TensorFlow `learn`

Just like `sklearn` is a convenient interface to traditional machine learning algorithms, `tf.contrib.learn` (formerly `skflow`) is a simplified interface to building and training deep neural nets. And now it comes free with every installation of TensorFlow!

It's worth looking at `learn` as the high-level API to TensorFlow even if you're not a fan of the syntax. This is because it's currently the only officially supported one. But you should know that there are many alternative high-level APIs that may have more intuitive interfaces. If interested, check out Keras, `tf.slim` (included with TF), or TFLearn.

Setup

To get started with `learn`, you only need to import it. We're also going to import this `estimator` function, which will help us craft general models.

```
# TF made EZ
import tensorflow.contrib.learn as learn
from tensorflow.contrib.learn.python.learn.estimators import estimator
```

We also want to import a few libraries for basic manipulation. Grab NumPy, `math`, and MatplotLib (optional). Of note here is `sklearn`, a general purpose machine learning library that tries to simplify model creation, training, and usage. We'll be mainly using it for convenient metrics, but you'll find it has a similar primary interface as `learn`.

```
# Some basics
import numpy as np
import math
import matplotlib.pyplot as plt
plt.ion()

# Learn more sklearn
# scikit-learn.org
import sklearn
from sklearn import metrics
```

Next we'll read in some data for processing. Since you're familiar with the font classification problem, let's stick with modeling that. For reproducibility, you can seed NumPy with your favorite number.

```
# Seed the data
np.random.seed(42)

# Load data
```

```
data = np.load('data_with_labels.npz')
train = data['arr_0']/255.
labels = data['arr_1']
```

For this exercise, split up your data into a training and validation set. `np.random.permutation` is useful for generating a random order of your input data, so let's use that much like we did in earlier modules.

```
# Split data into training and validation
indices = np.random.permutation(train.shape[0])
valid_cnt = int(train.shape[0] * 0.1)
test_idx, training_idx = indices[:valid_cnt],\
                         indices[valid_cnt:]
test, train = train[test_idx,:],\
              train[training_idx,:]
test_labels, train_labels = labels[test_idx],\
                            labels[training_idx]
```

`tf.contrib.learn` can be fickle about what data types it accepts. To play nicely, we need to recast our data. The image inputs will be `np.float32` instead of the default 64 bits. And our labels will be `np.int32` instead of `np.uint8`, even though this just takes up more memory.

```
train = np.array(train,dtype=np.float32)
test = np.array(test,dtype=np.float32)
train_labels = np.array(train_labels,dtype=np.int32)
test_labels = np.array(test_labels,dtype=np.int32)
```

Logistic Regression

Let's do a simple logistic regression example. This will be very quick and show off how `learn` makes straightforward models incredibly simple. First, we must create a listing of variables that our model expects as input. You might hope that this could be set with a simple argument, but it's actually this unintuitive `learn.infer_real_valued_columns_from_input` function. Basically, if you give your input data to this function, it will infer how many feature columns you have and what shape it should be in. In our linear model, we want to flatten our image to be 1-dimensional, so we reshape it when inferring the features.

```
# Convert features to learn style
feature_columns = learn.infer_real_valued_columns_from_input(train.reshape([-1,36*36]))
```

Now make a new variable called, `classifier`, and assign to it this `estimator.SKCompat` construction. This is a Scikit-Learn compatibility layer, allowing you use some of Scikit-Learn modules with your TensorFlow model.

Anyway, that's just dressing, what really creates the model is `learn.LinearClassifier`. This sets up the model, but does no training. So it only requires a couple arguments. The first is that funky `feature_columns` object, just letting your model know what to expect for input. The second, and last, required argument is its converse, how many output values the model should have. We have 5 fonts, so set `n_classes = 5`. That's the entire model specification!

```
# Logistic Regression
classifier = estimator.SKCompat(learn.LinearClassifier(
            feature_columns = feature_columns,
            n_classes=5))
```

To do the training, it takes just a single line. Call `classifier.fit` with your input data (reshaped, of course), output labels (note that these don't have to be one-hot format), and a few more parameters. `steps` determines how many batches the model will look at. That is, how many steps to take of the optimization algorithm. `batch_size` is as usual, the number of data points to use within an optimization step. So you can compute the number of epochs as the number of steps times the size of batches, divided by the number of data points in your training set. This may seem a little counterintuitive, but at least it's a quick specification, and you could easily write a helper function to convert between steps and epochs.

```
# One line training
# steps is number of total batches
# steps*batch_size/len(train) = num_epochs
classifier.fit(train.reshape([-1,36*36]),
               train_labels,
               steps=1024,
               batch_size=32)
```

