TP01-IntroductionToTensorflow

September 18, 2019

1 Introduction to Tensorflow

Tensorflow is one of the main libraries used for Deep Learning. Initialy, it was made for Tensor (nd-array) distributed computation and was extended with facilities for Machine Learning and Deep Learning.

For this tutorial we will use Tensorflow in interactive mode, which is not the optimal mode in term of computation speed but will provide us an easy way to get in touch with Tensorflow fundamental concepts.

[1]:
```
import numpy as np
import tensorflow as tf
session = tf.InteractiveSession()
```

1.1 tf.Tensor

`Tensor` is the fundamental class of Tensorflow. A huge difference with NumPy is that `Tensor` are immutable objects, that is they can only be assigned once. They can represent operation tree on arrays or arrays themselves (actually an operation tree with only one node).

[2]:
```
tf.zeros((2, 2))
```

[2]:
```
<tf.Tensor 'zeros:0' shape=(2, 2) dtype=float32>
```

To actually see the values of a `Tensor` you should `run` the tensor in the session or `eval` it.

[3]:
```
a = tf.zeros((2, 2))
session.run(a)
# the result of an evaluation is a ndarray !
```

[3]:
```
array([[0., 0.],
       [0., 0.]], dtype=float32)
```

[4]:
```
a.eval()  # equivalent of session.run(a)
```

[4]:
```
array([[0., 0.],
       [0., 0.]], dtype=float32)
```

Some ways to create Tensors:

[5]:
```
b = tf.ones((3, 3))
b.eval()
```

[5]:
```
array([[1., 1., 1.],
       [1., 1., 1.],
       [1., 1., 1.]], dtype=float32)
```

[6]:
```
c = tf.fill((2, 2), 7.)
c.eval()
```

[6]:
```
array([[7., 7.],
       [7., 7.]], dtype=float32)
```

[7]:
```
d = tf.constant([[1, 2, 3], [4, 5, 6]])
d.eval()
```

[7]:
```
array([[1, 2, 3],
       [4, 5, 6]], dtype=int32)
```

[8]:
```
e = tf.eye(4)
e.eval()
```

[8]:
```
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]], dtype=float32)
```

Most simple operations that can be done on `ndarray` can be done on `Tensors`. Results of operations are `Tensor` themselves.

[9]:
```
a = tf.constant([[1, 2, 3], [4, 5, 6]])
b = tf.constant([[7, 8, 9], [10, 11, 12]])
c = a + b  # symbolic definition of c
d = 2 * c  # symbolic definition of d
d.eval()  # c and d are only evaluated here
```

[9]:
```
array([[16, 20, 24],
       [28, 32, 36]], dtype=int32)
```

`d` contains the tree of the operations and no actual computation is made until the `eval` method is called.

[10]:
```
# get the operation of d
d.op
```

[10]:
```
<tf.Operation 'mul' type=Mul>
```

[11]:
```
# get the first input of the multiplication
d.op.inputs[0]
```

[11]:
```
<tf.Tensor 'mul/x:0' shape=() dtype=int32>
```

[12]:
```
# it is the 2 of 2*c
d.op.inputs[0].eval()
```

[12]:
```
2
```

[13]:
```
# Tensor representing a+b
d.op.inputs[1]
```

[13]:
```
<tf.Tensor 'add:0' shape=(2, 3) dtype=int32>
```

[14]:
```
# And so on...we can climb the tree
d.op.inputs[1].op.inputs[0].eval()
```

[14]:
```
array([[1, 2, 3],
       [4, 5, 6]], dtype=int32)
```

1.2 tf.placeholders

`Placeholders` are special kind of `Tensor` whose definition/values are unknown when building the operation tree. Nevertheless at least their type and shape must be known at creation time.

[15]:
```
x = tf.placeholder(tf.float32, (1, 5))
a = tf.ones((5, 1))
b = tf.matmul(x, a)
```

The value of `x` should be known when evaluating `b`. This is done through the `feed_dict` argument of the `eval` method.

[16]:
```
b.eval(feed_dict={x: [[1, 2, 3, 4, 5]]})
```

[16]:
```
array([[15.]], dtype=float32)
```

Placeholders will be later used to exchange data between the computer and the computation engine used by Tensorflow (CPU/GPU/TPU).

1.3 Symbolic calculus

Tensorflow can perform symbolic calculus on operation trees / Tensor. Let's play with some derivatives

[17]:
```
x = tf.placeholder(tf.float32, (1,))
fx = (x-1)**2
# As the output of gradients is a list, the coma after xp is mandatory
xp, = tf.gradients(fx, [x])
print(xp.eval(feed_dict={x: [0]}))
print(xp.eval(feed_dict={x: [1]}))
```

```
[-2.]
[0.]
```

1.4 tf.Variable

For now on, we have only seen *stateless* computation, also known as *functional programming*. To perform *imperative programming*, we need some mutable objects that can record changing data.

Tensorflow provides Variable object for that purpose. An initial value must be provide to the variable constructor. Moreover variables must be initialized before calling eval on them or derivated tree.

