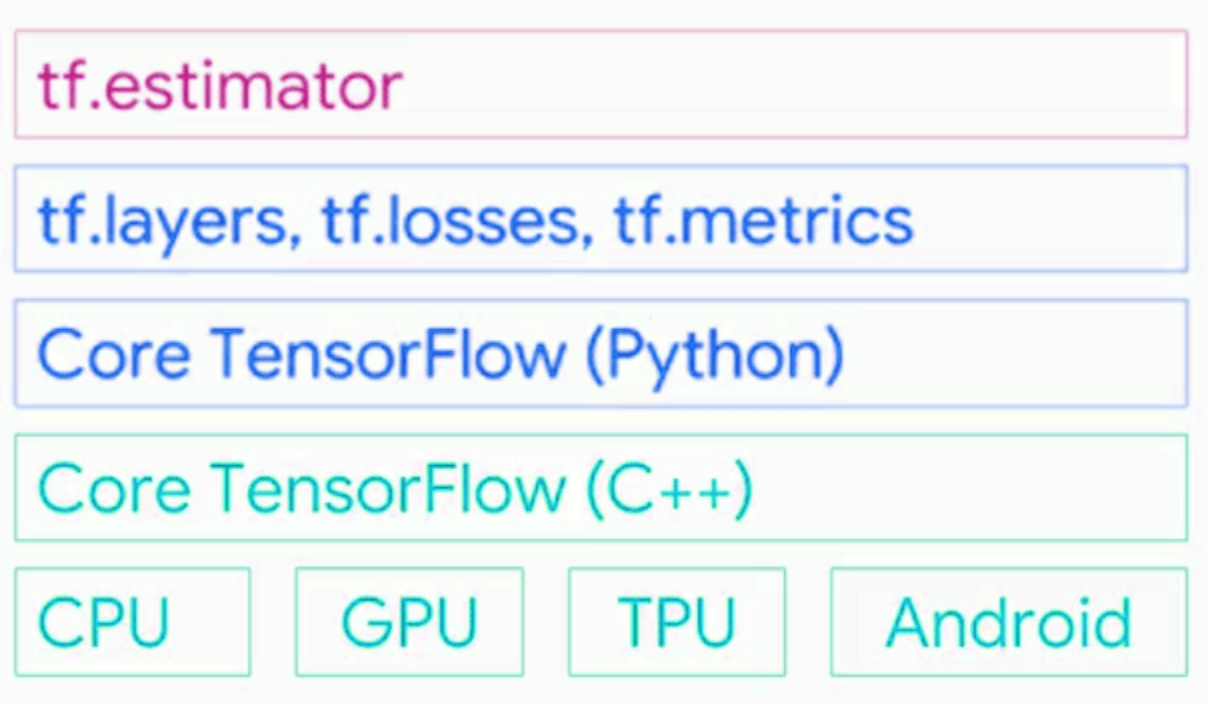
Tensorflow Basics

Tensors are objects that describe the linear relation between vectors, scalars, and other tensors. Tensors are nothing but multidimensional arrays. Tensorflow provides support to write the code according to your requirements and access to different kinds of tools. For example, we can write code in C++ and can call C++ code from python. Or we can write python code and call it by C++. tf.estimator.Estimator()



The lowest layer it supports two languages first is the Python language and the second C++ language. You can write it in any language in your comfort zone. It has a collection of different math libraries that help to create math functions easily.

It also provides support for processing like CPU, GPU, TPU and also runs on android mobiles.

Tf.layers:- tf.layers are used for method abstract so that you can customize the layers of neural networks.

```
pip install tensorflow / conda install tensorflow (Anaconda)
```

- This will install Tensorflow with GPU supported configurations.

```
pip install Tensorflow-gpu
```

Basic Data types of Tensorflow

The basic data types in the Tensorflow framework (Tensors)

Below shows each dimension of tensors.

- **Scalar –** O Dimensional Array
- **Vector –** 1 Dimensional Array
- **Matrix –** 2 Dimensional Array
- **3D Tensor –** 3 Dimensional Array
- **N – D Tensor –** N-dimensional array

Vector

An array of numbers, which is either continuous or discrete, is defined as a vector. Machine learning algorithms deal with fixed length vectors for better output generation. Machine learning algorithms deal with multidimensional data so vectors play a crucial role.

