# TensorFlow Architecture

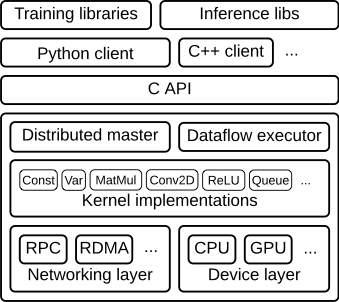
Scale: large-scale distributed training and inference.

Flexibility: flexible enough to support experimentation with new machine learning models.

## Overview

The TensorFlow runtime is a cross-platform library.

A C API separates user level code in different languages from the core runtime.



**Figure 1**

This document focuses on the following layers:

**Client**:

* + Defines the computation as a dataflow graph.
  + Initiates graph execution using a [**session**](https://www.github.com/tensorflow/tensorflow/blob/master/tensorflow/python/client/session.py).

**Distributed Master**

* + Prunes a specific subgraph from the graph, as defined by the arguments to Session.run().
  + Partitions the subgraph into multiple pieces that run in different processes and devices.
  + Distributes the graph pieces to worker services.
  + Initiates graph piece execution by worker services.

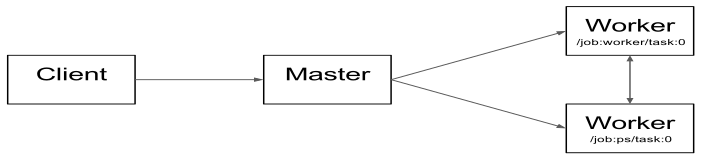
**Worker Services** (one for each task)

* + Schedule the execution of graph operations using kernel implementations appropriate to the available hardware (CPUs, GPUs, etc).
  + Send and receive operation results to and from other worker services.

**Kernel Implementations**

* + Perform the computation for individual graph operations.

Figure 2 illustrates the interaction of these components. "/job:worker/task:0" and "/job:ps/task:0" are both tasks with worker services.

"PS" stands for "parameter server": a task responsible for storing and updating the model's parameters. 

**Figure 2**

Note that the Distributed Master and Worker Service only exist in distributed TensorFlow.

## Client

Users write the client TensorFlow program that builds the computation graph.

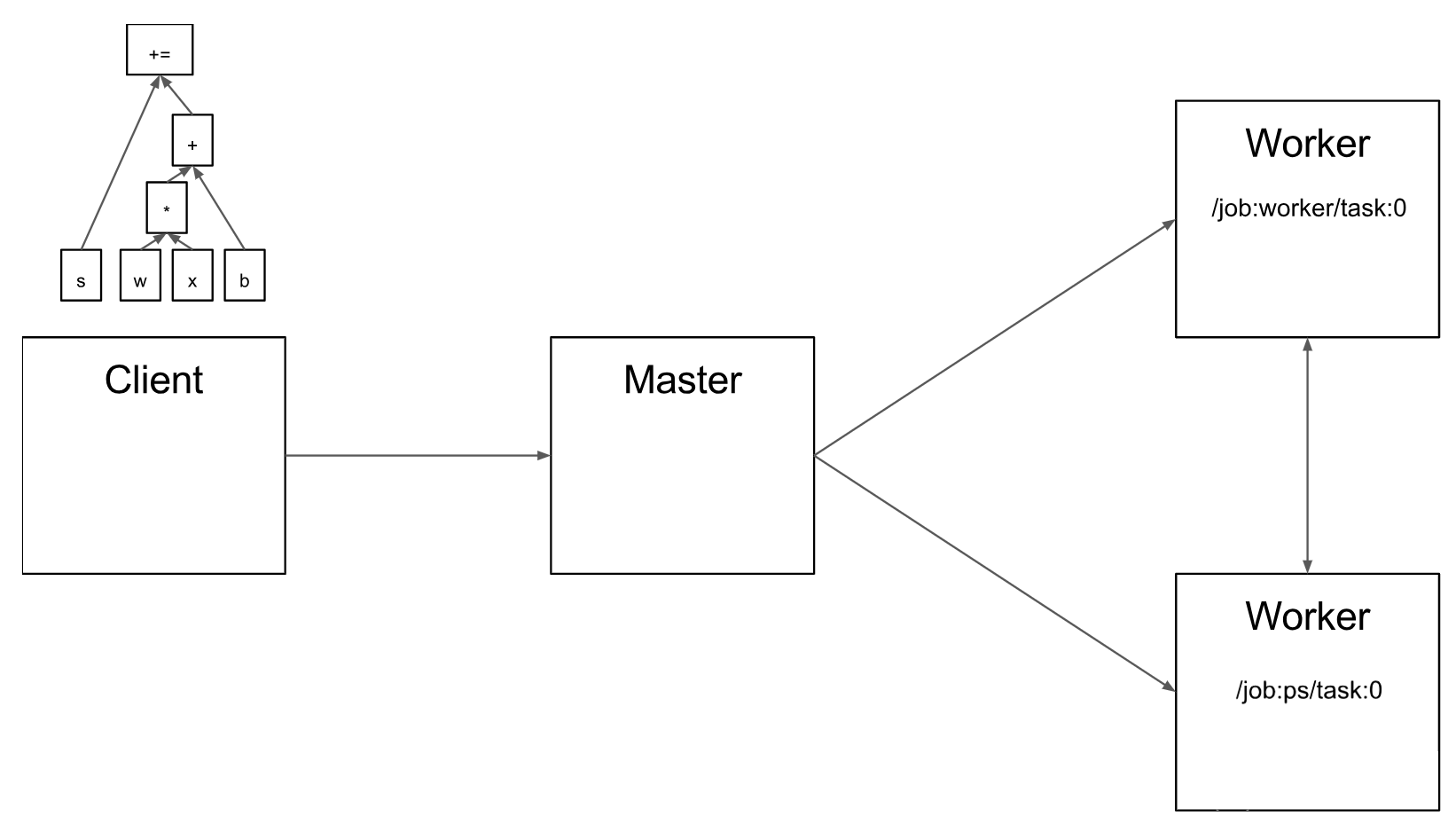
TensorFlow supports multiple client languages, and we have prioritized Python and C++, because our internal users are most familiar with these languages.

