Assignment 3

May 5, 2018

1 EECS 531: Computer Vision Assignment 3

David Fan

3/30/18

In this notebook we will be exploring different packages for training and configuring deep neural networks.

1.1 Set Up Keras

```
In [54]: import keras
         from keras.models import Sequential
         from keras.layers import Dense, Flatten, Conv2D, MaxPooling2D, Dropout
         from keras import backend as K
```

We'll be using the mnist dataset for these first tests to attempt to replicate the demo's work.

```
In [35]: from keras.datasets import mnist
         (imgTrain, labelTrain), (imgTest, labelTest) = mnist.load_data()
```

1.2 Format dataset

As Keras is essentially just an upper level API for different deep neural network package backends, we need to shape our image data to ensure that it matches the format that the backend we're using expects. Currently we're using Tensorflow as Keras's backend.

```
In [51]: def format_data(imgTrain, imgTest):
             imgRows, imgCols = 28, 28

             if K.image_data_format() == 'channels_first':
                 imgTrain = imgTrain.reshape(imgTrain.shape[0], 1, imgRows, imgCols)
                 imgTest  = imgTest.reshape(imgTest.shape[0], 1, imgRows, imgCols)
                 smpSize  = (1, imgRows, imgCols)
             else:
                 imgTrain = imgTrain.reshape(imgTrain.shape[0], imgRows, imgCols, 1)
                 imgTest  = imgTest.reshape(imgTest.shape[0], imgRows, imgCols, 1)
                 smpSize  = (imgRows, imgCols, 1)
```

```
            imgTrain = imgTrain.astype('float') / 255
            imgTest  = imgTest.astype('float') / 255

            print('Training set in shape of ', imgTrain.shape, ' with element type ', type(img
            print('Testing set in shape of  ', imgTest.shape, ' with element type ', type(img

            return (imgTrain, imgTest, smpSize)

        imgTrain, imgTest, smpSize = format_data(imgTrain, imgTest)
```

```
Training set in shape of  (60000, 28, 28, 1)  with element type  <class 'float'>
Testing set in shape of   (10000, 28, 28, 1)  with element type  <class 'float'>
```

Currently the labels are in the form of a single scalar with the numerical value of the handwritten digit. For our model we would prefer them as one-hot categorical vectors so we us a convenient built in keras function for this:

```
In [37]: def label_to_onehot(labelTrain, labelTest):
            ncat = 10

            onehotTrain = keras.utils.to_categorical(labelTrain, ncat)
            onehotTest  = keras.utils.to_categorical(labelTest, ncat)

            return (onehotTrain, onehotTest)

        onehotTrain, onehotTest = label_to_onehot(labelTrain, labelTest)
```

2 Replicating the Demo

2.1 Define the model

Now we use Keras to define our neural network. We first define the model as a sequential model which means it's just a linear stack of layers. This is a Keras built in model definition for simple model structures. Next, we add in a few layers to build the same model that the demo built: 1. A convolution transformation with 32 3x3 filters. Here we use the ReLU activation function. An explanation of the ReLU activation function can be found [here](https://en.wikipedia.org/wiki/Rectifier_(neural_networks). Essentially it just returns the positive part of its argument. We use it here due to its computational ease. 2. A max pooling layer with a 2x2 filter to perform non-linear down-sampling. 3. A flattening layer to reshape the data for the final linear transformation. 4. A regular linear transformation layer using softmax activation. Softmax is a really commonly used activation function for the final layer of a neural network. More detail can be found [here](https://en.wikipedia.org/wiki/Rectifier_(neural_networks).

```
In [50]: def simple_model(smpSize):
            model = Sequential()
            model.add(Conv2D(32, kernel_size=(3, 3),
                             activation='relu',
```

```
                        input_shape=smpSize))
    model.add(MaxPooling2D(pool_size=(2, 2)))
    model.add(Flatten())
    model.add(Dense(10, activation='softmax'))

    return model

model = simple_model(smpSize)
```

2.2 Compile the model

Here we compile our model. We use the cross-entropy loss function. A nice explanation of cross-entropy can be found here. Keras requires us to define an "optimizer" when we compile the model so we'll be using the Adadelta optimizer with the default parameters. We will specify the accuracy metric for the model output.

```
In [45]: def compile_simple(model):
             model.compile(loss=keras.losses.categorical_crossentropy,
                           optimizer=keras.optimizers.Adadelta(),
                           metrics=['accuracy'])

         compile_simple(model)
```

