

## **Assignment Two**

Denny Sabu  
Kevin Toft  
Lawrence Wong  
Mohaymen Ahmed

## Model Structure

Our initial model was built with a single softmax regression layer. The input is an array of 784 binary digits representing the handwritten digits in the MNIST data set. The desired output is a number 0-9 that best approximates that image. These nonlinear events were dealt with using the softmax regression.

## Loss and Accuracy

We used the cross entropy loss function:

$$-\sum y'_i \log(y_i)$$

This loss function was used to measure the accuracy of our created model and informs the shifting of hyperplanes towards the greatest margin. With this accuracy measure we adapted our model to better identify new data and bring the loss function to an acceptable range. Without an admissible loss function we have no way of knowing how to alter the weights of our model to become more accurate. If the loss function was not used, we would have an overfit model that would be inaccurate when presented with identification tasks it has not seen before. The loss function had an important effect on obtaining the accuracy we achieved.

A loss function measures how far apart the current model is from the provided data. We used a standard loss model for linear regression, which sums the squares of the deltas between the current model and the provided data. `linear_model - y` creates a vector where each element is the corresponding example's error delta. We call `tf.square` to square that error. Then, we sum all the squared errors to create a single scalar that abstracts the error of all examples using `tf.reduce_sum`.

The accuracy function measures the number of mistakes the finished model makes while identifying images in the testing phase. The percent of mistakes becomes the accuracy measure.

## Training Function and Loop

Originally we were using the Gradient Descent Optimizer and we minimized the cross entropy for the learning algorithm. With the extra credit we used the adam optimizer and we minimized the cross entropy to great effect.

Breaking up the data into separate parts is necessary for accuracy purposes. The rationale behind breaking up the training dataset into training, validation and test sets is that by holding out the entire data set, you can have a model that is able to predict values rather than just recall past inputs. The trade off is that when you have more data, the more accurate your model will be, so that if you're holding out data, then your model will be slightly less accurate. However, if you don't then you don't have a method of evaluating the model. It still makes sense to break the data up because even though we are not using every last bit of data to train our model, as long as we have a large enough number of inputs to train across, we will still have a model that is fairly accurate. For example, the graphs produced from the accuracy readings from earlier on this project show that the accuracy most radically fluxuates when the model is first initialized and rapidly improves. After the first initial climb, it appears to remain around the same accuracy measure, with only slight deviations as new data comes in. For this reason, we

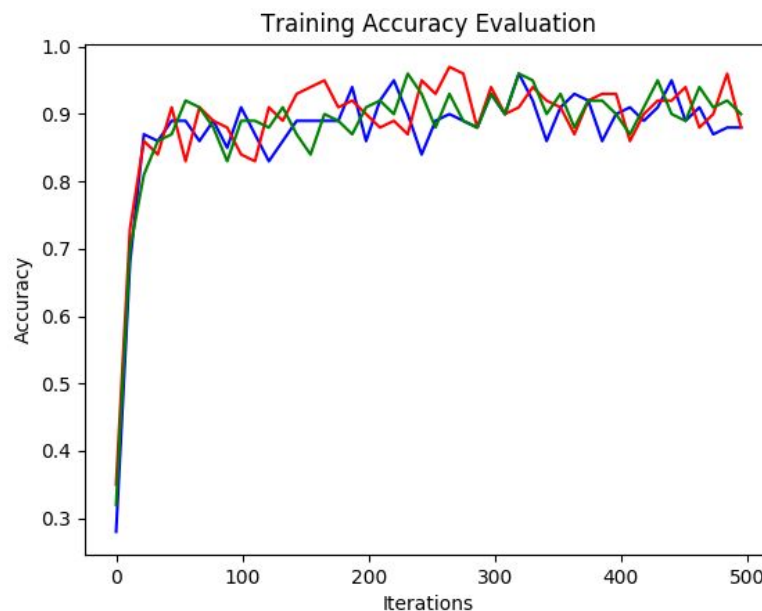
can allow a smaller set of data to train because we will still be floating around the same accuracy, that is within our bounds of error. Validation data is important so that we have isolated, new data that the model is not aware of while training is taking place so we can test how accurately the model is before being finalized. This improves its ability to predict what it observes later. Testing data evaluates the model and produces the accuracy metric of the model. This tells if our model is actually useable or not.

