Surf **Tensor Flow Neural Networks Demo**

Iris With Neural Networks

Name: Nishanth Bhaskara **July 7, 2016**

1) Introduction We are given 150 data points, each of which are 4 - dimensional vectors, and a corresponding class which corresponds to the type of flower that it is. The goal is to use the Tensor Flow library and two different types of neural networks to be able to cluster this data set into 3 groupings, and try to achieve accurate classification.

Part I: Deep Neural Networks

a) Methodology

We use a standard neural network architecture to classify our IRIS data. We first split our 150 data points into 120 for learning, and 30 for testing purposes. We should keep in mind that the small volume of data will adversely affect the accuracy of our results.

b) Code

Using tf.contrib.learn, we can construct a DNN classifier with just one line of code. Furthermore, it is easy to specify the number of layers we want, as well as the number of neurons in each of these hidden layers. Leaving out all of the set-up code, the actual training and testing happen in the lines of code shown below.

```
# Build 3 layer DNN with 10, 20, 10 units respectively.
classifier = tf.contrib.learn.DNNClassifier(hidden_units=[10, 20, 10],\
  n_classes=3)

# Fit model.
classifier.fit(x=x_train, y=y_train, steps=200)

# Evaluate accuracy.
accuracy_score = classifier.evaluate(x=x_test, y=y_test)["accuracy"]
print('Accuracy: {0:f}'.format(accuracy_score))
.
```

c) Results

Below we have the confusion matrix for our results

	Actual Class 1	Actual Class 2	Actual Class 3
Predicted Class 1	8	0	0
Predicted Class 2	0	14	0
Predicted Class 3	0	1	17

Below is the accuracy of our DNN on our test set.

```
Accuracy: 0.933333
```

Lastly, below we have the probabilities the classifier gave to each sample being in a class. In red is the correct class, and if there is no blue, that means the highest percentage also matched with the correct class. If there is blue text, it corresponds to the largest percentage/the classification that the DNN gave the sample.

Sample	Class 1	Class 2	Class 3	Correct Class
1	0.00255513	0.963146	0.0342992	1
2	6.48E-06	0.184529	0.815465	2
3	0.992481	0.0075186	1.13E-09	0
4	0.00165332	0.963993	0.0343533	1
5	0.00277461	0.918272	0.0789532	1
6	0.00380918	0.982331	0.01386	1
7	0.998071	0.00192946	5.75E-12	0
8	0.00012001	0.551666	0.448214	2
9	0.00522768	0.977295	0.0174774	1
10	2.44E-07	0.0207774	0.979222	2
11	3.47E-07	0.0387548	0.961245	2
12	0.992982	0.00701811	1.42E-09	0
13	2.34E-06	0.0456054	0.954392	2
14	0.00188548	0.852341	0.145773	1
15	0.00647622	0.975394	0.0181302	1
16	0.996285	0.00371502	9.37E-11	0
17	0.0422964	0.95599	0.00171316	1
18	0.994022	0.00597789	7.13E-10	0
19	0.995915	0.00408537	1.41E-10	0
20	6.72E-07	0.0203202	0.979679	2
21	0.996408	0.00359225	8.12E-11	0
22	0.00080587	0.827074	0.172121	1
23	1.56E-06	0.0597539	0.940245	2
24	0.00011481	0.482809	0.517077	1
25	0.00886843	0.981563	0.00956857	1
26	0.00130325	0.940863	0.0578341	1
27	0.997549	0.00245088	1.59E-11	0
28	0.00128333	0.971974	0.0267427	1
29	7.31E-08	0.0134275	0.986572	2
30	0.00433767	0.986691	0.00897091	1

