Tutorial I: Introduction to TensorFlow

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

The main feature of TF is the way we define operations. In regular programming we define a set of functions or methods on the objects. In TF we define a computational graph. Computational graph is a directed graph in which every node corresponds to an operation or variable. Variables can feed their value into operations, and operations can feed their output into other operations. Then, during execution we feed some data and/or parameters as input of the graph, and the graph produces the output.

00. Requirements

To run this notebooks you need Tensorflow and numpy installed.

Basic knowledge of Python can be acquired here and of Numpy here

Full documentation on Tensorflow functions is available in the reference. Sometimes functions' implementation might help to understand what is happening under the hood.

0. Cell execution

Press `Ctrl+Enter` or `Shift+Enter` on the next cell to execute the content

```
print('It works')
```

Navigate between cells with arrows. Press `Enter` to edit cell, `Esc` to exit. Press `a` or `b` too create a new cell above or below.

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>""")
```

2. Create our first graph

First we need to define the input for the graph. The easiest way is to define so called `placeholder`, where during the excecution we will feed in the input values.

```
input = tf.placeholder(name = 'input', shape=(), dtype=tf.float32)
```

Then we will define two simple operations. In most cases simple Python notation gives the desired result.

```
out1 = input + 5
```

```
out2 = input * out1
```

```
gr_disp.show(tf.get_default_graph().as_graph_def())
```

3. Run the graph

```
sess = tf.Session()
```

Session is used to compute the desired outputs, for example our defined `out1`.

```
#res1 = sess.run(out1)
```

If you will uncomment and run the above cell, you will get an error, indicating that the value for the `input` should be given. Here we will use feed dictionary, where we specify input as

```
res1 = sess.run(out1, feed_dict={input: 1})
print(res1)
#out1 = input+5 = 1+5 = 6
```

several values can be computed at the same time:

```
res1, res2 = sess.run((out1, out2), feed_dict={input: 3})
print(res1, res2)
#out1 = input+5 = 3+5 = 8
```

```
#out1 = (input+5)*input = (3+5)*3 = 8*3=24
```

It is important to remember that in principle its better NOT TO USE `feed_dict`: it is rather slow. There are several proper built-in mechanisms, that allow smooth data reading, in particular from disc (which is generally super slow!).

While in this course we will keep using `feed_dict`, since it's more visual, and helps to better understand what is going on, you are highly encouradged to read and follow the official guidelines related to the data streaming and handling.

4. Tensor operations

For ML tasks we often need to perform operations on high-dimensional data. Theese are represented as tensors in TF. For example we can calculate sum of squared values in an 1D array with 5 elements:

```
tf.reset_default_graph()
input_arr = tf.placeholder(name='input_arr', dtype=tf.float32, shape=(5,))
squared = tf.multiply(input_arr, input_arr)
out_sum = tf.reduce_sum(squared)
```

```
np_arr = np.asarray((1,2,3,4,5), dtype=np.float32)
with tf.Session() as sess:
    print(sess.run(out_sum, feed_dict={input_arr: np_arr}))
# squared = (1,4,9,16,25)
# out_sum = 55
```

Or we can do the same for several 1D arrays at once:

```
tf.reset_default_graph()
input_arr = tf.placeholder(name='input_arr', dtype=tf.float32, shape=(None, 5)) #No
squared = tf.multiply(input_arr, input_arr)
out_sum = tf.reduce_sum(squared, axis=1) # sum only along 1st axis

#Sample arrays of different size along first axis.
#They all can be fed into the input_arr placeholder since along first axis size is
np_arr1 = np.asarray([[1,2,3,4,5]], dtype=np.float32)
np_arr2 = np.asarray([[1,2,3,4,5], [2,3,4,5,6]], dtype=np.float32)
np_arr3 = np.asarray([[1,2,3,4,5], [2,3,4,5,6], [25,65,12,12,11], [1,2,3,4,5], [2,3
with tf.Session() as sess:
    print(sess.run(out_sum, feed_dict={input_arr: np_arr1}))
    print(sess.run(out_sum, feed_dict={input_arr: np_arr2}))
    print(sess.run(out_sum, feed_dict={input_arr: np_arr3}))
```

5. Excercise 1

Hint: You can use question mark to get description of function right from Jupyter notebook:

```
tf.multiply?
```

or `Shift+Tab` within the brackets to see function parameters (just `Tab` for google colab):

```
tf.reduce_sum(    )
```

Modify the code bellow to calculate mean of array's elements.

```
... #1. reset the graph
input_arr = tf.placeholder(name='input_arr', shape=(None, None), dtype=tf.float32)
out_mean = ... # 2.use reduce_mean to claculate mean along specified axes

np_arr = np.asarray([[1,2,3,4,5], [2,3,4,5,6], [25,65,12,12,11]], dtype=np.float32)
with tf.Session() as sess:
    res = ... # 3. calculate the out_mean
    print(res)

... #4. display the graph
```

6. Optimization problem

In ML we always try to optimize model parameters to minimize a loss function. TF allows for easy optimization problem solving. Let's see how this works. We will use a function f, parabolic with respect to the model parameter t: $f(x_1, x_2|t) = (x_1 * t - x_2)^2$. Here x_1 and x_2 are given values for which we will try to minimize value of function f.

We define `t` as a variable using `get_variable` and initialize it with a constant 0. Variables are by default trainable, *i.e.* their value will be changed during training.

```
tf.reset_default_graph()

t = tf.get_variable(name='t', dtype=tf.float32, shape=(), initializer=tf.constant_i
x1 = tf.placeholder(name='x1', dtype=tf.float32, shape=())
x2 = tf.placeholder(name='x2', dtype=tf.float32, shape=())
```

```
#function to be minimized
f = tf.square(t*x1-x2)
```

Next we create and optimizer: object that on each iteration adjusts values of all trainable parameters (in our case just `t` to minimize the value of `f`. As the name sugests it uses steepest gradient descent.

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.01).minimize(f)
```

```
#values of x1, x2 for which we will minimize f
x1_val = 3.
x2_val = 9.

#buffers to store intermidiate values of t and f to plot them later.
t_sv = []
f_sv = []

with tf.Session() as sess:
    #don't forget to initialize all variables!
    sess.run(tf.global_variables_initializer())

    #optimization works iteratively, adjusting the value of t on each step
    for itr in range (30):
        _ = sess.run(optimizer, feed_dict={x1:x1_val, x2:x2_val})
        f_val, t_val = sess.run([f, t], feed_dict={x1:x1_val, x2:x2_val})
        #save the current values of t and the function f
        t_sv.append(t_val)
        f_sv.append(f_val)
```

```
#just find the nice range for plotting
x0 = x2_val/x1_val
xhalf = max(abs(t_sv[0]-x0), 5.)
#fill array for parabola
t_all = np.arange(x0-xhalf, x0+xhalf, xhalf/50.)
f_all = np.asarray([(ti*x1_val-x2_val)*(ti*x1_val-x2_val) for ti in t_all])

#draw all
_, axs = plt.subplots(1, 2, figsize=(10,10))
axs[0].plot(t_all, f_all, 'b', t_sv, f_sv, 'g^')
axs[0].set_title('f(t | x1,x2)')
axs[0].legend(('f(t)', 'training iterations'),  loc='upper center')
axs[1].plot(f_sv)
axs[1].set_title('f(itr)')
```

7. Excercise 2

Try to modify `x1_val` and `x2_val` in the above code, as well as the `learning_rate` and `t` initialization value, and see how it affects convergence. Get an intuition on simple example, it is very useful!

Try to see when

1. convergence is too slow
2. oscillation near minimum occurs
3. divergence