ML0120EN-1.1-Review-TensorFlow-Hello-World

January 18, 2020

TENSORFLOW'S HELLO WORLD

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

Table of Contents

How does TensorFlow work?

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

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.

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:

[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 which we named as graph1.

[2]:
```
graph1 = tf.Graph()
```

Now we call the TensorFlow functions that construct new tf.Operation and tf.Tensor objects and add them to the graph1. As mentioned, each tf.Operation is a node and each tf.Tensor is an edge in the graph.

Lets add 2 constants to our graph. For example, calling tf.constant([2], name = 'constant_a') adds a single tf.Operation to the default graph. This operation produces the value 2, and returns a tf.Tensor that represents the value of the constant.
Notice: tf.constant([2], name="constant_a") creates a new tf.Operation named "constant_a" and returns a tf.Tensor named "constant_a:0".

[3]:
```
with graph1.as_default():
    a = tf.constant([2], name = 'constant_a')
    b = tf.constant([3], name = 'constant_b')
```

Lets look at the tensor **a**.

[4]:
```
a
```

[4]:
```
<tf.Tensor 'constant_a:0' shape=(1,) dtype=int32>
```

As you can see, it just show the name, shape and type of the tensor in the graph. We will see it's value when we run it in a TensorFlow session.

[5]:
```
# Printing the value of a
sess = tf.Session(graph = graph1)
result = sess.run(a)
print(result)
sess.close()
```

```
[2]
```

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

[6]:
```
with graph1.as_default():
    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:

[7]:
```
sess = tf.Session(graph = graph1)
```

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

[8]:
```
result = sess.run(c)
print(result)
```

```
[5]
```

Close the session to release resources:

[9]:
```
sess.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:

[10]:
```
with tf.Session(graph = graph1) as sess:
    result = sess.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 operations (In this case our constants and *tf.add*), and start a session to build a graph.

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:

Dimension

Physical Representation

Mathematical Object

In Code

Zero

Point

Scalar (Single Number)

[1]

One

Line

Vector (Series of Numbers)

[1,2,3,4,...]

Two

Surface

Matrix (Table of Numbers)

[[1,2,3,4,...], [1,2,3,4,...], [1,2,3,4,...],...]

Three

Volume

Tensor (Cube of Numbers)

[[[1,2,...], [1,2,...], [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:

[11]:
```
graph2 = tf.Graph()
with graph2.as_default():
    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],[8,9,10],[9,10,11]] ] )
with tf.Session(graph = graph2) as sess:
    result = sess.run(Scalar)
    print ("Scalar (1 entry):\n %s \n" % result)
    result = sess.run(Vector)
    print ("Vector (3 entries) :\n %s \n" % result)
    result = sess.run(Matrix)
    print ("Matrix (3x3 entries):\n %s \n" % result)
    result = sess.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]]]
```

tf.shape returns the shape of our data structure.

[12]:
```
Scalar.shape
```

[12]:
```
TensorShape([])
```

[13]:
```
Tensor.shape
```

[13]:
```
TensorShape([Dimension(3), Dimension(3), Dimension(3)])
```

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:

[14]:
```
graph3 = tf.Graph()
with graph3.as_default():
    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]])
    add_1_operation = tf.add(Matrix_one, Matrix_two)
    add_2_operation = Matrix_one + Matrix_two

with tf.Session(graph =graph3) as sess:
    result = sess.run(add_1_operation)
    print ("Defined using tensorflow function :")
    print(result)
    result = sess.run(add_2_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():

[15]:
```
graph4 = tf.Graph()
with graph4.as_default():
    Matrix_one = tf.constant([[2,3],[3,4]])
    Matrix_two = tf.constant([[2,3],[3,4]])

    mul_operation = tf.matmul(Matrix_one, Matrix_two)

with tf.Session(graph = graph4) as sess:
    result = sess.run(mul_operation)
    print ("Defined using tensorflow function :")
    print(result)
```

```
Defined using tensorflow function :
[[13 18]
 [18 25]]
```

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 in 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 structure (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 particularly 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. First of all, having tensors, why do we need variables?
TensorFlow variables are used to share and persistent some stats that are manipulated by our program. That is, when you define a variable, TensorFlow adds a tf.Operation to your graph. Then, this operation will store a writable tensor value that persists between tf.Session.run calls. So, you can update the value of a variable through each run, while you cannot update tensor (e.g a tensor created by tf.constant()) through multiple runs in a session.

