intro_to_tensorflow

December 13, 2020

1 Intro to TensorFlow

This notebook covers the basics of TF and shows you an animation with gradient descent trajectory.

2 TensorBoard

Plase note that if you are running on the Coursera platform, you won't be able to access the tensorboard instance due to the network setup there.

Run `tensorboard --logdir=./tensorboard_logs --port=7007` in bash.

If you run the notebook locally, you should be able to access TensorBoard on http://127.0.0.1:7007/

```
In [1]: import tensorflow as tf
        import sys
        sys.path.append("../..")
        from keras_utils import reset_tf_session
        s = reset_tf_session()
        print("We're using TF", tf.__version__)
```

```
Using TensorFlow backend.


We're using TF 1.2.1
```

3 Warming up

For starters, let's implement a python function that computes the sum of squares of numbers from 0 to N-1.

```
In [2]: import numpy as np

        def sum_python(N):
            return np.sum(np.arange(N)**2)
```

```
In [3]: %%time
        sum_python(10**5)
```

```
CPU times: user 1.66 ms, sys: 60 ţs, total: 1.72 ms
Wall time: 1.06 ms
```

```
Out[3]: 333328333350000
```

4 Tensoflow teaser

Doing the very same thing

```
In [4]: # An integer parameter
        N = tf.placeholder('int64', name="input_to_your_function")

        # A recipe on how to produce the same result
        result = tf.reduce_sum(tf.range(N)**2)
```

```
In [5]: # just a graph definition
        result
```

```
Out[5]: <tf.Tensor 'Sum:0' shape=() dtype=int64>
```

```
In [6]: %%time
        # actually executing
        result.eval({N: 10**5})
```

```
CPU times: user 5.16 ms, sys: 42 ţs, total: 5.2 ms
Wall time: 3.5 ms
```

```
Out[6]: 333328333350000
```

```
In [7]: # logger for tensorboard
        writer = tf.summary.FileWriter("tensorboard_logs", graph=s.graph)
```

5 How does it work?

1. Define placeholders where you'll send inputs
2. Make a symbolic graph: a recipe for mathematical transformation of those placeholders
3. Compute outputs of your graph with particular values for each placeholder

- `output.eval({placeholder: value})`
- `s.run(output, {placeholder: value})`

So far there are two main entities: "placeholder" and "transformation" (operation output) * Both can be numbers, vectors, matrices, tensors, etc. * Both can be int32/64, floats, booleans (uint8) of various size.

- You can define new transformations as an arbitrary operation on placeholders and other transformations

- `tf.reduce_sum(tf.arange(N)**2)` are 3 sequential transformations of placeholder `N`
- There's a tensorflow symbolic version for every numpy function
- `a+b, a/b, a**b, ...` behave just like in numpy
- `np.mean` -> `tf.reduce_mean`
- `np.arange` -> `tf.range`
- `np.cumsum` -> `tf.cumsum`
- If you can't find the operation you need, see the docs.

`tf.contrib` has many high-level features, may be worth a look.

```
In [8]: with tf.name_scope("Placeholders_examples"):
            # Default placeholder that can be arbitrary float32
            # scalar, vertor, matrix, etc.
            arbitrary_input = tf.placeholder('float32')

            # Input vector of arbitrary length
            input_vector = tf.placeholder('float32', shape=(None,))

            # Input vector that _must_ have 10 elements and integer type
            fixed_vector = tf.placeholder('int32', shape=(10,))

            # Matrix of arbitrary n_rows and 15 columns
            # (e.g. a minibatch of your data table)
            input_matrix = tf.placeholder('float32', shape=(None, 15))

            # You can generally use None whenever you don't need a specific shape
            input1 = tf.placeholder('float64', shape=(None, 100, None))
            input2 = tf.placeholder('int32', shape=(None, None, 3, 224, 224))

            # elementwise multiplication
            double_the_vector = input_vector*2

            # elementwise cosine
            elementwise_cosine = tf.cos(input_vector)

            # difference between squared vector and vector itself plus one
            vector_squares = input_vector**2 - input_vector + 1
```

```
In [9]: my_vector =  tf.placeholder('float32', shape=(None,), name="VECTOR_1")
        my_vector2 = tf.placeholder('float32', shape=(None,))
        my_transformation = my_vector * my_vector2 / (tf.sin(my_vector) + 1)
```

```
In [10]: print(my_transformation)
```

```
Tensor("truediv:0", shape=(?,), dtype=float32)
```

```
In [11]: dummy = np.arange(5).astype('float32')
         print(dummy)
         my_transformation.eval({my_vector: dummy, my_vector2: dummy[::-1]})
```

```
[ 0.  1.  2.  3.  4.]
```

```
Out[11]: array([ 0.        ,  1.62913239,  2.09501147,  2.62899613,  0.        ], dtype=float32)
```

```
In [12]: writer.add_graph(my_transformation.graph)
         writer.flush()
```

TensorBoard allows writing scalars, images, audio, histogram. You can read more on tensorboard usage here.

