatively skinny. 9's that are predicted as 3's have a curved bottom that comes out rather far to the left, indicating a possible 3. Those predicted as 8's have a bottom curve that nearly comes up to meet the middle. The one predicted as 7 has such a small top loop that it might have been interpreted as the flat line top of a 7. The one predicted as a 3 is so disjointed that it might be 2 closed loops of a horribly drawn 3.
9's were most misclassified as: 4

Part c: Classifying even/odd instead of digits

The modified code for part c is below

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt



# Import MINST data
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
# Parameters
learning_rate = 0.01
training_epochs = 25
batch_size = 100
display_step = 1

# iterate through digits
# odd digit 1's are in index 1
# even digit 1's are in index 0
def map_labels(labels):
    new_labels = np.zeros((labels.shape[0],2))
    for i in range(0,10):
        indices = np.where(labels[:,i] == 1)
        if (i % 2 == 0):
            new_labels[indices,0] = 1
        else:
            new_labels[indices,1] = 1
    return new_labels


images = mnist.test.images[:3000]
labels = map_labels(mnist.test.labels[:3000])


# tf Graph Input
x = tf.placeholder(tf.float32, [None, 784]) # mnist data image of shape 28*28=784

y = tf.placeholder(tf.float32, [None, 2]) # even / odd recognition ==> 2 classes

# Set model weights
W = tf.Variable(tf.zeros([784, 2]))
b = tf.Variable(tf.zeros([2]))

# Construct model
pred = tf.nn.softmax(tf.matmul(x, W) + b) # Softmax

# Minimize error using cross entropy
cost = tf.reduce_mean(-tf.reduce_sum(y*tf.log(pred), reduction_indices=1))
# Gradient Descent
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)

```

```
51 # Initialize the variables (i.e. assign their default value)
52 init = tf.global_variables_initializer()
53 # Start training
54 with tf.Session() as sess:
55     sess.run(init)
56
57     # Training cycle
58     for epoch in range(training_epochs):
59         avg_cost = 0.
60         total_batch = int(mnist.train.num_examples/batch_size)
61         # Loop over all batches
62         for i in range(total_batch):
63             batch_xs, batch_ys = mnist.train.next_batch(batch_size)
64             batch_ys = map_labels(batch_ys)
65             # Fit training using batch data
66             _, c = sess.run([optimizer, cost], feed_dict={x: batch_xs,
67                                                           y: batch_ys})
68             # Compute average loss
69             avg_cost += c / total_batch
70         # Display logs per epoch step
71         my_prediction = tf.argmax(pred,1)
72         correct_prediction = tf.equal(tf.argmax(pred, 1), tf.argmax(y, 1))
73         accuracy = tf.reduce_mean(tf.cast(correct_prediction,tf.float32))
74
75         if (epoch+1) % display_step == 0:
76             print("Epoch:", '%04d' % (epoch+1), "cost=", "{:.9f}".format(avg_cost))
77
78
79     print("Optimization Finished!")
80     print ("Final Accuracy:", accuracy.eval({x: images, y: labels}))
```

The main differences are in the new *map_labels*() function that takes in a group of labels, and assigns one hot labels, mapping digits to even/odd. This particular method allows the accuracy to be easily calculated through $tf.argmax$ and $tf.reduce\_mean$ on lines 72-73, and allows simple processing of input labels. The main changes for this code other than the *map_labels* function are

changing the y shape from [None, 10] $\rightarrow$ [None, 2]

changing the W shape from [None, 10] $\rightarrow$ [None,2]

changing the b shape from [10] $\rightarrow$ [2]

The output from running this code is below

```
1  Epoch: 0001 cost= 0.398432264
2  Epoch: 0002 cost= 0.329297669
3  Epoch: 0003 cost= 0.314435494
4  Epoch: 0004 cost= 0.305857888
5  Epoch: 0005 cost= 0.299956152
6  Epoch: 0006 cost= 0.295418508
7  Epoch: 0007 cost= 0.291828701
8  Epoch: 0008 cost= 0.288805550
9  Epoch: 0009 cost= 0.286255135
10 Epoch: 0010 cost= 0.284050495
11 Epoch: 0011 cost= 0.282173354
```

```
Epoch: 0012 cost= 0.280504572
Epoch: 0013 cost= 0.279000549
Epoch: 0014 cost= 0.277683871
Epoch: 0015 cost= 0.276547608
Epoch: 0016 cost= 0.275437350
Epoch: 0017 cost= 0.274490247
Epoch: 0018 cost= 0.273527861
Epoch: 0019 cost= 0.272722432
Epoch: 0020 cost= 0.271993787
Epoch: 0021 cost= 0.271269350
Epoch: 0022 cost= 0.270681579
Epoch: 0023 cost= 0.270030637
Epoch: 0024 cost= 0.269575211
Epoch: 0025 cost= 0.269076458
Optimization Finished!
Final Accuracy: 0.879
```