How to define a variable?
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 that assigns a value to the variable:

[16]:
```
v = tf.Variable(0)
```

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

To do this we use the tf.assign(reference_variable, value_to_update) command. tf.assign takes in two arguments, the reference_variable to update, and assign it to the value_to_update it by.

[17]:
```
update = tf.assign(v, v+1)
```

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:

[18]:
```
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:

```
[19]: with tf.Session() as session:
          session.run(init_op)
          print(session.run(v))
          for _ in range(3):
              session.run(update)
              print(session.run(v))
```

```
0
1
2
3
```

# Placeholders

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

If you want to feed data to a TensorFlow graph from outside a graph, 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 syntax is defined as:

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

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

So we create a placeholder:

[20]:
```
a = tf.placeholder(tf.float32)
```

And define a simple multiplication operation:

[21]:
```
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 into the model we call the session with an extra argument feed_dict in which we should pass a dictionary with each placeholder name followed by its respective data, just like this:

[22]:
```
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:

[23]:
```
dictionary={a: [ [ [1,2,3],[4,5,6],[7,8,9],[10,11,12] ] , [␣
 ↪[13,14,15],[16,17,18],[19,20,21],[22,23,24] ] ] }

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.constant, 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.html

[24]:
```
graph5 = tf.Graph()
with graph5.as_default():
    a = tf.constant([5])
    b = tf.constant([2])
    c = tf.add(a,b)
    d = tf.subtract(a,b)

with tf.Session(graph = graph5) as sess:
    result = sess.run(c)
    print ('c =: %s' % result)
    result = sess.run(d)
    print ('d =: %s' % result)
```

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

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

0.1 Want to learn more?

Running deep learning programs usually needs a high performance platform. **PowerAI** speeds up deep learning and AI. Built on IBM's Power Systems, **PowerAI** is a scalable software platform that accelerates deep learning and AI with blazing performance for individual users or enterprises. The **PowerAI** platform supports popular machine learning libraries and dependencies including TensorFlow, Caffe, Torch, and Theano. You can use PowerAI on IMB Cloud.

Also, you can use **Watson Studio** to run these notebooks faster with bigger datasets.___Watson Studio___ is IBM's leading cloud solution for data scientists, built by data scientists. With Jupyter notebooks, RStudio, Apache Spark and popular libraries pre-packaged in the cloud, **Watson Studio** enables data scientists to collaborate on their projects without having to install anything. Join the fast-growing community of **Watson Studio** users today with a free account at Watson Studio.This is the end of this lesson. Thank you for reading this notebook, and good luck on your studies.

0.1.1 Thanks for completing this lesson!

Notebook created by: Saeed Aghabozorgi and Rafael Belo Da Silva

0.1.2 References:

https://www.tensorflow.org/versions/r0.9/get_started/index.html http://jrmeyer.github.io/tutorial/2016/02/01/
Tutorial.html https://www.tensorflow.org/versions/r0.9/api_docs/python/index.html
https://www.tensorflow.org/versions/r0.9/resources/dims_types.html
https://en.wikipedia.org/wiki/Dimension https://book.mql4.com/variables/arrays
https://msdn.microsoft.com/en-us/library/windows/desktop/dn424131(v=vs.85).aspx

Copyright © 2018 Cognitive Class. This notebook and its source code are released under the terms of the MIT License.