Part II: Convolutional Neural Networks

a) Methodology

The fundamental unit of these networks, the neuron/perceptron, remains unchanged. It receives inputs which can be thought of as a vector $(x_0, x_1, x_2, ...)$ and computes the dot product with the weight vector $(w_0, w_1, w_2,)$, and saves this dot product in a number, call it A. We then apply a non-linearity to A (several to choose from), and save it into B. Then, B is returned in the form of the firing rate of the neuron/perceptron. This weight vector changes from neuron to neuron by minimization of a loss function. So how are

convolutional Neural Networks different? The answer is that these convolutional networks are specifically made for image recognition. An image with dimensions H x W x C would need that number of weights per neuron, causing the traditional model to be slow and bulky, not to mention possibly over fit the data. Convolutional Neural Networks take advantage of the fact that input can be thought of as 3 dimensional inputs. (height, width, color channels). The use of filters and partial connectivity from one layer to the next allow the same if not better performance while being exponentially more efficient.

b) Code

The code is commented describing each various layer of our convolutional neural network.

```
# The size of each training batch for out Convolutional Neural Network
# is a command line argument.
BATCHSIZE = (int)(sys.argv[1])
TrainingSteps = 120/BATCHSIZE


# Start our interactive Tensor Flow session.
sess = tf.InteractiveSession()

# Data sets
IRIS_TRAINING = "iris_training.csv"
IRIS_TEST = "iris_test.csv"

# Load datasets.
training_set = tf.contrib.learn.datasets.base.load_csv(filename=IRIS_TRAINING, \
        target_dtype=np.int)
test_set = tf.contrib.learn.datasets.base.load_csv(filename=IRIS_TEST, \
        target_dtype=np.int)

# Load training, test data into appropriate target.
x_train, x_test, y_train, y_test = training_set.data, test_set.data, \
  training_set.target, test_set.target

# Loading the data in a suitable fashion.
trainingCorrect = np.genfromtxt(IRIS_TRAINING, delimiter = ',', skip_header =
1, \
        usecols = (4), dtype = np.int)
testCorrect = np.genfromtxt(IRIS_TEST, delimiter = ',', skip_header =  1, \
        usecols = (4), dtype = np.int)

# Now we will intialize one_hot vectors with the right classifications.
trainingY = np.zeros([trainingCorrect.size,3], np.int)
for i in range(trainingCorrect.size):
        trainingY[i][trainingCorrect[i]] += 1

testY = np.zeros([testCorrect.size,3], np.int)
for i in range(testCorrect.size):
        testY[i][testCorrect[i]] += 1



# Set up place holders for the test data as x and the correct labels
```

```
# stored in y_ which is a one_hot vector.
x = tf.placeholder(tf.float32, shape=[None, 4])
y_ = tf.placeholder(tf.float32, shape=[None, 3])

# FIRST CONVOLUTIONAL LAYER
# We are going to be modeling our IRIS data as a 2 x 2 x 1 image
# since we have 4 features. The first two dimensions are the patch
# size, the third is the number of input channels, and the last
# dimension is how many output channels we have. We also initialize
# a bias vector for each output channel.
W_conv1 = weight_variable([1, 1, 1, 32])
b_conv1 = bias_variable([32])

# We reshape x to a 4D tensor. The second and third dimensions refer
# to height, and the fourth dimension is number of color channels.
x_image = tf.reshape(x, [-1,2,2,1])

# Convolve the x_image with the weight tensor, apply the RELU function,
# and then max-pool. Note that in this specific case we will not max-pool
# because max-pool is over 2x2 blocks and our entire "image" is a 2x2 block.
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = h_conv1

# SECOND CONVOLUTIONAL LAYER
W_conv2 = weight_variable([1, 1, 32, 64])
b_conv2 = bias_variable([64])
h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = h_conv2

# DENSELY CONNECTED LAYER
# 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.
W_fc1 = weight_variable([2 * 2 * 64, 1024])
b_fc1 = bias_variable([1024])
h_pool2_flat = tf.reshape(h_pool2, [-1, 2 * 2 * 64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

# 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.

keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

# READOUT LAYER
```

