

Machine Learning



Machine Learning

Lecture: TensorFlow

Ted Scully

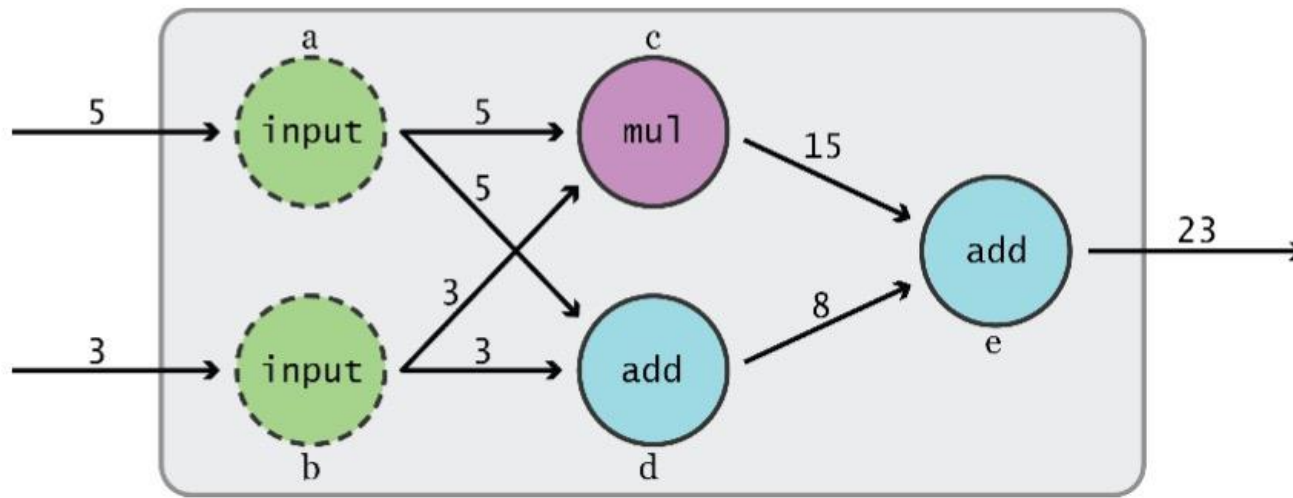
TensorFlow – Graph Mode and Eager Execution Mode

- ▶ There are some significant **disadvantages of the graph-based approach** as well. We will cover these later as it is easier to understand once you have written some TF code.
- ▶ An alternative to the graph mode of operation is also offered called **Eager Execution**.
- ▶ Eager execution is an interface in TF where **operations are executed immediately** as a program executes. This makes it much easier to get started with TF and can help speed up prototyping (there are other advantages that we will discuss later).
- ▶ While it may not be apparent now the differences between graph mode and eager execution mode become much easier to understand when you have written some TF code so we will defer it's discussion until then.

TensorFlow

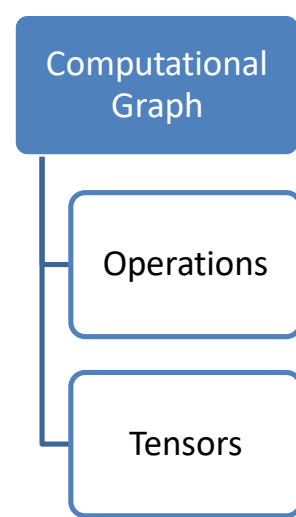
TensorFlow programs are usually structured into a **construction** phase, that assembles a graph, and an **execution** phase that uses a session to execute operations in the graph.

- ▶ With TF you initially build a computation graph. The graph defines the set of **operations** that you want performed on data structures known as **Tensors**. We can view the process as Tensor objects flowing through the computational graph.
- ▶ Then you use a **Session** to execute operations on the graph (more on sessions later).



TensorFlow – Computational Graph

- ▶ A computational graph is a series of TensorFlow operations arranged into a graph.
- ▶ The graph is composed of two types of objects.
 - ▶ **Operations:** The nodes of the graph. Operations describe calculations that consume and produce tensors.
 - ▶ **Tensors:** The edges in the graph. These represent the values that will flow through the graph.



TensorFlow - Operations



- ▶ In TensorFlow, nodes represent operations, which in turn express the transformation of data flowing through the graph.
- ▶ An operation can have **zero or more** inputs and produce **zero or more** outputs (!!)
- ▶ An operation may represent a mathematical equation such as simple addition, matrix multiplication, more neural network specific functions such as ReLU as well as just a TensorFlow variable or constant.
- ▶ Constants and variables are also viewed as operations in TF. Therefore, they can make up nodes in a computation graph.

