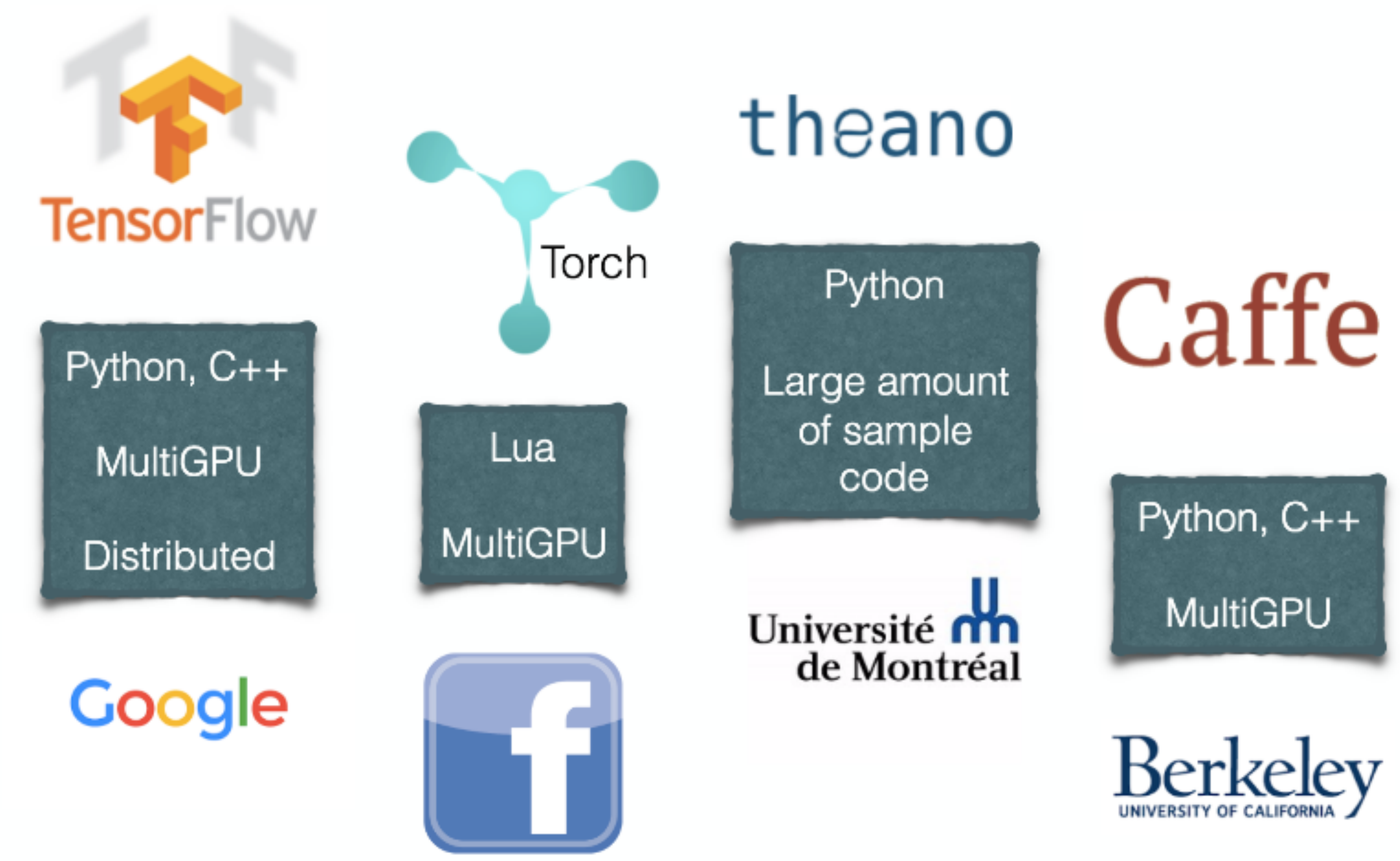
Deep Learning From Scratch

TensorFlow

Santi Seguí

http://datascience.barcelona/
www.santisegui.ml

TensorFlow: Just another library for Deep Learning?



What is **TensorFlow**

- TensorFlow was presented in 2015 by Google.
- Open Source project
- But, what does it actually do?
  - Provides primitives for defining **functions on tensors** and **automatically computing their derivatives**

What is a **Tensor**?

A typed multi-dimensional array

For example, you can represent a mini-batch of images as a 4-D array of floating point numbers with dimensions `[batch, height, width, channels]`.

Simple **Numpy** Recap

```
In [2]: # Simple Numpy Recap
        import numpy as np

        a = np.zeros((2,2)); b = np.ones((2,2))
        print np.sum(b,axis=1)

        [ 2.  2.]

In [3]: print a.shape

        (2, 2)

In [4]: print np.reshape(a,(1,4))

        [[ 0.  0.  0.  0.]]
```

Repeat in **TensorFlow**

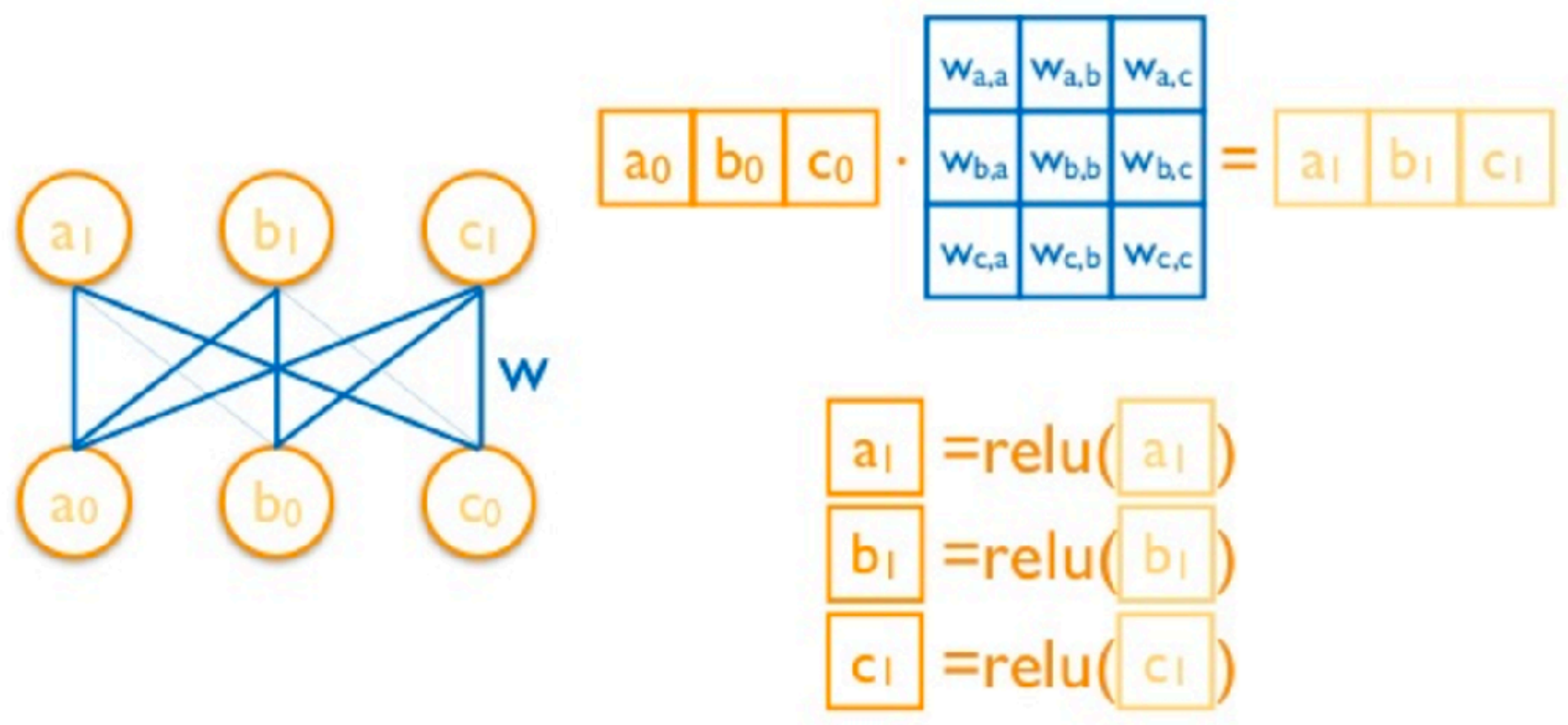
```
In [5]: import tensorflow as tf

In [6]: sess = tf.InteractiveSession()

In [7]: a = tf.zeros((2,2)); b = tf.ones((2,2))

In [8]: tf.reduce_sum(b, reduction_indices=1).eval()
Out[8]: array([ 2.,  2.], dtype=float32)

In [9]: a.get_shape()
Out[9]: TensorShape([Dimension(2), Dimension(2)])

In [10]: tf.reshape(a, (1, 4)).eval()
Out[10]: array([[ 0.,  0.,  0.,  0.]], dtype=float32)

In [11]: a = np.zeros((2,2))
         ta = tf.zeros((2,2))
         print a
         print ta

[[ 0.  0.]
 [ 0.  0.]]
Tensor("zeros_1:0", shape=(2, 2), dtype=float32)
```

What is **InteractiveSession?**

eval()**?**

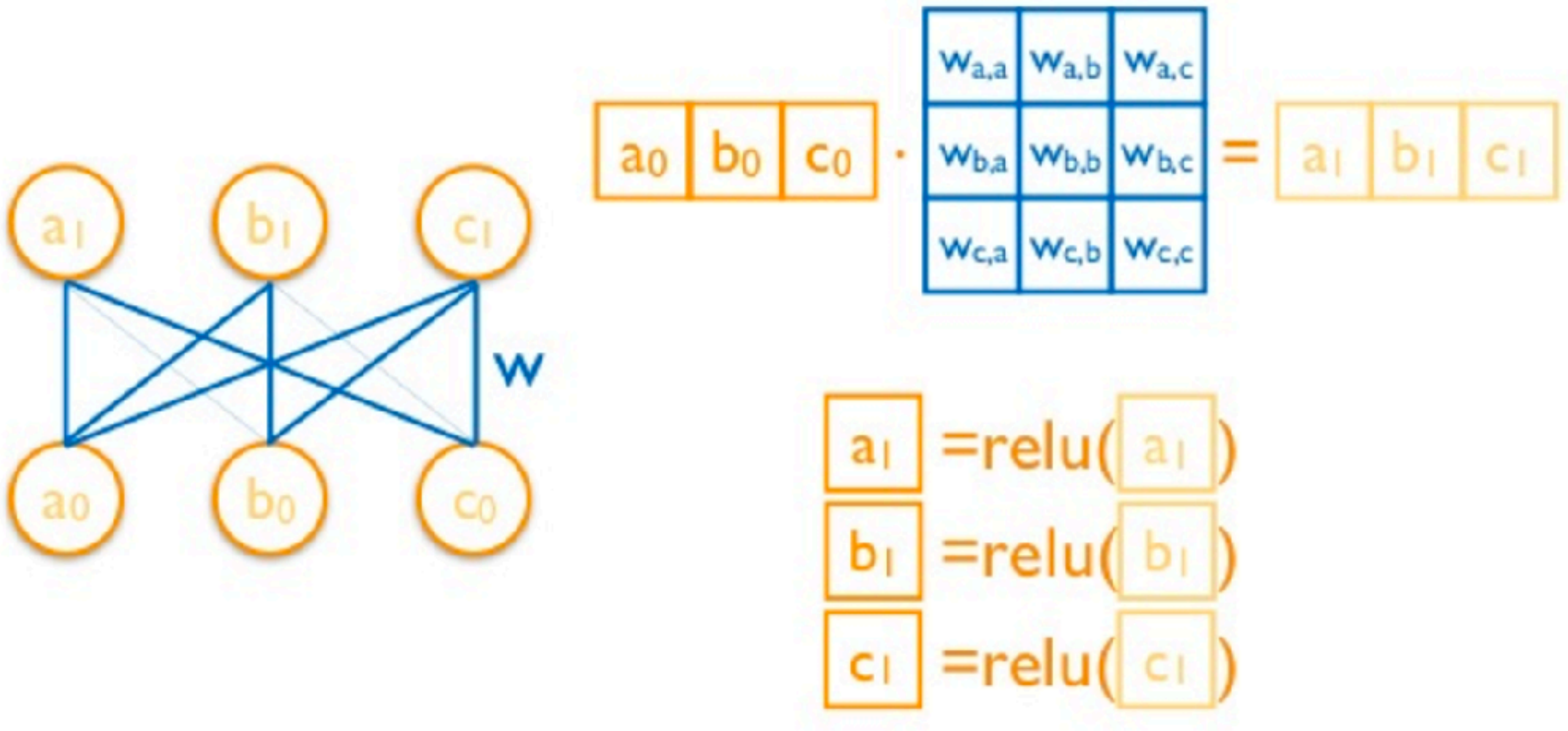
TensorFlow computations define a **computational graph** that has not a numerical value until explicit evaluation.

