TensorFlow: Estimators workshop

Gregory Teichert

February 2018

In this tutorial, we will learn how to use Deep Neural Networks (DNNs) using the TensorFlow Estimators module. We will look at two examples of predefined estimators, DNNRegressor and DNNClassifier. We will also create custom estimators, including our own versions of DNNRegressor and DNNClassifier, as well as a Convolutional Neural Network (CNN). These are based on the tutorials and code (with modifications) at www.tensorflow.org/versions/r1.4/get_started/estimator, www.tensorflow.org/versions/r1.4/get_started/mnist/beginners, and www.tensorflow.org/versions/r1.4/tutorials/layers [1]. To experiment with neural networks on toy data, take a look at http://playground.tensorflow.org/.

1 DNNRegressor

First, we consider nonlinear regression using DNNs. The given data has two input values and one output. We want a nonlinear function that maps two continuous inputs to one continuous output that models the given data.

We begin by importing the necessary modules, `tensorflow` and `numpy`. `shutil` is used for deleting previous training data, if desired.

```
import tensorflow as tf
import numpy as np
import shutil
```

The remaining modules are for plotting the results of the TensorFlow regression.

```
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
```

The following (optional) command lets us monitor the drop in the loss function during neural network training:

```
tf.logging.set_verbosity(tf.logging.INFO)
```

Now, we read in the data. For this regression example, we are reading from a .csv file with three columns (two inputs, one output) and 150 rows. We'll read it in using NumPy's `genfromtxt` function, directly forming a NumPy array from the .csv. We use slicing to take the first two columns as inputs or "features" and the last column as the output or "labels".

```
# Read in data
dataIn = np.genfromtxt('dataRegression_train.csv',delimiter=',')
features = dataIn[:,0:2]
labels = dataIn[:,2]
```

TensorFlow needs a place to store the data associated with this model. We specify the directory here, and we clear any previous training data. (If you do want to reuse a previously trained model, don't delete this folder.)

[1] As were the original tutorials from TensorFlow's website, these pages are licensed under the Creative Commons Attribution 3.0 License, and code samples are licensed under the Apache 2.0 License.

```
# Define and reset graph directory
model_dir = 'graphDNNRegressor'
shutil.rmtree(model_dir,ignore_errors=True)
```

Now we define the optimizer that TensorFlow will use to train the neural network (i.e. optimize the weights to minimize the loss function). TensorFlow has several options, including GradientDescentOptimizer, AdamOptimizer, AdadeltaOptimizer, etc. I've had good results with the AdagradOptimzer, which we use here. We need to specify a learning rate to initialize it.

```
# Define optimizer
optimizer=tf.train.AdagradOptimizer(learning_rate=0.1)
```

The Estimator class is TensorFlow's high-level tool for working with models. Here, we use a DNNRegressor object to perform nonlinear regression using a fully connected neural network. First, we specify the type and number of features, e.g. two numerical features. Now we define the structure of the DNN with the `hidden_units` argument, where, for example, `[50,130,25]` sets three hidden layers with 50 activation units in the first hidden layer, 130 in the second, and 25 in the final hidden layer:

```
# Get DNN
feature_columns = [tf.feature_column.numeric_column("x", shape=[2])]
dnn = tf.estimator.DNNRegressor(feature_columns=feature_columns,
                                hidden_units=[50,130,25],
                                model_dir=model_dir,
                                optimizer=optimizer)
```

For training, we define an input function, where we set the arrays `features` and `labels` to be the training data. We're using a variation of stochastic gradient descent, so not all training points are used at each optimization step. Instead, we specify a batch size of 10, where the data is shuffled. Also, we'll set the number of training steps directly, so we set no limit on the number of epochs. The `train` function performs the training using the input function.

```
# Fit (train) model
batch_size=10
train_input_fn = tf.estimator.inputs.numpy_input_fn(x={"x": features},
                                                    y=labels,
                                                    batch_size=batch_size,
                                                    num_epochs=None,
                                                    shuffle=True)
# Train model
dnn.train(input_fn=train_input_fn, steps=10000)
```

