ML0120EN-1.1-Review-TensorFlow-Hello-World

July 20, 2017

TENSORFLOW'S HELLO WORLD

In this notebook we will overview the basics of TensorFlow, learn it's structures and see what is the motivation to use it -

How does TensorFlow work?

- Building a Graph
- Defining multidimensional arrays using TensorFlow
- Why Tensors?
- Variables
- Placeholders
- Operations

How does TensorFlow work?

TensorFlow's capability to execute the code on different devices such as CPUs and GPUs is a consequence of it's specific structure:

TensorFlow defines computations as Graphs, and these are made with operations (also know as "ops"). So, when we work with TensorFlow, it is the same as defining a series of operations in a Graph.

To execute these operations as computations, we must launch the Graph into a Session. The session translates and passes the operations represented into the graphs to the device you want to execute them on, be it a GPU or CPU.

For example, the image below represents a graph in TensorFlow. W, x and b are tensors over the edges of this graph. *MatMul* is an operation over the tensors W and x, after that *Add* is called and add the result of the previous operator with b. The resultant tensors of each operation cross the next one until the end where it's possible to get the wanted result.

0.0.1 Importing TensorFlow

To use TensorFlow, we need to import the library. We imported it and optionally gave it the name "tf", so the modules can be accessed by **tf.module-name**:

```
In [1]: import tensorflow as tf
```

Building a Graph

As we said before, TensorFlow works as a graph computational model. Let's create our first graph.

To create our first graph we will utilize **source operations**, which do not need any information input. These source operations or **source ops** will pass their information to other operations which will execute computations.

To create two source operations which will output numbers we will define two constants:

```
In [2]: a = tf.constant([2])
        b = tf.constant([3])
```

After that, let's make an operation over these variables. The function `tf.add()` adds two elements (you could also use `c = a + b`).

```
In [3]: c = tf.add(a, b)
        # c = a + b is also a way to define the sum of the terms
```

Then TensorFlow needs to initialize a session to run our code. Sessions are, in a way, a context for creating a graph inside TensorFlow. Let's define our session:

```
In [4]: session = tf.Session()
```

Let's run the session to get the result from the previous defined 'c' operation:

```
In [5]: result = session.run(c)
        print result
```

```
[5]
```

Close the session to release resources:

```
In [6]: session.close()
```

To avoid having to close sessions every time, we can define them in a **with** block, so after running the **with** block the session will close automatically:

```
In [7]: with tf.Session() as session:
            result = session.run(c)
            print result
```

```
[5]
```

Even this silly example of adding 2 constants to reach a simple result defines the basis of TensorFlow. Define your edge (In this case our constants), include nodes (operations, like `tf.add`), and start a session to build a graph.

0.0.2 What is the meaning of Tensor?

In TensorFlow all data is passed between operations in a computation graph, and these are passed in the form of Tensors, hence the name of TensorFlow. The word **tensor** from new latin means "that which stretches". It is a mathematical object that is named **tensor** because an early application of tensors was the study of materials stretching under tension. The contemporary meaning of tensors can be taken as multidimensional arrays.

That's great, but... what are these multidimensional arrays?

Going back a little bit to physics to understand the concept of dimensions:

[Image Source]

The zero dimension can be seen as a point, a single object or a single item.

The first dimension can be seen as a line, a one-dimensional array can be seen as numbers along this line, or as points along the line. One dimension can contain infinite zero dimension/points elements.

The second dimension can be seen as a surface, a two-dimensional array can be seen as an infinite series of lines along an infinite line.

The third dimension can be seen as volume, a three-dimensional array can be seen as an infinite series of surfaces along an infinite line.

The Fourth dimension can be seen as the hyperspace or spacetime, a volume varying through time, or an infinite series of volumes along an infinite line. And so forth on...

As mathematical objects:

[Image Source]

Summarizing:

```
<td><b>Dimension</b></td>
<td><b>Physical Representation</b></td>
<td><b>Mathematical Object</b></td>
<td><b>In Code</b></td>

<td>Zero </td>
<td>Point</td>
<td>Scalar (Single Number)</td>
<td>[ 1 ]</td>

<td>One</td>
<td>Line</td>
<td>Vector (Series of Numbers) </td>
<td>[ 1,2,3,4,... ]</td>

<td>Two</td>
<td>Surface</td>
<td>Matrix (Table of Numbers)</td>
<td>[ [1,2,3,4,...], [1,2,3,4,...], [1,2,3,4,...],... ]</td>

<td>Three</td>
<td>Volume</td>
<td style="background-color:yellow;">Tensor (Cube of Numbers)</td>
<td>[ [[1,2,...], [1,2,...], [1,2,...],...], [[1,2,...], [1,2,...], [1,2,...],...], [[1,2,...]
```