First program with **TensorFlow**



Matrix operations

First program with **TensorFlow**

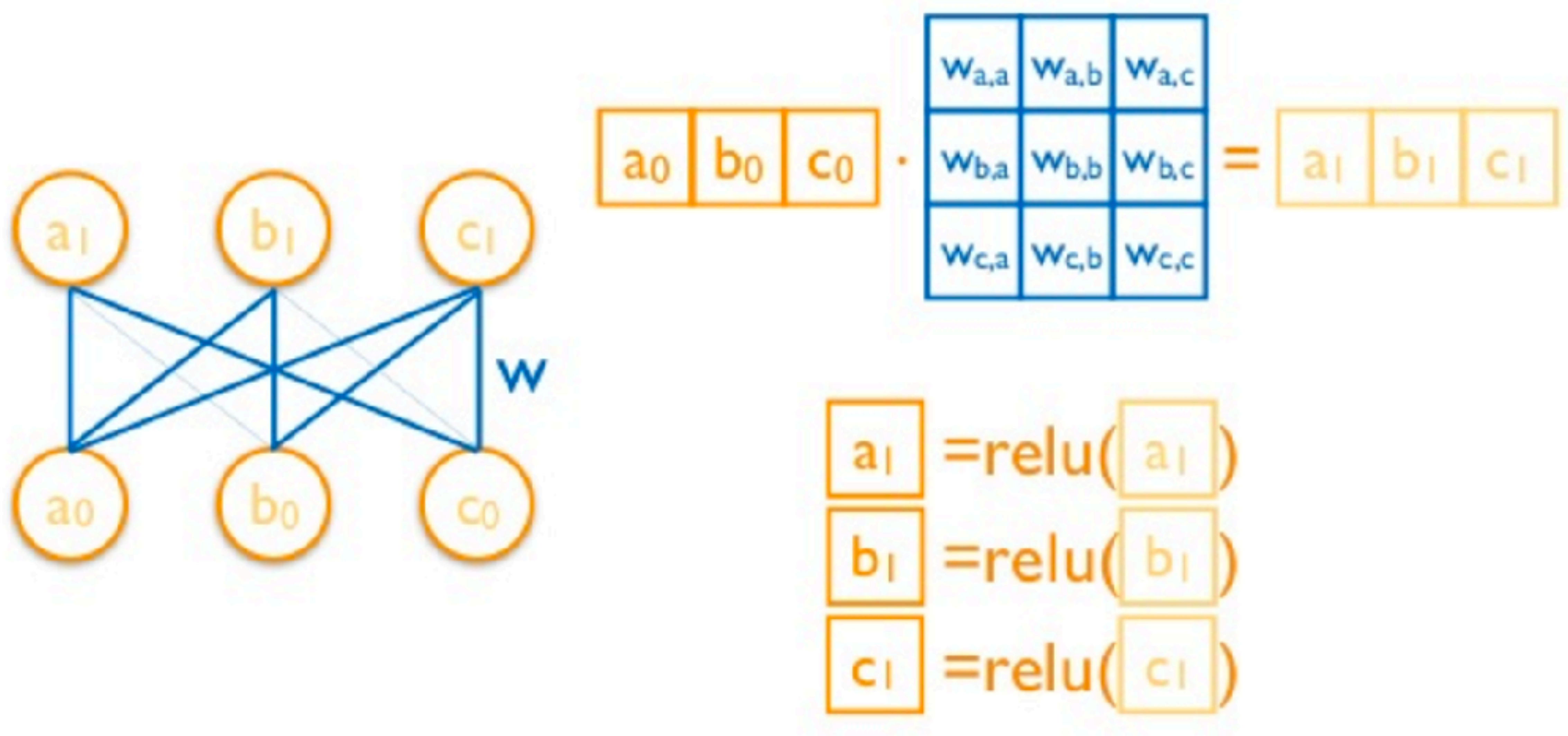



```
import tensorflow as tf


y = tf.matmul(x, w)

out = tf.nn.relu(y)
```

Matrix operations

Tensorflow operations

First program with **TensorFlow**

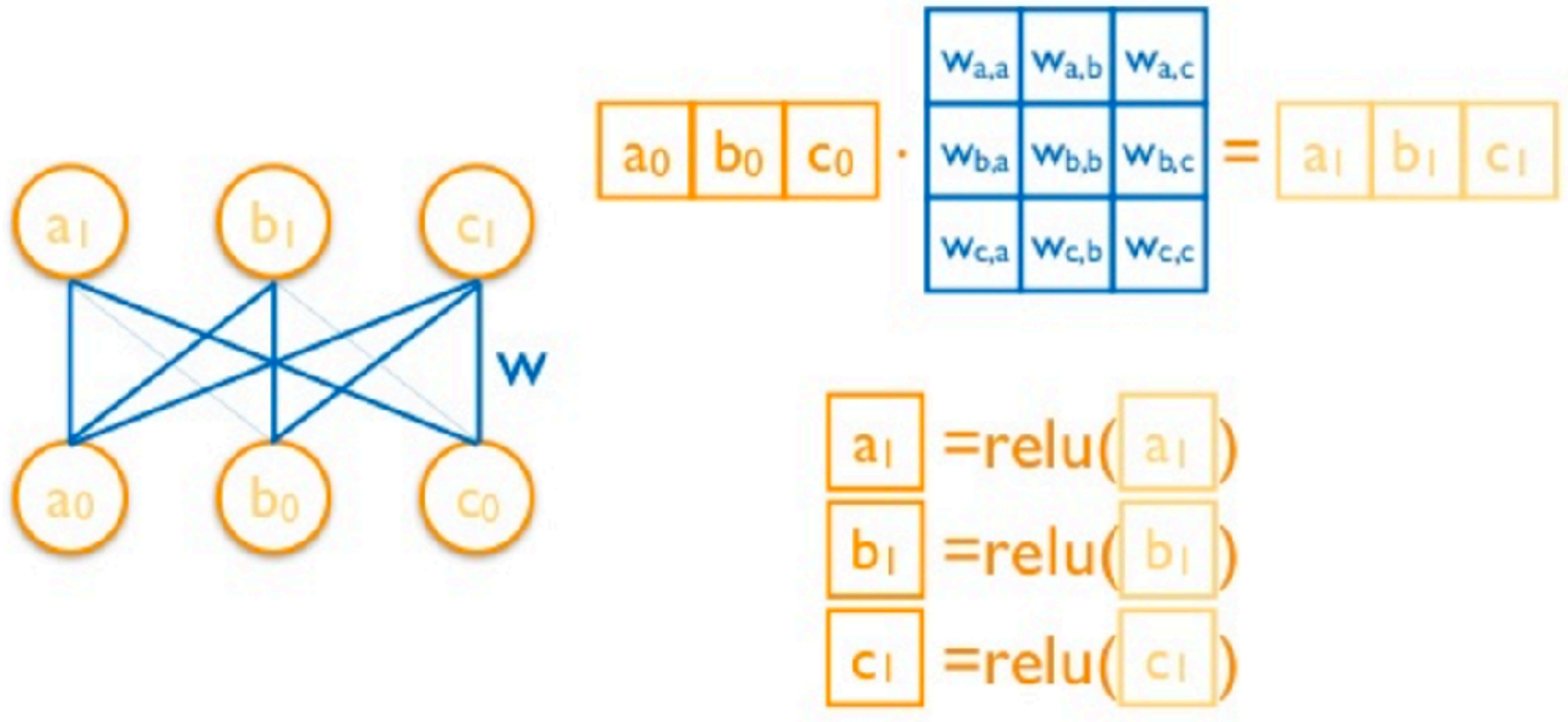



```
import tensorflow as tf

w = tf.Variable(tf.random_normal([3,3])
y = tf.matmul(x, w)

out = tf.nn.relu(y)
```

