Training neural networks

Glenn Bruns
CSUMB

Much material in this deck from Géron, Hands-on Machine Learning with Scikit-Learn and TensorFlow

Learning outcomes

After this lecture you should be able to:

- list some of the APIs built on top of TensorFlow
- build and train a DNN using TF
- tune the hyperparameters of your neural net

High-level APIs built over TF

- Keras (multiple backends: TF, Theaon, CNTK)
- TFLearn
- Pretty Tensor
- "do not confuse the high-level API TFLearn (no space) with the simplified interface TF Learn"
  - I'm confused
  - TFLearn, tflearn.org, developed by Aymeric Damien
  - TF Learn, tf.contrib.learn, part of the TF distribution

Building a neural net with pure TF

Step 1. Specify # of inputs, outputs, hidden layers

```
import tensorflow as tf

n_inputs = 28*28  # MNIST
n_hidden1 = 300
n_hidden2 = 100
n_outputs = 10
```

Notes:

- We're predicting the number from a handwritten digit
- There's one output for each possible number, 0 to 9

2. Specify placeholders for features and target

```
X = tf.placeholder(tf.float32, shape=(None, n_inputs), name="X")
y = tf.placeholder(tf.int64, shape=(None), name="y")
```

Notes:

- In the first line, None is used because the number of instances in a mini-batch is not yet known.
- Similarly the length of y will depend on the num. of instances.

3. Create the network

```
with tf.name_scope("dnn"):
    hidden1 = tf.layers.dense(X, n_hidden1, name="hidden1",
                              activation=tf.nn.relu)
    hidden2 = tf.layers.dense(hidden1, n_hidden2, name="hidden2",
                              activation=tf.nn.relu)
    logits = tf.layers.dense(hidden2, n_outputs, name="outputs")
```

Notes:

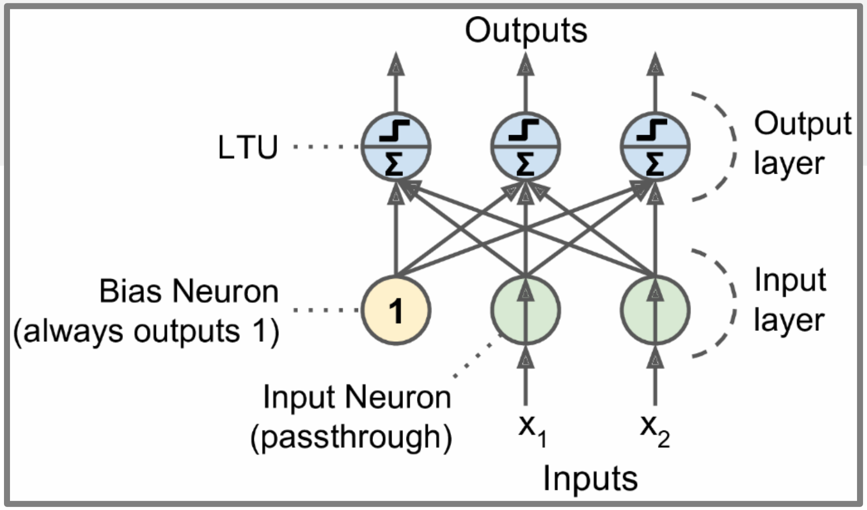
- first layer takes X as input, has n_hidden1 neurons
- activation function not specified for the last layer
- tf.layers.dense is preferred to contrib.layers.fully_connected, as used in text
- The tensorflow.contrib package is for experimental code that isn't yet part of the main TF API

Create a neuron layer "manually"

```
def neuron_layer(X, n_neurons, name, activation=None):
    with tf.name_scope(name):
        n_inputs = int(X.get_shape()[1])
        stddev = 2 / np.sqrt(n_inputs)
        init = tf.truncated_normal((n_inputs, n_neurons), stddev=stddev)
        W = tf.Variable(init, name="kernel")
        b = tf.Variable(tf.zeros([n_neurons]), name="bias")
        Z = tf.matmul(X, W) + b
        if activation is not None:
            return activation(Z)
        else:
            return Z
```

Notes:

- X is the input layer
- n_neurons is the number of neurons in the layer



4. Define loss function

```
with tf.name_scope("loss"):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y,
                                                              logits=logits)
    loss = tf.reduce_mean(xentropy, name="loss")
```

Notes:

- The first function applies softmax activation function then computes the cross entropy
  - software computes probabilities for the 10 outputs
  - cross entropy compares the probabilities with the target values (each target is a digit from 0-9) interpreted as one-hot vectors
- The second function take the average

5. Specify an optimizer

```
learning_rate = 0.01

with tf.name_scope("train"):
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    training_op = optimizer.minimize(loss)
```

Notes:

- notice how name scopes are being used for all the different pieces
- it's easy to plug in an alternative optimizer

6. Specify a performance measure

```
with tf.name_scope("eval"):
    correct = tf.nn.in_top_k(logits, y, 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
```

Notes:

- the performance measure and the loss function are separate things!
- We're using accuracy as the performance measure
  - i.e. on average, how often is prediction correct?
- in_top_k is checking whether the single highest logit is equal to the prediction y

7. Make initializer and saver

```
init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

Notes:

- remember that calling global_variables_initializer() doesn't initialize anything, it's creating a node to do that

Execution phase