6 Summary

- Tensorflow is based on computation graphs
- A graph consists of placeholders and transformations

7 Loss function: Mean Squared Error

Loss function must be a part of the graph as well, so that we can do backpropagation.

```
In [14]: with tf.name_scope("MSE"):
             y_true = tf.placeholder("float32", shape=(None,), name="y_true")
             y_predicted = tf.placeholder("float32", shape=(None,), name="y_predicted")
             # Implement MSE(y_true, y_predicted), use tf.reduce_mean(...)
             # mse = ### YOUR CODE HERE ###
             mse = tf.reduce_mean((y_true - y_predicted)**2)
         def compute_mse(vector1, vector2):
             return mse.eval({y_true: vector1, y_predicted: vector2})
```

```
In [15]: writer.add_graph(mse.graph)
         writer.flush()
```

```
In [16]: # Rigorous local testing of MSE implementation
         import sklearn.metrics
         for n in [1, 5, 10, 10**3]:
             elems = [np.arange(n), np.arange(n, 0, -1), np.zeros(n),
                      np.ones(n), np.random.random(n), np.random.randint(100, size=n)]
             for el in elems:
                 for el_2 in elems:
                     true_mse = np.array(sklearn.metrics.mean_squared_error(el, el_2))
                     my_mse = compute_mse(el, el_2)
                     if not np.allclose(true_mse, my_mse):
                         print('mse(%s,%s)' % (el, el_2))
                         print("should be: %f, but your function returned %f" % (true_mse, my_ms
                         raise ValueError('Wrong result')
```

8 Variables

Placeholder and transformation values are not stored in the graph once the execution is finished. This isn't too comfortable if you want your model to have parameters (e.g. network weights) that are always present, but can change their value over time.

Tensorflow solves this with `tf.Variable` objects. * You can assign variable a value at any time in your graph * Unlike placeholders, there's no need to explicitly pass values to variables when `s.run(...)`-ing * You can use variables the same way you use transformations

```
In [17]: # Creating a shared variable
         shared_vector_1 = tf.Variable(initial_value=np.ones(5),
                                       name="example_variable")
```

```
In [18]: # Initialize variable(s) with initial values
         s.run(tf.global_variables_initializer())

         # Evaluating the shared variable
         print("Initial value", s.run(shared_vector_1))
```

```
Initial value [ 1.  1.  1.  1.  1.]
```

```
In [19]: # Setting a new value
         s.run(shared_vector_1.assign(np.arange(5)))

         # Getting that new value
         print("New value", s.run(shared_vector_1))
```

```
New value [ 0.  1.  2.  3.  4.]
```

9 tf.gradients - why graphs matter

- Tensorflow can compute derivatives and gradients automatically using the computation graph
- True to its name it can manage matrix derivatives
- Gradients are computed as a product of elementary derivatives via the chain rule:

$$\frac{\partial f(g(x))}{\partial x} = \frac{\partial f(g(x))}{\partial g(x)} \cdot \frac{\partial g(x)}{\partial x}$$

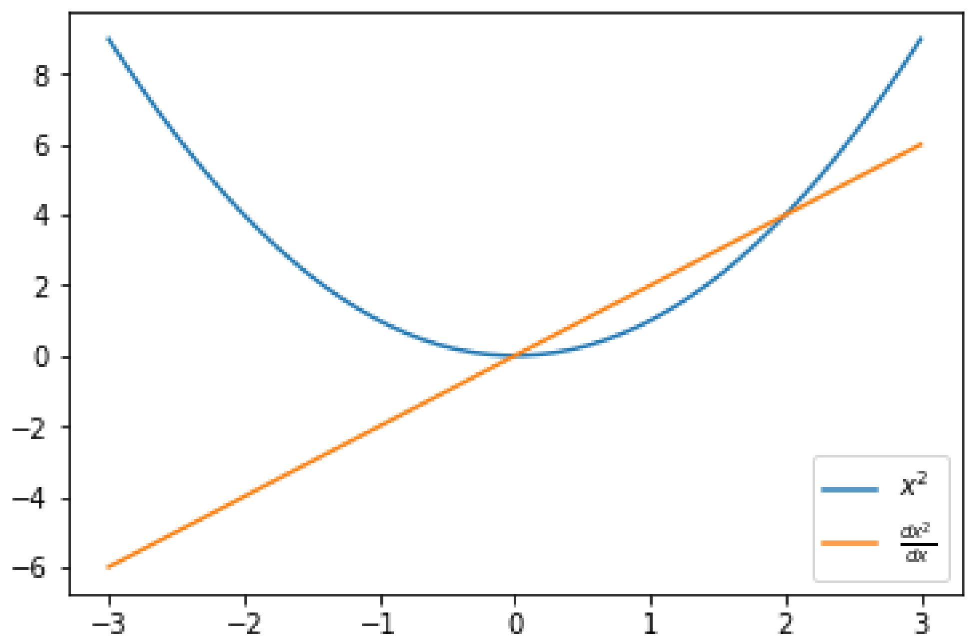
It can get you the derivative of any graph as long as it knows how to differentiate elementary operations

```
In [20]: my_scalar = tf.placeholder('float32')

         scalar_squared = my_scalar**2

         # A derivative of scalar_squared by my_scalar
         derivative = tf.gradients(scalar_squared, [my_scalar, ])
```