Matrix operations

Tensorflow operations

First program with **TensorFlow**

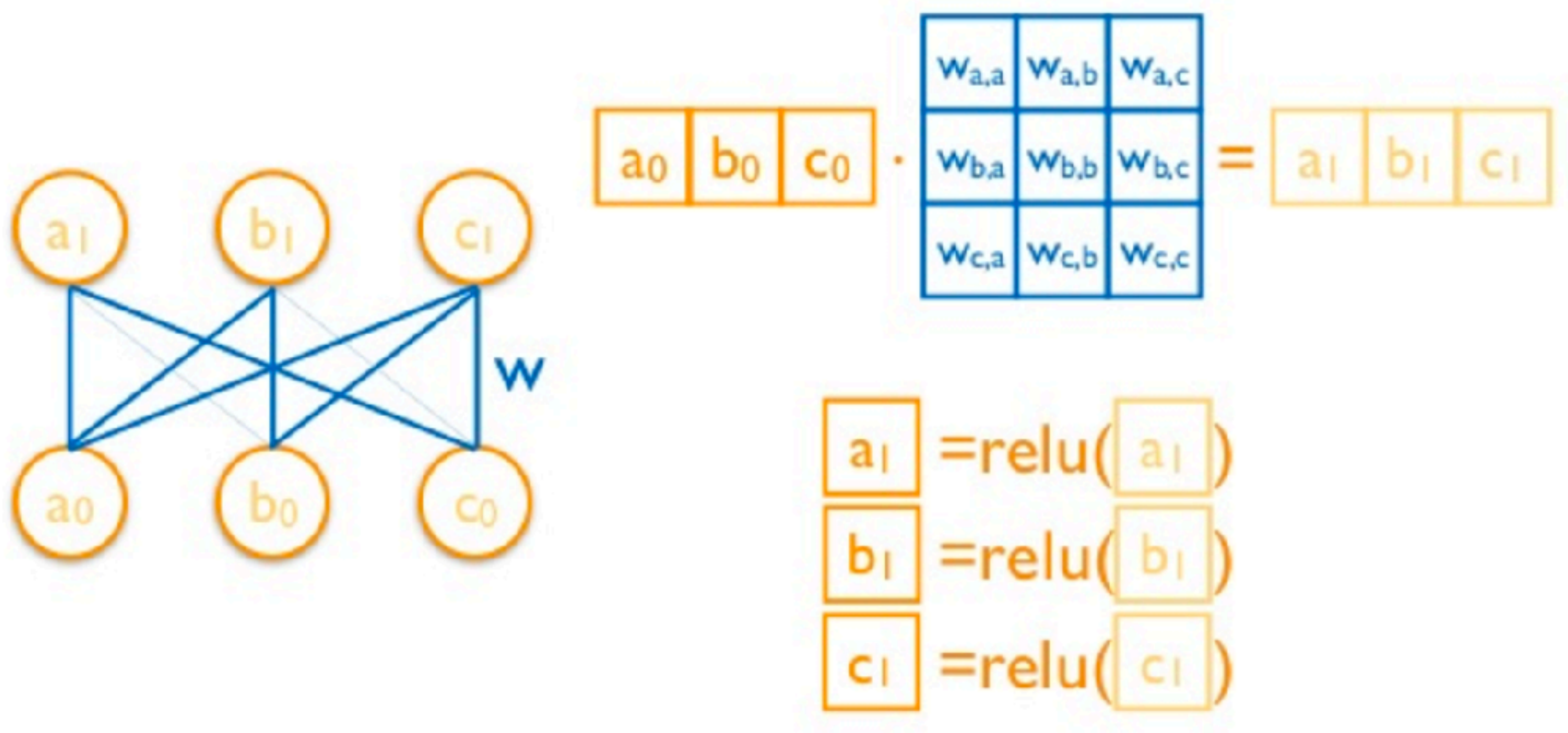



```
import tensorflow as tf
sess = tf.Session()
w = tf.Variable(tf.random_normal([3,3])
y = tf.matmul(x, w)

out = tf.nn.relu(y)
```

Matrix operations

Tensorflow operations

First program with **TensorFlow**

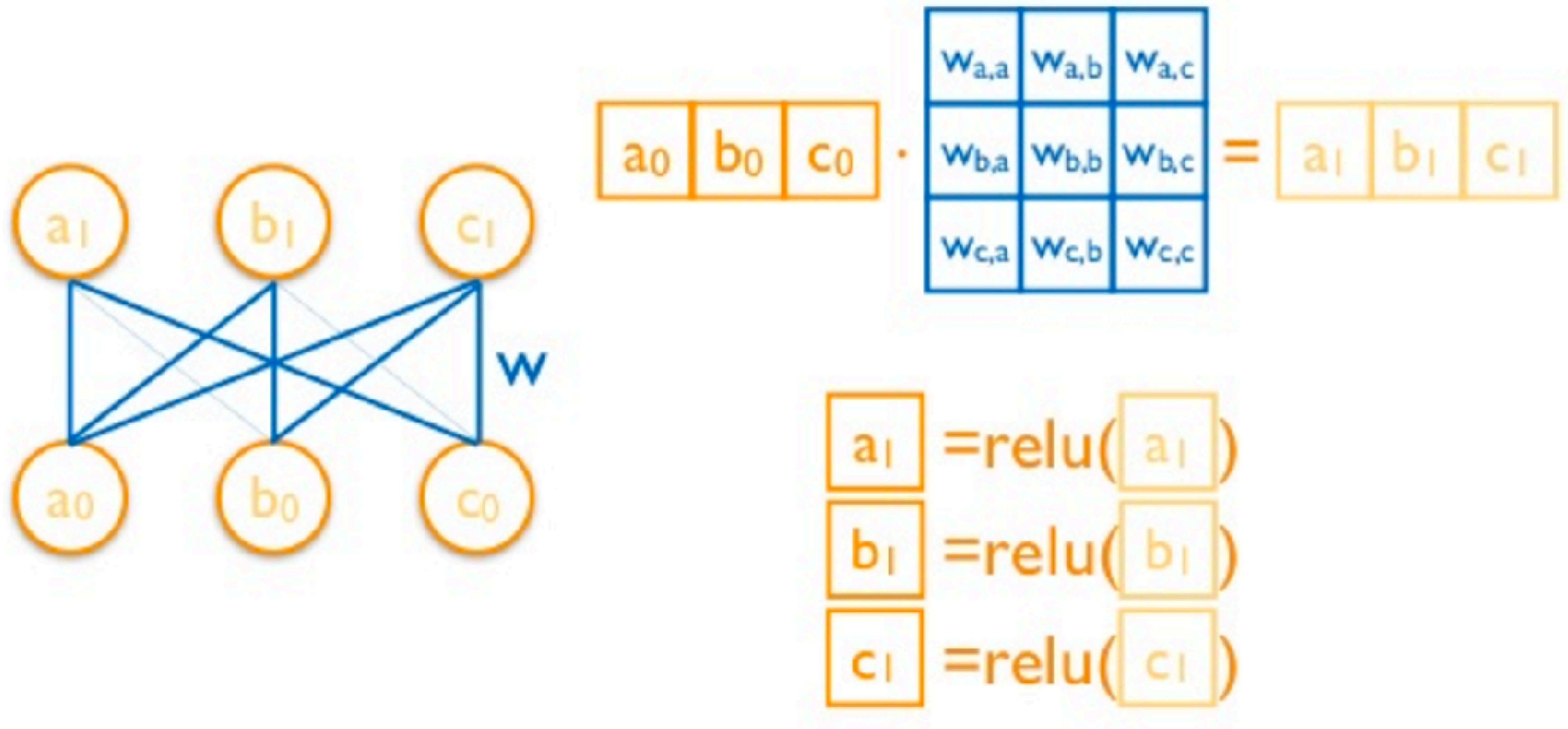



```
import tensorflow as tf
sess = tf.Session()
w = tf.Variable(tf.random_normal([3,3])
y = tf.matmul(x, w)

out = tf.nn.relu(y)

sess.run(tf.initialize_all_variables())
```

Matrix operations

Tensorflow operations

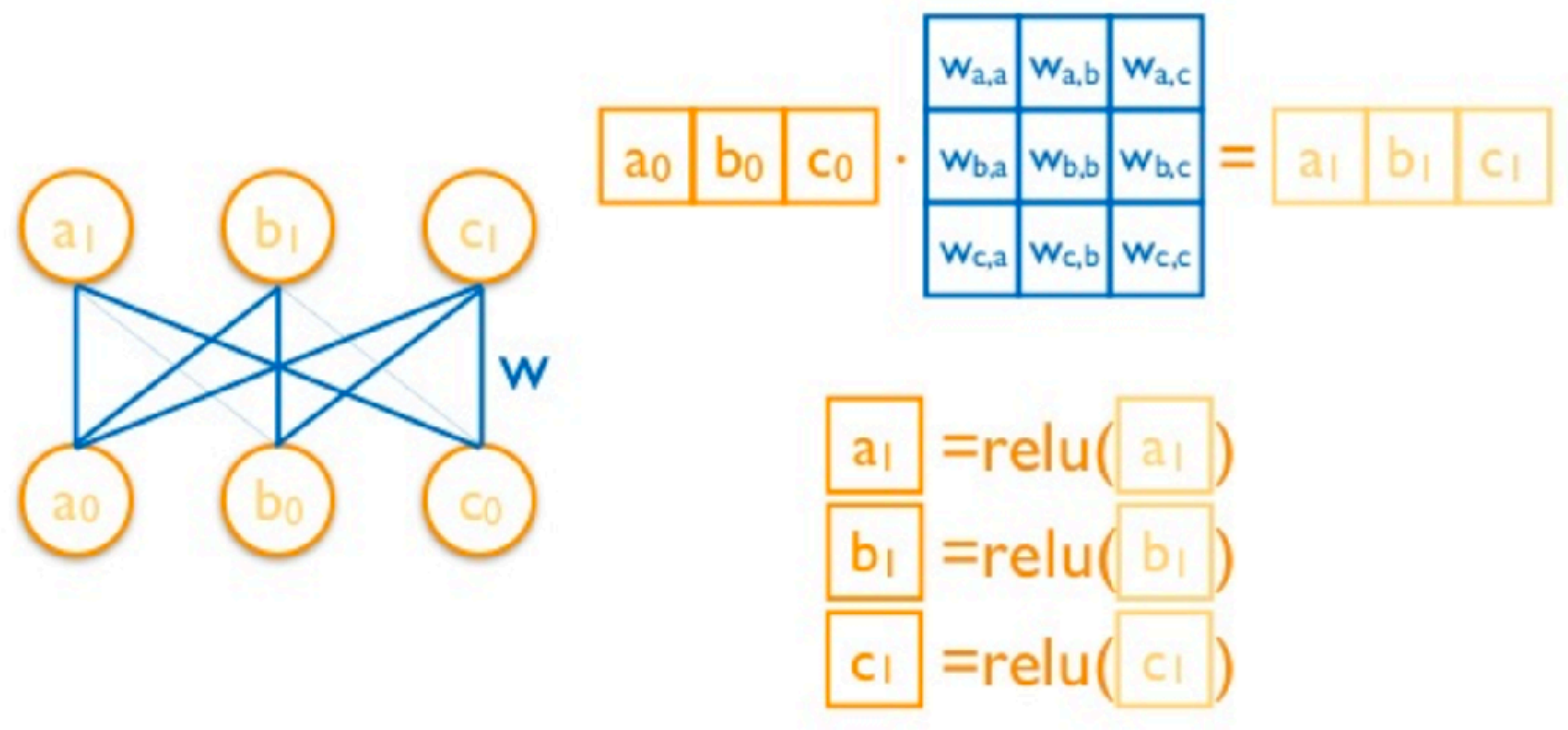
First program with **TensorFlow**




```
import tensorflow as tf
sess = tf.Session()
w = tf.Variable(tf.random_normal([3,3])
y = tf.matmul(x, w)

out = tf.nn.relu(y)

sess.run(tf.initialize_all_variables())
print sess.run(relu_out)
```

