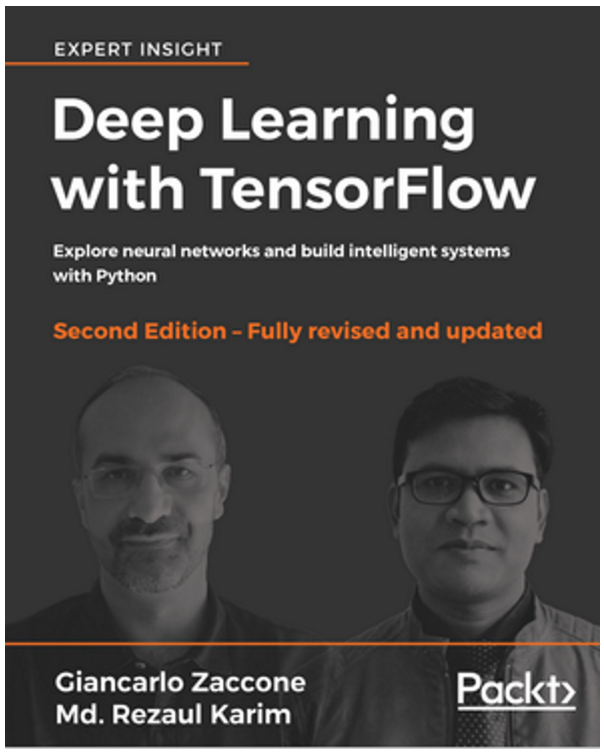
[Book] DL with TensorFlow (2nd Edition)

lunes, 1 de abril de 2019 12:44



- **Link:** https://www.packtpub.com/big-data-and-business-intelligence/deep-learning-tensorflow-second-edition
- **Repo:** https://github.com/dloperab/TensorFlow/tree/master/PacktPub/DL%2BTF

TF Overview

lunes, 1 de abril de 2019 15:18

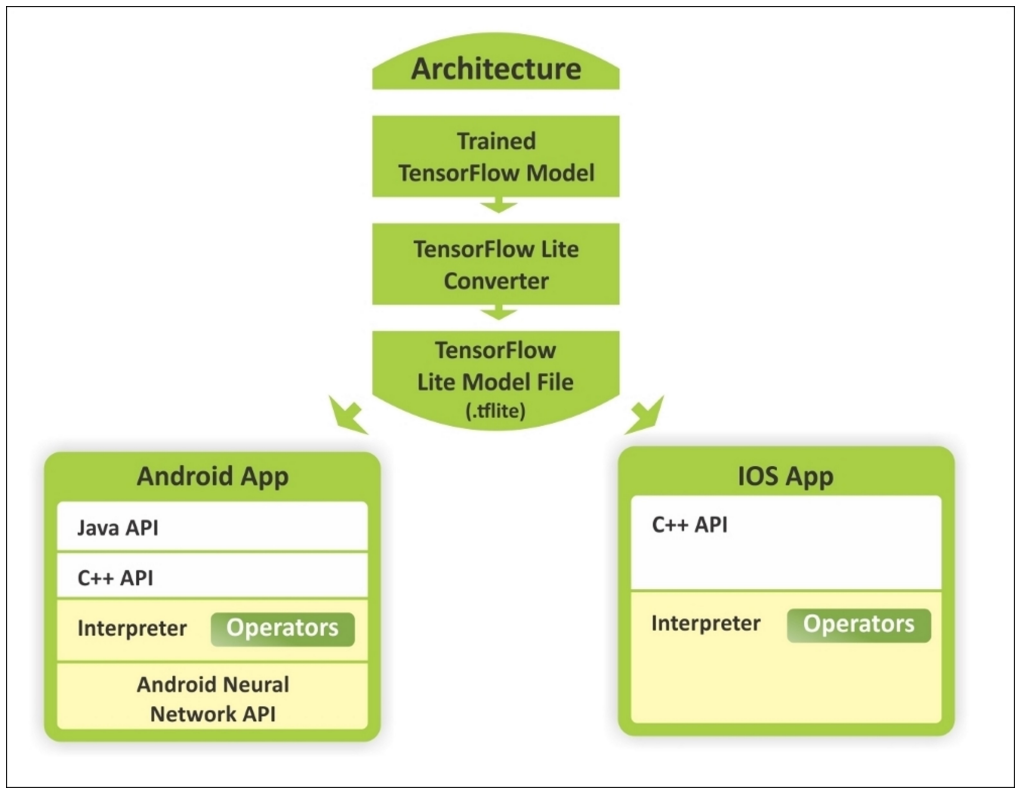
- TensorFlow is a mathematical software and an open source framework for deep learning developed by the Google Brain Team in 2011.
- Although the initial target of TensorFlow was to conduct research in ML and in Deep Neural Networks(DNNs), the system is general enough to be applicable to a wide variety of classical machine learning algorithm such as Support Vector Machine (SVM), logistic regression, decision trees, and random forest.

