Michael Russo

### Roman Stashchyshyn

### Ramanjit Mangat

MODEL BLOCK

We started off using a single linear layer. This only got us to about 92% accuracy, but it achieved this level of accuracy pretty quickly while training. In order to improve this, we integrated a multilayer convolutional neural network which was able to train with much higher accuracy. We expanded our model to have two convolutional layers. The first convolutional layer was able to compute 32 features for each 5x5 patch and the second convolutional layer was stacked on top and was able to compute 64 features for each 5x5 patch. This allowed us to implement learnable weights and biases. Each neuron receives several inputs, takes a weighted sum, and passes it through an activation function to respond with an output. A lot of the things we have developed for the regular single linear layer stayed similar, but the addition of the convolutional layers gave us an advantage. The reason for this is because it is taking a dot product between the convolutional filter and a small chunk of the sample image. The first and second convolutional layers have a max pool, which allowed the image size to reduce to 14X14. For every dot product taken, the result becomes a scalar, so when we convolve the complete image with the filter we are able to use these filters, which now become parameters that will be learned by the network SUBSEQUENTLY. So now since we added two convolutional layers and not just one, we do the same thing, except one step deeper. As we go deeper to the next convolutional layer, the filters are doing dot products to the input of the previous convolution layer. The convolutional layers allow for more subsequent learning, for better output of accuracy in the network.

In simpler words, convolutional layers apply a specified number of convolutional filters to each image. For each sub-region, the layer performs a set of mathematical operations to produce a single value in the output feature map. Our convolutional layers then applied a ReLU (rectified linear unit) activation function to the output, which then introduced NONLINEARITIES into the model.

In addition to our convolutional layers, the convolutional neuro network has pooling layers and dense “fully connected layers.” Pooling layers are used to downsample the image data extracted by the convolutional layers to reduce the dimensionality of the feature map in order to decrease processing time. The dense layers perform classification on the features extracted by the convolutional layers and down-sampled by the pooling layers. In a dense layer, every node in the layer is connected to every node in the proceeding layer.

Overall, these layers accept a tensor as input and return a transformed tensor as output. This makes it easy to connect one layer to another.

LOSS AND ACCURACY BLOCK

Our loss function is cross\_entropy = tf.reduce\_mean(tf.nn.softmax\_cross\_entropy\_with\_logits(labels=y\_, logits=y\_conv)). The loss function indicates how bad the models prediction was in a single example. We tried to minimize that while training across all examples. Loss function is not as big of a factor in predicting high accuracy in your model. there are other important things, such as the optimizer you use, which give you a better accuracy. We kept the same loss function from the first model that was only 92% accurate. Variance in models is more important when you have a variance in data.

The accuracy function we used was accuracy = tf.reduce\_mean(tf.cast(correct\_prediction, tf.float32), name='op\_accuracy'). The accuracy function takes in a list of Booleans. It then determins what fraction of the Booleans are correct and takes a mean of the numbers that are correct. For example, if 3 out of 4 predictions match the truth, the accuracy becomes 0.75. Here, our accuracy model is used because we need to see how accurate our model. we see that it is over 98% accurate in predicting what number the images display.

TRAINING FUNCTION BLOCK

In order to transition to a higher accuracy, we switched from the GradientDescentOptimizer to the AdamOptimizer. Typically, learning rates are configured naively at random by the user. At best, the user would leverage unpassed experiences, or other types of learning material, to gain the intuition on what is the best value to use in setting the learning rates. Furthermore, the learning rate affects how quickly our model can converge to local minima. In our case, we were given a good learning rate from the TensorFlow docs that was already optimal for quickly converging to a local minima in the MNIST image set.

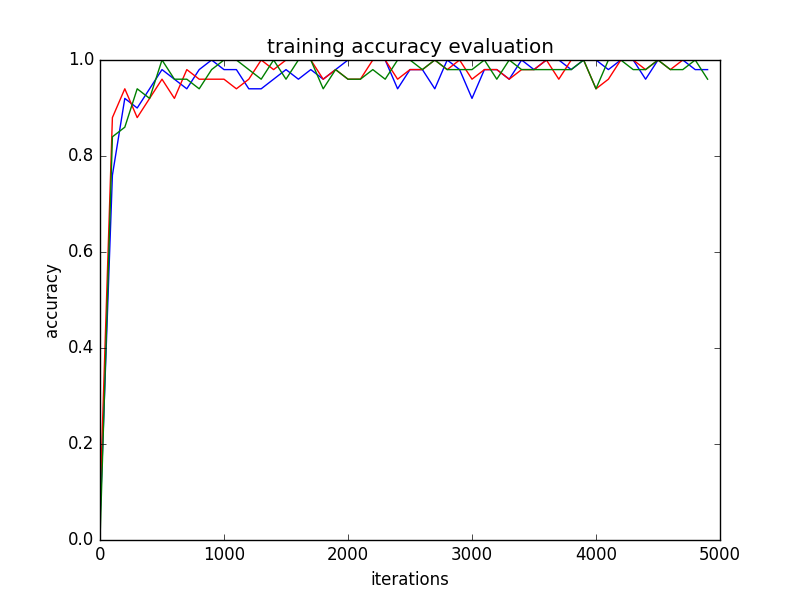
TRAINING LOOK BLOCK

We split up our training data set into training, validation, and test sets. The training data set is important because it is the actual data set that we used to train the model. In our case, since we are using a neural network, this includes the weights and biases. The model sees and learns from this data.

The validation set is used to evaluate a given model, but this is for frequent evaluation. Occasionally, the model sees the validation set, but never learns from it. We used the validation set results and update higher level hyperparameters. So, the validation set in a way affects the model, but indirectly through the computer programmer who may update the data received from it. Lastly, the test data set provides the gold standard used to evaluate the model. it is only used once a model is completely trained with train and validation data sets.

How you decide to split the three data sets depends on the programmer. However, it mainly depends on the total number of samples in your data and on the model you are actually training. Some models need substantial data to train upon. In this case, you would optimize for larger training sets. Overall, the train/test/validation split ratio is quite specific to your case and it gets easier to judge after building many models. In our case, this was the first model we built, so we followed what the TensorFLow docs recommended for the MNIST data set.

ACCURACY PLOT BLOCK



Legend for graph. Blue= train accuracy. Red = validation accuracy. Green = test accuracy

After 5000 steps, the test accuracy was 0.9868, which was much more accurate than using the single linear layer. As seen on the plot, after the first 100 iterations, all 3 accuracies jump from 0.20 to over 0.80 accuracy. After about 1000 interactions, all 3 accuracies fluctuate pretty regularly between 0.94 and 1.0, with the test accuracy having more of the high peaks, while the train accuracy has more of the low peaks. The validation accuracy is usually pretty similar to the test accuracy. The test data set is usually the highest because it provides the gold standard, after the validation and train accuracies have been completed. Therefore, it will most likely always yield the best results. The train accuracy had some lowers dips because the model is learning from this data.