TENSORFLOW'S HELLO WORLD

TENSORFLOW'S HELLO WORLD

Objective for this Notebook

1. How does TensorFlow work?
2. Building a Graph.
3. Meaning of Tensor?
4. Defining multidimensional arrays using TensorFlow.
5. How TensorFlow handles Variables.
6. What are these Placeholders and what do they do?
7. Learn Operations using TensorFlow.

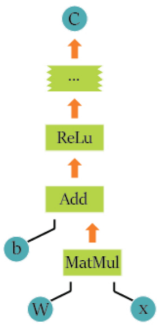
In this notebook we will overview the basics of TensorFlow, learn it's structure and see what is the motivation to use it

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.

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 *b*, 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.



With TensorFlow 2.x, **Eager Execution** is enabled by default. This allows TensorFlow code to be executed and evaluated line by line. Before version 2.x was released, every graph had to be run wihthin a TensorFlow session. This only allowed for the entire graph to be run all at once. This would make debugging the computation graph each time.

Installing TensorFlow

We begin by installing TensorFlow version 2.2.0 and its required prerequistes.

[ ]:
```
!pip install grpcio==1.24.3
!pip install tensorflow==2.2.0
```

Restart kernel for latest version of TensorFlow to be activated

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:

[2]:
```
import tensorflow as tf
if not tf.__version__ == '2.2.0':
    print(tf.__version__)
    raise ValueError('please upgrade to TensorFlow 2.2.0, or restart your Kernel (Kernel->Restart & Clear Output)')
```

IMPORTANT! => Please restart the kernel by clicking on "Kernel"->"Restart and Clear Outout" and wait until all output disapears. Then your changes are beeing picked up

---

tf.function and AutoGraph

Now we call the TensorFlow functions that construct new tf.Operation and tf.Tensor objects. 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]:
```
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: shape=(1,), dtype=int32, numpy=array([2], dtype=int32)>
```

As you can see, it just shows the name, shape and type of the tensor in the graph. We will see it's value by running the TensorFlow code as shown below.

[5]:
```
tf.print(a.numpy()[0])
```

```
2
```

Annotating the python functions with **tf.function** uses TensorFlow Autograph to create a TensorFlow static execution graph for the function. tf.function annotation tells TensorFlow Autograph to transform function *add* into TensorFlow control flow, which then defines the TensorFlow static execution graph.

[6]:
```
@tf.function
def add(a,b):
    c = tf.add(a, b)
    #c = a + b is also a way to define the sum of the terms
    print(c)
    return c
```

[7]:
```
result = add(a,b)
tf.print(result[0])
```

```
Tensor("Add:0", shape=(1,), dtype=int32)
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*), define a Python function named *add* and decorate it with using the *tf.function* annotator.

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:

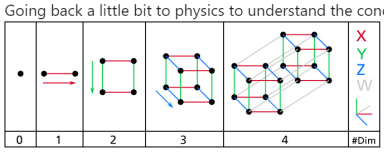


Image Source

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:

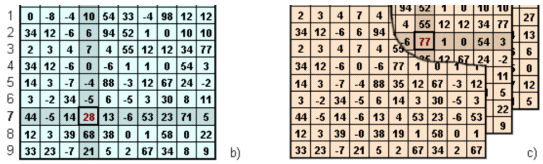


Image Source

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:

[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],[8,9,10],[9,10,11]] ] )

print ("Scalar (1 entry):\n %s \n" % Scalar)

print ("Vector (3 entries) :\n %s \n" % Vector)

print ("Matrix (3x3 entries):\n %s \n" % Matrix)

print ("Tensor (3x3x3 entries) :\n %s \n" % Tensor)
```

```
Scalar (1 entry):
 tf.Tensor(2, shape=(), dtype=int32)

Vector (3 entries) :
 tf.Tensor([5 6 2], shape=(3,), dtype=int32)

Matrix (3x3 entries):
 tf.Tensor(
[[1 2 3]
 [2 3 4]
 [3 4 5]], shape=(3, 3), dtype=int32)

Tensor (3x3x3 entries) :
 tf.Tensor(
[[[ 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]]], shape=(3, 3, 3), dtype=int32)
```

tf.shape returns the shape of our data structure.

[9]:
```
Scalar.shape
```

[9]:
```
TensorShape([])
```

[10]:
```
Tensor.shape
```

[10]:
```
TensorShape([3, 3, 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:

[11]:
```
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]])

@tf.function
def add():
    add_1_operation = tf.add(Matrix_one, Matrix_two)
    return add_1_operation



print ("Defined using tensorflow function :")
add_1_operation = add()
print(add_1_operation)
print ("Defined using normal expressions :")
add_2_operation = Matrix_one + Matrix_two
print(add_2_operation)
```

```
Defined using tensorflow function :
tf.Tensor(
[[3 4 5]
 [4 5 6]
 [5 6 7]], shape=(3, 3), dtype=int32)
Defined using normal expressions :
tf.Tensor(
[[3 4 5]
 [4 5 6]
 [5 6 7]], shape=(3, 3), dtype=int32)
```

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():

[12]:
```
Matrix_one = tf.constant([[2,3],[3,4]])
Matrix_two = tf.constant([[2,3],[3,4]])

@tf.function
def mathmul():
  return tf.matmul(Matrix_one, Matrix_two)


mul_operation = mathmul()

print ("Defined using tensorflow function :")
print(mul_operation)
```

```
Defined using tensorflow function :
tf.Tensor(
[[13 18]
 [18 25]], shape=(2, 2), dtype=int32)
```

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:

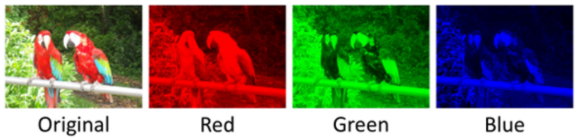


Image Source

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 persist 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. So, you can update the value of a variable through each run.

Let's first create a simple counter, by first initializing a variable *v* that will be increased one unit at a time:

[13]:
```
v = tf.Variable(0)
```

We now create a python method *increment_by_one*. This method will internally call *td.add* that takes in two arguments, the reference_variable to update, and assign it to the value_to_update it by.

[14]:
```
@tf.function
def increment_by_one(v):
        v = tf.add(v,1)
        return v
```

To update the value of the variable *v*, we simply call the *increment_by_one* method and pass the variable to it. We will invoke this method thrice. This method will increment the variable by one and print the updated value each time.

[15]:
```
for i in range(3):
    v = increment_by_one(v)
    print(v)
```

```
tf.Tensor(1, shape=(), dtype=int32)
tf.Tensor(2, shape=(), dtype=int32)
tf.Tensor(3, shape=(), dtype=int32)
```

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

[16]:
```
a = tf.constant([5])
b = tf.constant([2])
c = tf.add(a,b)
d = tf.subtract(a,b)


print ('c =: %s' % c)
------
```

```
print ('d =: %s' % d)
```

```
c =: tf.Tensor([7], shape=(1,), dtype=int32)
d =: tf.Tensor([3], shape=(1,), dtype=int32)
```

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.

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.

Thanks for completing this lesson!

Notebook created by: Romeo Kienzler , Saeed Aghabozorgi and Rafael Belo Da Silva

Updated to TF 2.X by Samaya Madhavan

References:

Change Log

Date (YYYY-MM-DD)	Version	Changed By	Change Description
2020-09-21	2.0	Srishti	Migrated Lab to Markdown and added to course repo in GitLab

---

https://www.tensorflow.org/versions/r0.9/get_started/index.html
http://jrmeyer.github.io/tutorial/2016/02/01/TensorFlow-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

---