A general overview of TensorFlow:

- TensorFlow is an open source framework from Google for scientific and numerical computation using data flow graphs that stand for TensorFlow's execution model.
- The data flow graphs used in TensorFlow help ML experts to perform more advanced and intensive training on their data to develop DL and predictive analytics models.
- As the name implies, TensorFlow includes operations that are performed by neural networks on multidimensional data arrays, that is, flow of tensors.
- Nodes in a flow graph correspond to mathematical operations, that is, addition, multiplication, matrix factorization, and so on; whereas, edges correspond to tensors that ensure communication between edges and nodes – that is, data flow and control flow.
- You can perform numerical computations on a CPU. However, with TensorFlow, it is also possible to distribute the training among multiple devices on the same system, especially if you have more than one GPU on your system that can share the computational load.
- Deploying a predictive or general-purpose model using TensorFlow is straightforward. Once you have constructed your neural network model after the required feature engineering, you can simply perform the training interactively and use the TensorBoard to visualize your TensorFlow graph, plot quantitative metrics about the execution of your graph, and show additional data like images that pass through it.
- If TensorFlow can access GPU devices, it will automatically distribute computations to multiple devices via a greedy process. Therefore, no special configuration is needed to utilize the cores of the CPU.
- Nevertheless, TensorFlow also allows the program to specify which operations will be on which device via name scope placement.
- Finally, after evaluating the model, you deploy it by feeding some test data to it.
- The main features offered by the latest release of TensorFlow are as follows:
    - **Faster computing:** The latest release of TensorFlow is incredibly fast. For example, the Inception-v3 model runs 7.3 times faster on 8 GPUs, and distributed Inception-v3 runs 58 times faster on 64 GPUs. (TF v1.7).
    - **Flexibility:** TensorFlow is not just a DL library. It comes with almost everything you need for powerful mathematical operations, thanks to its functions for solving the most difficult problems.
    - **Portability:** TensorFlow runs on Windows, Linux, and Mac machines, and on mobile computing platforms (that is, Android).
    - **Easy debugging:** TensorFlow provides the TensorBoard tool, which is useful for analyzing the models you develop.
    - **Unified API:** TensorFlow offers you a very flexible architecture that enables you to deploy computation to one or more CPUs or GPUs on a desktop, server, or mobile device with a single API.
    - **Transparent use of GPU computing:** TensorFlow now automates the management and optimization of the memory and the data used. You can now use your machine for large-scale and data-intensive GPU computing with NVIDIA's cuDNN and CUDA toolkits.
    - **Easy use:** TensorFlow is for everyone. It is not only suitable for students, researchers, DL practitioners, but also for professionals who work in the industries.
    - **Production-ready at scale:** TensorFlow has evolved into a neural network for machine translation at production scale. TensorFlow promises Python API stability, making it easier to choose new features without worrying too much about breaking your existing code.
    - **Extensibility:** TensorFlow is a relatively new technology, and it's still in active development.
    - **Support:** There is a large community of developers and users working together to make TensorFlow a better product, both by providing feedback and by actively contributing to the source code.
    - **Wide adoption:** Numerous tech giants use TensorFlow to increase their business intelligence, such as ARM, Google, Intel, eBay, Qualcomm, SAM, Dropbox, DeepMind, Airbnb, and Twitter.

TensorFlow v1.6 forwards:

- **Nvidia GPU support optimized:**
    - From TensorFlow v1.5, prebuilt binaries are now built against CUDA 9.0 and cuDNN 7.
    - However, from v1.6's release, TensorFlow prebuilt binaries use AVX instructions, which may break TensorFlow on older CPUs.
    - Nevertheless, since v1.5, an added support for CUDA on NVIDIA Tegra devices has been available.
- **Introducing TensorFlow Lite:**
    - TensorFlow Lite is TensorFlow's lightweight solution for mobile and embedded devices.
    - It enables low-latency inference of on-device machine learning models with a small binary size and fast performance supporting hardware acceleration.
    - TensorFlow Lite uses many techniques for achieving low latency like optimizing the kernels for specific mobile apps, pre-fused activations, quantized kernels that allow smaller and faster (fixed-point math) models, and in the future, leverage-specialized machine learning hardware to get the best possible performance for a particular model on a particular device.
    - A conceptual view on how to use trained model on Android and iOS devices using TensorFlow:



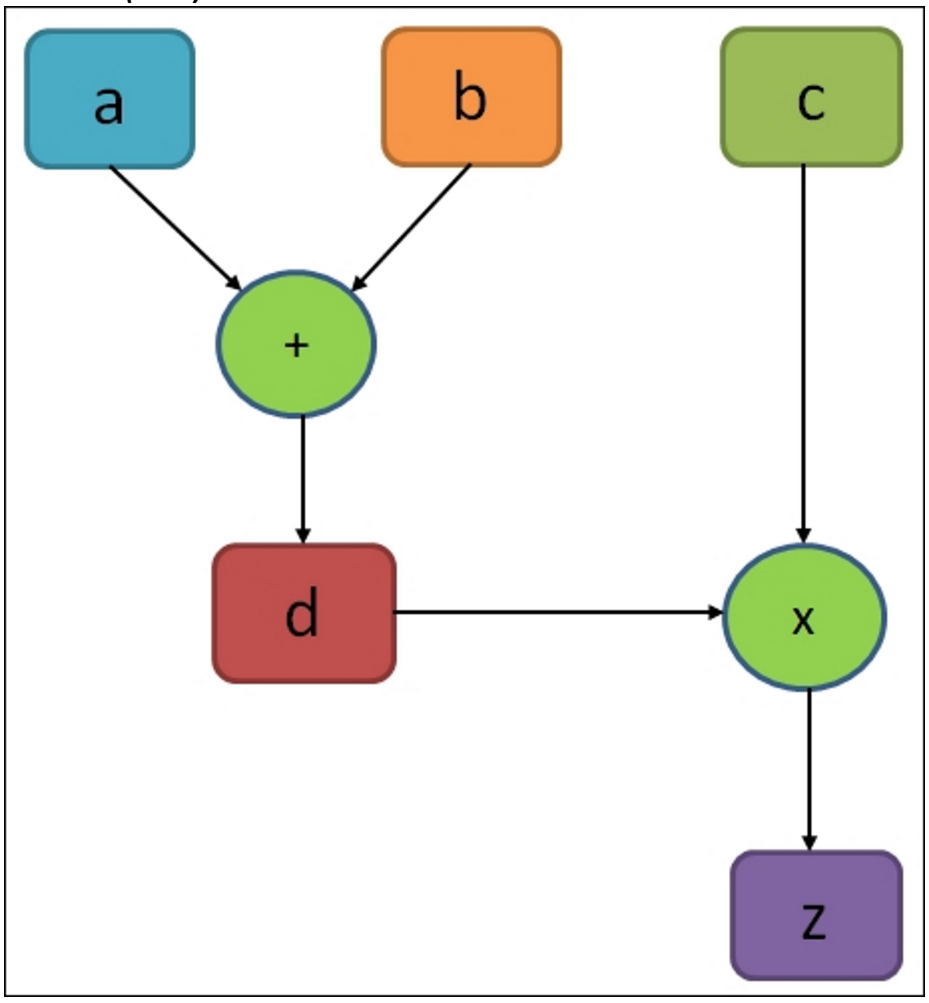
- Machine learning is changing the computing paradigm, and we see an emerging trend of new use cases on mobile and embedded
- devices. Consumer expectations are also trending toward natural, human-like interactions with their devices, driven by the camera and voice interaction models.
- Therefore, the user's expectations are no longer limited to the computer, and the computational power of mobile devices has also increased exponentially due to hardware acceleration, and frameworks such as the Android Neural Networks API and C++ API for iOS.
- As shown in the preceding figure, a pre-trained model can be converted into a lighter version to be running as an Android or iOS app.
- Therefore, widely available smart appliances create new possibilities for on-device intelligence. These allow us to use our smartphones to perform real-time computer vision and Natural Language Processing (NLP).

- **Eager execution:**
  - Eager execution is an interface for TensorFlow that provides an imperative programming style.
  - When you enable eager execution, TensorFlow operations (defined in a program) execute immediately.
  - It is to be noted that from TensorFlow v1.7, eager execution will be moved out of contrib. This means that using *tf.enable_eager_execution()* is recommended.
- **Optimized Accelerated Linear Algebra (XLA):** Pre v1.5 XLA was unstable and had a very limited number of features. However, v1.6 has more support for XLA.