To evaluate our model, we'll use `sklearn`'s metrics as usual. But the output of a basic `learn` model prediction is now a dictionary, within which are precomputed class labels as well as the probabilities and logits. To extract the class labels, use the key, `classes`.

```
# sklearn compatible accuracy
test_probs = classifier.predict(test.reshape([-1,36*36]))
sklearn.metrics.accuracy_score(test_labels,
        test_probs['classes'])
```

Dense Neural Network

While there are better ways to implement purely linear models, where TensorFlow and `learn` really shine are simplifying Dense Neural Networks with varying number of layers.

We're going to use the same input features, but now build a deep neural network with 2 hidden layers, first with 10 neurons and then 5. Creating this model will only take one line of Python code; it could not be easier.

The specification is similar to our linear model. We still need `SKCompat`, but now it's `learn.DNNClassifier`. For arguments, there's one additional requirement: the number of neurons on each hidden layer, passed as a list. This one simple argument, which really captures the essence of a DNN model, puts the power of deep learning at your finger tips.

There are some optional arguments to this as well, but we're only going to mention `optimizer`. This allows you to choose between different common optimizer routines such as `SGD` (Stochastic Gradient Descent) or `Adam`. Very convenient.

```
# Dense neural net
classifier = estimator.SKCompat(learn.DNNClassifier(
            feature_columns = feature_columns,
            hidden_units=[10,5],
            n_classes=5,
            optimizer='Adam'))
```

The training and evaluation occur exactly as they do with the linear model. Just for demonstration, we can also look at the confusion matrix created by this model. Note that we haven't trained much, so this model may not compete with our earlier creations using pure TensorFlow.

```
# Same training call
classifier.fit(train.reshape([-1,36*36]),
               train_labels,
               steps=1024,
               batch_size=32)

# simple accuracy
test_probs = classifier.predict(test.reshape([-1,36*36]))
sklearn.metrics.accuracy_score(test_labels,
        test_probs['classes'])

# confusion is easy
train_probs = classifier.predict(train.reshape([-1,36*36]))
conf = metrics.confusion_matrix(train_labels,
        train_probs['classes'])
print(conf)
```

Convolutional Neural Nets in `learn`

Convolutional Neural Nets power some of the most successful machine learning models out there, so we'd hope that `learn` supports them. In fact, the library supports using arbitrary TensorFlow code! You'll find that this is a blessing and a curse. Having arbitrary code available means you can use `learn` to do almost anything you can do with pure TensorFlow, giving maximum flexibility. But, the general interface tends to make the code more difficult to read and write. If you find yourself fighting wtih the interface to make some moderately complex model work in `learn`, it may be time to use pure TensorFlow or switch to another API.

To demonstrate this generality, we're going to build a simple convolutional neural network to attack our font classification problem. It will have one convolutional layer with 4 filters, followed by a flattening to a hidden dense layer with 5 neurons, and finally ending with the densely connected output logistic regression.

To get started, let's do a couple more imports. We want access to both generic TensorFlow, but we also need the `layers` module to call TensorFlow layers in a way that `learn` expects.

```
# Access general TF functions
import tensorflow as tf
import tensorflow.contrib.layers as layers
```

The generic interface forces us to write a function which creates the operations for our model. You may find this tedious, but that's the price of flexibility.

Start a new function called `conv_learn` with 3 arguments. `X` will be the input data, `y` will be the output labels (not yet one-hot encoded), and `mode` determines whether you are training or predicting. Note that you'll never directly interact with this function; you merely pass it to a constructor that expects these arguments. So if you wanted to vary the number or type of layers, you would need to write a new model function (or another function that would generate such a model function).

```
def conv_learn(X, y, mode):
```