Matrix operations **Tensorflow operations**

First program with **TensorFlow**

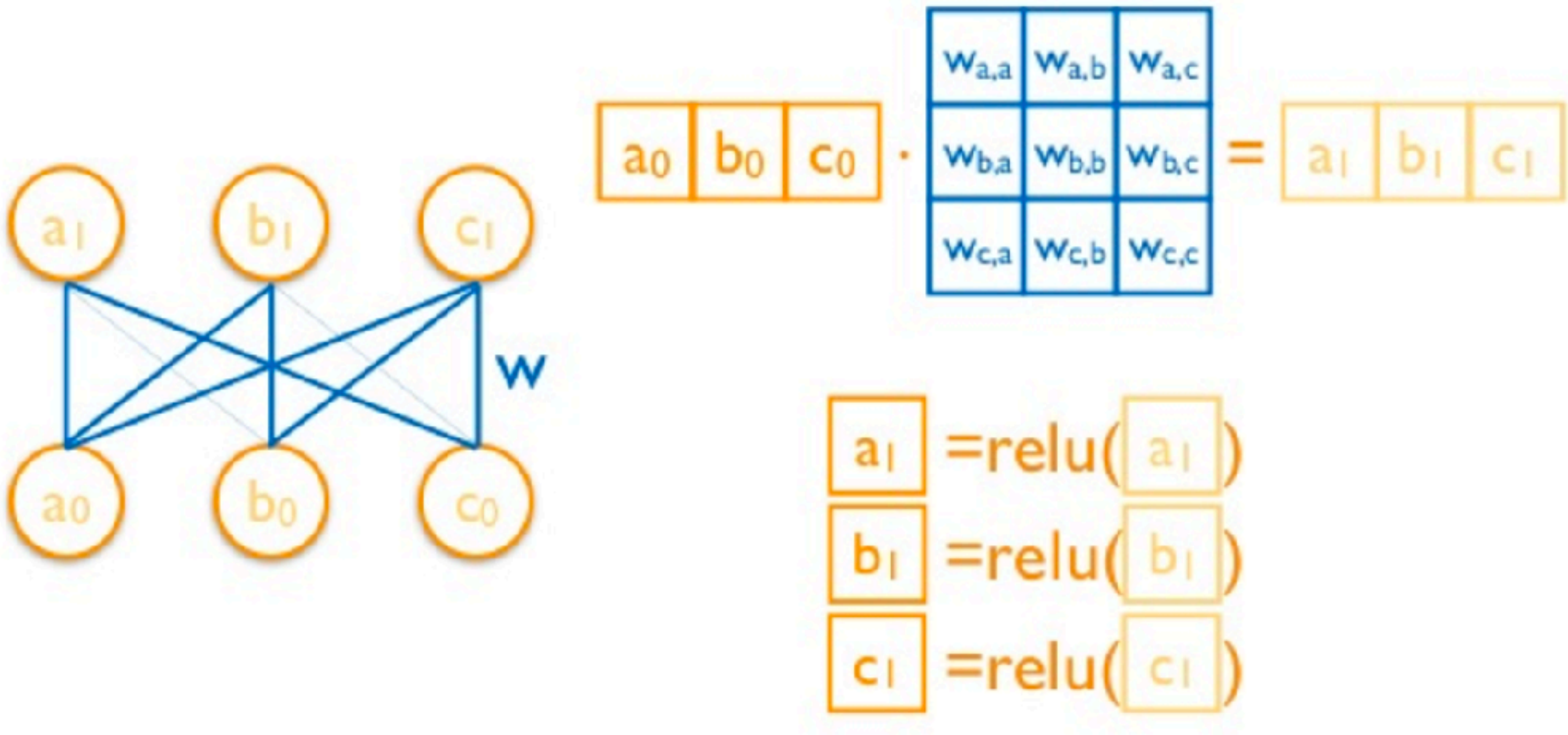



```
import tensorflow as tf
sess = tf.Session()
w = tf.Variable(tf.random_normal([3,3])
y = tf.matmul(x, w)

out = tf.nn.relu(y)

sess.run(tf.initialize_all_variables())
print sess.run(relu_out,
               feed_dict{x:np.array([[1.0,2.0,3.0]]})
```

Matrix operations

Tensorflow operations

First program with **TensorFlow**




```
import tensorflow as tf
sess = tf.Session()
w = tf.Variable(tf.random_normal([3,3])
y = tf.matmul(x, w)

out = tf.nn.relu(y)

sess.run(tf.initialize_all_variables())
print sess.run(relu_out,
               feed_dict{x:np.array([[1.0,2.0,3.0]]}
sess.close()
```

Matrix operations

Tensorflow operations

TensorFlow Session

A Session object **encapsulates the environment** in which Operation objects are executed, and Tensor objects are evaluated.
A session may **own resources**, such as variables, queues, and readers. It is important to release these resources when they are no longer required.

Different ways to use TensorFlow sessions:

1) Using the Session object:

```
a = tf.constant(5.0)
b = tf.constant(6.0)
c = a * b
sess = tf.Session()
print sess.run(c)
sess.close()
```

2) Using the context manager:

```
a = tf.constant(5.0)
b = tf.constant(6.0)
c = a * b
with tf.Session() as sess:
  print(c.eval())
```

3) Using Interactive Session:

```
sess = tf.InteractiveSession()
a = tf.constant(5.0)
b = tf.constant(6.0)
c = a * b
print(c.eval())
sess.close()
```

TensorFlow Session

```
with tf.Session() as sess:
  with tf.device("/gpu:1"):
    matrix1 = tf.constant([[3., 3.]])
    matrix2 = tf.constant([[2.],[2.]])
    product = tf.matmul(matrix1, matrix2)
    ...
```

"/cpu:0": The CPU of your machine
"/gpu:0": The GPU of your machine, if you have one.
"/gpu:1": The second GPU of your machine, etc…

TensorFlow Session

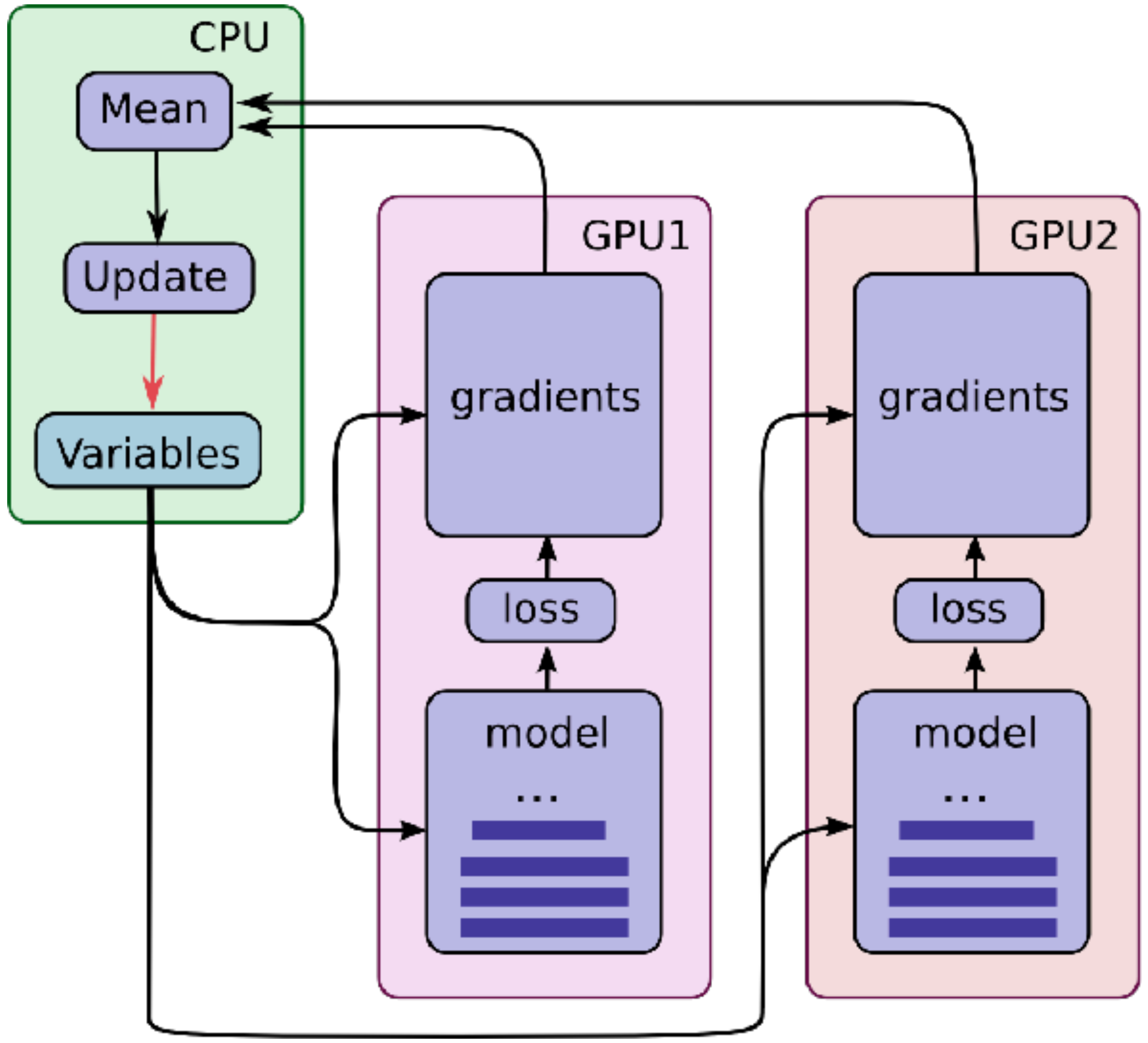
Launch the graph in a distributed session

```
with tf.Session("grpc://example.org:2222") as sess:
  # Calls to sess.run(...) will be executed on the cluster.
  ...
```

You can use "with tf.device():" statements to directly specify workers for particular parts of the graph:

```
with tf.device("/job:ps/task:0"):
  weights = tf.Variable(...)
  biases = tf.Variable(...)
```

TensorFlow Session



Strategy:

- Place an individual model replica on each GPU.
- Update model parameters synchronously by waiting for all GPUs to finish processing a batch of data.

Great Tutorial: https://www.tensorflow.org/tutorials/deep_cnn

TensorFlow Variables

- "When you train a model, you use variables to hold and update parameters. Variables are in-memory buffers containing tensors."

- "They mus … d to disk during and … d values to exercise or analyze the model."

Linear Regression $y = \mathbf{w}x + \mathbf{b}$

TensorFlow Variables

- Each **variable** defines a node in the graph, not the **result.**

TensorFlow Variables

```
In [5]: W = tf.Variable(tf.zeros([784, 10]))
        b = tf.Variable(tf.zeros([10]))
```

Why zeros?

```
In [6]: W = tf.Variable(tf.random_normal([784, 10], stddev=0.35), name = "weights")
        b = tf.Variable(tf.random_normal([10],stddev=0.35), name= "biases")
```

Variable initializers must be run explicitly before other ops in your model can be run. The easiest way to do that is to add an op that runs all the variable initializers, and run that op before using the model.

```
...
# Add an op to initialize the variables.
init_op = tf.initialize_all_variables()

# Later, when launching the model
with tf.Session() as sess:
    #Run the init operation.
    sess.run(init_op)
    ...
    # Use the model
    ...
```

Hands on!

- Let's try to do a counter!

```
state = 0
for i in range(3):
    state +=1
    print state
```

TensorFlow Code

```
state = tf.Variable(0, name = "counter")
new_value = tf.add(state, tf.constant(1))
update = tf.assign(state,new_value)

with tf.Session() as sess:
    sess.run(tf.initialize_all_variables())
    print(sess.run(state))
    for _ in range(3):
        sess.run(update)
        print(sess.run(state))
```

But, what about learning a model?

for instance, a **linear regression** model?