In [21]: derivative

Out[21]: [<tf.Tensor 'gradients/pow_1_grad/Reshape:0' shape=<unknown> dtype=float32>]

In [22]:
```
import matplotlib.pyplot as plt
%matplotlib inline

x = np.linspace(-3, 3)
x_squared, x_squared_der = s.run([scalar_squared, derivative[0]],
                                 {my_scalar:x})

plt.plot(x, x_squared,label="$x^2$")
plt.plot(x, x_squared_der, label=r"$\frac{dx^2}{dx}$")
plt.legend();
```

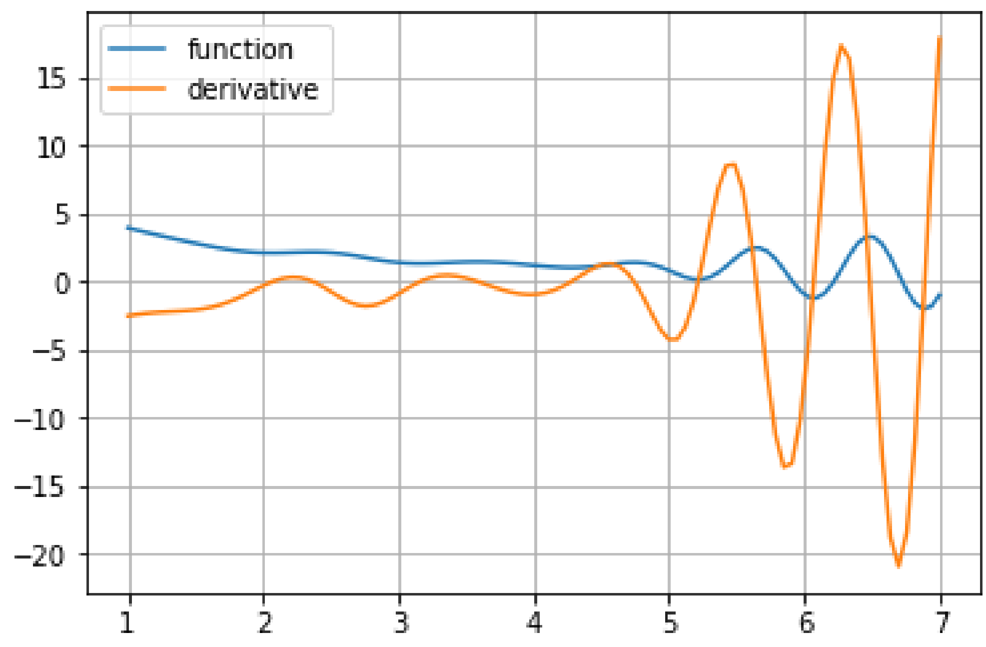


10 Why that rocks

In [23]:
```
my_vector = tf.placeholder('float32', [None])
# Compute the gradient of the next weird function over my_scalar and my_vector
# Warning! Trying to understand the meaning of that function may result in permanent br
weird_psychotic_function = tf.reduce_mean(
    (my_vector+my_scalar)**(1+tf.nn.moments(my_vector,[0])[1]) +
    1./ tf.atan(my_scalar))/(my_scalar**2 + 1) + 0.01*tf.sin(
    2*my_scalar**1.5)*(tf.reduce_sum(my_vector)* my_scalar**2
                      )*tf.exp((my_scalar-4)**2)/(
```

```
    1+tf.exp((my_scalar-4)**2))*(1.-(tf.exp(-(my_scalar-4)**2)
                                )/(1+tf.exp(-(my_scalar-4)**2)))**2

der_by_scalar = tf.gradients(weird_psychotic_function, my_scalar)
der_by_vector = tf.gradients(weird_psychotic_function, my_vector)
```