We can see how well the model generalizes by evaluating the loss function for validation data. This is done with data that was not used during training. There are some small changes to the input function

```
# Validate
dataInValid = np.genfromtxt('dataRegression_valid.csv',delimiter=',')
featuresValid = dataIn[:,0:2]
labelsValid = dataIn[:,2]
valid_input_fn = tf.estimator.inputs.numpy_input_fn(x={"x": featuresValid},
                                                    y=labelsValid,
                                                    batch_size=batch_size,
                                                    num_epochs=1,
                                                    shuffle=False)
loss = dnn.evaluate(input_fn=valid_input_fn)
print(loss)
```

Now we will create a set of inputs and see what the DNN predicts as the output. Note that we define yet another input function, this one for predictions. Also, the output from `estimator.predict` is a generator object, so we loop through each prediction to create a NumPy array for plotting a predicted surface.

```
# Create a prediction set
x_min = np.amin(features,axis=0)
x_max = np.amax(features,axis=0)
x_predict = np.mgrid[x_min[0]:x_max[0]:25j, x_min[1]:x_max[1]:25j].reshape(2,-1).T

predict_input_fn = tf.estimator.inputs.numpy_input_fn(x={"x": x_predict},
                                                      num_epochs=1,
```

```
                                                    shuffle=False)
predictions = dnn.predict(input_fn=predict_input_fn)
y_predict = np.array([p['predictions'] for p in predictions])
```

This last section uses the Matplotlib Python library to plot the training points and the predicted surface.

```
# Plot the actual and predicted values
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

x1 = x_predict[:,0].reshape(25,-1)
x2 = x_predict[:,1].reshape(25,-1)
y1 = y_predict.reshape(25,-1)
ax.scatter(features[:,0], features[:,1], labels, c='r', marker='o', label='Actual');
plt.hold(True);
ax.plot_surface(x1,x2,y1,cmap=cm.coolwarm,linewidth=0,rstride=1,cstride=1)

ax.set_xlabel('a/b')
ax.set_ylabel('c/b')
ax.set_zlabel('Energy')

plt.show()
```

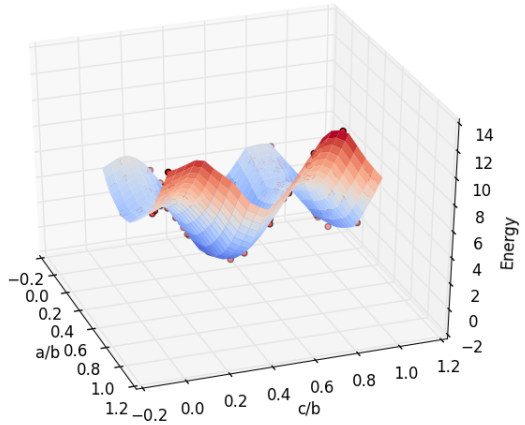

Figure 1: Plot of the training points and DNN predicted surface using the DNNRegressor estimator.

2 Custom DNNRegressor

Now, we will look at how we would construct DNNRegressor as a custom estimator. This is only for demonstration purposes, both to understand the structure of the DNN and to introduce custom estimators, which will be needed for the CNN.

We import the same modules as before.

```
import tensorflow as tf
import numpy as np
import shutil

import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D

tf.logging.set_verbosity(tf.logging.INFO)
```

Now we define the model function. This is what defines the custom estimator. As might be expected, it takes `features` and `labels` as arguments. The next argument, `mode`, defines whether the function is being used for training, evaluation (validation), or prediction. The `params` argument is a dictionary that can hold any other parameters you would like to have.

```
def model_fn(features, labels, mode, params):
```

The first step is define the neural network. In this example, we define the layers directly within `model_fn`, but the number of layers and neurons could be passed in as parameters. We use the `tf.layers` module to define each layer. Since we are creating a standard, fully-connected DNN, we define dense (or fully-connected) layers. The input to the first layer is taken from the dictionary `features`; we will name it "x".The output of each layer is the input for the following layer. We use the commonly used Rectified Linear Units, or `relu`, activation function for all layers except the last. (Since this is a regression problem, no activation function is applied to the output layer.) Since this returns a column vector (a 2D array), we reshape it to be 1D array.

```
    # Define DNN
    Layers = [2,50,130,25,1]
    a1 = tf.layers.dense(inputs=features["x"],units=Layers[1],activation=tf.nn.relu)
    a2 = tf.layers.dense(inputs=a1,units=Layers[2],activation=tf.nn.relu)
    a3 = tf.layers.dense(inputs=a2,units=Layers[3],activation=tf.nn.relu)
    f = tf.layers.dense

    # Reshape output layer to 1-dim Tensor to return predictions
    predictions = tf.reshape(f, [-1])
```