Basic Operations in TF

You can find basic mathematical operations [here](#).

tf.add (x,y)	Returns tensor with $x + y$ element-wise.
tf.subtract (x,y)	Returns tensor with $x - y$ element-wise.
tf.multiply (x,y)	Returns tensor $x * y$ element-wise.
tf.scalar_mul (scalar,x)	Multiplies scalar with tensor ($\text{scalar} * x$) and returns multiplied tensor.
tf.div (x,y,name=None)	Divides x / y elementwise and returns tensor
tf.reduce_max (x)	Computes the maximum of elements across dimensions of a tensor.
tf.reduce_mean (x)	Computes the mean of elements across dimensions of a tensor.
tf.round (x)	Rounds the values of a tensor to the nearest integer, element-wise.
tf.matmul (a, b)	Performs matrix multiplication on the arguments

Computational
Graph

Operations

Tensors

Basic Operations in TF

The following are the most commonly overloaded methods in TensorFlow.

- `tf.negative` (unary -)
- `tf.add` (binary +)
- `tf.subtract` (binary -)
- `tf.multiply` (binary elementwise *)
- `tf.floordiv` (binary `//` in Python 3)
- `tf.truediv` (binary `/` in Python 3)
- `tf.pow` (binary `**`)
- `tf.logical_and` (binary `&`)
- `tf.logical_or` (binary `|`)

It is important to note that `==` is **not overloaded**. `x == y` will return a Python boolean whether `x` and `y` refer to the same tensor. You need to use `tf.equal()` to check for element wise equality.

Computational
Graph

Operations

Tensors

Constants and Variables

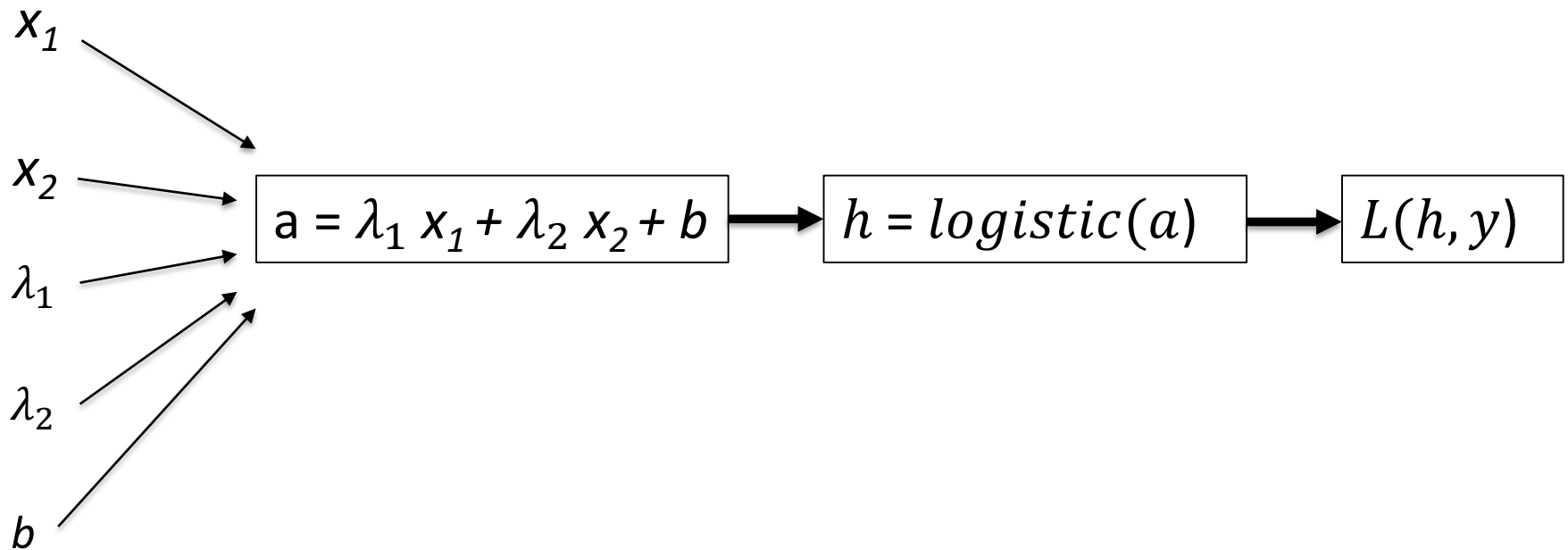
- ▶ It is often necessary to maintain **state** across evaluations of the graph, such as the weights and parameters of a neural network.
- ▶ For this purpose, TF provides **variables**, which are viewed as special operations that can be added to the computational graph.
- ▶ When creating a variable node for a TensorFlow graph, it is necessary to supply a tensor with which the variable is initialized upon graph execution.
- ▶ A **tf.Variable** represents a tensor whose value can be changed by running ops on it. Unlike tf.Tensor objects, a tf.Variable **exists outside the context of a single session.run call**.
- ▶ You can also create constant tensors. The value of these tensors won't change.