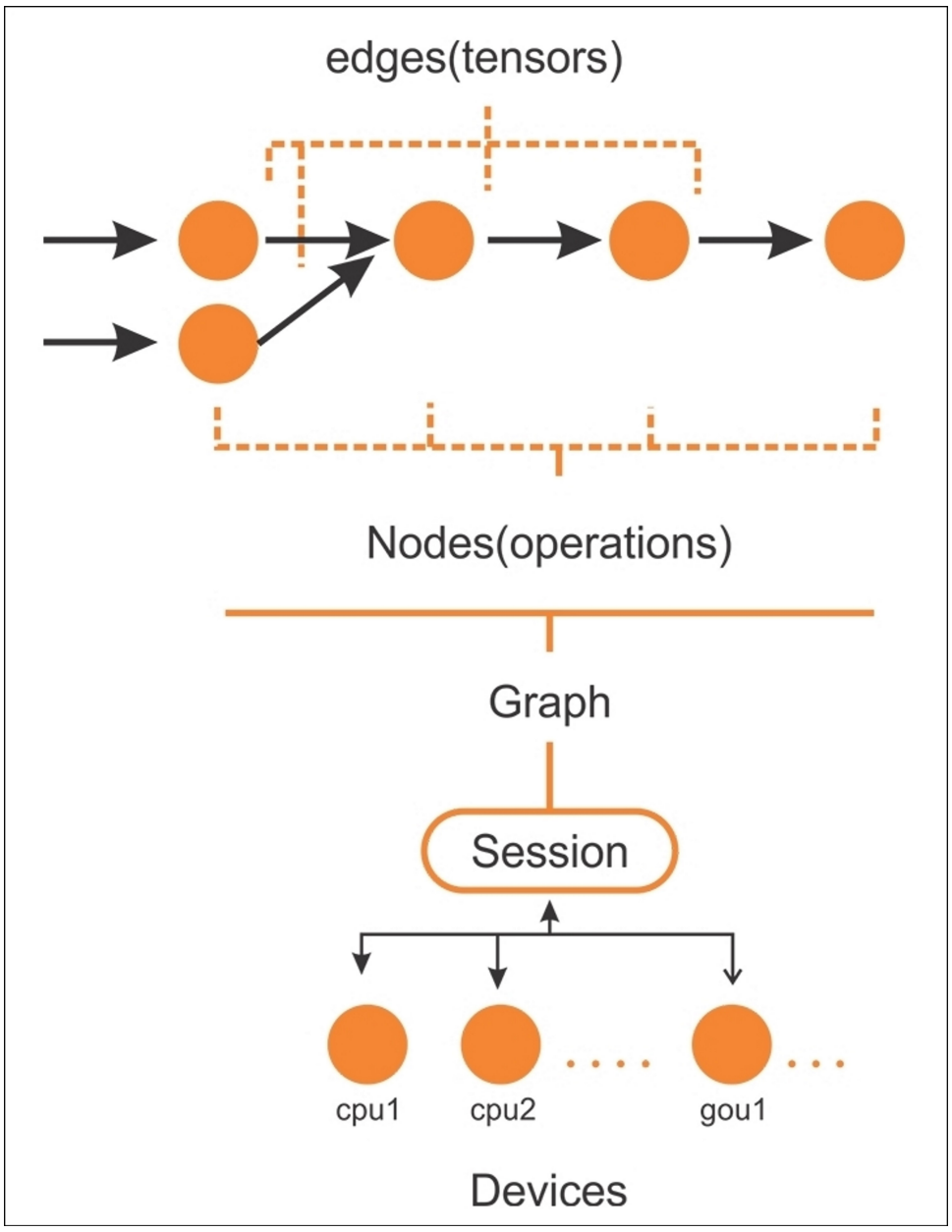
TF Computational Graph

lunes, 1 de abril de 2019 17:03

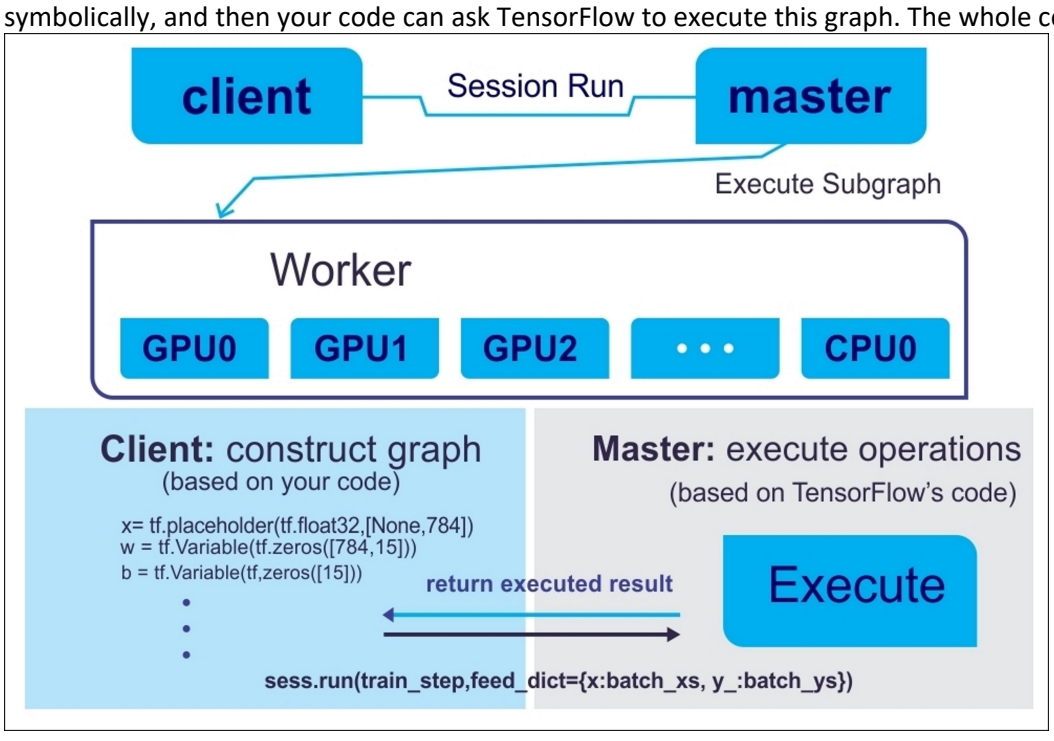
- When thinking of executing a TensorFlow program, we should be familiar with the concepts of graph creation and session execution. Basically, the first one is for building the model, and the second one is for feeding the data in and getting the results.
- Interestingly, TensorFlow does everything on the C++ engine, which means not even a little multiplication or addition is executed in Python. Python is just a wrapper.
- Fundamentally, the TensorFlow C++ engine consists of the following two things:
  - Efficient implementations of operations, such as convolution, max pool, and sigmoid for a CNN for example.
  - Derivatives of the forwarding mode operation.
- TensorFlow code consists of different operations. Even variable initialization is special in TensorFlow.
- When you are performing a complex operation with TensorFlow, such as training a linear regression, TensorFlow internally represents its computation using a data flow graph.
- The graph is called a computational graph, which is a directed graph consisting of the following:
  - A set of nodes, each one representing an operation.
  - A set of directed arcs, each one representing the data on which the operations are performed.
- TensorFlow has two types of edges:
  - **Normal:** They carry the data structures between the nodes. The output of one operation, that is, from one node, becomes the input for another operation. The edge connecting two nodes carries the values.
  - **Special:** This edge doesn't carry values, but only represents a **control dependency** between two nodes, say X and Y. It means that node Y will be executed only if the operation in X has already been executed, but before the relationship between operations on the data.
- The TensorFlow implementation defines control dependencies to enforce the order of otherwise independent operations as a way of controlling the peak memory usage.
- A computational graph is basically like a data flow graph. Next Figure shows a computational graph for a simple computation such as ***z = d x c = (a + b) x c***:



- In the preceding figure, the circles in the graph indicate the operations, while the rectangles indicate the computational graph. TensorFlow graph contains the following:
  - ***tf.Operation* objects:** These are the nodes in the graph. These are usually simply referred to as ops. An op is simply TITO (tensor-in-tensor-out). One or more tensors input and one or more tensors output.
  - ***tf.Tensor* objects:** These are the edges of the graph. These are usually simply referred to as tensors.
- Tensor objects flow between various ops in the graph. In the preceding figure, **d** is also an op. It can be a "constant" op whose output is a tensor that contains the actual value assigned to **d**.
- It is also possible to perform a **deferred execution** using TensorFlow. In a nutshell, once you have composed a highly compositional expression during the building phase of the computational graph, you can still evaluate it in the running session phase. Technically speaking, TensorFlow schedules the job and executes on time in an efficient manner.
- For example, parallel execution of independent parts of the code using the GPU is shown in the following figure:



- After a computational graph is created, TensorFlow needs to have an active session that is executed by multiple CPUs (and GPUs if available) in a distributed way. In general, you really don't need to specify whether to use a CPU or a GPU explicitly, since TensorFlow can choose which one to use.
- By default, a GPU will be picked for as many operations as possible; otherwise, CPU will be used. Nevertheless, generally, it allocates all GPU memory even if does not consume it.
- Here are the main components of a TensorFlow graph:
    - **Variables:** Used to contain values for the weights and biases between TensorFlow sessions.
    - **Tensors:** A set of values that pass between nodes to perform operations (aka. op).
    - **Placeholders:** Used to send data between the program and the TensorFlow graph.
    - **Session:** When a session is started, TensorFlow automatically calculates gradients for all the operations in the graph and uses them in a chain rule. In fact, a session is invoked when the graph is to be executed.
- Technically, the program you will be writing can be considered as a client. The client is then used to create the execution graph in C/C++ or Python symbolically, and then your code can ask TensorFlow to execute this graph. The whole concept gets clearer from the following figure:



- A computational graph helps to distribute the workload across multiple computing nodes with a CPU or GPU. This way, a neural network can be equated to a composite function where each layer (input, hidden, or output layer) can be represented as a function. To understand the operations performed on the tensors, knowing a good workaround for the TensorFlow programming model is necessary.

TF Code Structure

lunes, 1 de abril de 2019 17:19

- A TensorFlow program is generally divided into four phases when you have imported the TensorFlow library:
  - Construction of the computational graph that involves some operations on tensors.
  - Creation of a session.
  - Running a session; performed for the operations defined in the graph.
  - Computation for data collection and analysis.
- These main phases define the programming model in TensorFlow. Consider the following example, in which we want to multiply two numbers:

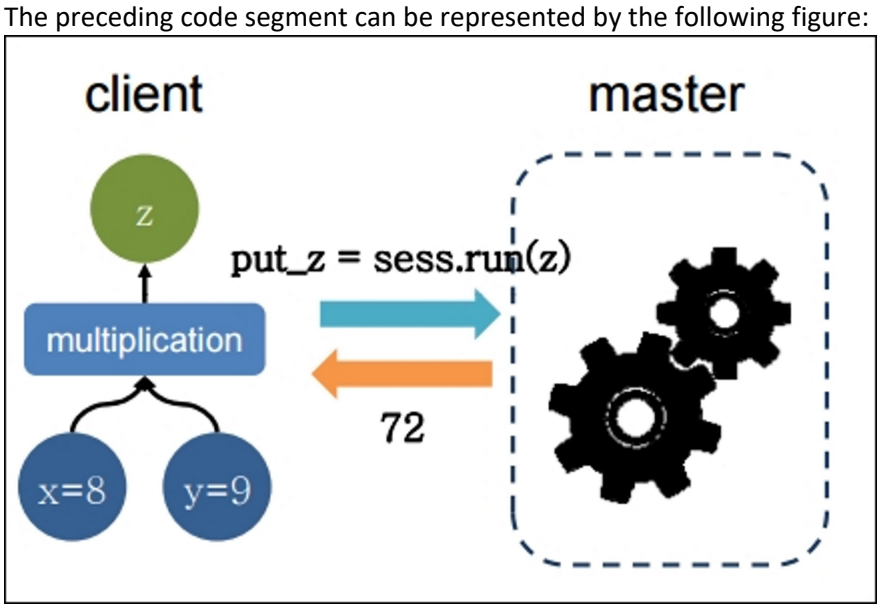
```
import tensorflow as tf # Import TensorFlow

x = tf.constant(8) # X op
y = tf.constant(9) # Y op
z = tf.multiply(x, y) # New op Z

sess = tf.Session() # Create TensorFlow session

out_z = sess.run(z) # execute Z op
sess.close() # Close TensorFlow session
print('The multiplication of x and y: %d' % out_z)# print result
```