As this is a convolutional model, we need to make sure our data is formatted correctly. In particular, this means reshaping the input to have not only the correct 2-dimensional shape (36 by 36), but also 1 "color" channel (the last dimension). This is part of a TensorFlow computation graph, so we use `tf.reshape`, not `np.reshape`. Likewise, because this is a generic graph, we want our outputs to be one-hot encoded. `tf.one_hot` provides that functionality. Note that we have to describe how many classes there are (5), what a "set" value should be (1), and what an "unset" value should be (0).

```
    # Ensure our images are 2d
    X = tf.reshape(X, [-1, 36, 36, 1])
    # We'll need these in one-hot format
    y = tf.one_hot(tf.cast(y, tf.int32), 5, 1, 0)
```

Now the real fun begins. To specify the convolutional layer, let's initialize a new scope, `conv_layer`. This will just make sure we don't clobber any variables. `layers.convolutional` provides the basic machinery. It accepts our input (a TensorFlow tensor), a number of outputs (really the number of kernels or filters), and the size of the kernel, here a 5 by 5 window. For an activation function, let's use Rectified Linear, which we can call from the main TensorFlow module. This gives us our basic convolutional output, `h1`.

Max pooling actually occurs exactly like it does in regular TensorFlow. No easier, no harder. `tf.nn.max_pool` with the usual kernel size and strides works as expected. Save this off into `p1`.

```python
# conv layer will compute 4 kernels for each 5x5 patch
with tf.variable_scope('conv_layer'):
    # 5x5 convolution, pad with zeros on edges
    h1 = layers.convolution2d(X, num_outputs=4,
            kernel_size=[5, 5],
            activation_fn=tf.nn.relu)
    # 2x2 Max pooling, no padding on edges
    p1 = tf.nn.max_pool(h1, ksize=[1, 2, 2, 1],
            strides=[1, 2, 2, 1], padding='VALID')
```

Now, to flatten the tensor at this point, we need to compute the number of elements in our would-be 1-dimensional tensor. One way to do this is multiplying all the dimension values (except the `batch_size`, which occupies the first position) together. This particular operation can occur outside the computation graph, so we use `np.product`. Once supplied with the total size, we can pass it to `tf.reshape` to reslice the intermediate tensor in the graph.

```python
# Need to flatten conv output for use in dense layer
p1_size = np.product(
          [s.value for s in p1.get_shape()[1:]])
p1f = tf.reshape(p1, [-1, p1_size ])
```

It's time for the densely connected layer. `layers` makes an appearance again, this time with the `fully_connectted` function (another name for a dense layer). This takes the previous layer, the number of neurons, and the activation function, again supplied by general TensorFlow.

For demonstration purposes, let's add a dropout here as well. `layers.dropout` provides the interface. As expected, it needs the previous layer as well as a probability of keeping a given node output. But it also needs this 'mode' argument that we passed into the original `conv_learn` function. All all this complex interface is saying is to only drop nodes during training. If you can handle that, we're almost through the model!

`python     # densely connected layer with 32 neurons and dropout h_fc1 = layers.fully_connected(p1f,             5,             activation_fn=tf.nn.relu) drop = layers.dropout(h_fc1, keep_prob=0.5,     is_training=mode == tf.contrib.learn.ModeKeys.TRAIN)`python

Now for some bad news. We need to write out the final linear model, loss function, and optimization parameters manually. This is something that can change from version to version, as it used to be easier on the user for some circumstances, but more difficult to maintain the backend. But let us persevere; it's really not too arduous.

Another `layers.fully_connected` layer creates the final logistic regression. Note that our activation here should be `None`, as it is purely linear. What handles the "logistic" side of the equation is the loss function. Thankfully, TensorFlow supplies a `softmax_cross_entropy` function, so we don't need to write this out manually. Given inputs, outputs, and a loss function, we can apply an optimization routine. Again, `layers.optimize_loss` minimizes the pain as well as the function in question. Pass it your loss node, optimizer (as a string), and a learning rate. Further, give it this `get_global_step()` parameter to ensure the optimizer handles decay properly.

Finally, our function needs to return a few things. One, it should report the predicted classes. Next, it must supply the loss node output itself. And finally, the training node must be available to external routines to actually execute everything.

```
    logits = layers.fully_connected(drop, 5, activation_fn=None)
    loss = tf.losses.softmax_cross_entropy(y, logits)
    # Setup the training function manually
    train_op = layers.optimize_loss(
        loss,
        tf.contrib.framework.get_global_step(),
        optimizer='Adam',
        learning_rate=0.01)
    return tf.argmax(logits, 1), loss, train_op
```

While specifying the model may be cumbersome, using it is just as easy as before. Now, use `learn.Estimator`, the most generic routine, and pass in your model function for `model_fn`. And don't forget the SKCompat!

