Deep Neural Networks with TensorFlow

A Quick Start Introduction using the Estimator Modules

Greg Teichert

Consulting for Statistics, Computing and Analytics Research

March 20, 2018
2-4pm

Workshop computer setup

- To use TensorFlow on Virtual Sites:
  - Choose HTML Access at `midesktop.umich.edu`
  - Launch "TensorFlow 1.5.0" from AppsAnywhere.
  - Open a Command Prompt (search "Command Prompt" using magnifying glass in bottom left corner) and enter the command:

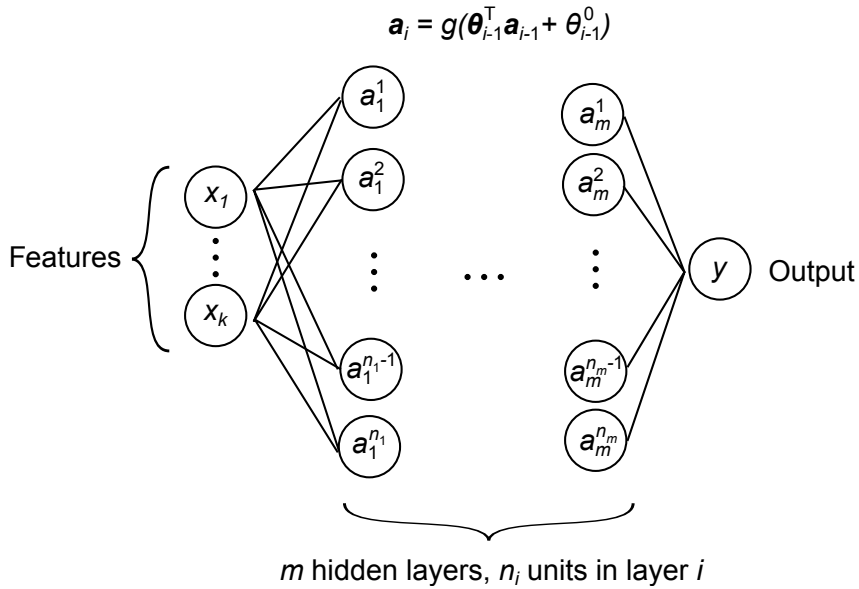
    ```
    C:\VApps\Tensorflow_Python\1.5.0\Scripts\pip3.6.exe install matplotlib
    ```
- Everyone:
  - Go to: `github.com/greght/Workshop-TensorFlow-Estimators`
  - Download the zip file.

Deep Neural Networks (DNNs)