Most of the training libraries are still Python-only, but C++ does have support for efficient inference.

The client creates a session, which sends the graph definition to the distributed master as a [tf.GraphDef](https://www.tensorflow.org/api_docs/python/tf/GraphDef) protocol buffer.

When the client evaluates a node or nodes in the graph, the evaluation triggers a call to the distributed master to initiate computation.

In Figure 3, the client has built a graph that applies weights (w) to a feature vector (x), adds a bias term (b) and saves the result in a variable (s).



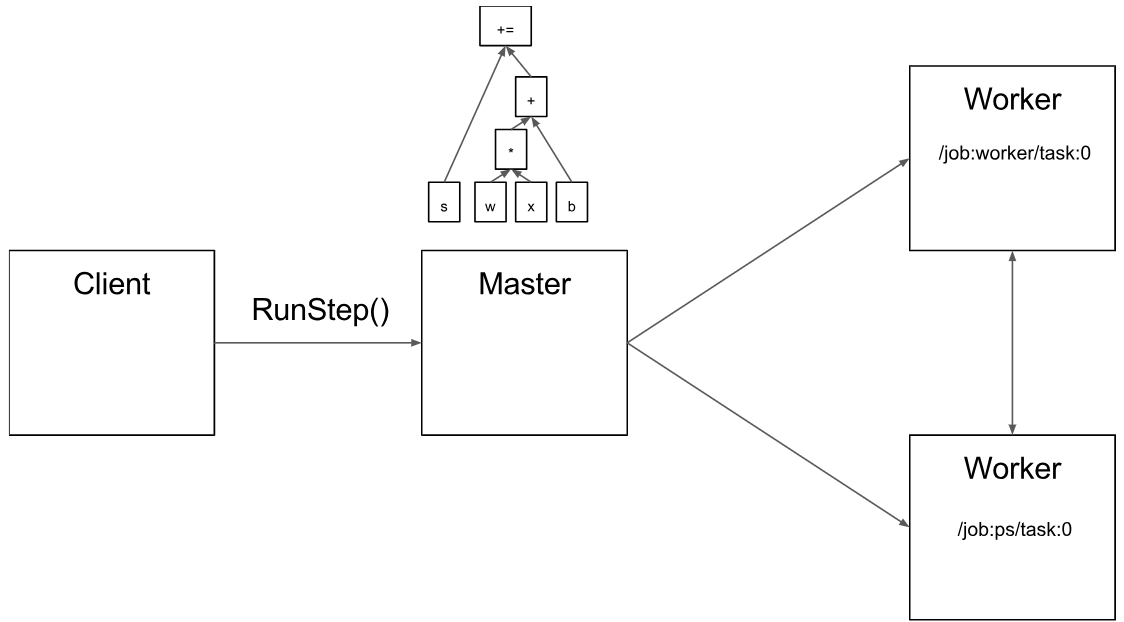
**Figure 3**

## Distributed master

The distributed master:

* prunes the graph to obtain the subgraph required to evaluate the nodes requested by the client,
* partitions the graph to obtain graph pieces for each participating device, and
* caches these pieces so that they may be re-used in subsequent steps.

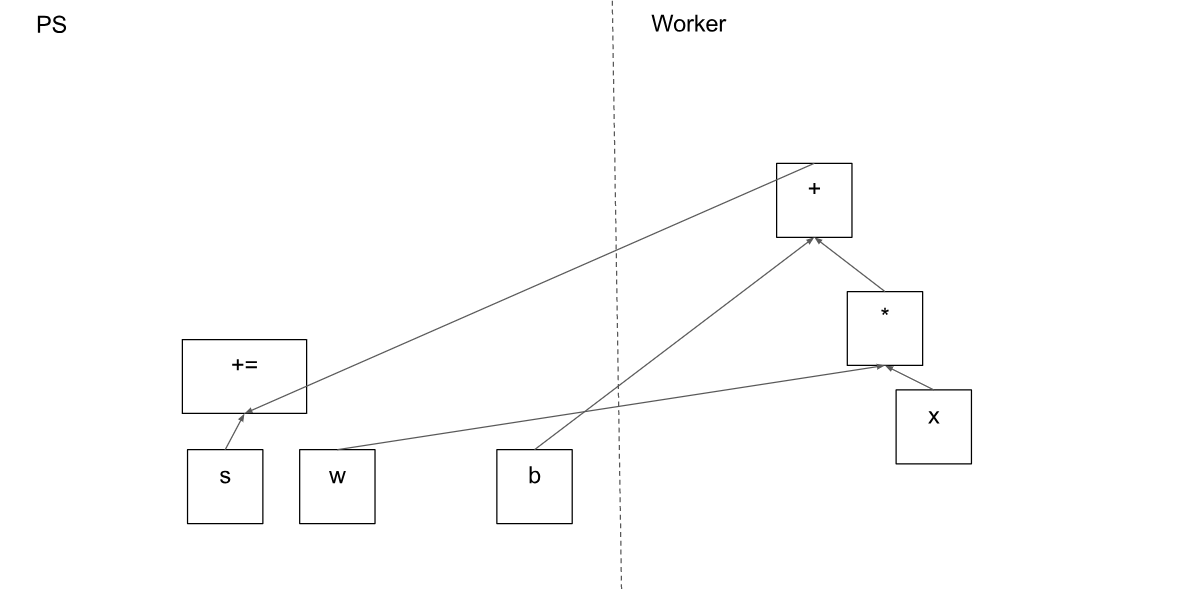
Since the master sees the overall computation for a step, it applies standard optimizations such as common subexpression elimination and constant folding.