In [24]:
```
# Plotting the derivative
scalar_space = np.linspace(1, 7, 100)

y = [s.run(weird_psychotic_function, {my_scalar:x, my_vector:[1, 2, 3]})
     for x in scalar_space]

plt.plot(scalar_space, y, label='function')

y_der_by_scalar = [s.run(der_by_scalar,
                         {my_scalar:x, my_vector:[1, 2, 3]})
                   for x in scalar_space]

plt.plot(scalar_space, y_der_by_scalar, label='derivative')
plt.grid()
plt.legend();
```



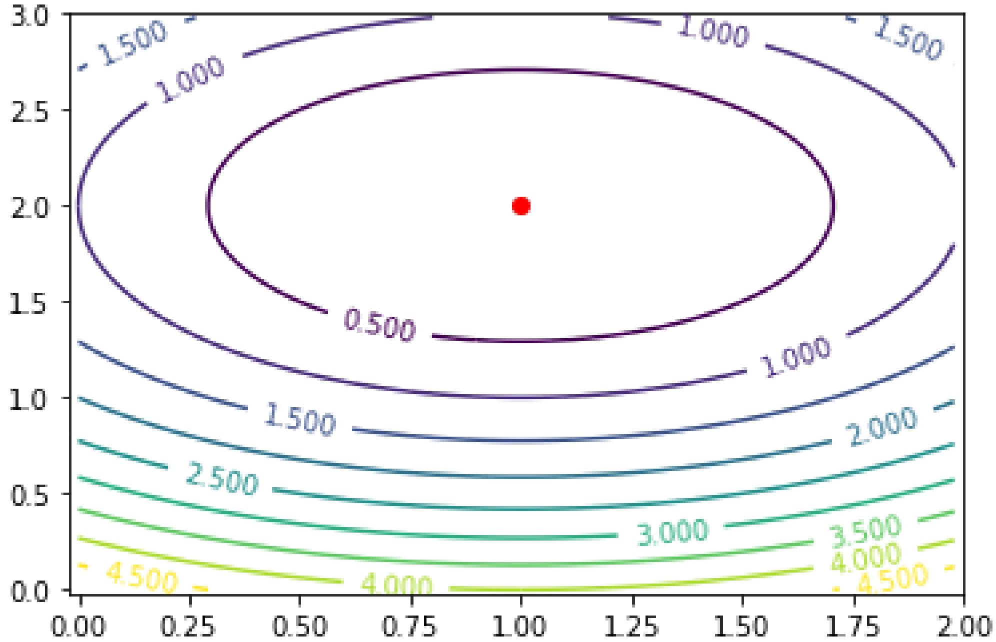
11 Almost done - optimizers

While you can perform gradient descent by hand with automatic gradients from above, tensorflow also has some optimization methods implemented for you. Recall momentum & rmsprop?

In [25]:
```
y_guess = tf.Variable(np.zeros(2, dtype='float32'))
y_true = tf.range(1, 3, dtype='float32')

loss = tf.reduce_mean((y_guess - y_true + 0.5*tf.random_normal([2]))**2)

step = tf.train.MomentumOptimizer(0.03, 0.5).minimize(loss, var_list=y_guess)
```

Let's draw a trajectory of a gradient descent in 2D

In [26]:
```
from matplotlib import animation, rc
import matplotlib_utils
from IPython.display import HTML, display_html

# nice figure settings
fig, ax = plt.subplots()
y_true_value = s.run(y_true)
level_x = np.arange(0, 2, 0.02)
level_y = np.arange(0, 3, 0.02)
X, Y = np.meshgrid(level_x, level_y)
Z = (X - y_true_value[0])**2 + (Y - y_true_value[1])**2
ax.set_xlim(-0.02, 2)
ax.set_ylim(-0.02, 3)
s.run(tf.global_variables_initializer())
ax.scatter(*s.run(y_true), c='red')
contour = ax.contour(X, Y, Z, 10)
ax.clabel(contour, inline=1, fontsize=10)
line, = ax.plot([], [], lw=2)

# start animation with empty trajectory
def init():
    line.set_data([], [])
    return (line,)

trajectory = [s.run(y_guess)]

# one animation step (make one GD step)
def animate(i):
    s.run(step)
    trajectory.append(s.run(y_guess))
    line.set_data(*zip(*trajectory))
    return (line,)

anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=100, interval=20, blit=True)
```



In [27]:
```
try:
    display_html(HTML(anim.to_html5_video()))
except (RuntimeError, KeyError):
    # In case the build-in renderers are unaviable, fall back to
    # a custom one, that doesn't require external libraries
    anim.save(None, writer=matplotlib_utils.SimpleMovieWriter(0.001))
```