  - The preceding code segment can be represented by the following figure:



- To make the preceding program more efficient, TensorFlow also allows exchanging data in your graph variables through **placeholders**. Now imagine the following code segment that does the same thing but more efficiently:

```
import tensorflow as tf

# Build a graph and create session passing the graph
with tf.Session() as sess:
    x = tf.placeholder(tf.float32, name="x")
    y = tf.placeholder(tf.float32, name="y")
    z = tf.multiply(x,y)

# Put the values 8,9 on the placeholders x,y and execute the graph
z_output = sess.run(z,feed_dict={x: 8, y:9})
print(z_output)
```

  - *with tf.Session() as sess*:
    - The session object (that is, sess) encapsulates the environment for the TensorFlow so that all the operation objects are executed, and Tensor objects are evaluated. We will see them in upcoming sections.
    - This object contains the computation graph, which, as we said earlier, contains the calculations to be carried out.
  - The following two lines define variables x and y, using a placeholder. Through a placeholder, you may define both an input (such as the variable x of our example) and an output variable (such as the variable y):
    -
```
x = tf.placeholder(tf.float32, name="x")
y = tf.placeholder(tf.float32, name="y")
```
    - **Placeholders** provide an interface between the elements of the graph and the computational data of the problem. They allow us to create our operations and build our computation graph without needing the data, instead of using a reference to it.
    - To define a **data** or **tensor** via the placeholder function, three arguments are required:
      - **Data type** is the type of element in the tensor to be fed.
      - **Shape** of the placeholder is the shape of the tensor to be fed (optional). If the shape is not specified, you can feed a tensor of any shape.
      - **Name** is very useful for debugging and code analysis purposes, but it is optional.
    - So, we can introduce the model that we want to compute with two arguments, the placeholder and the constant, that were previously defined. Next, we define the computational model.
  - The following statement, inside the **session**, builds the data structure of the product of x and y, and the subsequent assignment of the result of the operation to tensor z. Then it goes as follows:
    -
```
z = tf.multiply(x, y)
```
    - Since the result is already held by the placeholder z, we execute the graph through the *sess.run* statement.
  - Here, we feed two values to patch a tensor into a graph node. It temporarily replaces the output of an operation with a tensor value:
    -
```
z_output = sess.run(z,feed_dict={x: 8, y:9})
```

Eager execution with TensorFlow

- with eager execution for TensorFlow enabled, we can execute TensorFlow operations immediately as they are called from Python in an imperative way.

- With eager execution enabled, TensorFlow functions execute operations immediately and return concrete values. This is opposed to adding to a graph to be executed later in a *tf.Session* and creating symbolic references to a node in a computational graph.
- TensorFlow serves eager execution features through *tf.enable_eager_execution,* which is aliased with the following:
  - *tf.contrib.eager.enable_eager_execution*
  - *tf.enable_eager_execution*
- The tf.enable_eager_execution has the following signature:

```
tf.enable_eager_execution(
        config=None,
        device_policy=None
)
```

  - **config** is a *tf.ConfigProto* used to configure the environment in which operations are executed but this is an optional argument.
  - **device_policy** is also an optional argument used for controlling the policy on how operations requiring inputs on a specific device (for example, GPU0) handle inputs on a different device (for example, GPU1 or CPU).
- Now invoking the preceding code will enable the eager execution for the lifetime of your program. For example, the following code performs a simple multiplication operation in TensorFlow:

```
import tensorflow as tf

x = tf.placeholder(tf.float32, shape=[1, 1]) # a placeholder for variable x
y = tf.placeholder(tf.float32, shape=[1, 1]) # a placeholder for variable y
m = tf.matmul(x, y)

with tf.Session() as sess:
    print(sess.run(m, feed_dict={x: [[2.]], y: [[4.]]}))
```

  Output:

```
>>>
8.
```

  - However, with the eager execution, the overall code looks much simpler:

```
import tensorflow as tf

# Eager execution (from TF v1.7 onwards):
tf.eager.enable_eager_execution()
x = [[2.]]
y = [[4.]]
m = tf.matmul(x, y)

print(m)
```

  - Output:

```
>>>
tf.Tensor([[8.]], shape=(1, 1), dtype=float32)
```

- Can you understand what happens when the preceding code block is executed?
  - After eager execution is enabled, operations are executed as they are defined and Tensor objects hold concrete values, which can be accessed as *numpy.ndarray* through the *numpy()* method.
  - Note that eager execution cannot be enabled after TensorFlow APIs have been used to create or execute graphs. It is typically recommended to invoke this function at program startup and not in a library.