Scalar

Scalar can be defined as one-dimensional vector. Scalars are those, which include only magnitude and no direction. With scalars, we are only concerned with the magnitude. Examples of scalar include weight and height parameters of children.

Matrix

Matrix can be defined as multi-dimensional arrays, which are arranged in the format of rows and columns. The size of matrix is defined by row length and column length. Following figure shows the representation of any specified matrix.

Tensor Data Structure

Tensors are used as the basic data structures in TensorFlow language. Tensors represent the connecting edges in any flow diagram called the Data Flow Graph.

Tensors are defined as multidimensional array or list.

Tensors are identified by the following three parameters:

Rank

- Unit of dimensionality described within tensor is called rank.
- It identifies the number of dimensions of the tensor.
- A rank of a tensor can be described as the order or n-dimensions of a tensor defined.

Shape

- The number of rows and columns together define the shape of Tensor.

Type

Type describes the data type assigned to Tensor's elements.

A user needs to consider the following activities for building a Tensor:

- Build an n-dimensional array
- Convert the n-dimensional array.

Tensors: the basic

Every tensor has a name, a type, a rank and a shape.

- The **name** uniquely identifies the tensor in the computational graphs (for a complete understanding of the importance of the tensor name and how the full name of a tensor is defined, I suggest the reading of the article Understanding Tensorflow using Go).
- The **type** is the data type of the tensor, e.g.: a `tf.float32`, a `tf.int64`, a `tf.string`, …
- The **rank**, in the Tensorflow world (that's different from the mathematics world), is just the number of dimension of a tensor, e.g.: a scalar has rank 0, a vector has rank 1, …
- The **shape** is the number of elements in each dimension, e.g.: a scalar has a rank 0 and an empty shape `()`, a vector has rank 1 and a shape of `(D0)`, a matrix has rank 2 and a shape of `(D0, D1)` and so on.

Tensor's shape

To represent the shape of a Tensor in 3 different ways:

1. **Fully-known shape**: that are exactly the examples described above, in which we know the rank and the size for each dimension.
2. **Partially-known shape**: in this case, we know the rank, but we have an unknown size for one or more dimension (everyone that has trained a model

in batch is aware of this, when we define the input we just specify the feature vector shape, letting the batch dimension set to None, e.g.: (None, 28, 28, 1).

3. **Unknown shape and known rank**: in this case we know the rank of the tensor, but we don't know any of the dimension value, e.g.: (None, None, None).
4. **Unknown shape and rank**: this is the toughest case, in which we know nothing about the tensor; the rank nor the value of any dimension.

→Various Dimensions of TensorFlow

TensorFlow includes various dimensions. The dimensions are described in brief below:

One dimensional Tensor

One dimensional tensor is a normal array structure which includes one set of values of the same data type. Declaration

```
import numpy as np

tensor_1d = np.array([1.3, 1, 4.0, 23.99])

print tensor_1d
```

Two dimensional Tensors

Sequence of arrays are used for creating "two dimensional tensors".

```
import numpy as np

tensor_2d=np.array([(1,2,3,4),(4,5,6,7),(8,9,10,11),(12,13,14,15)])

print(tensor_2d)
```

```
 [[ 1 2 3 4]

 [ 4 5 6 7]
```

[8 9 10 11]

[12 13 14 15]]

Types of Tensors

Import tensorflow as tf

Constants are exactly what their names refer to. They are the fixed numbers in your equation. To define a constant, we can do this:

```
a = tf.constant(1, name='a_var')
b = tf.constant(2, name='b_bar')
```

Aside from the value 1, we can also provide a name such as "a_var" for the tensor which is separate from the Python variable name "a". It's optional.

After defining, if we print variable a, we'll have:

```
<tf.Tensor 'a_var:0' shape=() dtype=int32>
```

Variables are the model parameters to be optimized, for example, the weights and biases in your neural networks. Similarly, we can also define a variable and show its contents like this:

```
c = tf.Variable(a + b)
c
```

o/p `<tf.Variable 'Variable:0' shape=() dtype=int32_ref>`

all variables need to be initialized before use

```
init = tf.global_variables_initializer()
```

values for a and b, i.e., integers 1 and 2 are not showing up anywhere. That's an important characteristic of TensorFlow — "lazy

execution", meaning things are defined first, but not run. It's only executed when we tell it to do, which is done through the running of a session.