Computational
Graph

Operations

Tensors

Forward pass of Logistic Regression depicted as a computational graph.



Constants and Variables

Computational
Graph

Operations

Tensors

- ▶ Below are examples of how to create variables in TF.
- ▶ The following are some of the main parameters:
 - ▶ initial_value: A Tensor, or Python object convertible to a Tensor, which is the initial value for the Variable.
 - ▶ trainable: If True, this variable will be adjusted using an optimizer (more on this later)
 - ▶ name: Optional name for the variable within the graph.
- ▶ For a full listing of parameters click [here](#).

```
import tensorflow as tf

num2 = tf.Variable(0)
num2 = tf.Variable([1, 2], name= "numbers")

num3 = tf.Variable( tf.zeros((4, 4)), name = 'weights' )
```

Please note that you can also create variables using [tf.get_variable](#).

TensorFlow - Tensors

Computational
Graph

Operations

Tensors

- ▶ In TF, the edges of a graph represent data flowing from one operation to another and are referred to as tensors
- ▶ Tensors are the **data structures that ‘flow’ through the computational graph** that we create.
- ▶ A tf.Tensor has the following properties:
 - ▶ a **data type** (float32, int32, or string, for example)
 - ▶ a **shape**
- ▶ A tensor is a **homogenous** data structure.
- ▶ The shape (that is, the number of dimensions it has and the size of each dimension) may only be **partially known** (more on this later).

Tensors

Computational
Graph

Operations

Tensors

- ▶ The dimensions of a tensor is referred to it's **rank**.
- ▶ There are a host of operations build into TF that allow to create Tensors.
- ▶ Also as you being to gain exposure to TF you should see a high degree of simarity between TF and NumPy operations.

```
import tensorflow as tf

t1 = tf.zeros((4, 5))
t2 = tf.range(3, 15, 2)
t3 = tf.ones((2,2))
T4 = tf.random.uniform((10,20),dtype=tf.float32)
print (t1)
print (t1.shape)
```

Tensor("zeros_7:0", shape=(4, 5), dtype=float32)

(4, 5)

TensorFlow and NumPy

- ▶ You will hopefully be able to see a higher level of similarity between many of the functions used in NumPy and TensorFlow.

```
import tensorflow as tf
import numpy as np

a = np.zeros((4, 4))
a_t = tf.zeros((4, 4))

b = np.ones((4, 4))
b_t = tf.ones((4, 4))

print (np.sum(b, axis=1))
print (tf.reduce_sum(a, reduction_indices=[1]))

print (a.shape)
print (a_t.get_shape())

np.dot(a, b)
tf.matmul(a_t, b_t)

print (a[:, 1])
print (a_t[:, 1])
```

The main difference between NumPy arrays and TensorFlow Tensors are:

1. Tensors can use be used on a GPU or TPU.
2. Does support automatic calculation of derivatives

Tensor / NumPy Conversion

- ▶ Conversion between TensorFlow Tensors and NumPy ndarrays is straight-forward.
- ▶ TensorFlow operations automatically convert NumPy ndarrays to Tensors.
- ▶ NumPy operations automatically convert Tensors to NumPy ndarrays.
- ▶ Another useful method that can convert an existing NumPy data structure to a Tensor is **tf.convert_to_tensor**

Computational
Graph

Operations

Tensors

```
import tensorflow as tf
import numpy as np

arr = np.array([1, 5.5, 3, 15, 20])

t1 = tf.convert_to_tensor(arr, tf.float32)
print (type(t1))
```

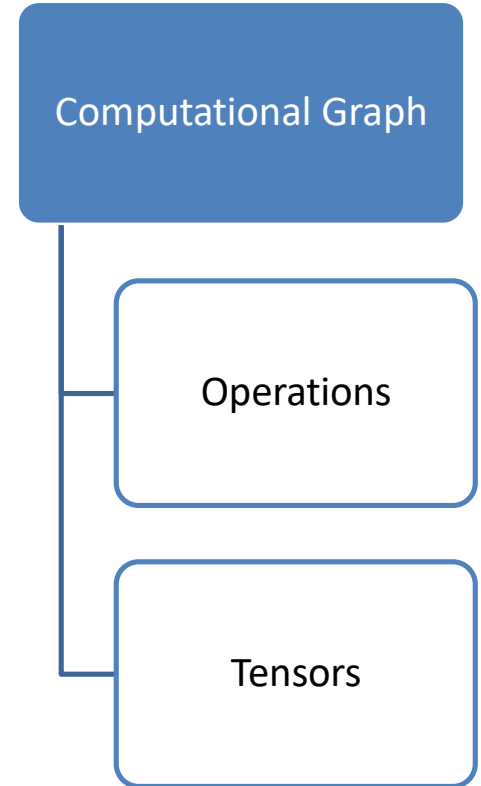
<class 'tensorflow.python.framework.ops.Tensor'>

