Tutorial II: Optimization in TensorFlow & NN introduction

Bern Winter School on Machine Learning, 28.01-01.02 2019
Mykhailo Vladymyrov

unpack libraries

if using colab, upload the `material.tgz` and run the next cell

```
!tar -xvzf material.tgz
```

1. Load necessary libraries

```
import sys

import numpy as np
import matplotlib.pyplot as plt
import IPython.display as ipyd
import tensorflow as tf

# We'll tell matplotlib to inline any drawn figures like so:
%matplotlib inline
plt.style.use('ggplot')
from utils import gr_disp

from IPython.core.display import HTML
HTML("""<style> .rendered_html code {
    padding: 2px 5px;
    color: #0000aa;
    background-color: #cccccc;
} </style>""")
```

1. Linear fit

Here we will solve optimization problem to perform linear regression. First we will generate training set of 160 data points and test set of 40, laying on a line with a random offset

$$y = a_0 x + b_0 + ofs$$

, where ofs is a random variable sampled from a normal distribution with standard deviation equal to s_0

```
a0 = 3#3
b0 = 5#5
s0 = 1#1

#all samples
x_all = np.arange(0, 10, .05)
n_all = x_all.shape[0]
ofs_all = s0*np.random.randn(n_all)
```

```
y_all = np.asarray([a0*x+b0+o for x, o in zip(x_all, ofs_all)])

#randomize order and get 80% for training
idx = np.random.permutation(n_all)
n_train = n_all*80//100

idx_train = idx[0:n_train]
idx_test = idx[n_train:]

x_train = x_all[idx_train]
y_train = y_all[idx_train]

x_test = x_all[idx_test]
y_test = y_all[idx_test]

plt.plot(x_train, y_train, "o", x_test, y_test, "b^")
plt.legend(('training points', 'test points'),  loc='upper left')
```

We will then define loss function as the mean of squared residuals (distance from line along y) for the points.

We will use [stochactic gradient descent]: on each iteration use only a fraction (`mini_batch_size`) of all training set. In many cases training set is huge and cannot be fed on each iteration in principle. Also it can sometimes help the optimizer to properly explore the manifold.

```
tf.reset_default_graph()

#here we have 2 trainable parameters, a and b
a = tf.get_variable(name='a', dtype=tf.float32, initializer=tf.random_normal(()))
b = tf.get_variable(name='b', dtype=tf.float32, initializer=tf.random_normal(()))
x = tf.placeholder(name='x', dtype=tf.float32, shape=(None))
y = tf.placeholder(name='y', dtype=tf.float32, shape=(None))

residual = y - tf.multiply(x,a) - b
residual2 = tf.square(residual)
loss = tf.reduce_mean(residual2)

optimizer = tf.train.GradientDescentOptimizer(0.005).minimize(loss) #0.005
mini_batch_size = 10 # 10

l_sv_train = []
l_sv_test = []
with tf.Session() as sess:
    #initialize all the variables
    sess.run(tf.global_variables_initializer())

    #iterate training for 201 epoch
    for epoch in range (201):
        #shuffle the data and perform stochastic gradient descent by runing over al
        idx = np.random.permutation(n_train)
        for mb in range(n_train//mini_batch_size):
            sub_idx = idx[mini_batch_size*mb:mini_batch_size*(mb+1)]
            sess.run(optimizer, feed_dict={x:x_train[sub_idx], y:y_train[sub_idx]})

        #evalueate and save test and training loss, to be plotted in the end
        l_val_test = sess.run(loss, feed_dict={x:x_test, y:y_test})
        l_val_train = sess.run(loss, feed_dict={x:x_train, y:y_train})
        if epoch%40==0:
            l_val, a_val, b_val = sess.run([loss, a, b], feed_dict={x:x_train, y:y_
            print(epoch, l_val, a_val, b_val)

        l_sv_train.append(l_val_train)
        l_sv_test.append(l_val_test)

end_fit_x = [x_all[0], x_all[-1]]
end_fit_y = [a_val*x+b_val for x in end_fit_x]
true_fn_y = [a0*x+b0 for x in end_fit_x]
fig, axs = plt.subplots(2, 1, figsize=(10,8))
axs[0].plot(x_train, y_train, 'ro', x_test, y_test, 'b^', end_fit_x, end_fit_y, 'g'
axs[0].legend(('training points', 'test points', 'final fit'),  loc='upper left')
```

```
ep_arr = np.arange(len(l_sv_train))
axs[1].plot(ep_arr, l_sv_train, 'r', ep_arr, l_sv_test, 'b')
axs[1].legend(('training loss', 'test loss'),  loc='upper right')
```

2. Excercise 1

Play with the true function parameters `a0, b0, s0` and the `mini_batch_size` value, check how it affects the convergence.

1. How change of `s0` affects convergance?
2. When one should stop training to prevent overfitting?

3. A bit of things

The training as we just saw is done iteratively, by adjusting the model parameters.

We perform optimization several times for all traininng dataset. Going through all this dataset is refered to as 'epoch'.