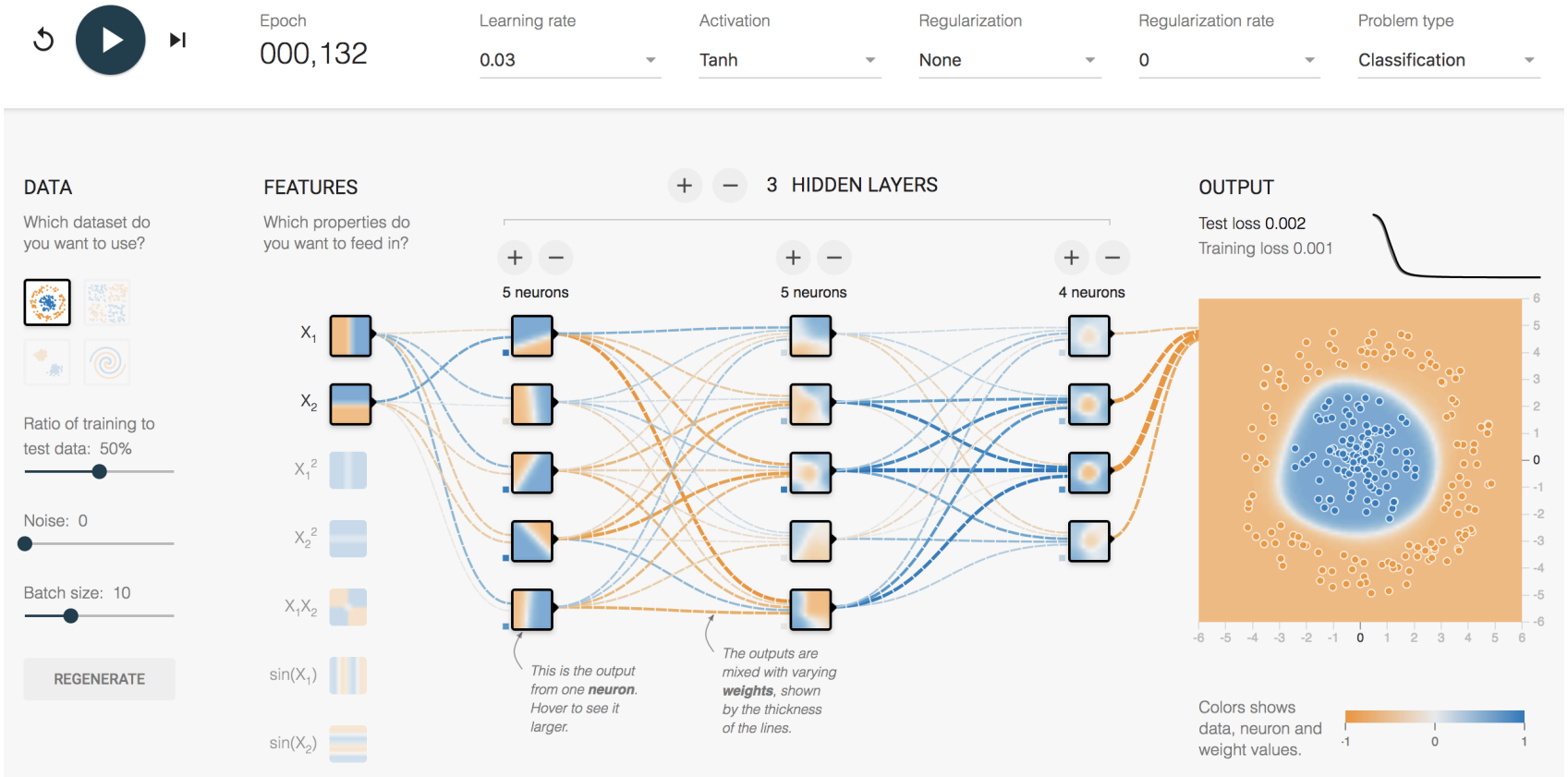
- A DNN is a composite function, combining linear combinations and nonlinear activation functions, e.g.

$$f(x) = A_2 g(A_1 g(A_0 x + b_0) + b_1) + b_2$$

- The coefficients in the linear combinations (weights) and the biases are optimized to "fit" given data (i.e. minimize an objective function, such as the mean squared error).



playground.tensorflow.org



Note: playground.tensorflow.org is an educational tool. It does not actually use the TensorFlow library, nor can you use it to train with your data.

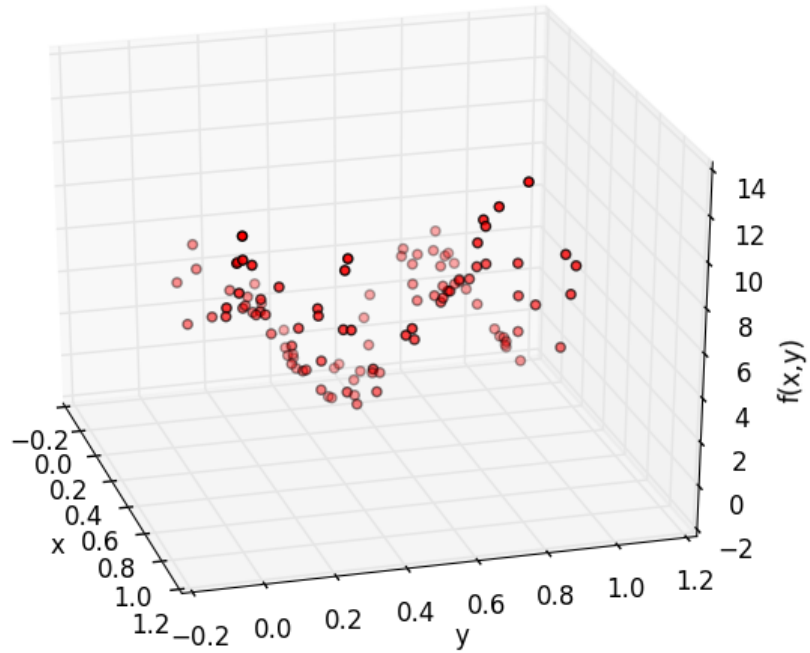
TensorFlow

- tensorflow.org
- Open-source software library widely used for deep learning.
- Most commonly used with Python.
- Provides high-level and low-level user interface functions.
- This workshop will focus on the high-level Estimator modules: nonlinear regression, classification, and Convolutional Neural Networks (CNNs).
- 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, www.tensorflow.org/versions/r1.4/tutorials/layers [1]

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

DNNRegressor

- Begin with DNNRegressor function, a pre-built Estimator.
- Uses a standard DNN to map continuous inputs to continuous outputs.
- Example in `TFintro_DNNRegressor.py`
- Data in example has two inputs, one output (slices parallel to x-axis are parabolic, slices parallel to y-axis are sinusoidal).