2.3 Fit the model

Here we tell Keras to fit our model. Not much about the parameters needs to be explained here except for the `batch_size` and the `epochs` parameters. `batch_size` defines the number of samples per gradient update while `epochs` defines the number of iterations over the entire input dataset.

```
In [46]: def fit_simple(model):
             model.fit(imgTrain, onehotTrain, validation_data=(imgTest, onehotTest), batch_size

         fit_simple(model)
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/3
60000/60000 [==============================] - 24s 397us/step - loss: 0.3214 - acc: 0.9097 - va
Epoch 2/3
60000/60000 [==============================] - 24s 404us/step - loss: 0.1229 - acc: 0.9655 - va
Epoch 3/3
60000/60000 [==============================] - 25s 415us/step - loss: 0.0888 - acc: 0.9747 - va
```

2.4 Evaluate the model

Here we tell Keras to evaluate our model and provide the metrics we defined earlier. We can now see the accuracy of our model on the mnist dataset:

```
In [47]: def evaluate(model):
             score = model.evaluate(imgTest, onehotTest, verbose=0)
             print('Test loss     :', score[0])
             print('Test accuracy :', score[1])

         evaluate(model)
```

```
Test loss     : 0.07534846454262734
Test accuracy : 0.9771
```

2.5 Fashion MNIST

Let's train the previous architecture on the Fashion MNIST dataset. The dataset is another built in Keras dataset that can be found on their website. It's the same shape and size as the MNIST dataset, but instead of handwritten digits it's articles of clothing. We're using this dataset for comparison because it doesn't require us to change anything about the shape of our network so we can make a quick and easy direct comparison.

```
In [48]: from keras.datasets import fashion_mnist
         (imgTrain, labelTrain), (imgTest, labelTest) = fashion_mnist.load_data()
```

```
In [49]: imgTrain, imgTest = format_data(imgTrain, imgTest)
         onehotTrain, onehotTest = label_to_onehot(labelTrain, labelTest)
         model = simple_model()
         compile_simple(model)
         fit_simple(model)
         evaluate(model)
```

```
Training set in shape of  (60000, 28, 28, 1)  with element type  <class 'float'>
Testing set in shape of   (10000, 28, 28, 1)  with element type  <class 'float'>
Train on 60000 samples, validate on 10000 samples
Epoch 1/3
60000/60000 [==============================] - 24s 405us/step - loss: 0.5669 - acc: 0.8036 - va
Epoch 2/3
60000/60000 [==============================] - 31s 508us/step - loss: 0.3713 - acc: 0.8706 - va
Epoch 3/3
60000/60000 [==============================] - 24s 396us/step - loss: 0.3278 - acc: 0.8859 - va
Test loss     : 0.3372094687461853
Test accuracy : 0.8837
```

3 Making a More Complex Network

First we need to get our MNIST data again.

```
In [52]: (imgTrain, labelTrain), (imgTest, labelTest) = mnist.load_data()
         imgTrain, imgTest, smpSize = format_data(imgTrain, imgTest)
         onehotTrain, onehotTest = label_to_onehot(labelTrain, labelTest)
```

```
Training set in shape of  (60000, 28, 28, 1)  with element type  <class 'float'>
Testing set in shape of   (10000, 28, 28, 1)  with element type  <class 'float'>
```

3.1 Define the model

This time around we're going to define a more complex CNN. We'll be using the following layers: 1. A convolutional layer using the ReLU activation function with 32 5x5 filters. 2. A max-pooling layer with a 2x2 filter. 3. A convolutional layer using the ReLU activation function with 64 5x5 filters. 4. A max-pooling layer with a 2x2 filter. 5. A flattening layer 6. A dense layer using the ReLU activation function 7. A dropout layer with rate 0.4 8. A final dense layer using softmax activation

```
In [55]: model = Sequential()
         model.add(Conv2D(32, kernel_size=(5, 5),
                          activation='relu',
                          input_shape=smpSize))
         model.add(MaxPooling2D(pool_size=(2, 2)))
         model.add(Conv2D(64, kernel_size=(5, 5),
                          activation='relu',
                          input_shape=smpSize))
         model.add(MaxPooling2D(pool_size=(2, 2)))
         model.add(Flatten())
         model.add(Dense(1024, activation='relu'))
         model.add(Dropout(.4))
         model.add(Dense(10, activation='softmax'))
```

3.2 Compile the model

```
In [56]: model.compile(loss=keras.losses.categorical_crossentropy,
                       optimizer=keras.optimizers.Adadelta(),
                       metrics=['accuracy'])
```