Building a graph

- ▶ At this stage we understand the elements of a computational graph in TF.
- ▶ Let's now try to build a graph for the following simple function

$(\text{num1} * \text{num2}) + \text{const2}$

Where num1 and num2 are variables and const2 is a constant value.



Building a Simple Graph

- ▶ Let's build a simple graph in TensorFlow.

```
import tensorflow as tf

num1 = tf.Variable(2.0)
num2 = tf.Variable(3.0)

const2 = tf.constant(2.0)

product = tf.multiply(num1, num2)
result = tf.add(product, const2)

print (result)
```

```
Tensor("Add_43:0", shape=(), dtype=float32)
```

Notice when we print out *result* we get the output above indicating that it is a node (called Add), that it provides a rank 0 tensor and dtype of float 32.

Building a Simple Graph

- ▶ Let's build a simple graph in TensorFlow for the illustration below.

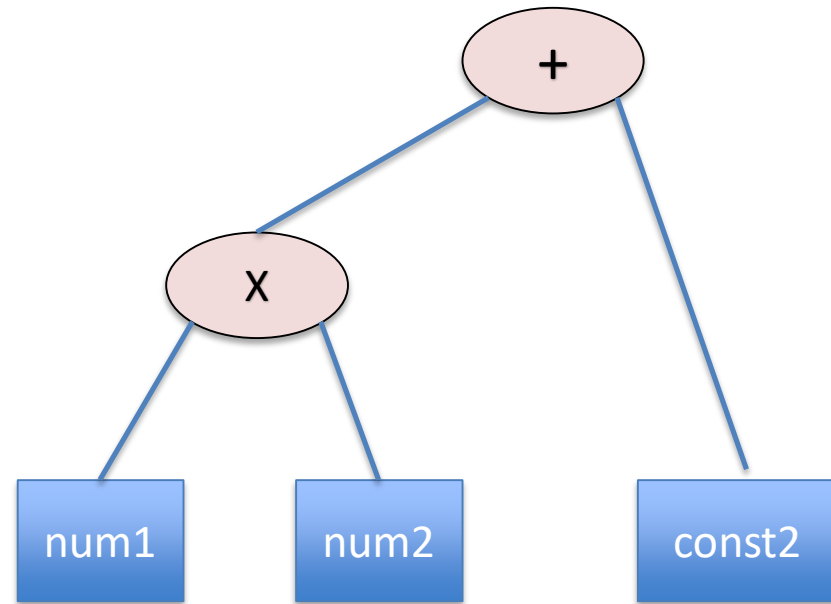
```
import tensorflow as tf

num1 = tf.Variable(2.0)
num2 = tf.Variable(3.0)

const2 = tf.constant(2.0)

product = tf.multiply(num1, num2, name =
"product")
result = tf.add(product, const2, name="sum")

print (result)
```



```
Tensor("sum:0", shape=(), dtype=float32)
```