If the function is only be used for predictions, we return the predictions using a TensorFlow `EstimatorSpec` object.

```
    # If called by 'predict' function...
    if mode == tf.estimator.ModeKeys.PREDICT:
        return tf.estimator.EstimatorSpec(mode=mode, predictions={'predictions': predictions
    })
```

If the function is being used for training or validation, the loss function is needed. We use the mean squared error from TensorFlow's collection of loss functions. This is all that is needed if being used for evaluation (validation).

```
    # Calculate loss using mean squared error
    loss = tf.losses.mean_squared_error(labels, predictions)

    # If called by 'evaluate' function...
    if mode == tf.estimator.ModeKeys.EVAL:
        return tf.estimator.EstimatorSpec(mode=mode, loss=loss)

tf.logging.set_verbosity(tf.logging.INFO)
```

For training, we also need to define our optimizer. We take the learning rate from the user defined `params` dictionary, and we specify that we want to minimize the loss function.

```
    # Define optimization
    optimizer = tf.train.AdagradOptimizer(learning_rate=params["learning_rate"])
    train_op = optimizer.minimize(loss=loss, global_step=tf.train.get_global_step())

    # Else, called by 'train' function
    return tf.estimator.EstimatorSpec(mode=mode, loss=loss, train_op=train_op)
```

And we're done! The main function is very similar to what we saw before. Let's look at any differences. To begin, there are no differences in reading the files.

```
if __name__ == '__main__':

    # Read in data
    dataIn = np.genfromtxt('dataRegression_train.csv',delimiter=',')
    features = dataIn[:,0:2]
    labels = dataIn[:,2]

    # Reset graph directory
    model_dir = 'graphDNNRegressor'
    shutil.rmtree(model_dir,ignore_errors=True)
```

The arguments to initialize a custom estimator are a little different than what we saw before. Here, we set the model function (defined above), the model directory (same as before), and the parameters dictionary (used in our example to define the learning rate, but it could be used for other parameters as well).

```
    # Get DNN
    model_params = {"learning_rate": 0.1}
    dnn = tf.estimator.Estimator(model_fn=model_fn,
                                 model_dir=model_dir,
                                 params=model_params)
```

The rest is exactly the same as what we saw with the predefined DNNRegressor.

```
    # Fit (train) model
    batch_size=10
    train_input_fn = tf.estimator.inputs.numpy_input_fn(x={"x": features},
                                                        y=labels,
                                                        batch_size=batch_size,
                                                        num_epochs=None,
                                                        shuffle=True)

    # Train
    dnn.train(input_fn=train_input_fn, steps=10000)

    # Validate
    dataInValid = np.genfromtxt('dataRegression_valid.csv',delimiter=',')
    featuresValid = dataIn[:,0:2]
    labelsValid = dataIn[:,2]
    valid_input_fn = tf.estimator.inputs.numpy_input_fn(x={"x": featuresValid},
                                                        y=labelsValid,
                                                        batch_size=batch_size,
                                                        num_epochs=1,
                                                        shuffle=False)
    loss = dnn.evaluate(input_fn=valid_input_fn)["loss"]
    print loss

    # Create a prediction set
    x_min = np.amin(features,axis=0)
    x_max = np.amax(features,axis=0)
    x_predict = np.mgrid[x_min[0]:x_max[0]:25j, x_min[1]:x_max[1]:25j].reshape(2,-1).T

    predict_input_fn = tf.estimator.inputs.numpy_input_fn(x={"x": x_predict},
                                                          num_epochs=1,
                                                          shuffle=False)
    predictions = dnn.predict(input_fn=predict_input_fn);
    y_predict = np.array([p['predictions'] for p in predictions])

    # Plot the actual and predicted values
    fig = plt.figure()
```

```
    ax = fig.add_subplot(111, projection='3d')

    x1 = x_predict[:,0].reshape(25,-1)
    x2 = x_predict[:,1].reshape(25,-1)
    y1 = y_predict.reshape(25,-1)
    ax.scatter(features[:,0], features[:,1], labels, c='r', marker='o', label='Actual');
    plt.hold(True);
    ax.plot_surface(x1,x2,y1,cmap=cm.coolwarm,linewidth=0,rstride=1,cstride=1)

    ax.set_xlabel('a/b')
    ax.set_ylabel('c/b')
    ax.set_zlabel('Energy')

    plt.show()
```