Import modules

Import modules (`shutil` used for deleting directories):

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

Import modules for plotting results:

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

Optional command to monitor the drop in the loss function (and other information) during neural network training:

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

Import data

Here, we read in data from a .csv file using NumPy.

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

Specify a directory to hold training information and delete previous info. (Don't delete if you want to reuse data.)

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

Optimizer

TensorFlow has several optimizers available, with different variations on gradient descent:

`GradientDescentOptimizer`, `AdadeltaOptimizer`, `AdamOptimizer`, `AdagradOptimizer`, etc.

Each accepts a learning rate (the step size). Experiment with the different optimizer types and learning rates.

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

DNNRegressor

Define the structure of the DNN. Here, we define three hidden layers, with 50, 130, and 25 neuron in each respective layer.

We can also specify the activation function here. The default is `tf.nn.relu`, but you can use others (see examples):

`tf.nn.sigmoid`, `tf.nn.softplus`, `tf.nn.tanh`, etc.

Experiment with the hidden units and activation function.

```
# 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,
                                activation_fn=tf.nn.relu,
                                optimizer=optimizer)
```

Training

Stochastic gradient descent methods use shuffled mini-batches instead of the entire data set for eaching training iteration.

Batch size is set in the input function. The number of training steps is set when `dnn.train` is called.

Experiment with batch size and number of steps.

```
# 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)
```

Validation

Check how well trained the model is by evaluating the loss function on validation data (not used in training).

```
# 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)
```

Prediction

We create a set of (x,y) points to use for prediction. The `dnn.predict` function returns a generator object, so we loop through to get the numerical values.

```
# 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])
```