When we do training its usually done in two loops. In outer loop we iterate over all epochs. For each epoch we usually split the dataset into small chuncks, 'mini-batches', and optimization it performed for all of those.

It is important that data doesn't go to the training pipeline in same order. So the overall scheme looks like this (pseudocode):

```
x,y = get_training_data()
for epoch in range(number_epochs):
   x_shfl,y_shfl = shuffle(x,y)

   for mb_idx in range(number_minibatches_in_batch):
       x_mb,y_mb = get_minibatch(x_shfl,y_shfl, mb_idx)

       optimize_on(data=x_mb, labels=y_mb)
```

Shuffling can be easily done using permuted indexes.

```
#some array
arr = np.array([110,111,112,113,114,115,116])

#we can get sub-array for a set of indexes, eg:
idx_1_3 = [1,3]
sub_arr_1_3 = arr[idx_1_3]
print (arr,'[',idx_1_3,']','->', sub_arr_1_3)


ordered_idx = np.arange(7)
permuteded_idx = np.random.permutation(7)
print(ordered_idx)
print(permuteded_idx)

permuted_arr = arr[permuteded_idx]
print (arr,'[',permuteded_idx,']','->', permuted_arr)
```

Some additional np things in this tutorial:

```
#index of element with highest value
np.argmax(permuted_arr)
```

```
arr2d = np.array([[0,1],[2,3]])
print(arr2d)
```

```
#flatten
arr_flat = arr2d.flatten()
#reshape
arr_4 = arr2d.reshape((4))
arr_4_1 = arr2d.reshape((4,1))

print (arr_flat)
print (arr_4)
print (arr_4_1)
```

4. Bulding blocks of a neural network

Neural network consists of layers of neurons. Each neuron perfroms 2 operations.

1. Calculate the linear transformation of the input vector $\bar{x}$:
$$z = \bar{W} \cdot \bar{x} + b = \sum W_i x_i + b$$
where $\bar{W}$ is vector of weights and b - bias.
2. Perform the nonlinear transformation of the result using activation function f
$$y = f(z)$$
Here we will use rectified linear unit activation.

In a fully connected neural network each layer is a set of N neurons, performing different transformations of all the same layer's inputs $\bar{x} = [x_i]$ producing output vector $\bar{y} = [y_j]_{i=1..N}$:
$$y_j = f(\bar{W}_j \cdot \bar{x} + b_j)$$
Since output of each layer forms input of next layer, one can write for layer l:
$$x_j^l = f(\bar{W}_j^l \cdot \bar{x}^{l-1} + b_j^l)$$
where $\bar{x}^0$ is network's input vactor.

To simplify building the network, we'll define a helper function, creating neuron layer with given number of outputs:

```
def fully_connected_layer(x, n_output, name=None, activation=None):
    """Fully connected layer.

    Parameters
    ----------
    x : tf.Tensor
        Input tensor to connect
    n_output : int
        Number of output neurons
    name : None, optional
        TF Scope to apply
    activation : None, optional
        Non-linear activation function

    Returns
    -------
```

```
    h, W : tf.Tensor, tf.Tensor
        Output of the fully connected layer and the weight matrix
    """
    if len(x.get_shape()) != 2:
        x = flatten(x, reuse=None)

    n_input = x.get_shape().as_list()[1]

    with tf.variable_scope(name or "fc", reuse=None):
        W = tf.get_variable(
            name='W',
            shape=[n_input, n_output],
            dtype=tf.float32,
            initializer=tf.contrib.layers.xavier_initializer())

        b = tf.get_variable(
            name='b',
            shape=[n_output],
            dtype=tf.float32,
            initializer=tf.constant_initializer(0.0))

        h = tf.nn.bias_add(
            name='h',
            value=tf.matmul(x, W),
            bias=b)

        if activation:
            h = activation(h)

        return h, W
```

In the case of classification, in the the last layer we use *softmax* transformation as non-linear transformation:

$$y_i = \sigma(\bar{z})_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

This will correspond to the one-hot labels that we use. Finally we will use the cross entropy as the loss function:

$$H(Y_{pred}, Y_{true}) = -\sum_i Y_{true,i} \log(Y_{pred,i})$$

5. Bulding a neural network

```
n_input = 10
n_output = 2
```

```
g = tf.Graph()
with g.as_default():
    X = tf.placeholder(name='X', dtype=tf.float32, shape=[None, n_input])
    Y = tf.placeholder(name='Y', dtype=tf.float32, shape=[None, n_output])

    #layer 1: 10 inputs -> 4, softmax activation
    L1, W1 = fully_connected_layer(X , 4, 'L1', activation=tf.sigmoid)
    L2, W2 = fully_connected_layer(L1 , 2, 'L2', activation=None)
    Y_onehot = tf.nn.softmax(L2, name='Logits')

    #prediction: onehot->integer
    Y_pred = tf.argmax(Y_onehot, axis=1, name='YPred')
```

```
print(X)
print(Y)
print(Y_pred)
```

```
gr_disp.show(g.as_graph_def())
```