3 DNNClassifier

We now turn to classification problems. The pre-made DNNClassifier estimator uses a fully-connected DNN to classify based on a set of feature values. The example used in the official tutorial referenced earlier is the classification of three types of irises based on four features. We'll use the same example and data in this example. You will see many similarities between this example and the DNNRegressor example.

We again load our modules. We won't be using Matplotlib in this example

```
import numpy as np
import tensorflow as tf
import shutil

tf.logging.set_verbosity(tf.logging.INFO)
```

The datasets are provided by TensorFlow: http://download.tensorflow.org/data/iris_training.csv, http://download.tensorflow.org/data/iris_test.csv. We'll go ahead and use TensorFlow's functions to load the data, then name them `features` and `labels` as before.

```
# Read in data
training_set = tf.contrib.learn.datasets.base.load_csv_with_header(
  filename="iris_training.csv",
  target_dtype=np.int,
  features_dtype=np.float32)
valid_set = tf.contrib.learn.datasets.base.load_csv_with_header(
  filename="iris_test.csv",
  target_dtype=np.int,
  features_dtype=np.float32)
features = np.array(training_set.data)
labels = np.array(training_set.target)
```

We again define the model directory and choose the AdagradOptimizer.

```
# Reset graph directory
model_dir = 'graphDNNClassifier'
shutil.rmtree(model_dir,ignore_errors=True)

# Define optimizer
optimizer=tf.train.AdagradOptimizer(learning_rate=0.1)
```

There are a couple of changes as we define the DNN. First, we have four features instead of two, which is reflected in the shape of the feature columns. Second, we specify the number of classes or categories into which we want the data to be classified. Since we have three varieties of iris, we set `n_classes=3`.

```
# Get DNN
feature_columns = [tf.feature_column.numeric_column("x", shape=[4])]
dnn = tf.estimator.DNNClassifier(feature_columns=feature_columns,
                                 hidden_units=[10, 20, 10],
                                 n_classes=3,
                                 model_dir=model_dir,
                                 optimizer=optimizer)
```

The training functions are the same as for the DNNRegressor.

```
# Fit (train) model
batch_size=10
train_input_fn = tf.estimator.inputs.numpy_input_fn(x={"x": features},
                                                    y=labels,
                                                    batch_size=batch_size,
                                                    num_epochs=None,
                                                    shuffle=True)
# Train model
dnn.train(input_fn=train_input_fn, steps=10000)
```

One difference that we see in validation is that, since it is a classification problem, we can output the accuracy instead of the value of the loss function. With the current parameters, accuracy should be around 0.96666 (nearly 97%).

```
# Validate
featuresValid = np.array(valid_set.data)
labelsValid = np.array(valid_set.target)
valid_input_fn = tf.estimator.inputs.numpy_input_fn(x={"x": featuresValid},
                                                    y=labelsValid,
                                                    batch_size=batch_size,
                                                    num_epochs=1,
                                                    shuffle=False)
accuracy_score = dnn.evaluate(input_fn=valid_input_fn)["accuracy"]
print "Accuracy: ", accuracy_score
```

To demonstrate prediction, we create two sets of feature values. After calling the `predict` function as before, the predicted iris types are returned.

```
# Classify two new flower samples.
new_samples = np.array([[6.4, 3.2, 4.5, 1.5],
                        [5.8, 3.1, 5.0, 1.7]], dtype=np.float32)
predict_input_fn = tf.estimator.inputs.numpy_input_fn(x={"x": new_samples},
                                                      num_epochs=1,
                                                      shuffle=False)

predictions = dnn.predict(input_fn=predict_input_fn)
predicted_classes = [p["classes"] for p in predictions]

print "New Samples, Class Predictions: ", predicted_classes
```

