Deep Learning with TensorFlow Tutorial

Deep Learning Winter School, University of Hull
January 22-23 2019
Nina Dethlefs, Code from Darren Bird

Our first tutorial was based on Keras [1]. Keras runs with two different backends that we can choose from – Theano [2] and TensorFlow [3]. This tutorial will give a brief introduction to TensorFlow using the same learning task as before – MNIST handwritten digit recognition.[1] See Figure 1 for a reminder of the way that images are represented in TensorFlow.

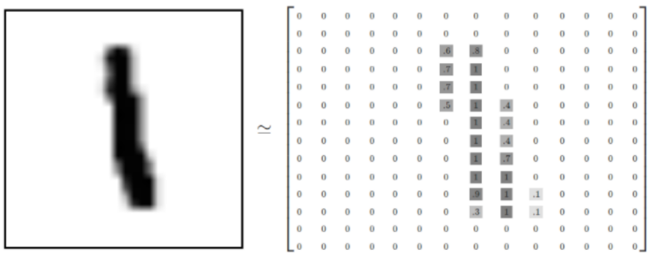
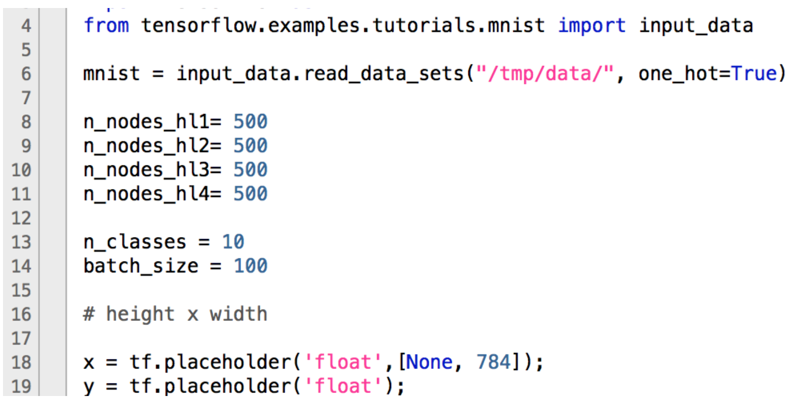


Figure 1: Example of digit representation for input to neural net.

Lets go through the code. As before, we import libraries and load the MNIST data from TensorFlow. Our model will have four hidden layers (500 output units each) and one output layer. We will have 10 classes and use a batch size of 100. Lines 18 and 19 represent the input as floats. The training data x has the shape (None, 784), where None can be variable, e.g. the number of examples, and each training example will have a length of 784 (the number of pixels).



```
 4  from tensorflow.examples.tutorials.mnist import input_data
 5
 6  mnist = input_data.read_data_sets("/tmp/data/", one_hot=True)
 7
 8  n_nodes_hl1= 500
 9  n_nodes_hl2= 500
10  n_nodes_hl3= 500
11  n_nodes_hl4= 500
12
13  n_classes = 10
14  batch_size = 100
15
16  # height x width
17
18  x = tf.placeholder('float',[None, 784]);
19  y = tf.placeholder('float');
```

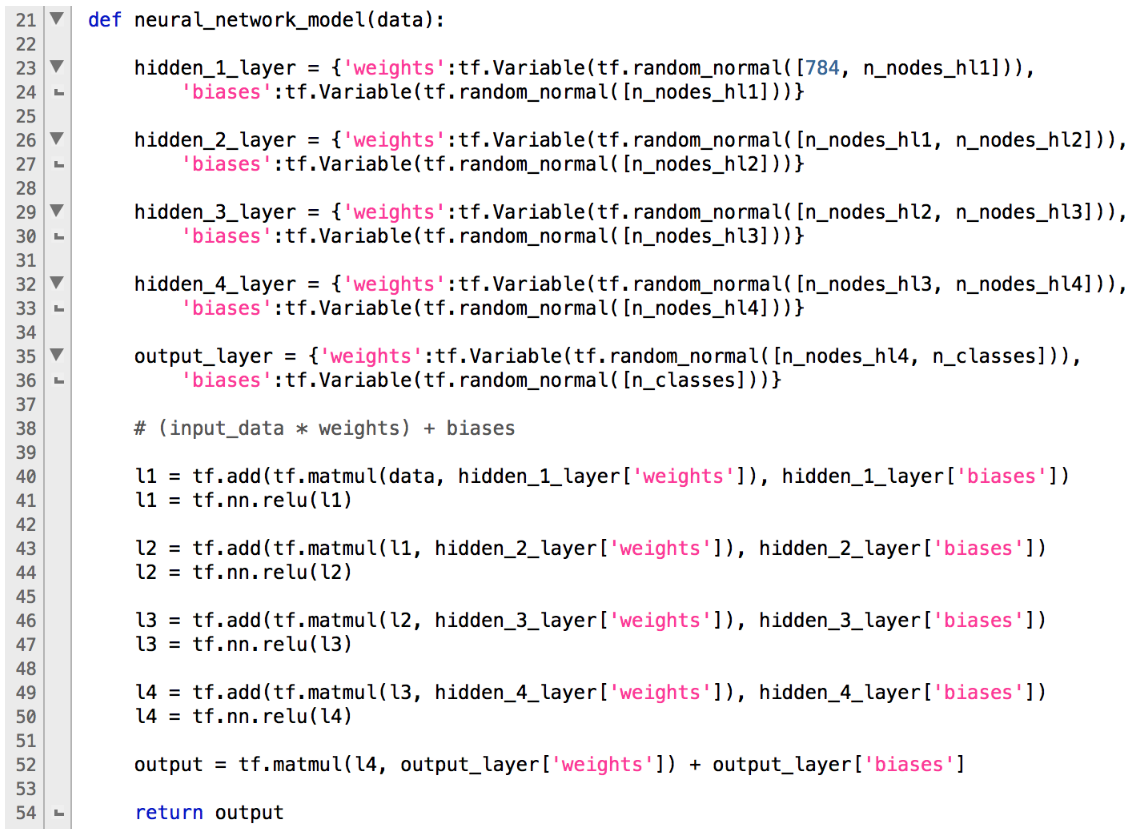
The method *neural_network_data* (lines 21-52) specifies the architecture of our neural net. We have 5 layers that get initiated with random values drawn from a normal distribution. Each layer specifies the number of input units it expects (784 for the first layer) and the number of outputs that it returns (500 for the first layer corresponding to *n_nodes_hl1* specified above). Each layer also explicitly specifies its biases.