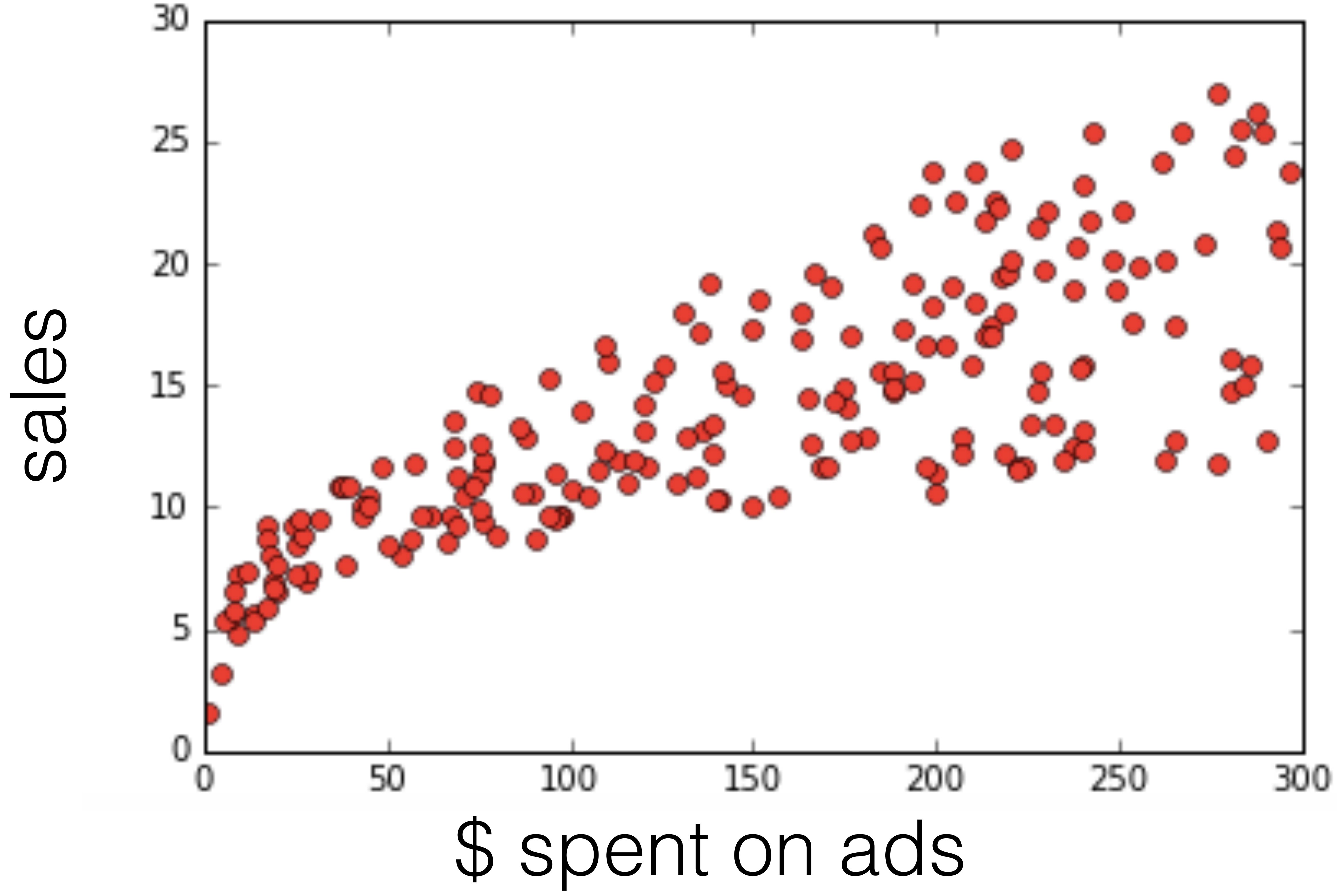
$$y = \mathbf{w}x + \mathbf{b}$$

Inference and parameters to learn
Loss function
Optimizer method
Data set

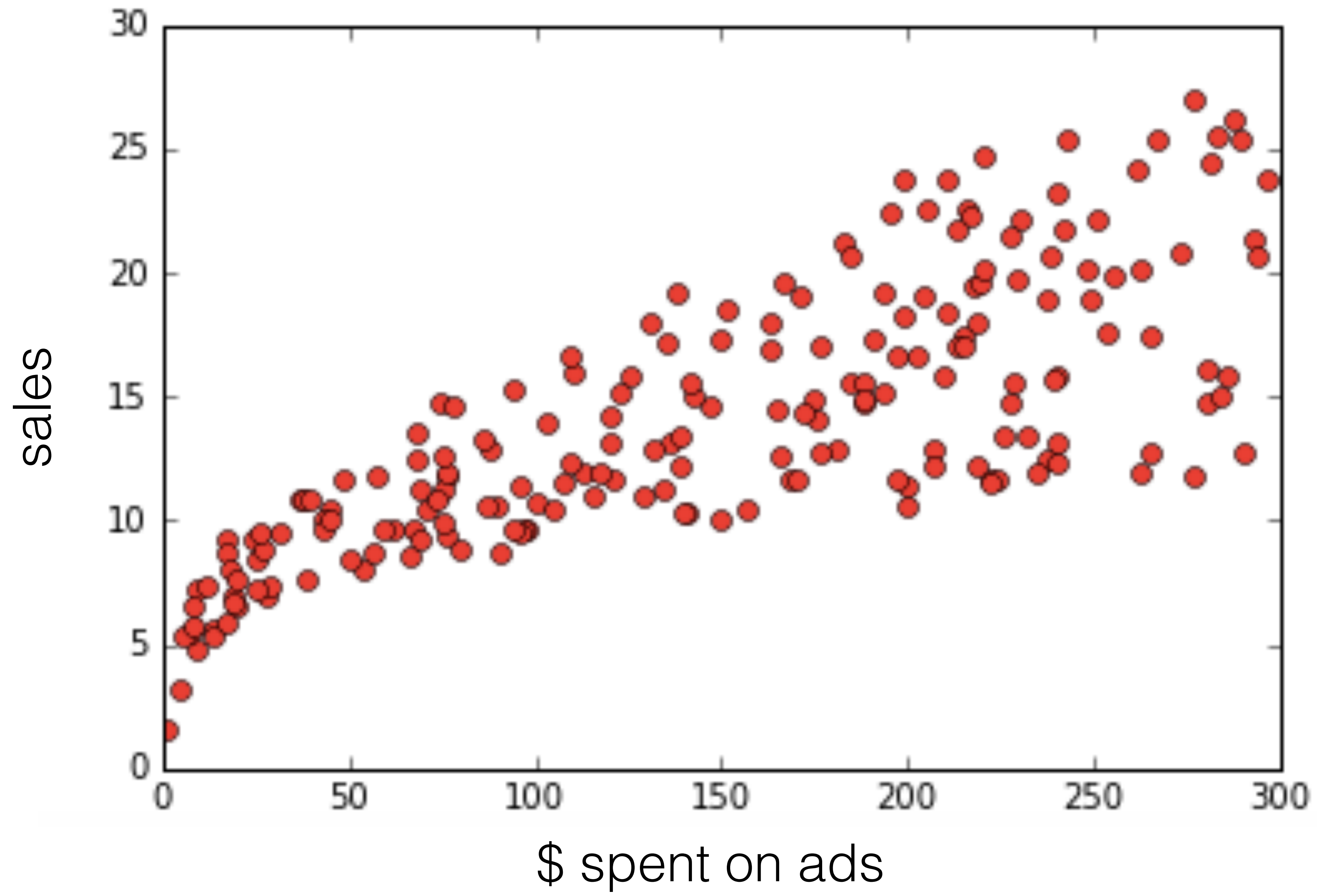
TensorFlow Mechanics

1. **Prepare the Data**
    1. Inputs and Placeholders
2. **Build the Graph**
    1. Inference
    2. Loss
    3. Training (optimizer)
3. **Train The model**
    1. The Session
    2. Compute Graph ops
    3. Train loop
4. **Evaluate the model**

A toy problem



TensorFlow Mechanics
Prepare data and inputs




```
In [1]: import tensorflow as tf
```

```
In [2]: input1 = tf.placeholder(tf.float32)
        input2 = tf.placeholder(tf.float32)
```

TensorFlow Graph

- "TensorFlow programs are usually structured into a **construction phase**, that **assembles** a **graph**, and an **execution phase** that uses a session to **execute ops** in the **graph**"

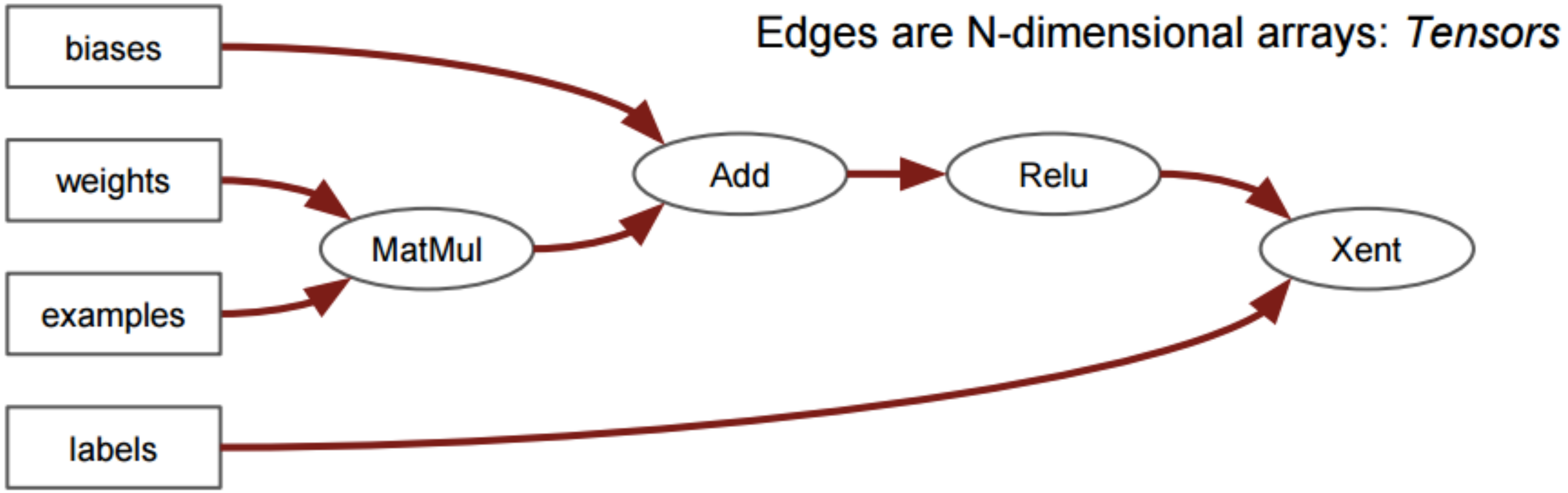
- All **computations** add **nodes** to global default **graph**

TensorFlow Graph

```python
import tensorflow as tf
# Define tf Graph Inputs
X = tf.placeholder("float",[None,1])
y = tf.placeholder("float",[None,1])

# Create Model variables
# Set model weights
W = tf.Variable(np.random.randn(), name="weight")
b = tf.Variable(np.random.randn(), name="bias")

# Construct a linear model
y_pred = tf.add(tf.mul(X, W), b)

# Minimize the squared errors
cost = tf.reduce_sum(tf.pow(y_pred-y,2))/(n_samples) #L2 loss

# Define the optimizer
optimizer = tf.train.AdamOptimizer(learning_rate).minimize(cost) #Gradient descent
```

$$J(W,b) = \frac{1}{N}\sum_{i=1}^{N}(y_i - (Wx_i + b))^2$$

TensorFlow Graph



TensorFlow Graph Loss Functions

- The loss() function further builds the graph by adding the required loss ops.
- The cost function to be minimized during training can be specified easily.