Building a Simple Graph

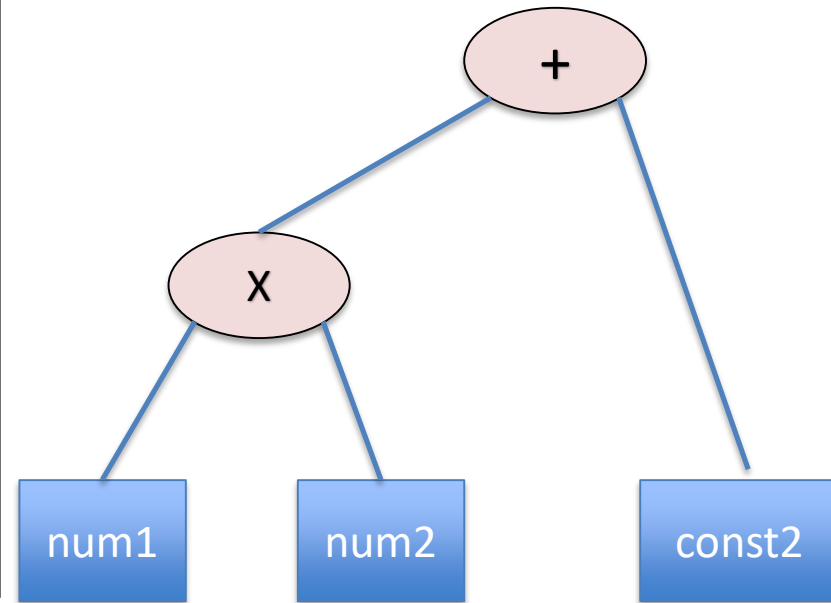
- ▶ Let's build a simple graph in TensorFlow for the illustration below.

```
import tensorflow as tf

num1 = tf.Variable(2.0)
num2 = tf.Variable(3.0)

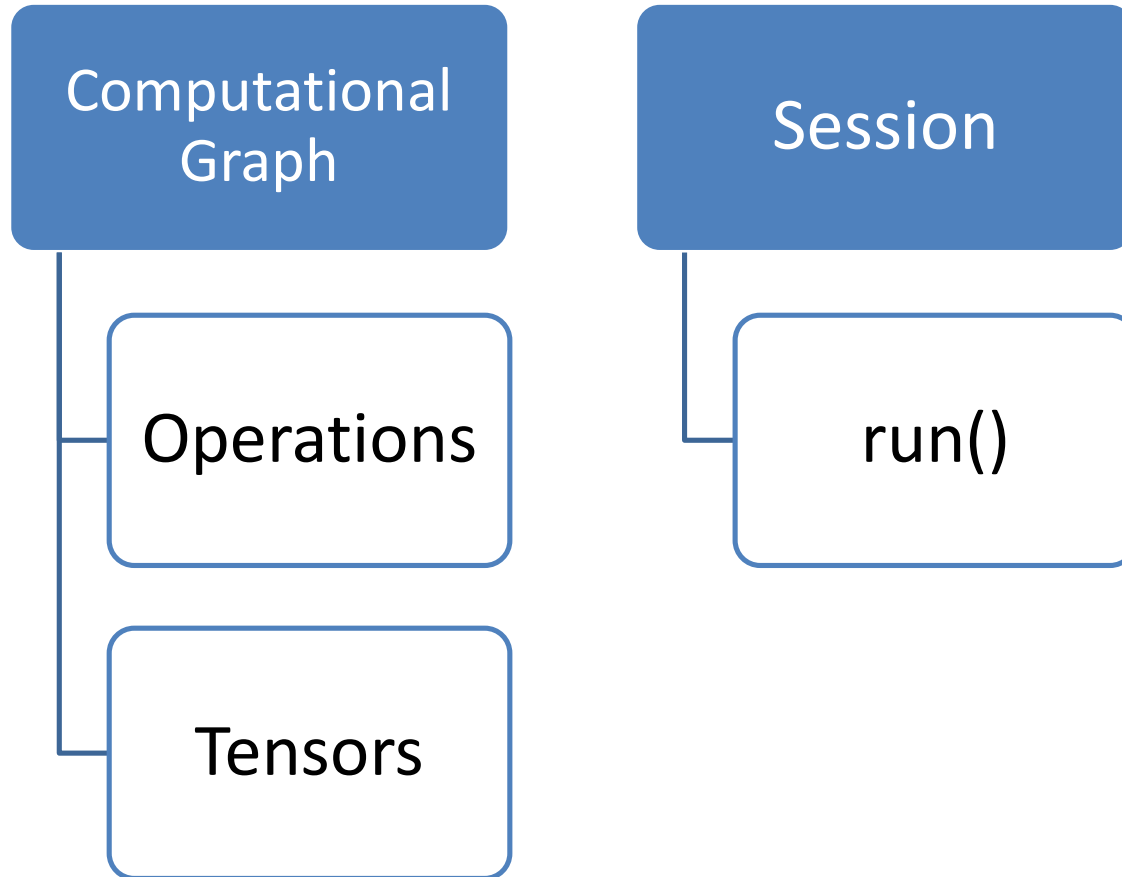
const2 = tf.constant(2.0)

product = tf.multiply(num1, num2, name =
"product")
result = tf.add(product, const2, name="sum")
print (result)
```