[1]http://yann.lecun.com/exdb/mnist/

Lines 35-45 feed the data through the network. We can see that *data* serves as input to the first layer, and the output of the first layer *l1* then serves as input to the second layer *l2*, etc. Each layer has an activation function. Weights at each layer are computed as:

$$(input\_data \times weights) + biases,$$

using TensorFlow operations *add* and *matmul* for matrix multiplication. Read through the code and make sure you can see how and where all of this is happening.

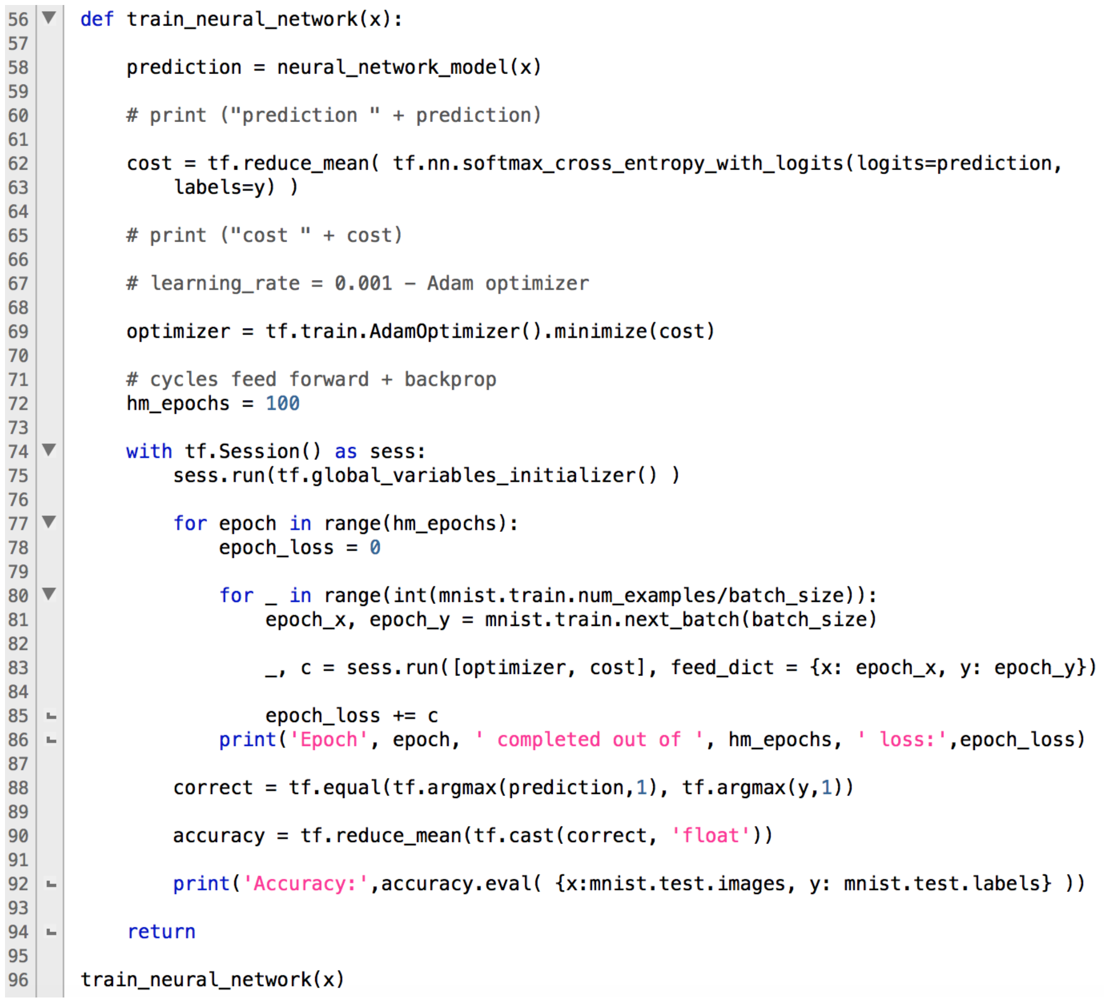


```
21  def neural_network_model(data):
22  
23      hidden_1_layer = {'weights':tf.Variable(tf.random_normal([784, n_nodes_hl1])),
24          'biases':tf.Variable(tf.random_normal([n_nodes_hl1]))}
25  
26      hidden_2_layer = {'weights':tf.Variable(tf.random_normal([n_nodes_hl1, n_nodes_hl2])),
27          'biases':tf.Variable(tf.random_normal([n_nodes_hl2]))}
28  
29      hidden_3_layer = {'weights':tf.Variable(tf.random_normal([n_nodes_hl2, n_nodes_hl3])),
30          'biases':tf.Variable(tf.random_normal([n_nodes_hl3]))}
31  
32      hidden_4_layer = {'weights':tf.Variable(tf.random_normal([n_nodes_hl3, n_nodes_hl4])),
33          'biases':tf.Variable(tf.random_normal([n_nodes_hl4]))}
34  
35      output_layer = {'weights':tf.Variable(tf.random_normal([n_nodes_hl4, n_classes])),
36          'biases':tf.Variable(tf.random_normal([n_classes]))}
37  
38      # (input_data * weights) + biases
39  
40      l1 = tf.add(tf.matmul(data, hidden_1_layer['weights']), hidden_1_layer['biases'])
41      l1 = tf.nn.relu(l1)
42  
43      l2 = tf.add(tf.matmul(l1, hidden_2_layer['weights']), hidden_2_layer['biases'])
44      l2 = tf.nn.relu(l2)
45  
46      l3 = tf.add(tf.matmul(l2, hidden_3_layer['weights']), hidden_3_layer['biases'])
47      l3 = tf.nn.relu(l3)
48  
49      l4 = tf.add(tf.matmul(l3, hidden_4_layer['weights']), hidden_4_layer['biases'])
50      l4 = tf.nn.relu(l4)
51  
52      output = tf.matmul(l4, output_layer['weights']) + output_layer['biases']
53  
54      return output
```