Session and Computational Graph

define a session and run it:

```
with tf.Session() as session:
    session.run(init)
    print(session.run(c))
```

Notice that within the session we run both the initialization of variables and the calculation of c. We defined c as the sum of a and b:

```
c = tf.Variable(a + b)
```

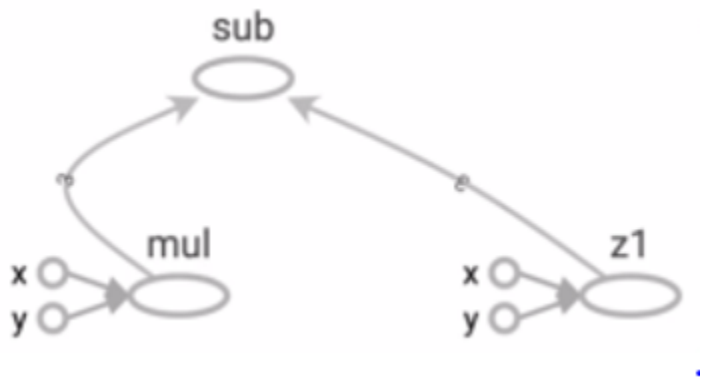
This, in TensorFlow and Deep Learning speak, is the "computational graph".

Graphs

Every line in our code written in TensorFlow is converted into an underlying chart.

Example:

```
session = tf.Session()
x = tf.constant([1, 2, 3], name = "x")
y = tf.constant([4, 5, 6], name = "y")
z1 = tf.add(x, y, name = "z1")
z2 = x * y
z3 = z2- z1
```



- **Nodes:** It represents mathematical operations.
- **Edges:** it represents the multidimensional array (Tensors) and shows how they communicate between them.

Placeholder

Another important tensor type is the **placeholder**. Its use case is to hold the place for data to be supplied.

For example, we defined a computational graph, and we have lots of training data, we can then use placeholders to indicate we'll feed these in later.

we have an equation like this:

$$y = ax^2 + bx + c$$

Instead of one single x input, we have a vector of x's. So we can use a placeholder to define x:

```
x = tf.placeholder(dtype=tf.float32)
```

We also need the coefficients. Let's use constants:

```
a = tf.constant(1, dtype=tf.float32)
b = tf.constant(-20, dtype=tf.float32)
c = tf.constant(-100, dtype=tf.float32)
```

Now let's make the computational graph and provide the input values for x:

```
y = a * (x ** 2) + b * x + c
x_feed = np.linspace(-10, 30, num=10)
```

And finally, we can run it:

```
with tf.Session() as sess:
  results = sess.run(y, feed_dict={x: x_feed})
print(results)
```

which gives us:

```
[ 200.          41.975304  -76.54321  -155.55554  -195.06174  -
195.06174  -155.55554   -76.54324    41.97534   200.
```