### Accuracy Plot / Conclusions

All 3 Accuracies are around the same percentages throughout the iterations. Since there is no radical difference between them, we have not under fitted or over fitted. This is partly due to the large training data set and the Gradient Descent Optimizer algorithm utilized.

From a theoretical perspective, one can expect to see the validation and test accuracy follow the training accuracy. The model improves after many training iterations with the help of validation data so as we use test data to test the model we should see output accuracy improve because our model is learning on a greater number of data samples. Both the validation and test accuracies should float around the reported accuracy of the training data because the training data is the baseline for how accurate our model is. We can expect slight, minute variations because of the held out data but all the accuracies should be within a reasonable margin of the training data.

*(Note that training, validation, and evaluation lines are blue, red and green respectively for all graphs)*



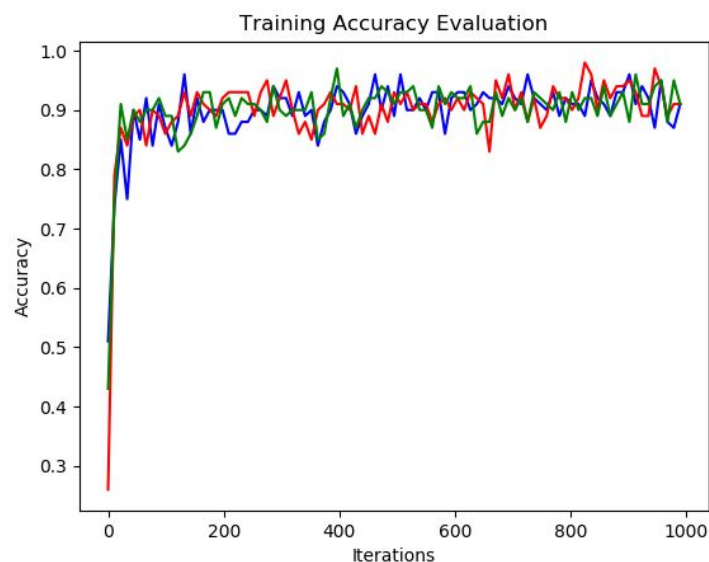
From the model it's apparent that initially the accuracy is poor but improves rapidly till it reaches the bounds of the algorithm where it seems to float around 93% accuracy. With each iteration the data continues to modify the network as the graph never seems to really stabilize but continues to bounce around the 90% - 95% range. This makes sense because, as it

receives new data, it continually updates the weights and biases of the network to best model the data that it has been trained on.

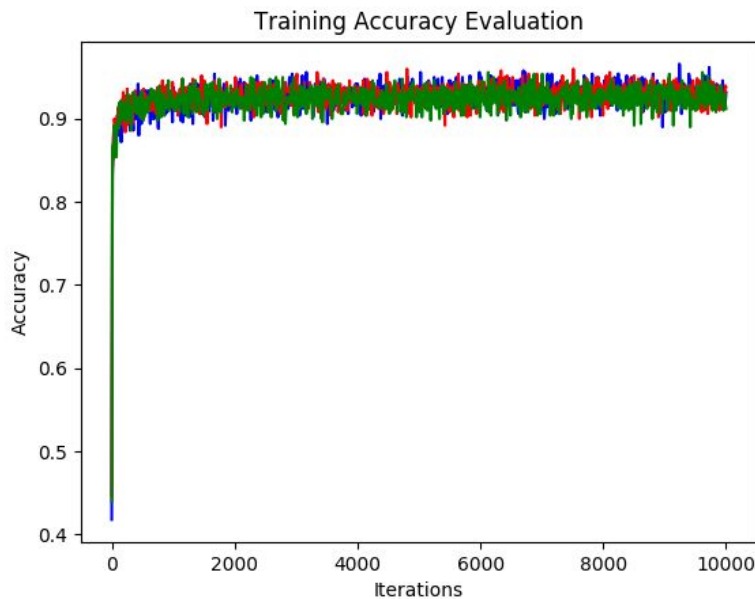
We saw an interesting effect when we increased the range of the for loop where we train, validate, and test our data. The consistency of the data varied more and we would see the accuracy plot dip as low as 80% when we were testing with 1000 iterations. This trend continued until we increased the batch size to 200, when we saw the line occasionally dip below the 90% mark but never approached the lows that we had seen previously.