**Figure 4**

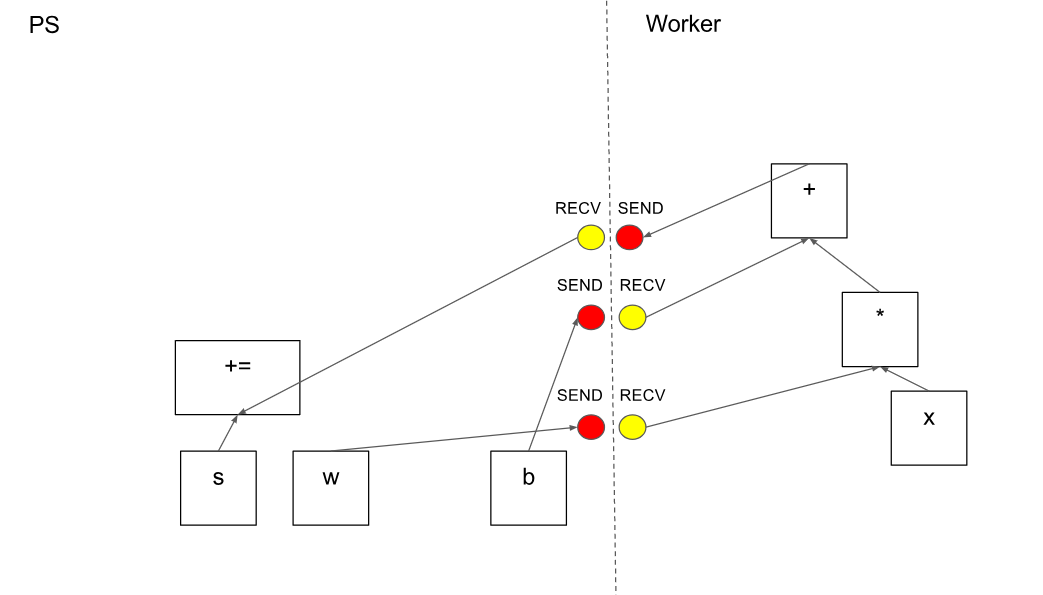
Figure 5 shows a possible partition of our example graph.

The distributed master has grouped the model parameters in order to place them together on the parameter server.



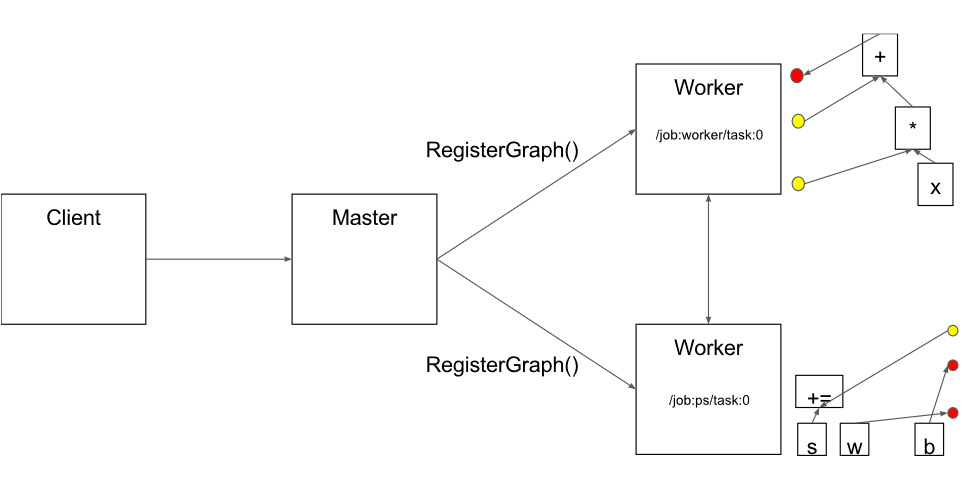
**Figure 5**

Where graph edges are cut by the partition, the distributed master inserts send and receive nodes to pass information between the distributed tasks (Figure 6).



**Figure 6**

The distributed master then ships the graph pieces to the distributed tasks.



**Figure 7**

## Worker Service

The worker service in each task:

* handles requests from the master,
* schedules the execution of the kernels for the operations that comprise a local subgraph, and
* mediates direct communication between tasks.

We optimize the worker service for running large graphs with low overhead.