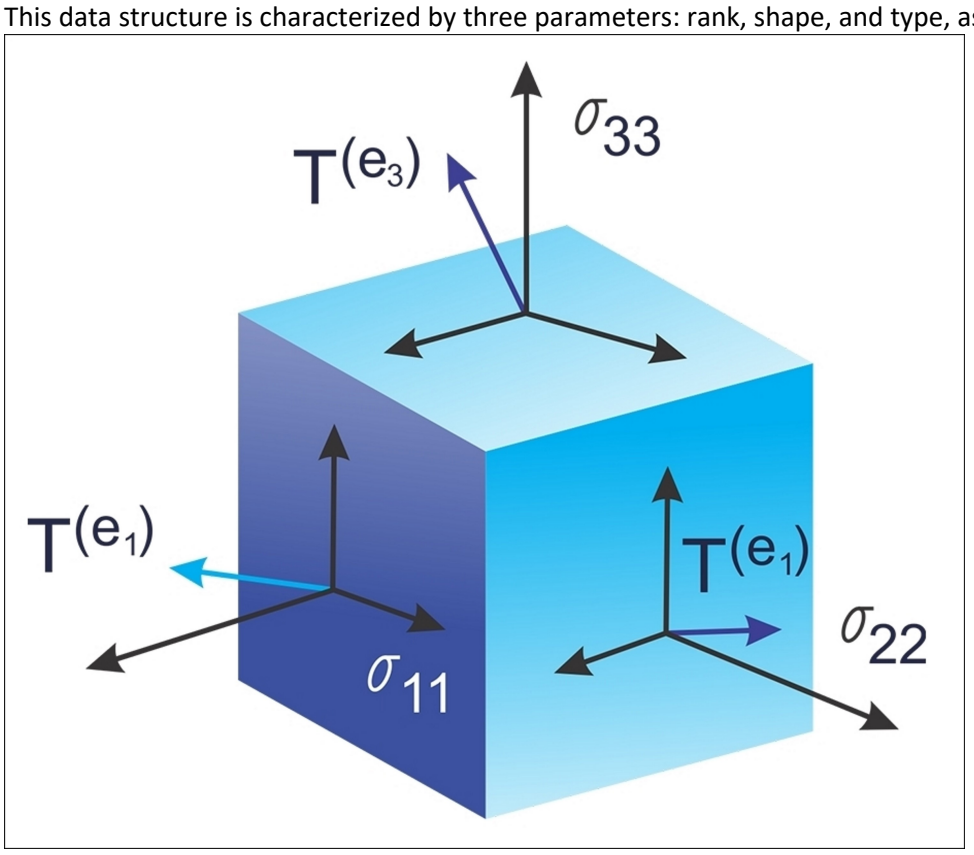
Data Model in TF

lunes, 1 de abril de 2019 18:32

- The data model in TensorFlow is represented by tensors.
- Without using complex mathematical definitions, we can say that a tensor (in TensorFlow) identifies a multidimensional numerical array.

Tensor

- "Tensors are geometric objects that describe linear relations between geometric vectors, scalars, and other tensors. Elementary examples of such relations include the dot product, the cross product, and linear maps. Geometric vectors, often used in physics and engineering applications, and scalars themselves are also tensors." (Definition of tensor from Wikipedia)
- This data structure is characterized by three parameters: rank, shape, and type, as shown in the following figure:



- A tensor can thus be thought of as the generalization of a matrix that specifies an element with an arbitrary number of indices.
- The syntax for tensors is more or less the same as nested vectors.
- Tensors just define the type of this value and the means by which this value should be calculated during the session. Therefore, they do not represent or hold any value produced by an operation.
- Some people love to compare NumPy and TensorFlow. However, in reality, TensorFlow and NumPy are quite similar in the sense that both are N-d array libraries.
- it's true that NumPy has n-dimensional array support, but it doesn't offer methods to create tensor functions and automatically compute derivatives (and it has no GPU support). The following figure is a short and one-to-one comparison of NumPy and TensorFlow:

Numpy	TensorFlow
a=np.zeros((2,2));b=np.ones((2,2))	a=tf.zeros((2,2)),b=tf.ones((2,2))
np.sum(b,axis=1)	tf.reduce_sum(a,reduction_indices=[1])
a.shape	a.get_shape()
np.reshape(a,(1,4))	tf.reshape(a,(1,4))
b*5+1	b*5+1
np.dot(a,b)	tf.matmul(a,b)
a[0,0], a[:,0], a[0,:]	a[0,0],a[:,0],a[0,:]

- Now let's see an alternative way of creating tensors before they could be fed by the TensorFlow graph:

```
>>> X = [[2.0, 4.0],
        [6.0, 8.0]] # X is a list of lists
>>> Y = np.array([[2.0, 4.0],
                  [6.0, 6.0]], dtype=np.float32)#Y is a Numpy array
>>> Z = tf.constant([[2.0, 4.0],
                     [6.0, 8.0]]) # Z is a tensor
```