3.3 Fit the model

```
In [57]: model.fit(imgTrain, onehotTrain, validation_data=(imgTest, onehotTest), batch_size=128

Train on 60000 samples, validate on 10000 samples
Epoch 1/3
60000/60000 [==============================] - 68s 1ms/step - loss: 0.1950 - acc: 0.9384 - val_
Epoch 2/3
60000/60000 [==============================] - 70s 1ms/step - loss: 0.0484 - acc: 0.9852 - val_
Epoch 3/3
60000/60000 [==============================] - 70s 1ms/step - loss: 0.0337 - acc: 0.9895 - val_


Out[57]: <keras.callbacks.History at 0x181e2af208>
```

3.4 Evaluate the model

```
In [58]: score = model.evaluate(imgTest, onehotTest, verbose=0)
         print('Test loss     :', score[0])
         print('Test accuracy :', score[1])
```

```
Test loss     : 0.03277401327027474
Test accuracy : 0.9886
```

We were able to squeeze nearly a whole percentage point more of accuracy with our more complex model!

4 Complex model in Tensorflow

For comparison purposes let's see how difficult it would be to create the exact same model using Tensorflow. We were using Tensorflow as Keras's backend, but here we're going to skip the Keras API and use Tensorflow directly.

```
In [60]: import numpy as np
         import tensorflow as tf
```

```
In [61]: tf.logging.set_verbosity(tf.logging.INFO)
```

```
In [94]: def cnn_model_fn(features, labels, mode):
             input_layer = tf.reshape(features["x"], [-1, 28, 28, 1])
             conv1 = tf.layers.conv2d(
                 inputs=input_layer,
                 filters=32,
                 kernel_size=[5,5],
                 padding="same",
                 activation=tf.nn.relu)
             pool1=tf.layers.max_pooling2d(inputs=conv1, pool_size=[2,2], strides=2)
             conv2=tf.layers.conv2d(
                 inputs=pool1,
                 filters=64,
                 kernel_size=[5,5],
                 padding="same",
                 activation=tf.nn.relu)
             pool2 = tf.layers.max_pooling2d(inputs=conv2, pool_size=[2,2], strides=2)

             pool2_flat = tf.reshape(pool2, [-1, 7*7*64])
             dense=tf.layers.dense(inputs=pool2_flat, units=1024, activation=tf.nn.relu)
             dropout=tf.layers.dropout(inputs=dense,rate=0.4, training=mode == tf.estimator.Mo

             logits = tf.layers.dense(inputs=dropout, units=10)

             predictions = {
                 'classes': tf.argmax(input=logits, axis=1),
```

```
            'probabilities': tf.nn.softmax(logits, name="softmax_tensor")
        }

        if mode == tf.estimator.ModeKeys.PREDICT:
            return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)

        onehot_labels = tf.one_hot(indices=tf.cast(labels, tf.int32), depth=10)
        loss=tf.losses.softmax_cross_entropy(onehot_labels=onehot_labels, logits=logits)
        if mode == tf.estimator.ModeKeys.TRAIN:
            optimizer = tf.train.AdagradOptimizer(learning_rate=0.01)
            train_op = optimizer.minimize(
                loss=loss,
                global_step=tf.train.get_global_step())
            return tf.estimator.EstimatorSpec(mode=mode, loss=loss, train_op=train_op)

        eval_metric_ops = {
            "accuracy": tf.metrics.accuracy(
                labels=labels, predictions=predictions["classes"])
        }
        return tf.estimator.EstimatorSpec(mode=mode, loss=loss, eval_metric_ops=eval_metri