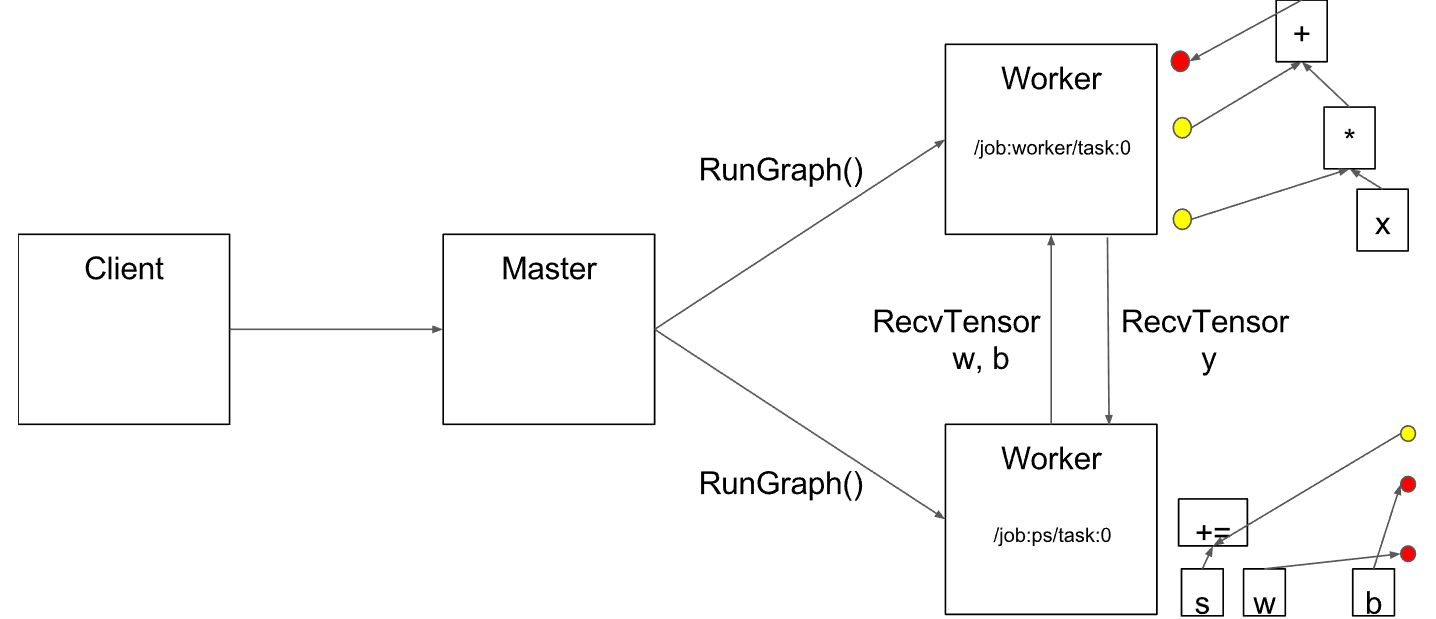
The worker service dispatches kernels to local devices and runs kernels in parallel when possible, for example by using multiple CPU cores or GPU streams.

We specialize Send and Recv operations for each pair of source and destination device types:

* Transfers between local CPU and GPU devices use the cudaMemcpyAsync() API to overlap computation and data transfer.
* Transfers between two local GPUs use peer-to-peer DMA, to avoid an expensive copy via the host CPU.

For transfers between tasks, TensorFlow uses multiple protocols, including:

* gRPC over TCP.
* RDMA over Converged Ethernet.



**Figure 8**

## Kernel Implementations

The runtime contains over 200 standard operations including mathematical, array manipulation, control flow, and state management operations.

Each of these operations can have kernel implementations optimized for a variety of devices.

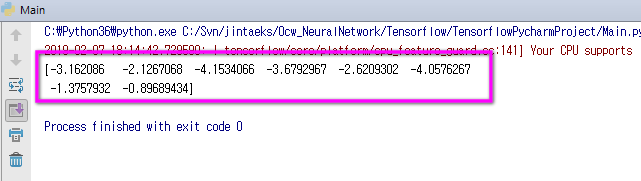
Many of the operation kernels are implemented using Eigen::Tensor, which uses C++ templates to generate efficient parallel code for multicore CPUs and GPUs;

however, we liberally use libraries like cuDNN where a more efficient kernel implementation is possible.