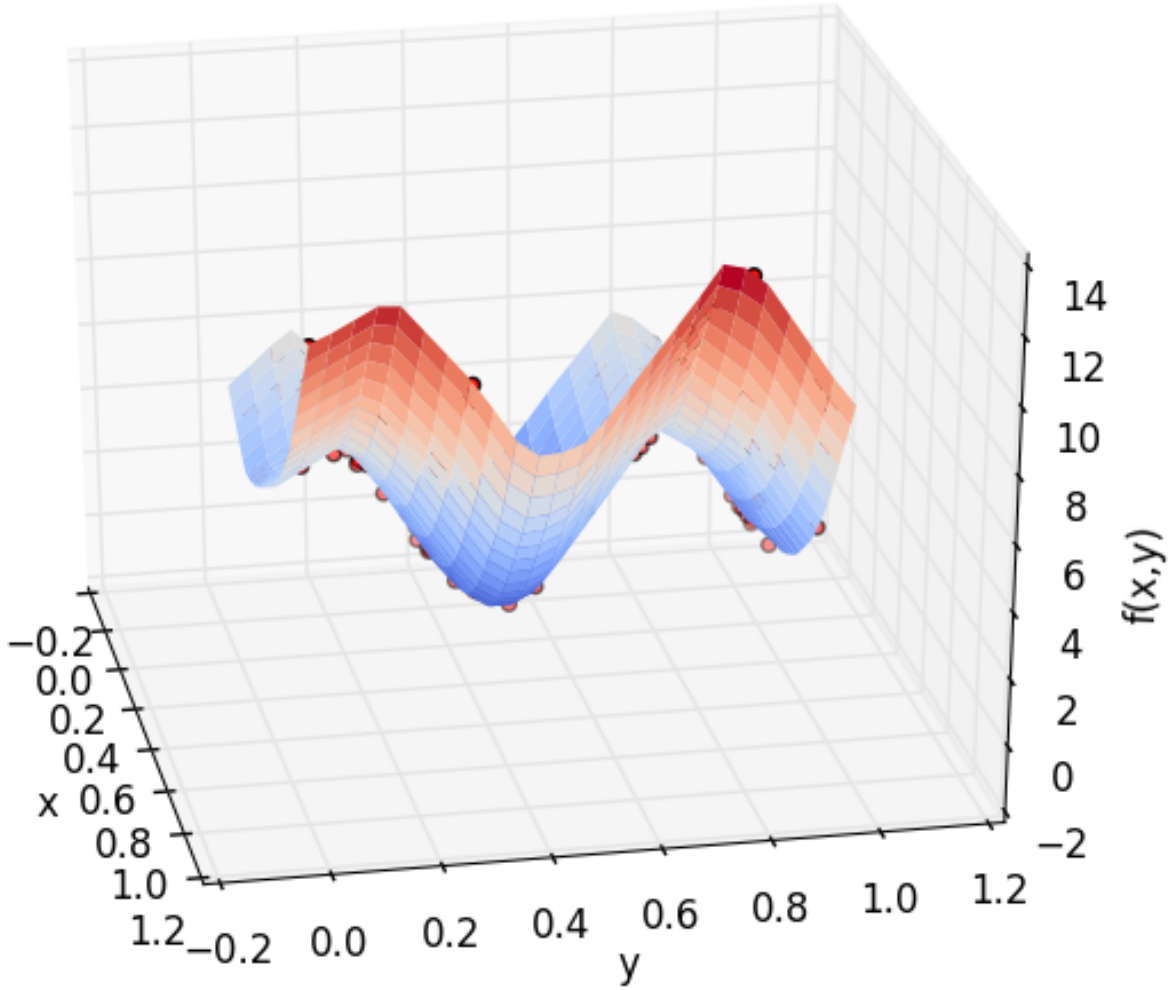
Plotting

The rest of the code plots the data and predicted surface with the Python Matplotlib library (this code is independent of TensorFlow).

```
# 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')
ax.plot_surface(x1,x2,y1,cmap=cm.coolwarm,linewidth=0,rstride=1,
    cstride=1)

ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('f(x,y)')

plt.show()
```

Results

Using the current settings in the code, we get the following fit:



Minimum example for training

```
import tensorflow as tf
import numpy as np

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

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

# 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],
                                optimizer=optimizer)

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

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

DNNRegressor as a custom Estimator

- Custom Estimators can be defined for DNNs with nonstandard structure.
- CNNs are not available as a pre-made estimator.
- We will construct the DNNRegressor as a custom Estimator as an example.

Model function

The model function is the core of a custom Estimator.

It can accept feature and label data, the "mode" (train, predict, or evaluate), and other parameters in a dictionary object `param`.

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

Define the DNN with `tf.layers`

We use "dense" layers with a given size/# of units. 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.

Since this is a regression problem, no activation function is applied to the output layer.

```
# 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(inputs=a3,units=Layers[4],activation=None)
```

`f` is a column vector (a 2D array), so we reshape it to be 1D array.

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

Return predictions

If the function is being used for predictions, we return a predictions dictionary 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})
```

Loss, Evaluation

For regression, we define the loss as a standard mean squared error:

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

If performing evaluation (validation), we return the loss.

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

Training

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)
```

That's it for the `model_fn`!

Instantiating a custom Estimator

The only difference in the main function between the pre-made and the custom DNNRegressor is in declaring an object of the Estimator class.

Pre-made:

```
optimizer=tf.train.AdagradOptimizer(learning_rate=0.1)
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,
                                activation_fn=tf.nn.relu,
                                optimizer=optimizer)
```

Custom:

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

You could add `hidden_units` and `optimizer` to `params` if you wanted for the custom Estimator.

DNNClassifier

- Consider the pre-made DNNClassifier object.
- Maps feature values to a category.
- Use the example of irises
  - Four features: sepal length, sepal width, petal length, petal width
  - Three classes: Iris setosa, Iris virginica, Iris versicolor

Figure: Iris versicolor, by Danielle Langlois (CC BY-SA 3.0), commons.wikimedia.org/w/index.php?curid=248095

Import modules

There are a few changes between the DNNRegressor and this DNNClassifier example.

Import modules (no plotting in this example, but a couple extra modules for downloading data):

```
import tensorflow as tf
import numpy as np
import shutil, os
from six.moves.urllib.request import urlopen

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

Download data

We download the data from tensorflow.org (this code isn't TensorFlow specific):

```
# Read in dat, download first if necessary
if not os.path.exists("iris_training.csv"):
    raw = urlopen(
        "http://download.tensorflow.org/data/iris_training.csv").
     read()
    with open("iris_training.csv", "wb") as f:
        f.write(raw)

if not os.path.exists("iris_test.csv"):
    raw = urlopen(
        "http://download.tensorflow.org/data/iris_test.csv").read()
    with open("iris_test.csv", "wb") as f:
        f.write(raw)
```

Import data

We import the .csv data with a built-in TensorFlow function and store them as NumPy arrays:

```
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)
```

Optimizer

We specify a model directory and optimizer as in the DNNRegressor example:

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

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

DNNClassifier

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)
```

Training

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)
```

Validation

`valid_input_fn` is the same as for DNNRegressor, but we get the accuracy instead of the value of the loss function.