```

```
In [95]: mnist = tf.contrib.learn.datasets.load_dataset("mnist")
         train_data = mnist.train.images
         train_labels = np.asarray(mnist.train.labels, dtype=np.int32)
         eval_data = mnist.test.images
         eval_labels = np.asarray(mnist.test.labels, dtype=np.int32)
```

```
Extracting MNIST-data/train-images-idx3-ubyte.gz
Extracting MNIST-data/train-labels-idx1-ubyte.gz
Extracting MNIST-data/t10k-images-idx3-ubyte.gz
Extracting MNIST-data/t10k-labels-idx1-ubyte.gz
```

```
In [96]: mnist_classifier = tf.estimator.Estimator(model_fn=cnn_model_fn, model_dir = "/tmp/mni
```

```
INFO:tensorflow:Using default config.
INFO:tensorflow:Using config: {'_model_dir': '/tmp/mnist_convnet_model', '_tf_random_seed': Non
```

```
In [97]: tensors_to_log = {"probabilities": "softmax_tensor"}
         logging_hook = tf.train.LoggingTensorHook(tensors=tensors_to_log, every_n_iter=50)
```

```
In [98]: train_input_fn = tf.estimator.inputs.numpy_input_fn(
             x={"x": train_data},
             y=train_labels,
             batch_size=128,
             num_epochs=3,
             shuffle=True)
         mnist_classifier.train(
```

```
            input_fn=train_input_fn,
            hooks=[logging_hook])

INFO:tensorflow:Calling model_fn.
WARNING:tensorflow:From /Users/david/anaconda3/envs/531/lib/python3.6/site-packages/tensorflow/
Instructions for updating:

Future major versions of TensorFlow will allow gradients to flow
into the labels input on backprop by default.

See tf.nn.softmax_cross_entropy_with_logits_v2.

INFO:tensorflow:Done calling model_fn.
INFO:tensorflow:Create CheckpointSaverHook.
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Saving checkpoints for 1 into /tmp/mnist_convnet_model/model.ckpt.
INFO:tensorflow:probabilities = [[0.09947185 0.09428356 0.09020665 ... 0.09754863 0.10479341 0
 [0.0967101  0.09268875 0.09332853 ... 0.1029065  0.09900474 0.11861431]
 [0.10116234 0.10283292 0.09386651 ... 0.10139182 0.09759238 0.11258373]
 ...
 [0.09149599 0.11180939 0.09391053 ... 0.08764391 0.09967451 0.1082143 ]
 [0.09633005 0.1092521  0.08950268 ... 0.08119695 0.10351996 0.10277978]
 [0.09181795 0.13115564 0.09331577 ... 0.08530012 0.11110679 0.09617934]]
INFO:tensorflow:loss = 2.3007069, step = 1
INFO:tensorflow:probabilities = [[0.16294982 0.01564095 0.18915127 ... 0.04732589 0.10805734 0
 [0.02334074 0.03296491 0.25243276 ... 0.01475261 0.03117588 0.06313489]
 [0.24458086 0.00708354 0.08159756 ... 0.04739382 0.04835165 0.05682703]
 ...
 [0.01033448 0.12818936 0.06270529 ... 0.10654939 0.05971275 0.17056784]
 [0.02407295 0.20687887 0.07200382 ... 0.07012981 0.07408266 0.08678001]
 [0.0261441  0.03804741 0.10530289 ... 0.23520046 0.0615381  0.12177011]] (14.981 sec)
INFO:tensorflow:global_step/sec: 3.34925
INFO:tensorflow:probabilities = [[0.01014767 0.00271574 0.00891862 ... 0.02328641 0.04799221 0
 [0.00837994 0.1959683  0.04187519 ... 0.00278944 0.67123    0.0049191 ]
 [0.00051308 0.00029656 0.00059476 ... 0.02793286 0.00751582 0.70151377]
 ...
 [0.03709065 0.00799334 0.35414723 ... 0.28755927 0.0586246  0.07911873]
 [0.00011939 0.00015525 0.99561226 ... 0.00000662 0.00080754 0.0000436 ]
 [0.00066493 0.00038886 0.84478563 ... 0.00000463 0.00245093 0.00021767]] (14.877 sec)
INFO:tensorflow:loss = 0.48616648, step = 101 (29.858 sec)