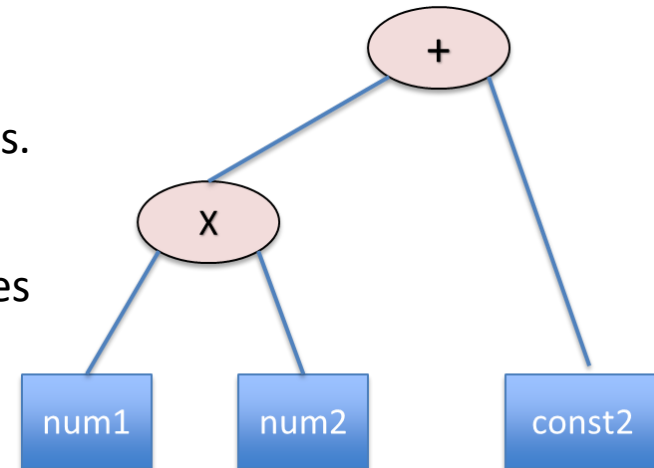
- ▶ It is important to understand that the code written above does not perform any computation.
- ▶ It just creates a computation graph. Even the variables are not initialized yet.
- ▶ To evaluate this graph, you need to open a **TensorFlow session** and use it to initialize the variables and evaluate result.

Building a graph



Session

- ▶ In TensorFlow, the **execution of operations** and **evaluation of tensors** is performed in a special environment referred to as a **session**
 - ▶ The Session interface manages the resources and computation performed in the computational graph.
 - ▶ It takes care of initialisation of variables, placing operations onto devices such as CPUs and GPUs and running them, and it also holds all the variable values.
- ▶ The **Session interface** of the TensorFlow library provides a run() routine, which is the entry point for executing either parts of the graph or the entire computational graph.
 - ▶ This method takes as input the nodes in the graph whose tensors should be computed and returned
 - ▶ Upon invocation of run, TensorFlow will start at the requested output nodes and work backwards, examining the graph dependencies and finally computing the output of the requested nodes.



Session

- ▶ As we mentioned the execution of operations and the evaluation of tensors is performed in a special environment referred to as a **Session**
- ▶ A method of creating a Session is shown below. The code `tf.Session` creates a session and assigns it to the object `sess`. The session is automatically closed after the indented block of code.

```
import tensorflow as tf

num1 = tf.Variable(2.0)
num2 = tf.Variable(3.0)
const2 = tf.constant(2.0)

product = tf.multiply(num1, num2, name = "product")
result = tf.add(product, const2, name="sum")

sess = tf.Session()

sess.run( tf.global_variables_initializer() )

answer = sess.run(result)
print (answer)
sess.close()
```

The function `tf.global_variables_initializer()` actually creates a node in the graph and when this is executed using `run` all variables are initialized.

Session

```
import tensorflow as tf

num1 = tf.Variable(2.0)
num2 = tf.Variable(3.0)
const2 = tf.constant(2.0)

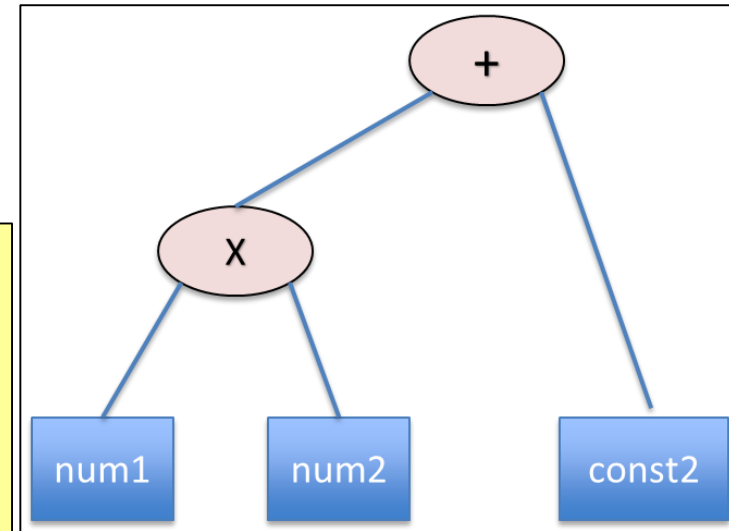
product = tf.multiply(num1, num2, name = "product")
result = tf.add(product, const2, name="sum")

sess = tf.Session()

sess.run( tf.global_variables_initializer() )

answer = sess.run(result)
print (answer)

sess.close()
```



If we want to run and solve for any node we can use `sess.run(node)`. It will return the result.

Notice we must also close the session to release any resources.