This is because we are using a larger batch of data to train our model. We can adapt for more input allowing for a larger difference between the handwriting samples. When we train on smaller batches of data, we are not generalizing enough, and we could even say we are overfitting. Below is a figure of running 1000 iterations with a batch size of 100:



As we see here, the accuracy of our graph suffers with a smaller batch size. As we get new data, our current model does not have enough information to accurately model the data and we see the accuracy suffer.



When we increase the number of iterations to 10,000 (pictured above), with a batch size of 500, we see that the model shoots up in accuracy to above .90 and then fluxuates within that range as it continues to get more data and adapt its model to what it has previously seen.

### Integration with A\*

We first thought, that the model created will approximate the level of accuracy of our Agent in A\*, but there may be outliers. Our agent on the graph identifies the image as the wrong digit value every single time with the first basic model. This is extremely odd since the plot says it would be accurate with at least 93% certainty. Sometimes our generated model has a higher max training and testing value than a previous model generated from the same model code. With another model created from the same instance, our agent reports back the correct digit for most images but this appears to be through sheer luck of this single model being a bit more accurate with a max of 98%.

We think the difference can be explained by the premise that machine learning is always limited by the amount of data. This is a case of overfitting.

### Extra Credit:

Results → 0.992 Max Train Accuracy and 1.0 Max Test Accuracy

You might be wondering how we got such dank results, let me explain...

There were 4 steps in this implementation:

- Create a softmax regression function that is a model for recognizing MNIST digits, based on looking at every pixel in the image

- Use Tensorflow to train the model to recognize digits by having it "look" at thousands of examples (and run our first Tensorflow session to do so)
- Check the model's accuracy with our test data
- Build, train, and test a multilayer convolutional neural network to improve the results

Our loss function is the cross-entropy between the target and the softmax activation function applied to the model's prediction using `soft_max_cross_entropy_with_logits` instead of the old `tf.reduce_mean` and `tf.reduce_sum`. This is an improvement since `tf.nn.softmax_cross_entropy_with_logits` internally applies the softmax on the model's unnormalized model prediction and sums across all classes, and `tf.reduce_mean` takes the average over these sums.

To create the model, we created a lot of weights and biases. We initialized weights with a small amount of noise for symmetry breaking, and to prevent 0 gradients. Since we're using ReLU a.k.a Rectified Linear Unit neurons, we initialized them with a slightly positive initial bias to avoid "dead neurons". Rectifier neurons have the following advantages:

- Biological plausibility: One-sided, compared to the antisymmetry of tanh.
- Sparse activation: For example, in a randomly initialized network, only about 50% of hidden units are activated (having a non-zero output).
- Better gradient propagation: Fewer vanishing gradient problems compared to sigmoidal activation functions that saturate in both directions.<sup>[13]</sup>
- Efficient computation: Only comparison, addition and multiplication.
- Scale-invariant:  $\max(0, ax) = a \max(0, x)$  for  $a \geq 0$ .

Instead of doing this repeatedly while we build the model, we created two handy functions to do it for us.

```
def weight_variable(shape):  
    initial = tf.truncated_normal(shape, stddev=0.1)  
    return tf.Variable(initial)  
  
def bias_variable(shape):  
    initial = tf.constant(0.1, shape=shape)  
    return tf.Variable(initial)
```

Next was adding Convolution and Pooling to our weights using `tf.nn.max_pool` and `tf.nn.conv2d` with the following functions.

```
def conv2d(x, W):  
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')
```

```
def max_pool_2x2(x):  
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1], strides=[1, 2, 2, 1],  
        padding='SAME')
```

### Adding the first layer:

It will consist of convolution, followed by max pooling. The convolution will compute 32 features for each 5x5 patch. Its weight tensor will have a shape of [5, 5, 1, 32]. The first two dimensions are the patch size, the next is the number of input channels, and the last is the number of output channels. We will also have a bias vector with a component for each output channel. To apply the layer, we first reshape `x` to a 4d tensor, with the second and third dimensions corresponding to image width and height, and the final dimension corresponding to the number of color channels. We then convolve `x_image` with the weight tensor, add the bias, apply the ReLU function, and finally max pool. The `max_pool_2x2` method will reduce the image size to 14x14.