INFO:tensorflow:probabilities = [[0.00009292 0.00025598 0.01759389 ... 0.00000065 0.00079261 0
 [0.00023898 0.00014943 0.00072459 ... 0.00339273 0.08739515 0.04159139]
 [0.00212298 0.00182488 0.00016852 ... 0.00009217 0.00445666 0.00027431]
 ...
 [0.98725003 0.00000002 0.00023351 ... 0.00000019 0.00043408 0.00000033]
 [0.0000641  0.00000037 0.00000034 ... 0.99783736 0.00002933 0.00176801]
```

```
 [0.00019792 0.96796304 0.00419816 ... 0.00513519 0.0018126  0.00291081]] (16.492 sec)
INFO:tensorflow:global_step/sec: 3.17555
INFO:tensorflow:probabilities = [[0.00042438 0.00521736 0.02258301 ... 0.01975702 0.05622964 0
 [0.00000522 0.00008564 0.9955758  ... 0.00000028 0.00021611 0.00001443]
 [0.00017883 0.00025233 0.9952721  ... 0.00015711 0.00053635 0.00045557]
 ...
 [0.9948944  0.00000008 0.0023525  ... 0.00011878 0.00001626 0.00005656]
 [0.03194182 0.00000335 0.18936646 ... 0.00011831 0.00070013 0.00235979]
 [0.00004581 0.00001719 0.00193001 ... 0.00400541 0.00600153 0.00079912]] (14.998 sec)
INFO:tensorflow:loss = 0.3363195, step = 201 (31.490 sec)
INFO:tensorflow:probabilities = [[0.00236716 0.00015754 0.00124899 ... 0.00008524 0.04167992 0
 [0.0000001  0.0000048  0.00000895 ... 0.00003156 0.00016102 0.00330366]
 [0.00025821 0.00051284 0.00001336 ... 0.00028808 0.00108309 0.00009956]
 ...
 [0.9899733  0.00000001 0.00088849 ... 0.00008061 0.00001562 0.00005295]
 [0.00014943 0.95851785 0.00403992 ... 0.00634    0.00496228 0.00621394]
 [0.00000149 0.0000159  0.00047916 ... 0.00017804 0.00403377 0.01547192]] (14.788 sec)
INFO:tensorflow:global_step/sec: 3.39807
INFO:tensorflow:probabilities = [[0.000213   0.00004493 0.00011842 ... 0.00032703 0.00143717 0
 [0.00030415 0.02948698 0.93913656 ... 0.00355162 0.00157145 0.00004089]
 [0.01141334 0.00284136 0.00468177 ... 0.00062344 0.931893   0.00105556]
 ...
 [0.01445172 0.03559377 0.06544033 ... 0.00005384 0.7115627  0.00016761]
 [0.00038292 0.00000148 0.12296771 ... 0.00000047 0.00011692 0.00000569]
 [0.99716455 0.0000001  0.00066401 ... 0.0000034  0.00006644 0.00000011]] (14.640 sec)
INFO:tensorflow:loss = 0.25406113, step = 301 (29.429 sec)
INFO:tensorflow:probabilities = [[0.0001615  0.9864659  0.00065953 ... 0.00148887 0.00272454 0
 [0.0000491  0.00000046 0.00004642 ... 0.00454162 0.00600808 0.93559086]
 [0.00000014 0.00000001 0.00000045 ... 0.00000036 0.00007045 0.00005346]
 ...
 [0.00000728 0.00501082 0.00200361 ... 0.00000386 0.00190206 0.00012154]
 [0.00000014 0.00000009 0.00000053 ... 0.9998073  0.00000025 0.00007947]
 [0.         0.         0.00000028 ... 0.00000015 0.00007039 0.00003984]] (14.915 sec)
INFO:tensorflow:global_step/sec: 3.29327
INFO:tensorflow:probabilities = [[0.00004788 0.99126536 0.00008544 ... 0.00224134 0.00386411 0
 [0.0000003  0.00000043 0.00000089 ... 0.9996269  0.00010403 0.00021026]
 [0.05886825 0.00383966 0.83194935 ... 0.00402604 0.00941157 0.00020704]
 ...
 [0.980802   0.00000103 0.00019383 ... 0.00059249 0.00002863 0.00003399]
 [0.00003563 0.00000102 0.00069949 ... 0.00000321 0.00005589 0.00000101]
 [0.00000027 0.00001492 0.00001834 ... 0.00001397 0.00004829 0.00239975]] (15.451 sec)
INFO:tensorflow:loss = 0.17895089, step = 401 (30.365 sec)
INFO:tensorflow:probabilities = [[0.00001499 0.00002202 0.0002814  ... 0.01720001 0.00046539 0
 [0.00003993 0.00059335 0.00002182 ... 0.00660945 0.0020574  0.98186284]