Session

- ▶ While the code on the previous slides is perfectly fine it is more common to execute a session using the python **'with' command**.
- ▶ The 'with' command will automatically clean up any resources when the code in the with block finishes.

```
import tensorflow as tf

num1 = tf.Variable(2.0)
num2 = tf.Variable(3.0)
const2 = tf.constant(2.0)

product = tf.multiply(num1, num2)
result = tf.add(product, const2)

with tf.Session() as sess:

    sess.run(tf.global_variables_initializer())
    answer = sess.run(result)
    print (answer)
```

Session

- Remember we created a number of Tensor objects in a earlier slide. To obtain the value of these tensors we must still initialize a Session and get the session to return the values.

```
t1 = tf.zeros((4, 5))
```

```
t2 = tf.range(3, 15, 2)
```

```
t3 = tf.ones((2,2))
```

```
with tf.Session() as sess:
```

```
    print (sess.run(t1))
```

```
    print (sess.run(t2))
```

```
    print (sess.run(t3))
```

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

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

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

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

```
[ 3 5 7 9 11 13]
```

```
[[1. 1.]
```

```
[1. 1.]]
```


Lifecycle of a Node

- ▶ The code below starts a session and runs the graph to evaluate y: TensorFlow automatically detects that **y depends on x**, which depends on w, so it first evaluates w, then x, then y, and returns the value of y.
- ▶ On the next line the code runs the graph to evaluate z. Once again, TensorFlow detects that it must first evaluate w and x.
- ▶ It is important to note that it **will not reuse the result of the previous evaluation** of w and x. In short, the preceding code evaluates w and x twice.

```
import tensorflow as tf

w = tf.constant(4)
x = tf.add(w, 3)
y = tf.add(x, 2)
z = tf.add(x, 1)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    print(sess.run(y)) # 9
    print(sess.run(z)) # 8
```

Lifecycle of a Node

- ▶ The code below starts a session and runs the graph to evaluate `y`: TensorFlow automatically detects that **`y` depends on `x`**, which depends on `w`, so it first evaluates `w`, then `x`, then `y`, and returns the value of `y`.
- ▶ On the next line the code runs the graph to evaluate `z`. Once again, TensorFlow detects that it must first evaluate `w` and `x`.
- ▶ It is important to note that it **will not reuse the result of the previous evaluation** of `w` and `x`. In short, the preceding code evaluates `w` and `x` twice.

```
import tensorflow as tf

w = tf.constant(4)
x = tf.add(w, 3)
y = tf.add(x, 2)
z = tf.add(x, 1)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    print(sess.run(y)) # 9
    print(sess.run(z)) # 8
```

All node values are dropped between graph runs, except variable values, which are maintained by the session across graph runs.

A variable starts its life when its initializer is run, and it ends when the session is closed.

Lifecycle of a Node

- ▶ If you want to evaluate y and z efficiently, without evaluating w and x twice as in the previous code, you must ask TensorFlow to evaluate both y and z in just one 'graph run' as shown below.

```
import tensorflow as tf

w = tf.constant(4)
x = tf.add(w, 3)
y = tf.add(x, 2)
z = tf.add(x, 1)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    resultY, resultZ = sess.run([y, z])

    print(resultY) # 9
    print(resultZ) # 8
```

Notice we can provide a list of nodes to the run session and it will evaluate them in a single run.

Resetting the Default Graph

- ▶ The default graph can persist between runs of a program, which can mean that you are continually adding nodes to the graph over and over again with each run.
- ▶ To avoid this problem and potential problem that can arise as a result I would suggest you reset the default graph at the start of your programming using **`tf.reset_default_graph()`**.

```
import tensorflow as tf

tf.reset_default_graph()

w = tf.constant(4)
x = tf.add(w, 3)
y = tf.add(x, 2)
z = tf.add(x, 1)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())

    resultY, resultZ = sess.run([y, z])

    print(resultY) # 9
    print(resultZ) # 8
```

Placeholder Nodes

- ▶ Placeholder nodes act as containers for **training data** that will pass into our computational graph during the execution phase.
- ▶ **So a placeholder is simply a Tensor that we will assign data to at a later date.** It allows us to create our operations and build our computation graph, without needing the data.
- ▶ To create a placeholder node, you must call the `placeholder()` function and specify the output tensor's data type. You can also specify its shape (optional).

```
import tensorflow as tf

tf.reset_default_graph()

x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)

ans = tf.add(x, y)

with tf.Session() as sess:
    answer = sess.run(ans, feed_dict={x:2.0, y:3.0})
    print (answer)
```

Placeholder Nodes

- ▶ Placeholder nodes act as containers for **training data** that will pass into our computational graph during the execution phase.
- ▶ **So a placeholder is simply a Tensor that we will assign data to at a later date.** It allows us to create our operations and build our computation graph, without needing the data.
- ▶ To create a placeholder node, you must call the `placeholder()` function and specify the output tensor's data type. You can also specify its shape (optional).

```
import tensorflow as tf

tf.reset_default_graph()

x = tf.placeholder(tf.float32)
y = tf.placeholder(tf.float32)

ans = tf.add(x, y)

with tf.Session() as sess:
    answer = sess.run(ans, feed_dict={x:2.0, y:3.0})
    print (answer)
```

So how do we specify values that we will feed to a placeholder object. The `run` function in a session takes a **dictionary argument** called `feed_dict` that allows us to specify the values for the placeholders.