### Adding the second layer:

The second layer will have 64 features for each 5x5 patch. Now that the image size has been reduced to 7x7, we add a fully-connected layer with 1024 neurons to allow processing on the entire image. We reshape the tensor from the pooling layer into a batch of vectors, multiply by a weight matrix, add a bias, and apply a ReLU.

### Adding Dropout:

To reduce overfitting, we will apply dropout before the readout layer. We create a placeholder for the probability that a neuron's output is kept during dropout. This allows us to turn dropout on during training, and turn it off during testing. TensorFlow's `tf.nn.dropout` op automatically handles scaling neuron outputs in addition to masking them, so dropout just works without any additional scaling.

Finally, we added a layer, just like for the one layer softmax regression above.

### Training and Evaluation:

To train and evaluate it we used code that is nearly identical to that for the simple one layer SoftMax network from the first attempt. The differences are that:

- We replaced the steepest gradient descent optimizer with the more sophisticated ADAM optimizer.
- We included the additional parameter `keep_prob` in `feed_dict` to control the dropout rate.
- We added logging to every 100th iteration in the training process.

The final test set accuracy after running this code should be approximately 99.2% and hit 100% for training accuracy with 20,000 iterations. You can see this graph below. With a greater

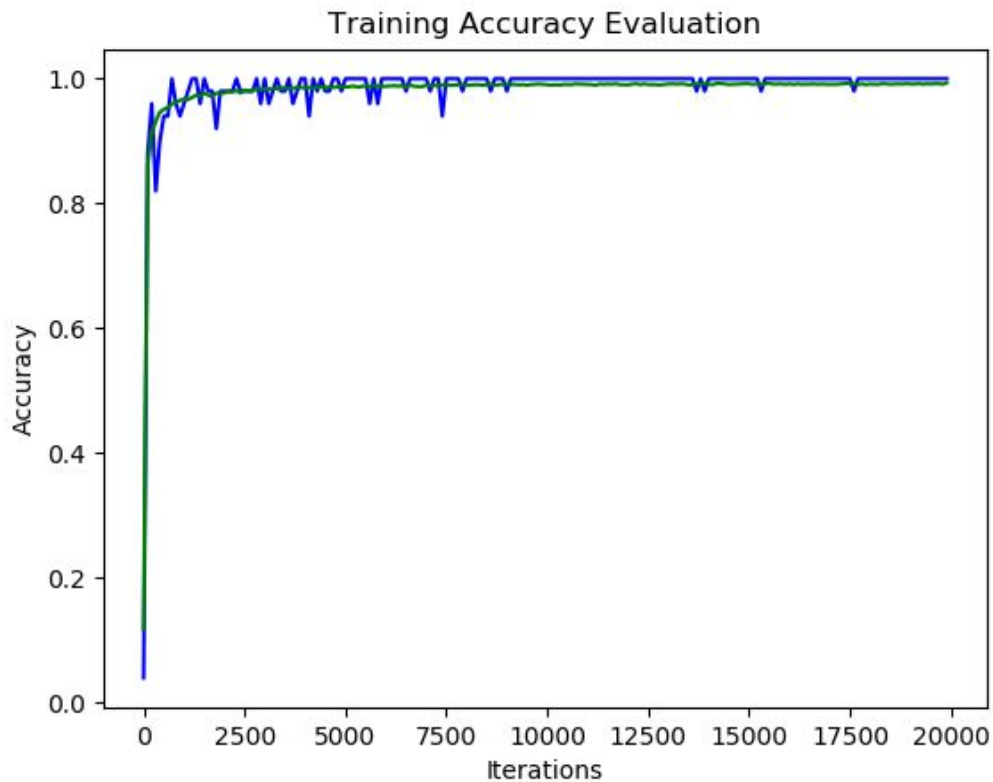
number of iterations like 50,000 then we suspect that the result would be slightly higher on average.

Moral of the story: We have learned how to quickly and easily build, train, and evaluate a fairly sophisticated deep learning model using TensorFlow despite being mathematically illiterate.