 [0.00052749 0.00003903 0.00541881 ... 0.01104139 0.67614526 0.26397383]
 ...
 [0.0000057  0.00001262 0.00099355 ... 0.00858089 0.00205586 0.9800236 ]
 [0.0000527  0.00058065 0.00025106 ... 0.00031513 0.00156917 0.01524691]
```

```
 [0.00009903 0.00001076 0.00102465 ... 0.00000015 0.00017447 0.00000075]] (16.709 sec)
INFO:tensorflow:global_step/sec: 3.11106
INFO:tensorflow:probabilities = [[0.9932152  0.00000527 0.00106191 ... 0.000057   0.00206752 0
 [0.00015326 0.9814527  0.00143052 ... 0.00248656 0.01206923 0.00005225]
 [0.00262255 0.9535782  0.00575175 ... 0.0051392  0.0142943  0.00251858]
 ...
 [0.00001446 0.00023711 0.9945314  ... 0.00001321 0.00052462 0.00001646]
 [0.00008489 0.9955089  0.00047261 ... 0.00122107 0.00159668 0.00026624]
 [0.00169409 0.00000738 0.00284478 ... 0.00000999 0.00012265 0.00000629]] (15.434 sec)
INFO:tensorflow:loss = 0.17285481, step = 501 (32.143 sec)
INFO:tensorflow:probabilities = [[0.00024216 0.0000168  0.0000552  ... 0.9795584  0.00016303 0
 [0.00006816 0.01125861 0.1579321  ... 0.00016555 0.01277759 0.00070321]
 [0.00000011 0.00000002 0.99934345 ... 0.00000001 0.00000756 0.00000029]
 ...
 [0.00202413 0.00002248 0.12166131 ... 0.00054266 0.00012068 0.00511299]
 [0.00077236 0.00008473 0.9130675  ... 0.00031921 0.00882435 0.00174548]
 [0.00016801 0.00000501 0.00000267 ... 0.0000012  0.00045741 0.00000844]] (15.857 sec)
INFO:tensorflow:global_step/sec: 3.19306
INFO:tensorflow:probabilities = [[0.00000616 0.0000031  0.00000116 ... 0.00000024 0.00001345 0
 [0.00000897 0.00000484 0.00002906 ... 0.00010212 0.00047083 0.02779429]
 [0.00097636 0.03511612 0.00201167 ... 0.00003552 0.00479637 0.0025092 ]
 ...
 [0.00000029 0.00000007 0.0009391  ... 0.00000003 0.00000176 0.00000051]
 [0.00001704 0.00008252 0.0000096  ... 0.00000406 0.02056686 0.00003669]
 [0.0000268  0.00000121 0.00001106 ... 0.0180513  0.00018495 0.97128356]] (15.461 sec)
INFO:tensorflow:loss = 0.20480746, step = 601 (31.319 sec)
INFO:tensorflow:probabilities = [[0.9989011  0.00000002 0.0000951  ... 0.00000296 0.00061079 0
 [0.00000001 0.00000005 0.00000496 ... 0.9999913  0.00000004 0.00000315]
 [0.00000035 0.00000008 0.00004391 ... 0.         0.00000713 0.        ]
 ...
 [0.6331035  0.00005086 0.0014994  ... 0.00204592 0.00507527 0.00311168]
 [0.9987174  0.0000001  0.0003866  ... 0.00006101 0.00000951 0.00071851]
 [0.999954   0.00000002 0.00000674 ... 0.0000005  0.00000153 0.0000112 ]] (14.252 sec)
INFO:tensorflow:global_step/sec: 3.54988
INFO:tensorflow:probabilities = [[0.00008914 0.9919952  0.00024178 ... 0.00578251 0.00093296 0
 [0.00001386 0.00002614 0.00210979 ... 0.00008718 0.0001348  0.00699584]
 [0.00023731 0.00004146 0.0014921  ... 0.00004484 0.9925707  0.00424605]
 ...
 [0.00000063 0.0000153  0.00000524 ... 0.00057431 0.00062672 0.8320926 ]
 [0.00002464 0.00000032 0.00088321 ... 0.00000059 0.00004074 0.0000173 ]
 [0.00000975 0.00001524 0.9903094  ... 0.00882894 0.0000039  0.00001754]] (13.918 sec)
INFO:tensorflow:loss = 0.104220234, step = 701 (28.170 sec)
INFO:tensorflow:probabilities = [[0.998895   0.00000363 0.00040399 ... 0.00004001 0.00000599 0
 [0.0031018  0.00000261 0.00003349 ... 0.00015691 0.01229582 0.0119734 ]
 [0.00000018 0.00000002 0.00000339 ... 0.00007247 0.00007766 0.00059537]
 ...
 [0.00002519 0.00051321 0.00772043 ... 0.00001007 0.00247321 0.00001831]