4 Custom DNNClassifier

We will again look at how to define the pre-made estimator as a custom estimator. The initial lines, as with the DNNRegressor, are the same for the custom estimator.

```
import tensorflow as tf
import numpy as np
import shutil

tf.logging.set_verbosity(tf.logging.INFO)
```

We move into the model function. The initial definition of the deep neural network is the same as we saw with the custom DNNRegressor. We define a few dense layers, with ReLU activation functions on all but the last layer. That was sufficient when performing regression, but an additional function will need to be applied to use the DNN for classification.

```
def model_fn(features, labels, mode, params):

    # Define DNN
    Layers = [2,10,20,10,3]
    a1 = tf.layers.dense(inputs=features["x"], units=Layers[1], activation=tf.nn.relu)
    a2 = tf.layers.dense(inputs=a1, units=Layers[2], activation=tf.nn.relu)
    a3 = tf.layers.dense(inputs=a2, units=Layers[3], activation=tf.nn.relu)
    logits = tf.layers.dense(inputs=a3, units=Layers[4], activation=None)
```

If the output were binary (true or false), we could apply a sigmoid or similar function to acheive a smoothed binary output. In this example, there are three possible categories, so we apply the softmax function to get the probability of each category being true, where the probablities add up to 1:

$$p_i = \frac{\exp(f_i)}{\sum_j \exp(f_j)} \tag{1}$$

where p_i is the probability of category i being true and f_i is the i-th output of the DNN. The predicted category or class is simply the category that had the highest output value f_i. We output both of these values when the model function is used for prediction.

```
    predictions = {
        "classes": tf.argmax(input=logits, axis=1),
        "probabilities": tf.nn.softmax(logits, name="softmax_tensor")
    }

    # If called by 'predict' function...
    if mode == tf.estimator.ModeKeys.PREDICT:
        return tf.estimator.EstimatorSpec(mode=mode, predictions=predictions)
```

The loss function is also different than the MSE used in regression. For classification, we use the cross entropy for the loss function. This requires that the labels from the data be in the same format as our predictions. In other words, the data labels are given as 1, 2, or 3, but we need them to be [1,0,0], [0,1,0], or [0,0,1]. This conversion is done with the `tf.one_hot` function. The cross entropy can then be computed:

$$\text{loss} = \frac{1}{n_{\text{samples}}} \sum_j^{n_{\text{samples}}} \sum_i^{n_{\text{classes}}} \hat{p}_i^j \log(p_i^j) \tag{2}$$

where $\hat{p}_i^j$ is the data and p_i^j is the prediction for class i, sample j. The softmax and cross entropy functions are applied at the same time with the function `tf.losses.softmax_cross_entropy`.

```
    # Calculate loss
    onehot_labels = tf.one_hot(indices=tf.cast(labels, tf.int32), depth=3)
    loss = tf.losses.softmax_cross_entropy(onehot_labels=onehot_labels, logits=logits)
```

For evaluation or validation, the function returns the accuracy and the loss.

```
    # If called by 'evaluate' function...
    if mode == tf.estimator.ModeKeys.EVAL:
        eval_metric_ops = {"accuracy": tf.metrics.accuracy(labels=labels,
                                                           predictions=predictions["classes"
    ])}
        return tf.estimator.EstimatorSpec(mode=mode, loss=loss,
                                          eval_metric_ops=eval_metric_ops)
```

Finally, we define the optimization for when the function is used in training.

```
    # Define optimization
    optimizer = tf.train.AdagradOptimizer(learning_rate=params["learning_rate"])
    train_op = optimizer.minimize(loss=loss, global_step=tf.train.get_global_step())

    # Else, called by 'train' function
    return tf.estimator.EstimatorSpec(mode=mode, loss=loss, train_op=train_op)
```