- Linear regression

$$\frac{1}{n}\sum_{i=1}^{n}(y_i - (w^T x_i + w_0))^2$$

- Logistic regression

$$\frac{1}{n}\sum_{i=1}^{n}\log(1 + e^{-y_i(w^T G_i + w_0)})$$

- SVM

$$\frac{1}{n}\sum_{i=1}^{n}\max(0, -y_i(w^T x_i + w_0))$$

TensorFlow Graph Optimitzation Functions

- AdamOptimizer
- GradientDescentOptimizer
- AdagradOptimizer
- AdadeltaOptimizer
- MomentumOptimizer
- FtrlOptimizer
- RMSPropOptimizer

TensorFlow Train the model

1. Create the Session
2. Compute Graph ops
3. Train loop

TensorFlow Train the model Feeding data

- TensorFlow's **feed** mechanism lets you inject **data** into any Tensor in a **computation graph**.

- While you can replace any Tensor with feed data, including variables and constants, the best practice is to use a **placeholder** op node. A placeholder exists solely to serve as the target of feeds. It is not initialized and contains no data

- A **feed_dict** is a python dictionary mapping from tf.placeholder vars (or their names) to data (numpy arrays, lists, etc.).

TensorFlow Train the model Feeding data

Placeholders and feed dictionaries

```
In [1]: import tensorflow as tf

In [2]: input1 = tf.placeholder(tf.float32)
        input2 = tf.placeholder(tf.float32)

In [3]: output = tf.mul(input1,input2)

In [4]: print output

        Tensor("Mul:0", dtype=float32)

In [5]: with tf.Session() as sess:
            print(sess.run([output], feed_dict={input1:[7.],input2:[3.]}))

        [array([ 21.], dtype=float32)]
```

TensorFlow Train the model

Train the model

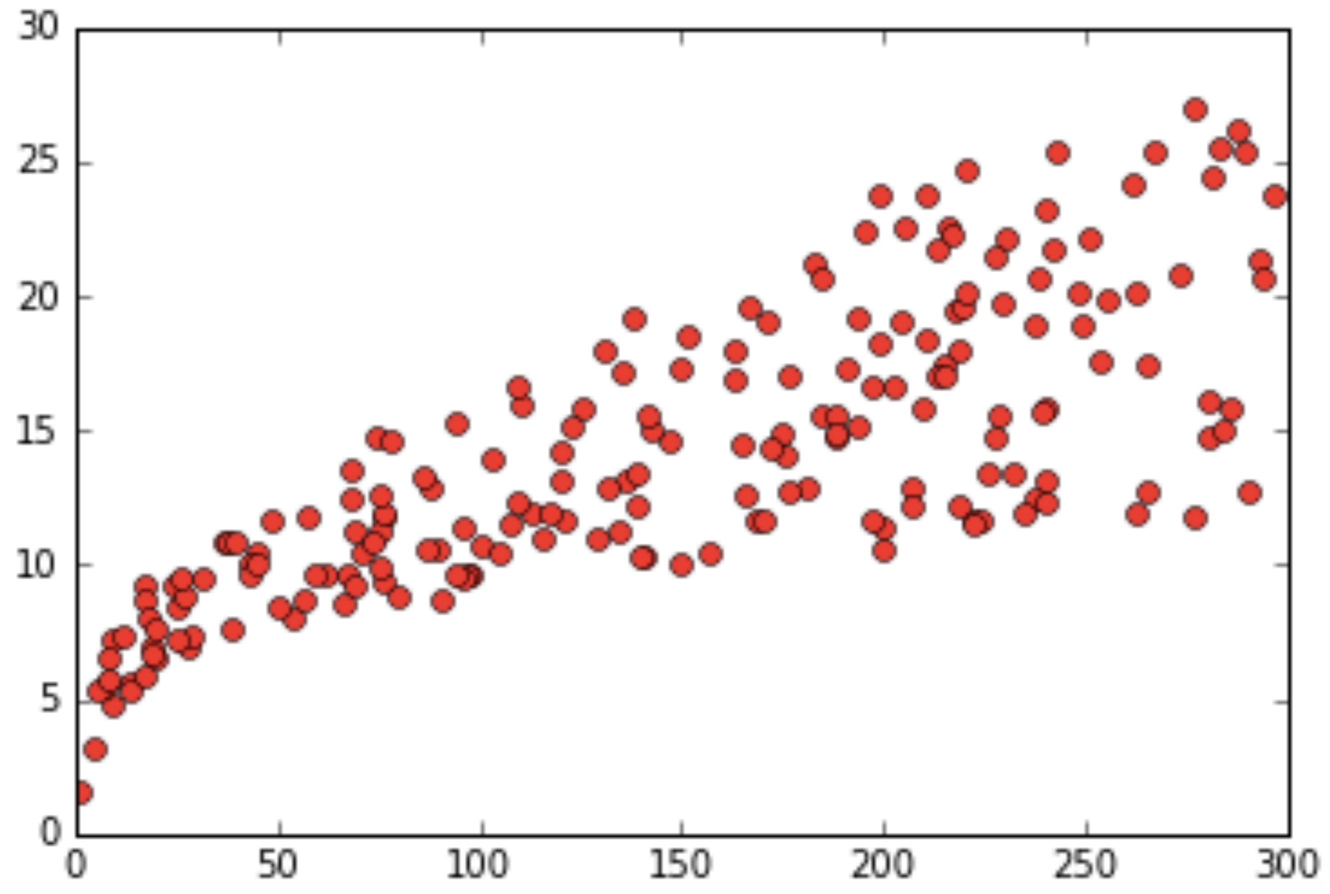
- Now that we have defined our **model** and **training cost function**, it is straightforward to train using TensorFlow. Because TensorFlow knows the entire computation graph, it can use **automatic differentiation** to find the gradients of the cost with respect to each of the variables.

- TensorFlow has a variety of builtin optimization algorithms.

Hands on!

3.1 Linear Regression.ipynb

Linear Regression $y = wx + b$

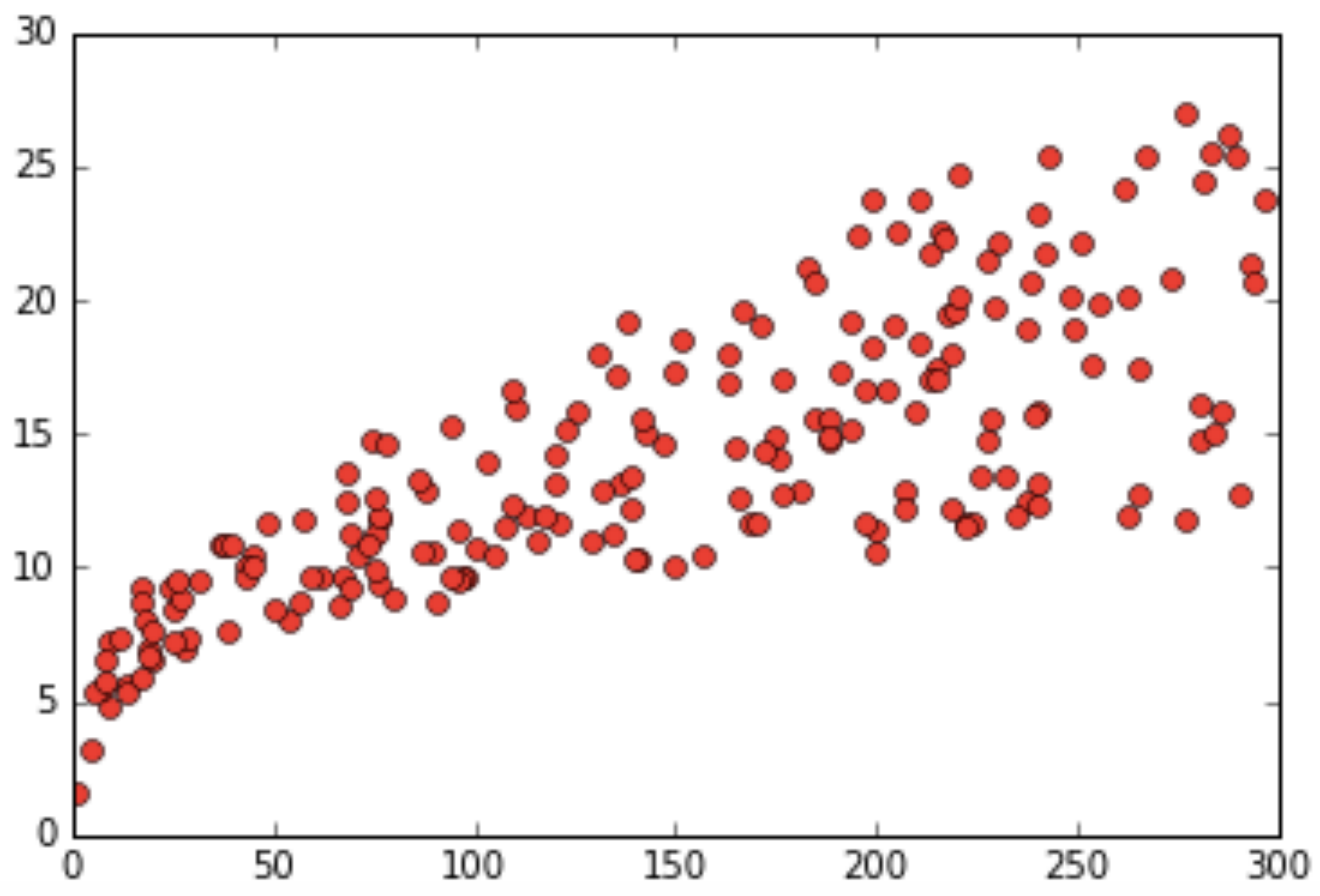


Cost Function: $$J(W,b) = \frac{1}{N}\sum_{i=1}^{N}(y_i - (Wx_i + b))^2$$

Linear Regression Example

```
In [3]: # Load data. Advertising dataset from "An Introduction to Statistical Learning",
        # textbook by Gareth James, Robert Tibshirani, and Trevor Hastie
        import numpy as np
        data = pd.read_csv('dataset/Advertising.csv',index_col=0)
        train_X = data[['TV']].values

        train_Y = data.Sales.values
        train_Y = train_Y[:,np.newaxis]

        n_samples = train_X.shape[0]
        print n_samples
        print train_X.shape, train_Y.shape
        plt.plot(train_X, train_Y, 'ro', label='Original data')
        plt.show()
```

```
200
(200, 1) (200, 1)
```