 [0.00309345 0.00000324 0.00327289 ... 0.00000009 0.00221275 0.00000212]
```

```
 [0.         0.         0.00000002 ... 0.00000049 0.00000187 0.00000167]] (13.897 sec)
INFO:tensorflow:global_step/sec: 3.59832
INFO:tensorflow:probabilities = [[0.00000364 0.00000065 0.00000749 ... 0.00103713 0.00120563 0
 [0.00000025 0.         0.00000007 ... 0.00057486 0.00005794 0.9992786 ]
 [0.00010304 0.00005136 0.00084699 ... 0.00004474 0.9961092  0.00049064]
 ...
 [0.00000872 0.99936527 0.00010193 ... 0.00025864 0.00006508 0.00001087]
 [0.00000539 0.00000106 0.0000739  ... 0.00000001 0.00003874 0.00000027]
 [0.00009479 0.00000081 0.00020065 ... 0.00367061 0.00343911 0.9764811 ]] (13.894 sec)
INFO:tensorflow:loss = 0.09241407, step = 801 (27.792 sec)
INFO:tensorflow:probabilities = [[0.00000014 0.         0.00004226 ... 0.00002107 0.99651504 0
 [0.00005293 0.00003043 0.99556726 ... 0.0002578  0.00033151 0.00000156]
 [0.0001427  0.00011539 0.00005581 ... 0.00000024 0.01112207 0.00000893]
 ...
 [0.00006988 0.00016914 0.00007003 ... 0.00000529 0.99946445 0.00008   ]
 [0.00003021 0.00003937 0.00029659 ... 0.00024722 0.02006368 0.02902307]
 [0.82749665 0.00375674 0.00214909 ... 0.00098224 0.14736533 0.00094381]] (13.990 sec)
INFO:tensorflow:global_step/sec: 3.59797
INFO:tensorflow:probabilities = [[0.00000029 0.00000071 0.00001483 ... 0.00000067 0.00326849 0
 [0.00487515 0.00000021 0.01259146 ... 0.00101555 0.00005064 0.97917604]
 [0.02070434 0.00239331 0.0231076  ... 0.00153181 0.8409621  0.03759935]
 ...
 [0.00008158 0.00000062 0.00028023 ... 0.00000019 0.00051591 0.00000034]
 [0.00000004 0.0000007  0.02634886 ... 0.00007645 0.00003947 0.00002193]
 [0.00000747 0.00008074 0.00058243 ... 0.99422705 0.0000973  0.00448534]] (13.803 sec)
INFO:tensorflow:loss = 0.13635287, step = 901 (27.792 sec)
INFO:tensorflow:probabilities = [[0.00005812 0.9833618  0.00012194 ... 0.00043682 0.01139194 0
 [0.00000546 0.00000009 0.00000355 ... 0.00001621 0.00531549 0.7319386 ]
 [0.00002319 0.00000003 0.00019807 ... 0.00000004 0.0001751  0.00001763]
 ...
 [0.00000005 0.00000017 0.9999683  ... 0.00000033 0.00000497 0.00000001]
 [0.00041681 0.06608314 0.0006275  ... 0.00719141 0.09801018 0.60668343]
 [0.00000097 0.00014767 0.99923277 ... 0.0000005  0.0000343  0.00000003]] (13.899 sec)
INFO:tensorflow:global_step/sec: 3.62237
INFO:tensorflow:probabilities = [[0.00273848 0.00145952 0.04449766 ... 0.035784   0.05996764 0
 [0.00000007 0.00000019 0.00000246 ... 0.         0.00027508 0.00000325]
 [0.00002328 0.9995358  0.00022966 ... 0.00001023 0.00012138 0.00000062]
 ...
 [0.00851184 0.00004971 0.00708007 ... 0.00063622 0.97763604 0.00129483]
 [0.00000048 0.00024196 0.999688   ... 0.0000156  0.00000442 0.        ]
 [0.00003293 0.9966472  0.00054133 ... 0.00064462 0.00027689 0.00007806]] (13.707 sec)
INFO:tensorflow:loss = 0.20102808, step = 1001 (27.605 sec)
INFO:tensorflow:probabilities = [[0.00001095 0.00011895 0.00011281 ... 0.00005362 0.00672962 0
 [0.00000015 0.00000038 0.00007134 ... 0.9958668  0.00001111 0.00259174]
 [0.00016138 0.00009309 0.10117069 ... 0.00006071 0.3896886  0.00401816]
 ...
 [0.         0.00000008 0.00002143 ... 0.00000007 0.00002158 0.00006958]
 [0.9993332  0.00000004 0.00062094 ... 0.00000004 0.00001424 0.0000011 ]
```

```
 [0.00065956 0.11441199 0.00535392 ... 0.00281536 0.7867398  0.04977809]] (13.788 sec)
INFO:tensorflow:global_step/sec: 3.63371