The final method *train_neural_network* (see next page) specifies a loss function, optimiser and training over a certain number of epochs. Read through the code and make sure you understand what is happening and how. You will see that TensorFlow is more low-level than Keras – this can give the developer more control over what is happening but can also lead to longer development times when a higher level specification is sufficient for our learning task.

Exercise - Train the neural net and modify parameters.

- Change the Adam optimiser for training, e.g. to AdaGrad, and observe the effects: `https://www.tensorflow.org/api_docs/python/tf/train/AdamOptimizer`
- Change the cost function that helps minimise the error over time and observe the effects: `https://www.tensorflow.org/api_docs/python/tf/nn/softmax_cross_entropy_with_logits`
- Train a more complex neural network with TensorFlow, e.g. a CNN (follow this tutorial: `https://www.tensorflow.org/tutorials/estimators/cnn`) or an RNN (follow this tutorial: `https://www.tensorflow.org/tutorials/recurrent`). Note that the RNN uses a different task than MNIST.



```python
def train_neural_network(x):

    prediction = neural_network_model(x)

    # print ("prediction " + prediction)

    cost = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(logits=prediction,
        labels=y) )

    # print ("cost " + cost)

    # learning_rate = 0.001 - Adam optimizer

    optimizer = tf.train.AdamOptimizer().minimize(cost)

    # cycles feed forward + backprop
    hm_epochs = 100

    with tf.Session() as sess:
        sess.run(tf.global_variables_initializer() )

        for epoch in range(hm_epochs):
            epoch_loss = 0

            for _ in range(int(mnist.train.num_examples/batch_size)):
                epoch_x, epoch_y = mnist.train.next_batch(batch_size)

                _, c = sess.run([optimizer, cost], feed_dict = {x: epoch_x, y: epoch_y})

                epoch_loss += c
            print('Epoch', epoch, ' completed out of ', hm_epochs, ' loss:',epoch_loss)

        correct = tf.equal(tf.argmax(prediction,1), tf.argmax(y,1))

        accuracy = tf.reduce_mean(tf.cast(correct, 'float'))

        print('Accuracy:',accuracy.eval( {x:mnist.test.images, y: mnist.test.labels} ))

    return

train_neural_network(x)
```

References

[1] F. Chollet, “Keras,” https://github.com/fchollet/keras, 2016.

[2] Theano Development Team, “Theano: A Python framework for fast computation of mathematical expressions,” *arXiv e-prints*, vol. abs/1605.02688, May 2016. [Online]. Available: http://arxiv.org/abs/1605.02688

[3] M. Abadi, A. Agarwal, P. Barham, and *et al.*, “TensorFlow: Large-scale machine learning on heterogeneous systems,” 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/