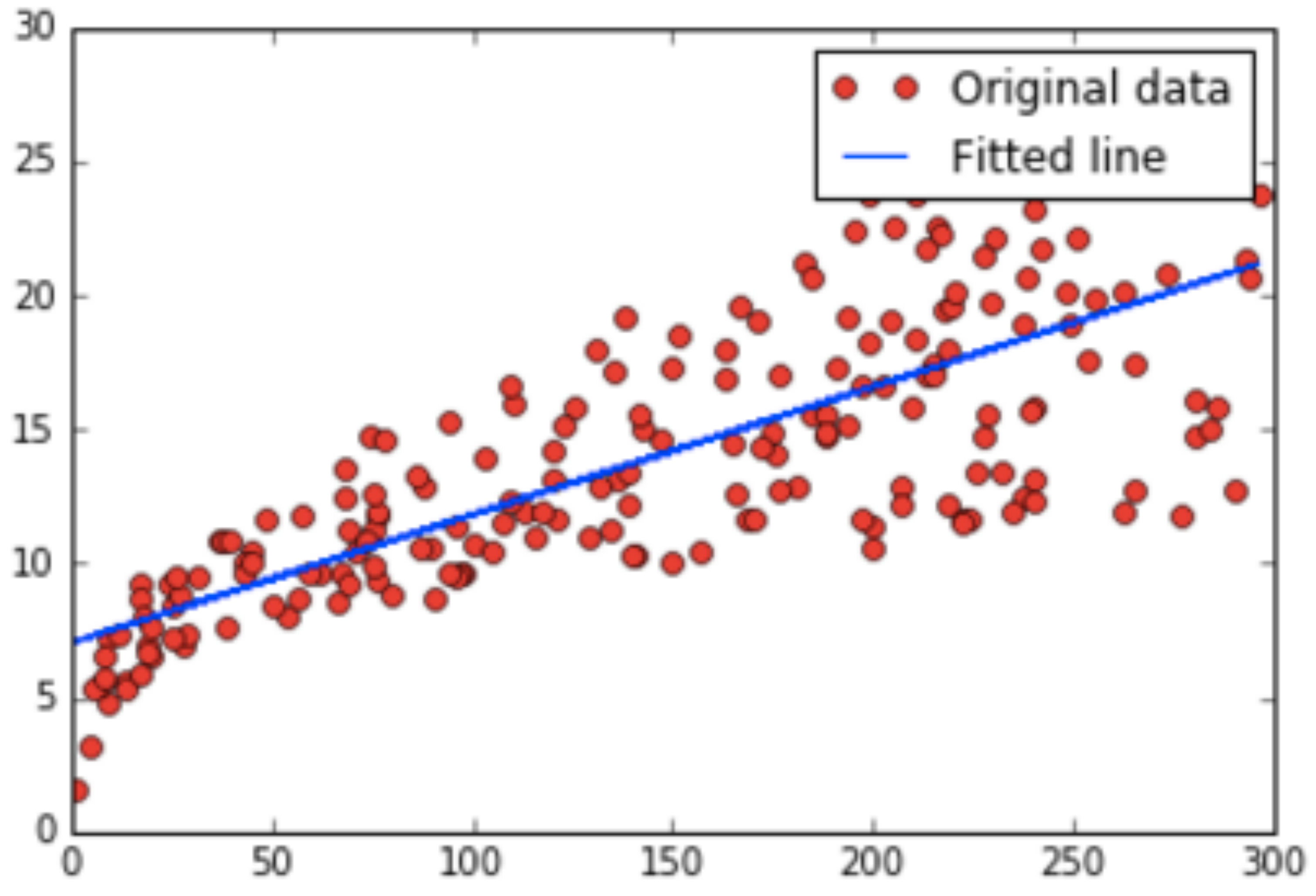


Linear Regression Example

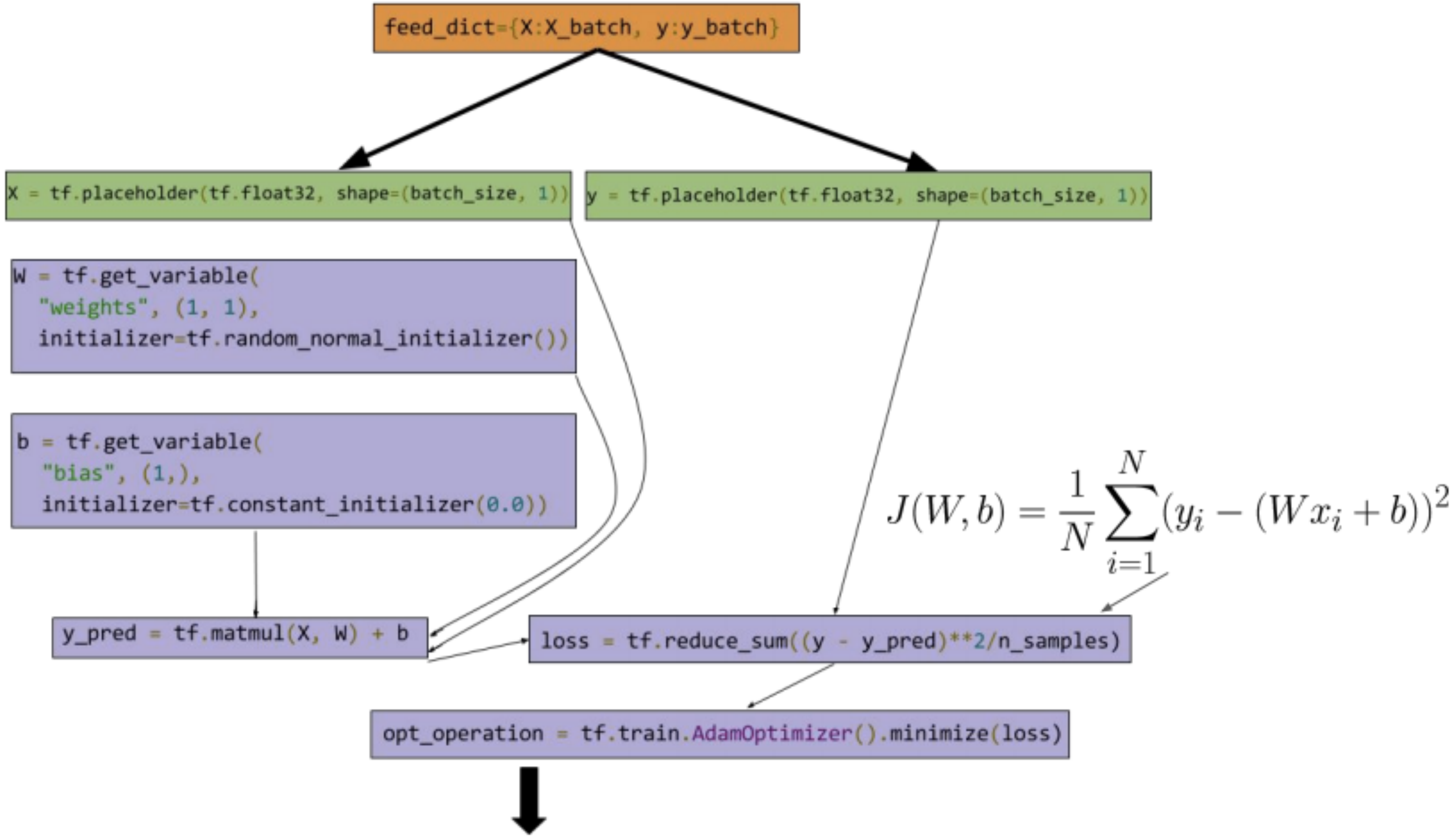
In [6]:
```
# Initializing the variables
init = tf.initialize_all_variables()
# Launch the graph
with tf.Session() as sess:
    sess.run(init)

    # Fit all training data
    for epoch in range(training_epochs):
        sess.run(optimizer, feed_dict={X: train_X, y: train_Y})

        #Display logs per epoch step
        if epoch % display_step == 0:
            print "Epoch:", '%04d' % (epoch+1), "cost=", \
                "{:.9f}".format(sess.run(cost, feed_dict={X: train_X, y:train_Y})), \
                "W=", sess.run(W), "b=", sess.run(b)

    print "Optimization Finished!"
    print "cost=", sess.run(cost, feed_dict={X: train_X, y: train_Y}), \
          "W=", sess.run(W), "b=", sess.run(b)

    #Graphic display
    plt.plot(train_X, train_Y, 'ro', label='Original data')
    plt.plot(train_X, sess.run(W) * train_X + sess.run(b), label='Fitted line')
    plt.legend()
    plt.show()
```

Linear Regression Example

```
Epoch: 0001 cost= 11.784294128 W= 0.085883 b= -0.113122
Epoch: 0201 cost= 5.516845703 W= 0.0548285 b= 5.59834
Epoch: 0401 cost= 5.256753445 W= 0.0478319 b= 6.97454
Epoch: 0601 cost= 5.256326675 W= 0.0475391 b= 7.03211
Epoch: 0801 cost= 5.256326675 W= 0.0475367 b= 7.03259
Epoch: 1001 cost= 5.256326675 W= 0.0475367 b= 7.03259
Epoch: 1201 cost= 5.256326675 W= 0.0475367 b= 7.03259
Epoch: 1401 cost= 5.256327629 W= 0.0475367 b= 7.03259
Epoch: 1601 cost= 5.256326675 W= 0.0475367 b= 7.03259
Epoch: 1801 cost= 5.256326675 W= 0.0475367 b= 7.03259
Optimization Finished!
cost= 5.25679 W= 0.047714 b= 7.03275
```



Linear Regression Example



Logistic Regression?

- Hands on!

Which data is stored ?

In [6]:
```
# Initializing the variables
init = tf.initialize_all_variables()
# Launch the graph
with tf.Session() as sess:
    sess.run(init)

    # Fit all training data
    for epoch in range(training_epochs):
        sess.run(optimizer, feed_dict={X: train_X, y: train_Y})

        #Display logs per epoch step
        if epoch % display_step == 0:
            print "Epoch:", '%04d' % (epoch+1), "cost=", \
                "{:.9f}".format(sess.run(cost, feed_dict={X: train_X, y:train_Y})), \
                "W=", sess.run(W), "b=", sess.run(b)

    print "Optimization Finished!"
    print "cost=", sess.run(cost, feed_dict={X: train_X, y: train_Y}), \
          "W=", sess.run(W), "b=", sess.run(b)

    #Graphic display
    plt.plot(train_X, train_Y, 'ro', label='Original data')
    plt.plot(train_X, sess.run(W) * train_X + sess.run(b), label='Fitted line')
    plt.legend()
    plt.show()
```

Can we obtain **W** and **b** outside the session?

Which data is stored ?