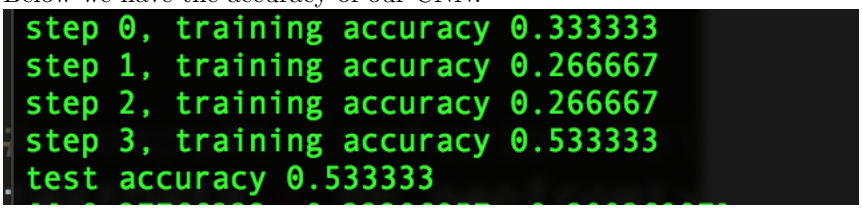
```
# Finally, we add a softmax layer,
W_fc2 = weight_variable([1024, 3])
b_fc2 = bias_variable([3])
y_conv=tf.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)

# TRAIN AND EVALUATE MODEL

# TRAINING MODEL
cross_entropy = tf.reduce_mean(-tf.reduce_sum(y_ * tf.log(y_conv),\
 reduction_indices=[1]))
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
sess.run(tf.initialize_all_variables())
for i in range(TrainingSteps):
  train_accuracy = accuracy.eval(feed_dict={x: training_set.data[BATCHSIZE * \
    i: BATCHSIZE * (i + 1)], y_: trainingY[BATCHSIZE * i: BATCHSIZE * \
    (i + 1)], keep_prob: 1.0})
  print("step %d, training accuracy %g"%(i, train_accuracy))
  train_step.run(feed_dict={x: training_set.data[BATCHSIZE * i: BATCHSIZE * \
    (i + 1)], y_: trainingY[BATCHSIZE * i: BATCHSIZE * (i + 1)], \
     keep_prob: 0.5})

# TESTING MODEL ON TEST SET
print("test accuracy %g"%accuracy.eval(feed_dict={
    x: x_test, y_: testY, keep_prob: 1.0}))

# Printing out the probabilities of the classification for each one.
print(y_conv.eval(feed_dict={x: x_test, y_:testY, keep_prob: 1.0}))
.
```

c) Results

Keep in mind that this entire neural network is made specifically images with several hundreds of feautures, in which case the methodology of using filters has great results. In this case where we are modeling our data with $2x2$ grayscale images, we can obviously expect to have subpar (in this case horrendous) results. Below we have the confusion matrix for our results.

	Actual Class 1	Actual Class 2	Actual Class 3
Predicted Class 1	8	0	0
Predicted Class 2	0	0	0
Predicted Class 3	0	14	8

Below we have the accuracy of our CNN.



Lastly, below we have the probabilities the classifier gave to each sample being in a class. In red is the correct class, and if there is no blue, that means the highest percentage also matched with the correct class. If there is blue text, it corresponds to the largest percentage/the classification that the CNN gave the sample.

Sample	Class 1	Class 2	Class 3	Correct Class
1	0.371671	0.128967	0.499363	1
2	0.347279	0.124356	0.528366	2
3	0.451697	0.133066	0.415237	0
4	0.374397	0.126433	0.49917	1
5	0.367205	0.131802	0.500992	1
6	0.367615	0.127403	0.504982	1
7	0.490878	0.126206	0.382916	0
8	0.352531	0.124637	0.522832	2
9	0.377021	0.128422	0.494557	1
10	0.332063	0.124131	0.543805	2
11	0.337877	0.121939	0.540184	2
12	0.489055	0.132413	0.378533	0
13	0.357668	0.128142	0.51419	2
14	0.362174	0.132888	0.504938	1
15	0.359206	0.13208	0.508714	1
16	0.47979	0.12973	0.39048	0
17	0.377297	0.132242	0.490461	1
18	0.463917	0.128534	0.407548	0
19	0.474379	0.130029	0.395592	0
20	0.348093	0.129438	0.522469	2
21	0.4654	0.128958	0.405641	0
22	0.369799	0.127585	0.502616	1
23	0.346053	0.12286	0.531087	2
24	0.347834	0.127693	0.524473	1
25	0.37169	0.130374	0.497936	1
26	0.365526	0.126312	0.508163	1
27	0.492564	0.125909	0.381527	0
28	0.362047	0.124957	0.512996	1
29	0.347554	0.123151	0.529295	2
30	0.366439	0.127011	0.506551	1

Conclusions: We obviously see that our DNN had a much higher accuracy than our CNN. We can attribute this to the fact that the small amount of features in our data set don't work well with the CNN methodology. In this case obviously a normal fully connected neural network works much better.