[18]:
```
w = tf.Variable([[1., 2.], [3., 4.]])
session.run(w.initializer)  # we must initialize variable before using them
w.eval()
```

```
WARNING:tensorflow:From /home/rherault/.local/venvs/spyder/lib/python3.7/site-
packages/tensorflow/python/framework/op_def_library.py:263: colocate_with (from
tensorflow.python.framework.ops) is deprecated and will be removed in a future
version.
Instructions for updating:
Colocations handled automatically by placer.
```

[18]:
```
array([[1., 2.],
       [3., 4.]], dtype=float32)
```

[19]:
```
# you can create operation to assign new values to variables
inc = w.assign(w + 1.)
session.run(inc)  # run the operation
w.eval()
```

[19]:
```
array([[2., 3.],
       [4., 5.]], dtype=float32)
```

[20]:
```
session.run(inc)  # run the operation again
w.eval()
```

[20]:
```
array([[3., 4.],
       [5., 6.]], dtype=float32)
```

Variables will be later used as memories inside the computation engine used by Tensorflow (CPU/GPU/TPU), for example as parameters of a model to be trained.

1.5 Example: minimizing a function by a gradient descent

Let's try to minimize $(3x - w)^2$ over w where x is an input provides by an user.

[21]:
```
#import numpy as np
#import tensorflow as tf
#session = tf.InteractiveSession()

# x is an input from the user so it is a placeholder
x = tf.placeholder(tf.float32, (1,))
# w is the parameter that we will do the minimization over so it is a variable
w = tf.Variable([1], dtype=tf.float32)
# lr the learning rate (Tensor Constant)
lr = tf.constant([1e-1])

nstep = 20

fw = (3*x-w)**2
gw, = tf.gradients(fw, w)
gradientstep = w.assign(w-lr*gw)

userinput = np.array([input()])

# initialize the variable
```

```
session.run(w.initializer)
for i in range(nstep):
    # get the curent position
    fwvalue, wvalue = session.run([fw, w], feed_dict={x: userinput})
    print("Step %d: f(%f)=%f" % (i, wvalue, fwvalue))
    # perform a gradient step
    session.run([gradientstep], feed_dict={x: userinput})
fwvalue, wvalue = session.run([fw, w], feed_dict={x: userinput})
print("Step %d: f(%f)=%f" % (nstep, wvalue, fwvalue))
```

```
5
Step 0: f(1.000000)=196.000000
Step 1: f(3.800000)=125.439995
Step 2: f(6.040000)=80.281601
Step 3: f(7.832000)=51.380226
Step 4: f(9.265600)=32.883343
Step 5: f(10.412480)=21.045336
Step 6: f(11.329985)=13.469012
Step 7: f(12.063988)=8.620168
Step 8: f(12.651190)=5.516909
Step 9: f(13.120952)=3.530823
Step 10: f(13.496761)=2.259727
Step 11: f(13.797409)=1.446225
Step 12: f(14.037928)=0.925583
Step 13: f(14.230342)=0.592374
Step 14: f(14.384274)=0.379119
Step 15: f(14.507419)=0.242636
Step 16: f(14.605935)=0.155287
Step 17: f(14.684748)=0.099384
Step 18: f(14.747798)=0.063606
Step 19: f(14.798239)=0.040708
Step 20: f(14.838591)=0.026053
```

1.6 Your turn

Write a code to minimize $(< w, x > +b)^2$ over w and b where w and x are vectors of size 2, b a scalar. x is given by the user (no need to be interactive).

To do the scalar product of x and y you can use:
`tf.reduce_sum(tf.multiply(x,y))`

[22]:
```
#import numpy as np
#import tensorflow as tf
#session = tf.InteractiveSession()

# x is an input from the user so it is a placeholder
x = tf.placeholder(tf.float32, (2,))
# w is the parameter that we will do the minimization over so it is a variable
w = tf.Variable([1, 1], dtype=tf.float32)
```

```
b = tf.Variable([1], dtype=tf.float32)
# lr the learning rate (Tensor Constant)
lr = tf.constant([1e-2])

nstep = 20

f = (tf.reduce_sum(tf.multiply(w, x))+b)**2
gw, gb = tf.gradients(f, [w, b])
gradientstepw = w.assign(w-lr*gw)
gradientstepb = b.assign(b-lr*gb)

userinput = np.array([1., -3.])

# initialize the variable
session.run([w.initializer, b.initializer])
for i in range(nstep):
    # get the curent position
    fvalue, = session.run(f, feed_dict={x: userinput})
    print("Step %d: %f" % (i, fvalue))
    # perform a gradient step
    session.run([gradientstepw, gradientstepb], feed_dict={x: userinput})
fvalue, wvalue, bvalue = session.run([f, w, b], feed_dict={x: userinput})
print("Step %d: %f, w:%s, b:%s" % (nstep, fvalue, wvalue, bvalue))
```

```
Step 0: 1.000000
Step 1: 0.608400
Step 2: 0.370151
Step 3: 0.225200
Step 4: 0.137011
Step 5: 0.083358
Step 6: 0.050715
Step 7: 0.030855
Step 8: 0.018772
Step 9: 0.011421
Step 10: 0.006948
Step 11: 0.004227
Step 12: 0.002572
Step 13: 0.001565
Step 14: 0.000952
Step 15: 0.000579
Step 16: 0.000352
Step 17: 0.000214
Step 18: 0.000130
Step 19: 0.000079
Step 20: 0.000048, w:[1.0902777  0.72916794], b:[1.0902777]
```