Defining multidimensional arrays using TensorFlow Now we will try to define such arrays using TensorFlow:

```
In [8]: Scalar = tf.constant([2])
        Vector = tf.constant([5,6,2])
        Matrix = tf.constant([[1,2,3],[2,3,4],[3,4,5]])
        Tensor = tf.constant( [ [[1,2,3],[2,3,4],[3,4,5]] , [[4,5,6],[5,6,7],[6,7,8]] , [[7,8,9
        with tf.Session() as session:
            result = session.run(Scalar)
            print "Scalar (1 entry):\n %s \n" % result
            result = session.run(Vector)
            print "Vector (3 entries) :\n %s \n" % result
            result = session.run(Matrix)
            print "Matrix (3x3 entries):\n %s \n" % result
            result = session.run(Tensor)
            print "Tensor (3x3x3 entries) :\n %s \n" % result

Scalar (1 entry):
 [2]

Vector (3 entries) :
 [5 6 2]

Matrix (3x3 entries):
 [[1 2 3]
 [2 3 4]
 [3 4 5]]

Tensor (3x3x3 entries) :
 [[[ 1  2  3]
  [ 2  3  4]
  [ 3  4  5]]

 [[ 4  5  6]
  [ 5  6  7]
  [ 6  7  8]]

 [[ 7  8  9]
  [ 8  9 10]
  [ 9 10 11]]]
```

Now that you understand these data structures, I encourage you to play with them using some previous functions to see how they will behave, according to their structure types:

```
In [9]: Matrix_one = tf.constant([[1,2,3],[2,3,4],[3,4,5]])
        Matrix_two = tf.constant([[2,2,2],[2,2,2],[2,2,2]])
```

```
        first_operation = tf.add(Matrix_one, Matrix_two)
        second_operation = Matrix_one + Matrix_two

        with tf.Session() as session:
            result = session.run(first_operation)
            print "Defined using tensorflow function :"
            print(result)
            result = session.run(second_operation)
            print "Defined using normal expressions :"
            print(result)
```

```
Defined using tensorflow function :
[[3 4 5]
 [4 5 6]
 [5 6 7]]
Defined using normal expressions :
[[3 4 5]
 [4 5 6]
 [5 6 7]]
```

With the regular symbol definition and also the TensorFlow function we were able to get an element-wise multiplication, also known as **Hadamard product**.

But what if we want the regular matrix product?

We then need to use another TensorFlow function called **`tf.matmul()`**:

```
In [10]: Matrix_one = tf.constant([[2, 3], [3, 4]])
         Matrix_two = tf.constant([[2, 3], [3, 4]])

         first_operation = tf.matmul(Matrix_one, Matrix_two)
         second_operation = Matrix_one * Matrix_two

         with tf.Session() as session:
             result = session.run(first_operation)
             print "Defined using tensorflow function tf.matmul() :"
             print result
             result = session.run(second_operation)
             print "Defined using regular matrix multiplication (i.e. element-wise multiplicati
             print result
```

```
Defined using tensorflow function tf.matmul() :
[[13 18]
 [18 25]]
Defined using regular matrix multiplication (i.e. element-wise multiplication) :
[[ 4  9]
 [ 9 16]]
```

We could also define this multiplication ourselves, but there is a function that already does that, so no need to reinvent the wheel!

Why Tensors?

The Tensor structure helps us by giving the freedom to shape the dataset the way we want.

And it is particularly helpful when dealing with images, due to the nature of how information in images are encoded,

Thinking about images, its easy to understand that it has a height and width, so it would make sense to represent the information contained in it with a two dimensional strucutre (a matrix)... until you remember that images have colors, and to add information about the colors, we need another dimension, and thats when Tensors become particulary helpful.

Images are encoded into color channels, the image data is represented into each color intensity in a color channel at a given point, the most common one being RGB, which means Red, Blue and Green. The information contained into an image is the intensity of each channel color into the width and height of the image, just like this:

[Image Source]

So the intensity of the red channel at each point with width and height can be represented into a matrix, the same goes for the blue and green channels, so we end up having three matrices, and when these are combined they form a tensor.

Variables

Now that we are more familiar with the structure of data, we will take a look at how TensorFlow handles variables.

To define variables we use the command **`tf.Variable()`**. To be able to use variables in a computation graph it is necessary to initialize them before running the graph in a session. This is done by running **`tf.global_variables_initializer()`**.