Again, the changes in the main function are slight when compared to the pre-made DNNClassifier. The loading of data is the same.

```
if __name__ == '__main__':

    # Read in data
    training_set = tf.contrib.learn.datasets.base.load_csv_with_header(
        filename="iris_training.csv",
        target_dtype=np.int,
        features_dtype=np.float32)
    valid_set = tf.contrib.learn.datasets.base.load_csv_with_header(
        filename="iris_test.csv",
        target_dtype=np.int,
        features_dtype=np.float32)
    features = np.array(training_set.data)
    labels = np.array(training_set.target)

    # Reset graph directory
    model_dir = 'graphDNNClassifier_Custom'
    shutil.rmtree(model_dir,ignore_errors=True)
```

The instantiation of the estimator object is different than when using the pre-defined classifier. It is, however, the same as when using the custom DNNRegressor earlier.

```
    # Get DNN
    model_params = {"learning_rate": 0.1}
    dnn = tf.estimator.Estimator(model_fn=model_fn,
                                 model_dir=model_dir,
                                 params=model_params)
```

From here to the end, all is the same as in the example with the pre-defined classifier.

```
    # Fit (train) model
    batch_size=10
    train_input_fn = tf.estimator.inputs.numpy_input_fn(x={"x": features},
                                                        y=labels,
                                                        batch_size=batch_size,
                                                        num_epochs=None,
                                                        shuffle=True)
    # Train model
    dnn.train(input_fn=train_input_fn, steps=10000)

    # Validate
    featuresValid = np.array(valid_set.data)
    labelsValid = np.array(valid_set.target)
    valid_input_fn = tf.estimator.inputs.numpy_input_fn(x={"x": featuresValid},
                                                        y=labelsValid,
                                                        batch_size=batch_size,
                                                        num_epochs=1,
                                                        shuffle=False)
    accuracy_score = dnn.evaluate(input_fn=valid_input_fn)["accuracy"]
    print "Accuracy: ", accuracy_score
```

```
# Classify two new flower samples.
new_samples = np.array([[6.4, 3.2, 4.5, 1.5],
                        [5.8, 3.1, 5.0, 1.7]], dtype=np.float32)
predict_input_fn = tf.estimator.inputs.numpy_input_fn(x={"x": new_samples},
                                                      num_epochs=1,
                                                      shuffle=False)

predictions = dnn.predict(input_fn=predict_input_fn)
predicted_classes = [p["classes"] for p in predictions]

print "New Samples, Class Predictions: ", predicted_classes
```

5 Convolutional Neural Network (CNN)

The final example will present how to construct a Convolutional Neural Network as a custom estimator, since there is no pre-defined CNN estimator. CNNs are commonly used in image classification. The pixel grayscale or RGB values are the input, and the category is the output. We will be using the MNIST handwritten digits data as an example For a more in depth explanation of CNNs and MNIST, see the original tutorial at www.tensorflow.org/versions/r1.4/tutorials/layers.

By now, it's no surprise that the first lines are the same as before...

```
import tensorflow as tf
import numpy as np
import shutil

tf.logging.set_verbosity(tf.logging.INFO)
```

...and that the main changes will happen in the `model_fn` function.

```
def model_fn(features, labels, mode, params):
```

The MNIST images are 28×28 pixel grayscale images (i.e. each pixel has a single grayscale value). Each sample has the feature values given as a vector; this is reshaped into a tensor that is $28 \times 28 \times 1$. We do not specify the number of samples, which is reflected by the "-1".

```
  # Input Layer
  input_layer = tf.reshape(features["x"], [-1, 28, 28, 1])
```

The first convolutional layer is applied. This can be thought of as sweeping a filter across the image, looking for a match in different locations of the image. This allows a CNN to classify an object independently of its location in the image; e.g. an image with a ball in the top left would be as easily recognized as having a ball as an image with a ball in the bottom right. In the current layer, we use 8 filters with a size of 5×5 pixels. The result would be $24 \times 24 \times 8$, but setting `padding="same"` adds zeros to the edges of the output to maintain the dimensions 28×28. We use the ReLU activation functions.

```
  # Convolutional Layer #1
  conv1 = tf.layers.conv2d(inputs=input_layer,
                           filters=8,
                           kernel_size=[5, 5],
                           padding="same",
                           activation=tf.nn.relu)
```