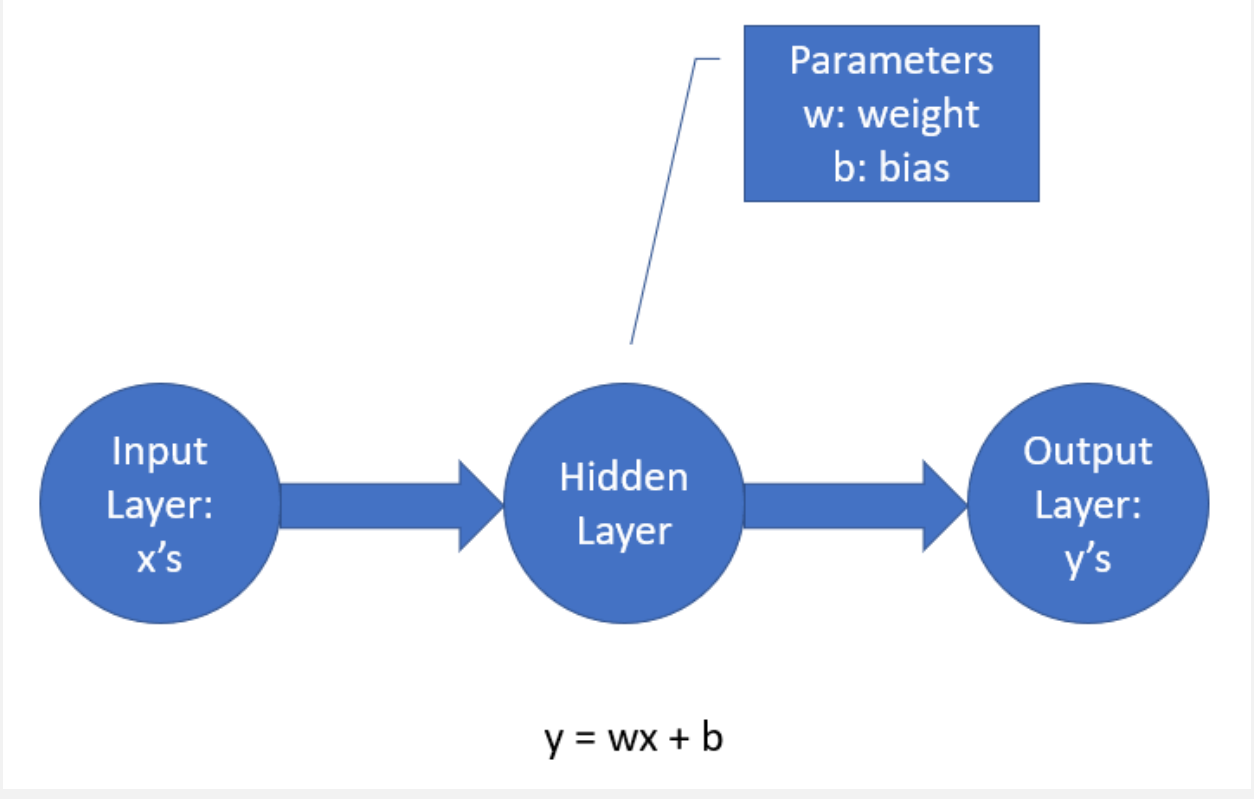
In TensorFlow, there are placeholders that are similar to variables that anyone could define, even during the runtime by using the feed_dict argument. The feed_dict argument is used in TensorFlow to feed values to these placeholders, to avoid getting an error that prompts you to feed a value for placeholders in the TensorFlow.

→Build a linear regression model, a neural network :

- we have a bunch of x, y value pairs, and we need to find the best fit line.
- First, since both x and y have values to be fed in the model, we'll define them as placeholders:

```
x = tf.placeholder(dtype=tf.float32, shape=(None, 1))
y_true = tf.placeholder(dtype=tf.float32, shape=(None, 1))
```

- The number of rows is defined as None to have the flexibility of feeding in any number of rows we want.
- Next, we need to define a model.
- In this case here, our model has just one layer with one weight and one bias.



TensorFlow allows us to define neural network layers very easily:

```
linear_model = tf.layers.Dense(
                    units=1,
                    bias_initializer=tf.constant_initializer(1))
y_pred = linear_model(x)
```

- The number of units is set to be one since we only have one node in the hidden layer.
- we need to have a loss function and set up the optimization method.
- The loss function is basically a way to measure how bad our model is when measured using the training data,
- so of course, we want it to be minimized.
- here use the gradient descent algorithm to optimize this loss function

- `optimizer=tf.train.GradientDescentOptimizer(0.01)`
  `train = optimizer.minimize(loss)`

- Then we can initialize all the variables.

- In this case here, all our variables including weight and bias are part of the layer we defined above.

- `init = tf.global_variables_initializer()`

- Lastly, we can supply the training data for the placeholders and start the training:

```
x_values = np.array([[1], [2], [3], [4]])
y_values = np.array([[0], [-1], [-2], [-3]])
with tf.Session() as sess:
  sess.run(init)
  for i in range(1000):
    _, loss_value = sess.run((train, loss),
                             feed_dict={x: x_values, y_true:
y_values})
```

and we can get the weights and make the predictions like so:

```
weights = sess.run(linear_model.weights)
bias = sess.run(linear_model.bias)
preds = sess.run(y_pred,
                 feed_dict={x: x_values})
```

which yields these predictions:

```
[[-0.00847495]  [-1.0041066 ]  [-1.9997383 ]  [-2.99537   ]]
```

-