INFO:tensorflow:probabilities = [[0.00024568 0.00367791 0.09590948 ... 0.8493384  0.00414044 0
 [0.998539   0.0000001  0.00004748 ... 0.00000063 0.00000101 0.00005181]
 [0.         0.00000018 0.00000933 ... 0.0000004  0.00000096 0.        ]
 ...
 [0.00000002 0.00029081 0.98824173 ... 0.00001473 0.00004819 0.00006508]
 [0.00002221 0.9904969  0.00017271 ... 0.0030445  0.00477487 0.00013575]
 [0.00000036 0.00000022 0.00001098 ... 0.00003837 0.00076187 0.00013274]] (13.732 sec)
INFO:tensorflow:loss = 0.120576896, step = 1101 (27.520 sec)
INFO:tensorflow:probabilities = [[0.00000713 0.00013066 0.0000117  ... 0.00095722 0.00082413 0
 [0.00029359 0.9947943  0.00249471 ... 0.00095691 0.00067203 0.00004693]
 [0.         0.00000108 0.00497717 ... 0.9949904  0.00000028 0.00000003]
 ...
 [0.00132246 0.00002755 0.00438002 ... 0.01605156 0.00597121 0.9648326 ]
 [0.00000297 0.00000436 0.00000757 ... 0.00000004 0.00072237 0.00005678]
 [0.00002027 0.0003414  0.00018124 ... 0.00528606 0.01055471 0.9570233 ]] (13.743 sec)
INFO:tensorflow:global_step/sec: 3.63433
INFO:tensorflow:probabilities = [[0.00000115 0.00000024 0.00006409 ... 0.00000049 0.99989223 0
 [0.9999963  0.00000001 0.00000004 ... 0.00000039 0.00000005 0.0000001 ]
 [0.00000017 0.00000681 0.00004699 ... 0.01007283 0.00071382 0.9847719 ]
 ...
 [0.00045139 0.00166069 0.00060404 ... 0.00085713 0.00844635 0.23027934]
 [0.00000004 0.00000012 0.00009884 ... 0.9998012  0.00000025 0.00001307]
 [0.00045432 0.00000431 0.00001925 ... 0.0000003  0.00025881 0.00003793]] (13.772 sec)
INFO:tensorflow:loss = 0.07438943, step = 1201 (27.516 sec)
INFO:tensorflow:probabilities = [[0.0001483  0.9917613  0.00040647 ... 0.00208627 0.0004143  0
 [0.00000023 0.00000021 0.188586   ... 0.00037548 0.00016426 0.00002359]
 [0.00000339 0.99786735 0.00009669 ... 0.00081882 0.00017196 0.00059526]
 ...
 [0.00000014 0.00001192 0.00005456 ... 0.00000101 0.0000035  0.00038591]
 [0.00000014 0.00017943 0.00022085 ... 0.0008475  0.06253193 0.12329737]
 [0.00029388 0.0000499  0.00001067 ... 0.0002599  0.00015177 0.06137759]] (13.817 sec)
INFO:tensorflow:Saving checkpoints for 1290 into /tmp/mnist_convnet_model/model.ckpt.
INFO:tensorflow:Loss for final step: 0.07628272.
```

```
Out[98]: <tensorflow.python.estimator.estimator.Estimator at 0x181b7b7518>
```

```
In [99]: eval_input_fn = tf.estimator.inputs.numpy_input_fn(
             x={"x": eval_data},
             y=eval_labels,
             num_epochs=3,
             shuffle=False)
         eval_results = mnist_classifier.evaluate(input_fn=eval_input_fn)
         print(eval_results)
```

```
INFO:tensorflow:Calling model_fn.
INFO:tensorflow:Done calling model_fn.
```

```
INFO:tensorflow:Starting evaluation at 2018-05-05-22:40:11
INFO:tensorflow:Graph was finalized.
INFO:tensorflow:Restoring parameters from /tmp/mnist_convnet_model/model.ckpt-1290
INFO:tensorflow:Running local_init_op.
INFO:tensorflow:Done running local_init_op.
INFO:tensorflow:Finished evaluation at 2018-05-05-22:40:31
INFO:tensorflow:Saving dict for global step 1290: accuracy = 0.9738, global_step = 1290, loss =
{'accuracy': 0.9738, 'loss': 0.08406767, 'global_step': 1290}
```

Interestingly we get worse accuracy than when we implemented the model in Keras. This is likely due to a difference in implementations and a lack of optimization for using Tensorflow.