With the current parameters, accuracy should be around 0.96666 (nearly 97%).

```
# Validate
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
```

Prediction

`predict_input_fn` is also the same as before.

For example, create two sets of feature values. The predicted iris types are returned.

```
# Classify two new flower samples.
x_predict = 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": x_predict},
                    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
```

With the current settings, the code will probably predict classes 1 and 2, respectively, for the two new samples.

DNNClassifier as a custom Estimator

For the custom defined DNNClassifier, we again focus on the model function.

```
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)
```

The definition of the hidden layers is the same as for the custom DNNRegressor.

Return predictions

Since this is classification, apply the softmax function:

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

where p_i is probability of category i being true, f_i is i-th component of `logits`.

Highest probability $\Longrightarrow$ predicted category.

```
    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)
```

Loss

The loss function for classification: cross entropy

Data label format: Given as 0, 1, or 2, needs 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:

$$\text{cross\_entropy} = \frac{1}{n_{\text{samples}}} \sum_{j}^{n_{\text{samples}}} \sum_{i}^{n_{\text{classes}}} \hat{p}_i^j \log(p_i^j)$$

where $\hat{p}_i^j$ is the data and p_i^j is the prediction for class i, sample j.

Loss, Evaluation

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)
```

Training

We define the optimizer and training step as in the DNNRegressor example.

```
    # 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)
```

That completes `model_fn` for classification.

Instantiating the custom DNNClassifier

Again, the only difference in the main function between the pre-made and the custom DNNClassifier is in declaring an object of the Estimator class.

Pre-made:

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

Custom:

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

Convolutional Neural Network (CNN)

- Image recognition is often done with CNNs.
- CNNs perform classification by adding new types of layers, primarily "convolutions" and "pooling".
- The "convolution": scanning a filter across the image.
- The "pooling": take the most significant features from a group of pixels.
- Some nice explanations of CNNs by Adam Geitgey and ujjwalkarn.
- Our example will use the MNIST database of handwritten digits.
- No pre-made Estimator available from TensorFlow.

Custom CNNClassifier

The main difference between the custom CNNClassifier and the DNNClassifier is in the construction of the layers in `model_fn`.

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

The MNIST images are 28×28 pixel grayscale images. Each sample can be represented by a $28 \times 28 \times 1$ tensor.

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])
```

Convolutional layer

The first convolutional layer is applied. This involves sweeping a filter across the image (see this example).

Convolution $\Longrightarrow$ translational invariance (it doesn't matter where the object of interest is located).

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

Max pooling involves looking at clusters of the output (in this example, 2×2 clusters), and sets the maximum filter value as the value for the cluster.

I.e. a match anywhere in the cluster $\implies$ a match for the cluster.

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)
```

Pooling reduces the size of the neural net, speeding up computations.

2nd convolution and pooling

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

```
# 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)
```

Fully-connected layer

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)
```

Dropout

Dropout can help prevent overfitting (can be used in other types of neural networks). 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` (softmax, cross entropy, etc.) is the same as the custom classifier example.

Load data

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

Training and evaluation is done as before.

With the setup in this example, you should achieve an accuracy of over 99%:

```
INFO:tensorflow:Saving dict for global step 5000:
accuracy = 0.9917, global_step = 5000, loss = 0.023755284
```

Experiment

Experiment with the following aspects of the different neural networks:

- Number of layers
- Units per layer
- Learning rate
- Batch size
- Number of training steps
- Optimizer type, e.g. GradientDescentOptimizer, AdadeltaOptimizer, AdamOptimizer, AdagradOptimizer
- Activation function, e.g relu, sigmoid, softplus, tanh
- Dropout rate
- Number of filters
- Pooling size and stride

Summary of resources

tensorflow.org
playground.tensorflow.org
github.com/greght/Workshop-TensorFlow-Estimators
www.tensorflow.org/versions/r1.4/get_started/estimator
www.tensorflow.org/versions/r1.4/get_started/mnist/beginners
www.tensorflow.org/versions/r1.4/tutorials/layers
CNN articles: by Adam Geitgey and ujjwalkarn.