Max pooling is then applied to the output. Max pooling involves looking at clusters of the output (in this example, 2×2 clusters), and set the maximum filter value as the value for the cluster. So if a filter identifies a match anywhere in the cluster, the entire cluster is labeled as having a match for that filter. Since we are also using stride of 2, the clusters don't overlap, reducing the output to $14 \times 14 \times 8$.

```
  # Pooling Layer #1
  pool1 = tf.layers.max_pooling2d(inputs=conv1, pool_size=[2, 2], strides=2)
```

A second convolutional layer, followed by max pooling, is used. The resulting dimensions are now 7×16.

```
  # Convolutional Layer #2 and Pooling Layer #2
  conv2 = tf.layers.conv2d(inputs=pool1,
                           filters=16,
                           kernel_size=[5, 5],
                           padding="same",
                           activation=tf.nn.relu)
  pool2 = tf.layers.max_pooling2d(inputs=conv2, pool_size=[2, 2], strides=2)
```

The 3D tensor is converted back to a 1D tensor of length $7 \times 7 \times 16 = 784$ to act as input for a dense or fully-connected layer, the same type used with the previous regression and classification examples.

```
  # Dense Layer
  pool2_flat = tf.reshape(pool2, [-1, 7 * 7 * 16])
  dense = tf.layers.dense(inputs=pool2_flat, units=200, activation=tf.nn.relu)
```

A regularization method called dropout is used here to prevent overfitting. This could have been used in the other examples as well. Dropout randomly drops weights temporarily from the network . In this example, dropout happens at a rate of 40% (i.e. 40% of weights are temporarily set to zero at each training iteration).

```
  dropout = tf.layers.dropout(inputs=dense,
                              rate=0.4,
                              training=(mode == tf.estimator.ModeKeys.TRAIN))
```

The rest of `model_fn` is the same as the previous classifier example. The output of the neural network has the softmax and cross entropy functions applied, the optimization is defined, etc.

```
  # Logits Layer
  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)

  # Calculate Loss (for both TRAIN and EVAL modes)
  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 called by 'evaluate' function...
  if mode == tf.estimator.ModeKeys.EVAL:
    eval_metric_ops = {"accuracy": tf.metrics.accuracy(labels=labels,
                                                       predictions=predictions["classes"])}
    return tf.estimator.EstimatorSpec(mode=mode, loss=loss,
                                      eval_metric_ops=eval_metric_ops)

  # Define optimization
  optimizer = tf.train.AdagradOptimizer(learning_rate=params["learning_rate"])
  train_op = optimizer.minimize(loss=loss, global_step=tf.train.get_global_step())

  # Else, called by 'train' function
  return tf.estimator.EstimatorSpec(mode=mode, loss=loss, train_op=train_op)
```

The MNIST dataset is divided into training and validation data. These are provided through the TensorFlow library and loaded as NumPy arrays.

```
if __name__ == '__main__':

    # Load datasets
    mnist = tf.contrib.learn.datasets.load_dataset("mnist")
    train_data = mnist.train.images # Returns np.array
    train_labels = np.asarray(mnist.train.labels, dtype=np.int32)
    eval_data = mnist.test.images # Returns np.array
    eval_labels = np.asarray(mnist.test.labels, dtype=np.int32)

    # Reset graph directory
    shutil.rmtree('graphCNN',ignore_errors=True)
```

Training and evaluation is done as before. With the setup in this example, you should achieve an accuracy of over 99%.

```
    # Get DNN
    model_params = {"learning_rate": 0.1}
    dnn = tf.estimator.Estimator(model_fn=model_fn,
                                 model_dir='graphCNN',
                                 params=model_params)

    # Fit (train) model
    batch_size=100
    train_input_fn = tf.estimator.inputs.numpy_input_fn(x={"x": train_data},
                                                        y=train_labels,
                                                        batch_size=batch_size,
```

```
                                                    num_epochs=None,
                                                    shuffle=True)

# Train
dnn.train(input_fn=train_input_fn, steps=5000)

# Validate
valid_input_fn = tf.estimator.inputs.numpy_input_fn(x={"x": eval_data},
                                                    y=eval_labels,
                                                    batch_size=batch_size,
                                                    num_epochs=1,
                                                    shuffle=False)

accuracy = dnn.evaluate(input_fn=valid_input_fn)["accuracy"]
print "Accuracy: ", accuracy
```