  - Here, X is a list, Y is an n-dimensional array from the NumPy library, and Z is a TensorFlow tensor object. Now let's see their types:

```
>>> print(type(X))
>>> print(type(Y))
>>> print(type(Z))

#Output
<class 'list'>
<class 'numpy.ndarray'>
<class 'tensorflow.python.framework.ops.Tensor'>
```

- However, a more convenient function that we're formally dealing with tensors as opposed to the other types is *tf.convert_to_tensor()* function as follows:

```
t1 = tf.convert_to_tensor(X, dtype=tf.float32)
t2 = tf.convert_to_tensor(Z, dtype=tf.float32)
```

- Now let's see their types using the following code:

```
>>> print(type(t1))
>>> print(type(t2))

#Output:
<class 'tensorflow.python.framework.ops.Tensor'>
<class 'tensorflow.python.framework.ops.Tensor'>
```

Rank and shape

- A unit of dimensionality called rank describes each tensor.
- It identifies the number of dimensions of the tensor.
- For this reason, a rank is known as order or n–dimensions of a tensor.
- A rank zero tensor is a **scalar**, a rank one tensor is a **vector**, and a rank two tensor is a **matrix**.
- The following code defines a TensorFlow scalar, vector, matrix, and cube_matrix. In the next example, we will show how rank works:

```
import tensorflow as tf
scalar = tf.constant(100)
vector = tf.constant([1,2,3,4,5])
matrix = tf.constant([[1,2,3],[4,5,6]])

cube_matrix = tf.constant([[[1],[2],[3]],[[4],[5],[6]],[[7],[8],[9]]])

print(scalar.get_shape())
print(vector.get_shape())
print(matrix.get_shape())
print(cube_matrix.get_shape())
```

- The results are printed here:

```
>>>
()
(5,)
(2, 3)
(3, 3, 1)
>>>
```

- The shape of a tensor is the number of rows and columns it has. Now we will see how to relate the shape of a tensor to its rank:

```
>>scalar.get_shape()
TensorShape([])

>>vector.get_shape()
TensorShape([Dimension(5)])

>>matrix.get_shape()
TensorShape([Dimension(2), Dimension(3)])

>>cube.get_shape()
TensorShape([Dimension(3), Dimension(3), Dimension(1)])
```

Data type

Data type	Python type	Description
`DT_FLOAT`	`tf.float32`	32-bit floating point
`DT_DOUBLE`	`tf.float64`	64-bit floating point
`DT_INT8`	`tf.int8`	8-bit signed integer
`DT_INT16`	`tf.int16`	16-bit signed integer
`DT_INT32`	`tf.int32`	32-bit signed integer
`DT_INT64`	`tf.int64`	64-bit signed integer
`DT_UINT8`	`tf.uint8`	8-bit 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-bit floating points: real and imaginary parts
`DT_COMPLEX128`	`tf.complex128`	Complex number made of two 64-bit floating points: real and imaginary parts
`DT_QINT8`	`tf.qint8`	8-bit signed integer used in quantized Ops
`DT_QINT32`	`tf.qint32`	32-bit signed integer used in quantized Ops
`DT_QUINT8`	`tf.quint8`	8-bit unsigned integer used in quantized Ops

Variables

- Variables are TensorFlow objects used to hold and update parameters.
- A variable must be initialized so that you can save and restore it to analyze your code later on.
- Variables are created by using either tf.Variable() or tf.get_variable() statements.
- Whereas tf.get_varaiable() is recommended but tf.Variable() is lower-label abstraction.

Fetches

- To fetch the output of an operation, the graph can be executed by calling run() on the session object and passing in the tensors.
- Apart from fetching a single tensor node, you can also fetch multiple tensors.

```
import tensorflow as tf
constant_A = tf.constant([100.0])
constant_B = tf.constant([300.0])
constant_C = tf.constant([3.0])

sum_ = tf.add(constant_A,constant_B)
mul_ = tf.multiply(constant_A,constant_C)

with tf.Session() as sess:
    result = sess.run([sum_,mul_])# _ means throw away afterwards
    print(result)
```

```
>>>
[array(400.],dtype=float32),array([ 300.],dtype=float32)]
```

- It should be noted that all the ops that need to be executed (that is, in order to produce tensor values) are run once (not once per requested tensor).

Feeds and placeholders

- There are four methods of getting data into a TensorFlow program:
  - **The Dataset API:** This enables you to build complex input pipelines from simple and reusable pieces of distributed filesystems and perform complex operations. Using the Dataset API is recommended if you are dealing with large amounts of data in different data formats. The Dataset API introduces two new abstractions to TensorFlow for creating a feedable dataset: *tf.contrib.data.Dataset* (by creating a source or applying transformation operations) and *tf.contrib.data.Iterator*.
  - **Feeding:** This allows us to inject data into any tensor in a computation graph.
  - **Reading from files:** This allows us to develop an input pipeline using Python's built-in mechanism for reading data from data files at the beginning of the graph.
  - **Preloaded data:** For a small dataset, we can use either constants or variables in the TensorFlow graph to hold all the data.
- Feeding using feed_dict argument is the least efficient way to feed data into a TensorFlow execution graph and should only be used for small experiments needing small dataset. It can also be used for debugging.

- a