To update the value of a variable, we simply run an assign operation **`tf.assign()`** that assigns a value to the variable:

```
In [11]: # define variable
         state = tf.Variable(0)
```

Let's first create a simple counter, a variable that increases one unit at a time:

```
In [12]: one = tf.constant(1)
         new_value = tf.add(state, one)

         # update the value of a variable
         update = tf.assign(state, new_value)
```

Variables must be initialized by running an initialization operation after having launched the graph. We first have to add the initialization operation to the graph:

```
In [13]: # initialize variables
         init_op = tf.global_variables_initializer()
```

We then start a session to run the graph, first initialize the variables, then print the initial value of the **state** variable, and then run the operation of updating the **state** variable and printing the result after each update:

```
In [14]: state

Out[14]: <tf.Variable 'Variable:0' shape=() dtype=int32_ref>

In [15]: one

Out[15]: <tf.Tensor 'Const_10:0' shape=() dtype=int32>

In [16]: new_value

Out[16]: <tf.Tensor 'Add_2:0' shape=() dtype=int32>

In [17]: update

Out[17]: <tf.Tensor 'Assign:0' shape=() dtype=int32_ref>

In [18]: with tf.Session() as session:
             session.run(init_op)            # initialize all variables
             print(session.run(state))       # print the initial value of the state variable
             for x in range(3):
                 print "--- Round", x, " ---"
                 session.run(update)
                 print session.run(state)

0
--- Round 0  ---
1
--- Round 1  ---
2
--- Round 2  ---
3
```

Placeholders

Now we know how to manipulate variables inside TensorFlow, but what about feeding data outside of a TensorFlow model?

If you want to feed data to a TensorFlow model from outside a model, you will need to use placeholders.

So what are these placeholders and what do they do?

Placeholders can be seen as "holes" in your model, "holes" which you will pass the data to, you can create them using **`tf.placeholder(datatype)`**, where `datatype` specifies the type of data (integers, floating points, strings, booleans) along with its precision (8, 16, 32, 64) bits.

The definition of each data type with the respective python sintax is defined as:

Data type	Python type	Description
DT_FLOAT	tf.float32	32 bits floating point.
DT_DOUBLE	tf.float64	64 bits floating point.

Data type	Python type	Description
DT_INT8	tf.int8	8 bits signed integer.
DT_INT16	tf.int16	16 bits signed integer.
DT_INT32	tf.int32	32 bits signed integer.
DT_INT64	tf.int64	64 bits signed integer.
DT_UINT8	tf.uint8	8 bits unsigned integer.
DT_STRING	tf.string	Variable length byte arrays. Each element of a Tensor is a byte array.
DT_BOOL	tf.bool	Boolean.
DT_COMPLEX64	tf.complex64	Complex number made of two 32 bits floating points: real and imaginary parts.
DT_COMPLEX128	tf.complex128	Complex number made of two 64 bits floating points: real and imaginary parts.
DT_QINT8	tf.qint8	8 bits signed integer used in quantized Ops.
DT_QINT32	tf.qint32	32 bits signed integer used in quantized Ops.

Data type	Python type	Description
DT_QUINT8	tf.quint8	8 bits unsigned integer used in quantized Ops.

[Table Source]

So we create a placeholder:

```
In [19]: a = tf.placeholder(tf.float32)
```

And define a simple multiplication operation:

```
In [20]: b = a * 2
```

Now we need to define and run the session, but since we created a "hole" in the model to pass the data, when we initialize the session we are obligated to pass an argument with the data, otherwise we would get an error.

To pass the data to the model we call the session with an extra argument `feed_dict` in which we should pass a dictionary with each placeholder name folowed by its respective data, just like this:

```
In [21]: with tf.Session() as sess:
             result = sess.run(b, feed_dict={a: 3.5})
             print result
```

```
7.0
```

Since data in TensorFlow is passed in form of multidimensional arrays we can pass any kind of tensor through the placeholders to get the answer to the simple multiplication operation:

```
In [22]: dictionary={a: [ [ [1,2,3],[4,5,6],[7,8,9],[10,11,12] ] , [ [13,14,15],[16,17,18],[19

         with tf.Session() as sess:
             result = sess.run(b, feed_dict = dictionary)
             print result
```

```
[[[  2.   4.   6.]
  [  8.  10.  12.]
  [ 14.  16.  18.]
  [ 20.  22.  24.]]

 [[ 26.  28.  30.]
  [ 32.  34.  36.]
  [ 38.  40.  42.]
  [ 44.  46.  48.]]]
```

Operations

Operations are nodes that represent the mathematical operations over the tensors on a graph. These operations can be any kind of functions, like add and subtract tensor or maybe an activation function.

tf.matmul, **tf.add**, **tf.nn.sigmoid** are some of the operations in TensorFlow. These are like functions in python but operate directly over tensors and each one does a specific thing.

Other operations can be easily found in: https://www.tensorflow.org/versions/r0.9/api_docs/python/index

```
In [24]: a = tf.constant([5])
         b = tf.constant([2])
         c = tf.add(a, b)
         d = tf.subtract(a, b)

         with tf.Session() as session:
             result = session.run(c)
             print 'c =: %s' % result
             result = session.run(d)
             print 'd =: %s' % result

c =: [7]
d =: [3]
```

tf.nn.sigmoid is an activiation function, it's a little more complicated, but this function helps learning models to evaluate what kind of information is good or not.