Training works exactly as before, just note that we don't need to reshape the inputs here, since that's handled inside the function.

To predict with the model, you can simply call `classifier.predict`, but note that you get as output your first argument returned by the function. We opted to return the class, but it would also be reasonable to return the probabilities from the softmax function as well. That's all the basics of `tf.contrib.learn` models!

```
# Use generic estimator with our function
classifier = estimator.SKCompat(
         learn.Estimator(
         model_fn=conv_learn))

classifier.fit(train,train_labels,
                steps=1024,
                batch_size=32)
```

```python
# simple accuracy
metrics.accuracy_score(test_labels,classifier.predict(test))
```

Extracting Weights

While training and prediction are the core uses of models, it's important to be able to study the inside of models as well. Unfortunately, this API makes it difficult to extract parameter weights. Thankfully, this section provides some simple examples of a weakly documented feature to get the weights out of `tf.contrib.learn` models.

To pull out the weights of a model, we really need to get the value from certain points in the underlying TensorFlow computation graph. TensorFlow provides various ways to do this, but the first problem is just figuring out what your variable of interest is called.

A list of variable names in your `learn` graph is available, but it's buried under the hidden attribute, `_estimator`. Calling `classifier._estimator.get_variable_names()` returns a list of strings of your various names. Many of these will be uninteresting, like the `OptimizeLoss` entries. In our case, we're looking for `conv_layer` and `fully_connected` elements.

```python
# See layer names
print(classifier._estimator.get_variable_names())
```

```
['OptimizeLoss/beta1_power',
 'OptimizeLoss/beta2_power',
 'OptimizeLoss/conv_layer/Conv/biases/Adam',
 'OptimizeLoss/conv_layer/Conv/biases/Adam_1',
 'OptimizeLoss/conv_layer/Conv/weights/Adam',
 'OptimizeLoss/conv_layer/Conv/weights/Adam_1',
 'OptimizeLoss/fully_connected/biases/Adam',
 'OptimizeLoss/fully_connected/biases/Adam_1',
 'OptimizeLoss/fully_connected/weights/Adam',
 'OptimizeLoss/fully_connected/weights/Adam_1',
 'OptimizeLoss/fully_connected_1/biases/Adam',
 'OptimizeLoss/fully_connected_1/biases/Adam_1',
 'OptimizeLoss/fully_connected_1/weights/Adam',
 'OptimizeLoss/fully_connected_1/weights/Adam_1',
 'OptimizeLoss/learning_rate',
 'conv_layer/Conv/biases',
 'conv_layer/Conv/weights',
 'fully_connected/biases',
 'fully_connected/weights',
 'fully_connected_1/biases',
 'fully_connected_1/weights',
```

```
'global_step']
```

Figuring out which entry is the layer you're looking for can be a challenge. Here, `conv_layer` is obviously from our convolutional layer. But you see 2 `fully_connected` elements, one is our dense layer at flattening, and one is the output weights. It turns out they're named in the order specified. We created the dense hidden layer first, so it gets the basic `fully_connected` name, while the output layer came last, so it has a `_1` tacked onto it. If you're unsure, you can always look at the shapes of the weight arrays, depending on the shape of your model.

To actually get at the weights, it's another arcane call. This time, `classifier._estimator.get_variable_value`, supplied with the variable name string, produces a NumPy array with the relevant weights. Try it out for the convolutional weights and biases as well as the dense layers.

```python
# Convolutional Layer Weights
print(classifier._estimator.get_variable_value(
        'conv_layer/Conv/weights'))
print(classifier._estimator.get_variable_value(
        'conv_layer/Conv/biases'))

# Dense Layer
print(classifier._estimator.get_variable_value(
        'fully_connected/weights'))

# Logistic weights
print(classifier._estimator.get_variable_value(
        'fully_connected_1/weights'))
```

Now armed with the esoteric knowledge of how to peer inside `tf.contrib.learn` neural nets, you're more than capable with this high-level API. While it is convenient in many situations, it can be cumbersome in others. Never be afraid to pause and consider switching to another library; use the right machine learning tool for the right machine learning job.