```
n_epochs = 20
n_batches = 50
with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
        acc_test = accuracy.eval(feed_dict={X: mnist.test.images,
                                            y: mnist.test.labels})
        print(epoch, "Train accuracy:", acc_train,
                     "Test accuracy:",  acc_test)
    save_path = saver.save(sess, "./my_model_final.ckpt")
```

Notes:

- accuracy is evaluated at end of each epoch
- training accuracy: use batch, test accuracy: use test set
- with every eval() we need a feed_dict

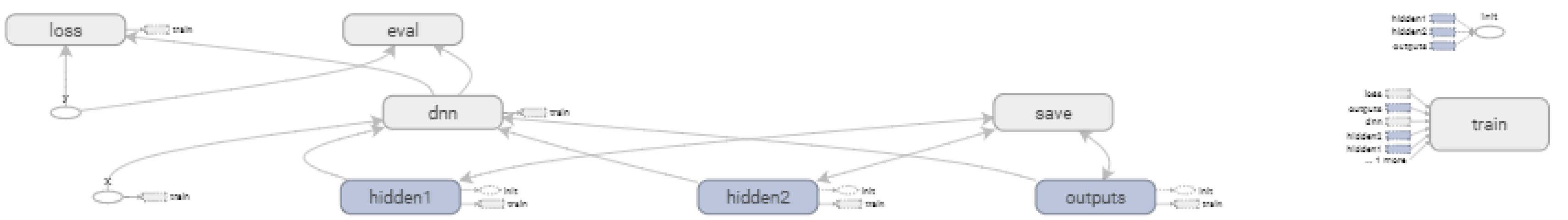
Summary of building the net

1. define inputs
2. specify placeholders for training, target data
3. create the network
4. define a loss function
5. specify an optimizer
6. specify a performance measure
7. make initializer and saver

Questions:

- does the optimizer depend on the loss function?
- does the performance measure depend on the optimizer?

Viewing the neural network



Using the classifier

```
with tf.Session() as sess:
    saver.restore(sess, "./my_model_final.ckpt")
    X_new_scaled = […]    # new images
    Z = logits.eval(feed_dict={X: X_new_scaled})
    y_pred = np.argmax(Z, axis=1)
```

Notes:

- restore() loads the trained model parameters from disk
- the parts of the net related to training ("loss", "eval") are no longer needed
- eval() is used on logits, so no y values are needed in feed_dict
- no softmax is involved here! We're just getting the output with the highest value

Tuning the network parameters

- number of hidden layers
  - with more layers, exponentially fewer total neurons can be used to model complex functions
  - hierarchical concept
  - start with a couple, ramp up
- number of neurons per hidden layers
  - fewer and fewer neurons per layer, as you move to output layer
- activation functions
  - often, ReLU for hidden layers (good performance)
  - output layer: often softmax for classification, nothing for regression

please read this section of our text carefully for details

Summary

- building and training a network with TF.Learn
- building, training, and using a network with pure TF
- tips on tuning a network

Training an MLP with TF.Learn

MINST data set: 70,000 images of handwritten digits (28 x 28 pixels each)

Each image is labeled with the correct number

A standard ML data set (see Geron chapter 3)

```
from tensorflow.examples.tutorials.mnist import input_data

# get mnist data
mnist = input_data.read_data_sets("/tmp/data/")

# build training, test sets
# I don't see that the data is scaled, as the book says it should be
X_train = mnist.train.images
X_test = mnist.test.images
y_train = mnist.train.labels.astype("int")
y_test = mnist.test.labels.astype("int")
```

Build a classifier for MNIST

```
# "specifies the configurations for an Estimator run"
config = tf.contrib.learn.RunConfig(tf_random_seed=42)

# "creates FeatureColumn obects for inputs defined by input x"
feature_cols = tf.contrib.learn.infer_real_valued_columns_from_input(X_train)

# two hidden layers, one with 300 neurons, the other with 100
# softmax output layer with 10 neurons
dnn_clf = tf.contrib.learn.DNNClassifier(hidden_units=[300,100], n_classes=10,
                                         feature_columns=feature_cols, config=config)
dnn_clf = tf.contrib.learn.SKCompat(dnn_clf) # if TensorFlow >= 1.1
dnn_clf.fit(X_train, y_train, batch_size=50, steps=40000)
```

Our text explains little about this, and the tf.contrib.learn documentation is very minimal

Classification results

```
from sklearn.metrics import accuracy_score, log_loss

# check the accuracy
y_pred = dnn_clf.predict(X_test)
accuracy_score(y_test, y_pred['classes'])

# further evaluation of results
y_pred_proba = y_pred['probabilities']
log_loss(y_test, y_pred_proba)
```

output:

```
y_pred = dnn_clf.predict(X_test)
INFO:tensorflow:Restoring parameters from
C:\Users\Glenn\AppData\Local\Temp\tmpynv7n9ys\model.ckpt-40000

accuracy_score(y_test, y_pred['classes'])
Out[663]: 0.9824000000000005

y_pred_proba = y_pred['probabilities']
log_loss(y_test, y_pred_proba)
Out[666]: 0.069003421592055081
```

This is very good given so little work to build the model