***BEHOLD THE GLORY!***

Training Evaluation = BLUE

Testing Evaluation = GREEN

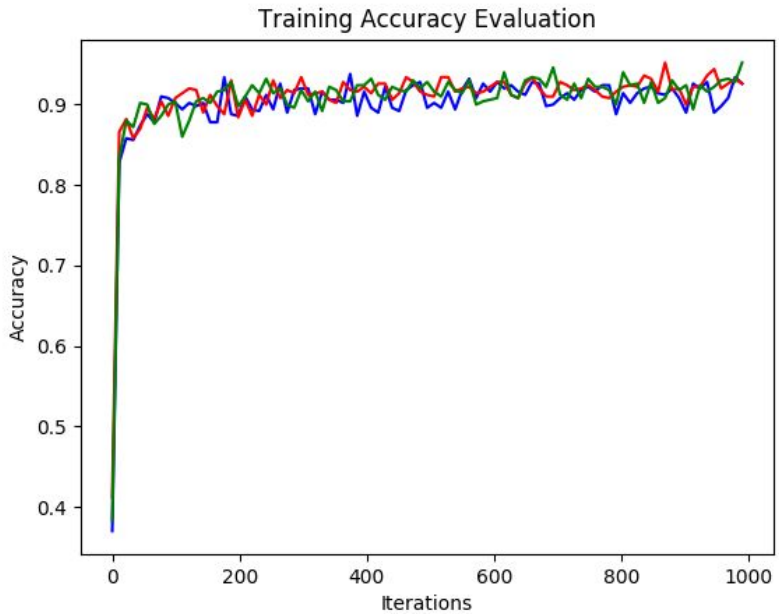
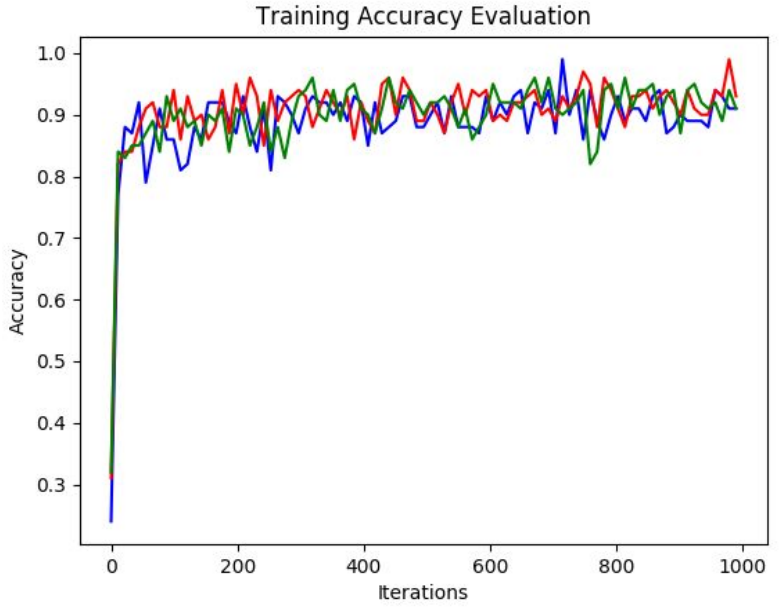




Intro to Artificial Intelligence  
Assignment 2

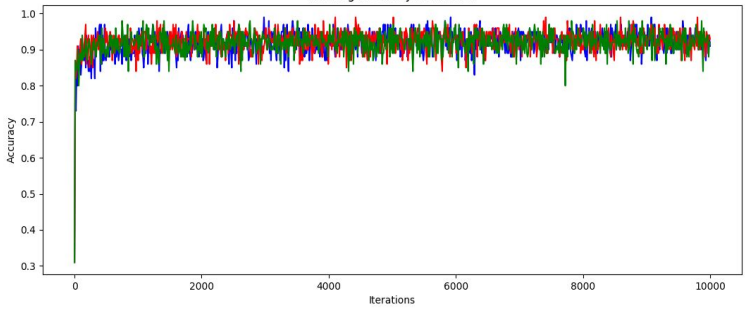
Below are the various tests we attempted on the original model:

test\_train\_model\_ASSIGNMENT\_FILE.py

Iterations	Batch size	
1000	500	
1000	100	

# Intro to Artificial Intelligence

## Assignment 2

10000	100	<p>Training Accuracy Evaluation</p> 
10000	500	<p>Training Accuracy Evaluation</p> 