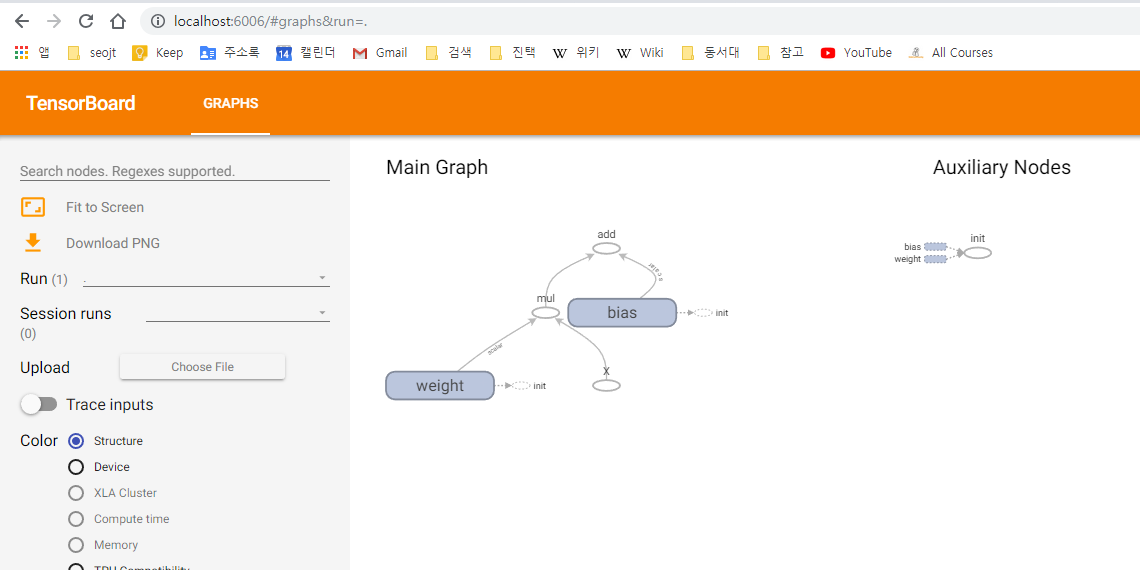
# Example

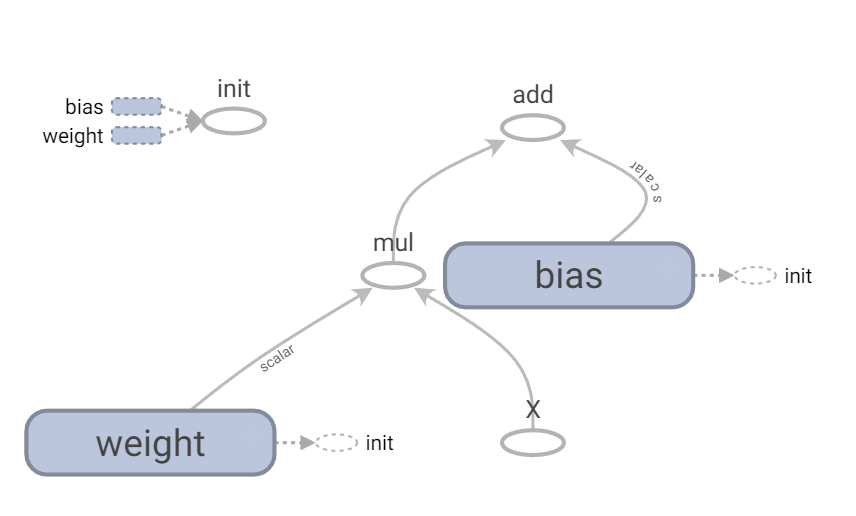
**import** tensorflow **as** tf  
**import** numpy **as** np  
  
tf.reset\_default\_graph()  
  
test\_X = np.asarray([6.83, 4.668, 8.9, 7.91, 5.7, 8.7, 3.1, 2.1])  
n\_samples = test\_X.shape[0]  
  
X = tf.placeholder(tf.float32, name=**"X"**)  
  
W = tf.Variable(np.random.randn(), name=**"weight"**)  
b = tf.Variable(np.random.randn(), name=**"bias"**)  
  
*# Construct a linear model*S = W \* X + b  
  
*# Initializing the variables*init = tf.global\_variables\_initializer()  
  
*# Launch the graph***with** tf.Session() **as** sess:  
 *# Load initialized variables in current session* sess.run(init)  
 writer = tf.summary.FileWriter(**"./graphs"**, sess.graph)  
 result = sess.run(S, feed\_dict={X: test\_X})  
 print( result )

## Output



## TensorBoard





## Variable vs. Placeholder

There's not much related between tf.Variable and tf.placeholder in my opinion. You use a Variable if you need to store state. You use a placeholder if you need to input external data.

If you are not building a model, you should still use tf.placeholder if you want to insert external data that you don't necessarily have while you're defining the graph. If you are not building a model, you still need tf.Variable if you want to store some kind of result of your computation while the graph is being run.

@