In [9]:
```
# Initializing the variables
init = tf.initialize_all_variables()
# Launch the graph
with tf.Session() as sess:
    sess.run(init)

    # Fit all training data
    for epoch in range(training_epochs):
        sess.run(optimizer, feed_dict={X: train_X, y: train_Y})

        #Display logs per epoch step
        if epoch % display_step == 0:
            print "Epoch:", '%04d' % (epoch+1), "cost=", \
                "{:.9f}".format(sess.run(cost, feed_dict={X: train_X, y:train_Y})), \
                "W=", sess.run(W), "b=", sess.run(b)

    print "Optimization Finished!"
    print "cost=", sess.run(cost, feed_dict={X: train_X, y: train_Y}), \
          "W=", sess.run(W), "b=", sess.run(b)

    #Graphic display
    w = sess.run(W)
    plt.plot(train_X, train_Y, 'ro', label='Original data')
    plt.plot(train_X,  w * train_X + sess.run(b), label='Fitted line')
    plt.legend()
    plt.show()
```

Input Data

- Most of the times we do **not** have **enough memory to load all data from training** set and compute the gradients.
  - Let's see an example using a batch

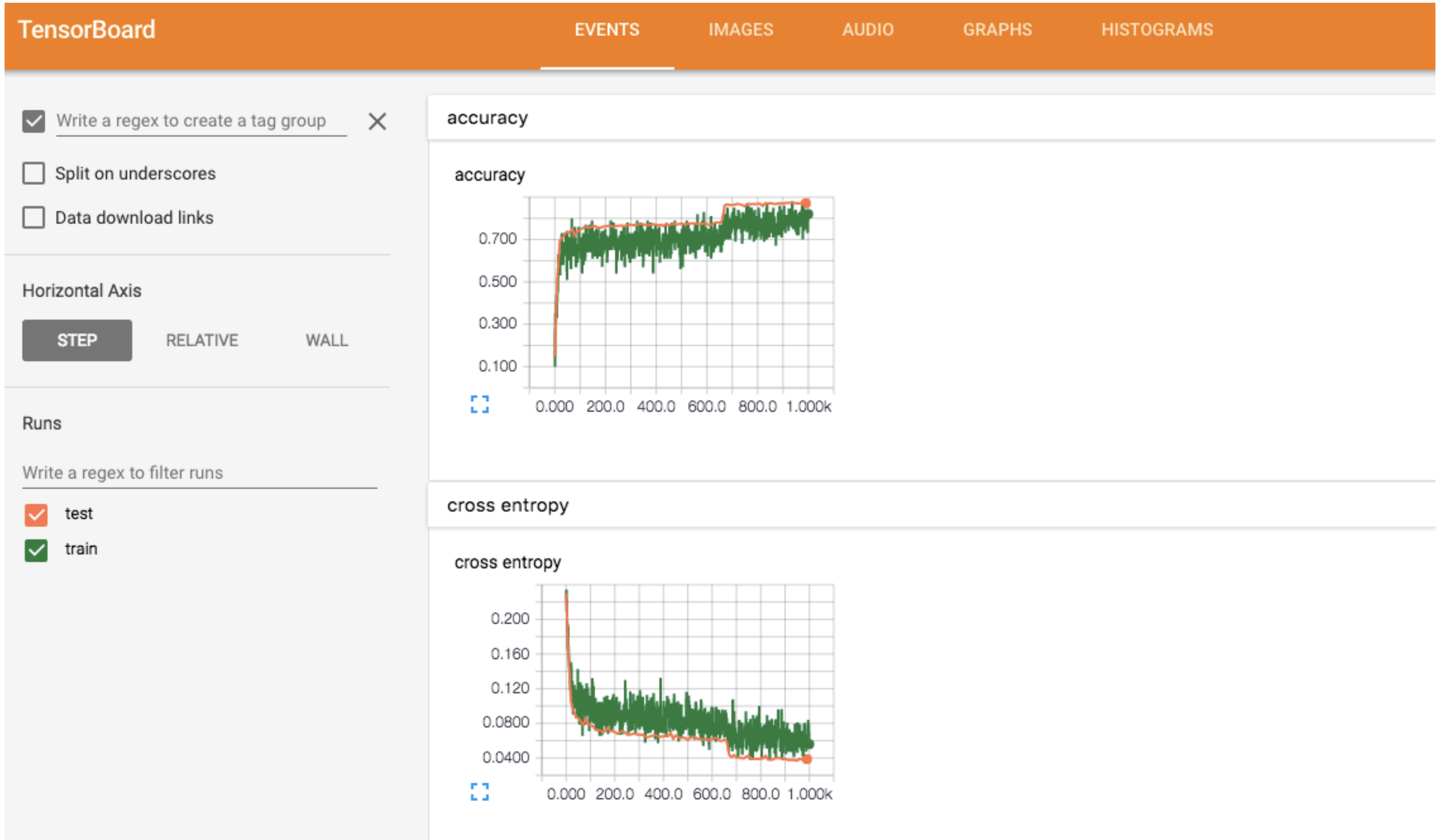
Input Data: Batch

In [2]:
```
#import tensorflow
import tensorflow as tf
import numpy as np

# tf Graph Input
X = tf.placeholder("float", [None, 784]) # mnist data image of shape 28*28=784
y = tf.placeholder("float", [None, 10]) # 0-9 digits recognition => 10 classes

# Create model
# Set model weights
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))

# Construct model
y_pred = tf.nn.softmax(tf.matmul(X, W)) # Softmax
```

Input Data: Batch

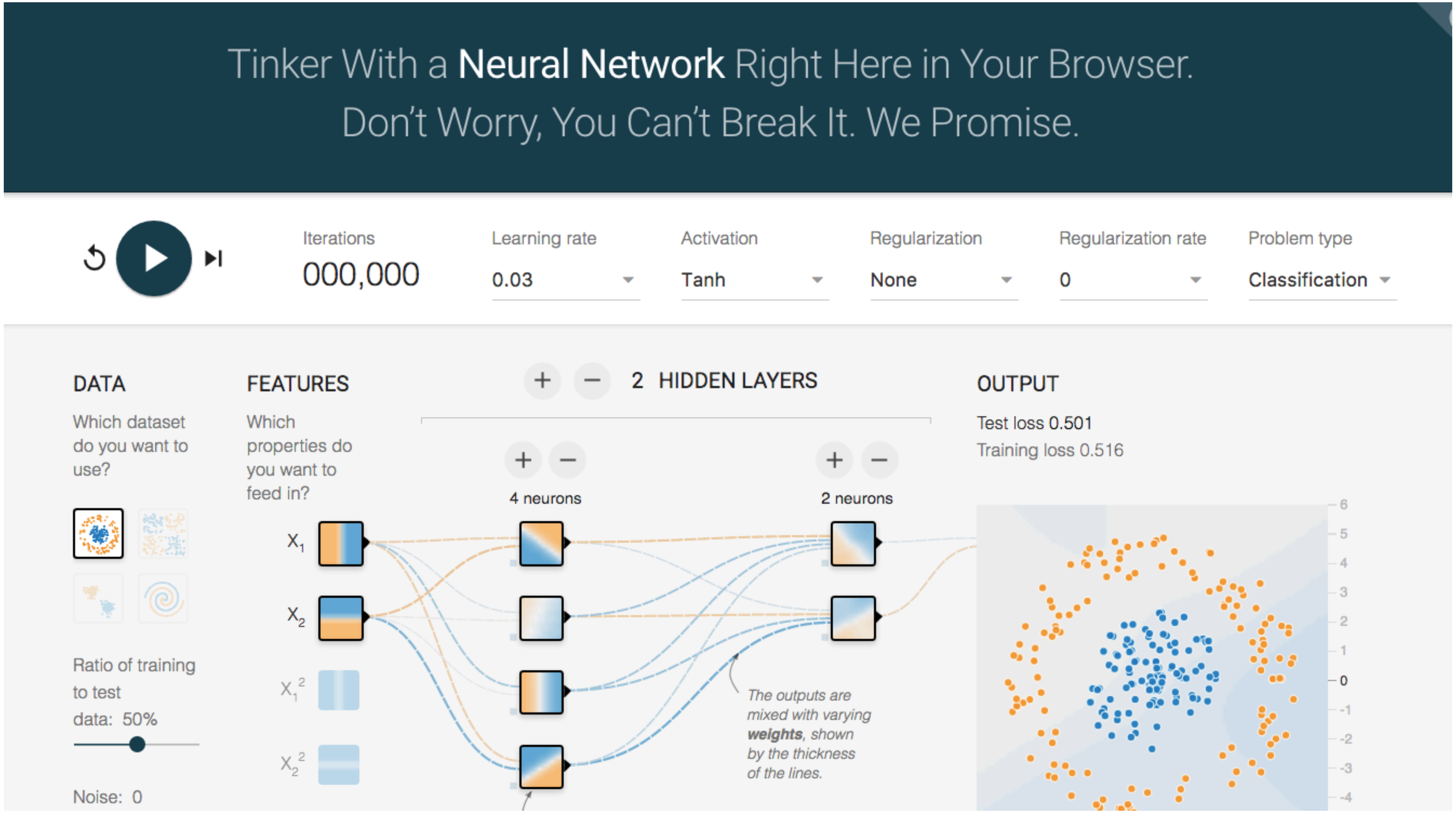
In [5]:
```
# Initializing the variables
init = tf.initialize_all_variables()

# Launch the graph
with tf.Session() as sess:
    sess.run(init)

    # Training cycle
    for epoch in range(training_epochs):
        avg_cost = 0.
        total_batch = int(mnist.train.num_examples/batch_size)
        # Loop over all batches
        for i in range(total_batch):
            batch_xs, batch_ys = mnist.train.next_batch(batch_size)
            # Fit training using batch data
            sess.run(optimizer, feed_dict={X: batch_xs, y: batch_ys})
            # Compute average loss
            avg_cost += sess.run(cost, feed_dict={X: batch_xs, y: batch_ys})/total_batch
        # Display logs per epoch step
        if epoch % display_step == 0:
            print "Epoch:", '%04d' % (epoch+1), "cost=", "{:.9f}".format(avg_cost)

    print "Optimization Finished!"

    # Test model
    correct_prediction = tf.equal(tf.argmax(y_pred, 1), tf.argmax(y, 1))
    # Calculate accuracy
    accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
    print "Accuracy:", accuracy.eval({X: mnist.test.images, y: mnist.test.labels})
```

TensorBoard



https://www.tensorflow.org/tensorboard/index.html

http://playground.tensorflow.org/



Keras: Deep Learning Library for Theano and Tensorflow

- The core data structure of eras is a **model**, a way to organize layers. The main type of model is the **Sequential**, a linear stack of layers.

```
from keras.model import Sequential

model = Sequential()


from keras.layers import Dense, Activation

model.add(Dense(output_dim, input_dim))
model.add(Activation("relu")
model.add(Dense(output_dim=10))
model.add(Activation("softmax))
```

Keras: Deep Learning Library for Theano and Tensorflow

- The core data structure of eras is a **model**, a way to organize layers. The main type of model is the **Sequential**, a linear stack of layers.

```
#Once your model looks good, configure its learning process with .compile():

model.compile(loss='categorical_crossentropy', optimizer='sgd', metrics=['accuracy'])

model.fit(X_train, Y_train, nb_epoch=5, batch_size=32)
```