Placeholder Nodes – Specifying Shape

- ▶ We can also **specify a shape** on a placeholder object. For example, I can specify that A is a placeholder object that consists of floats in the shape of a 2*3 tensor.

```
import tensorflow as tf

A = tf.placeholder(tf.float32, shape=(2, 3))
B = tf.add(A, 5)

with tf.Session() as sess:

    B_val_1 = sess.run(B, feed_dict={A: [[4, 5, 6], [7, 8, 9]]})
    print (B_val_1)
```

Placeholder Nodes

- ▶ It's also possible to **partially specify the dimensions**.
- ▶ If we specify None for a dimension, it means “any size.” Notice below this allows us to reuse the same placeholder and provide arguments with different dimensions.
- ▶ We specify that data fed into placeholder A must have 3 columns of data but the number of row is not fixed.

```
[[ 9. 10. 11.]  
 [12. 13. 14.]]  
  
[[ 6. 7. 8.]]
```

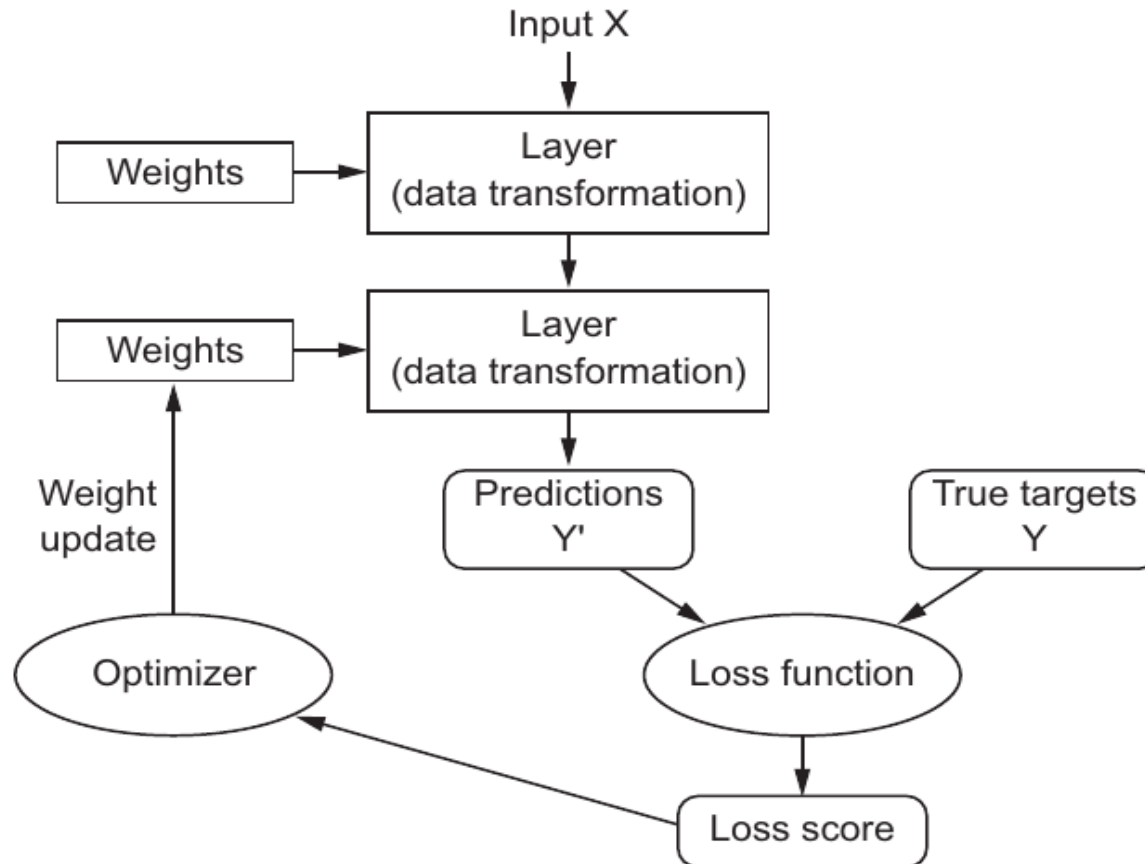
```
A = tf.placeholder(tf.float32, shape=(None, 3))  
B = tf.add(A, 5)
```

```
with tf.Session() as sess:
```

```
    B_val_1 = sess.run(B, feed_dict={ A: [[4, 5, 6], [7, 8, 9]] })  
    print (B_val_1)
```

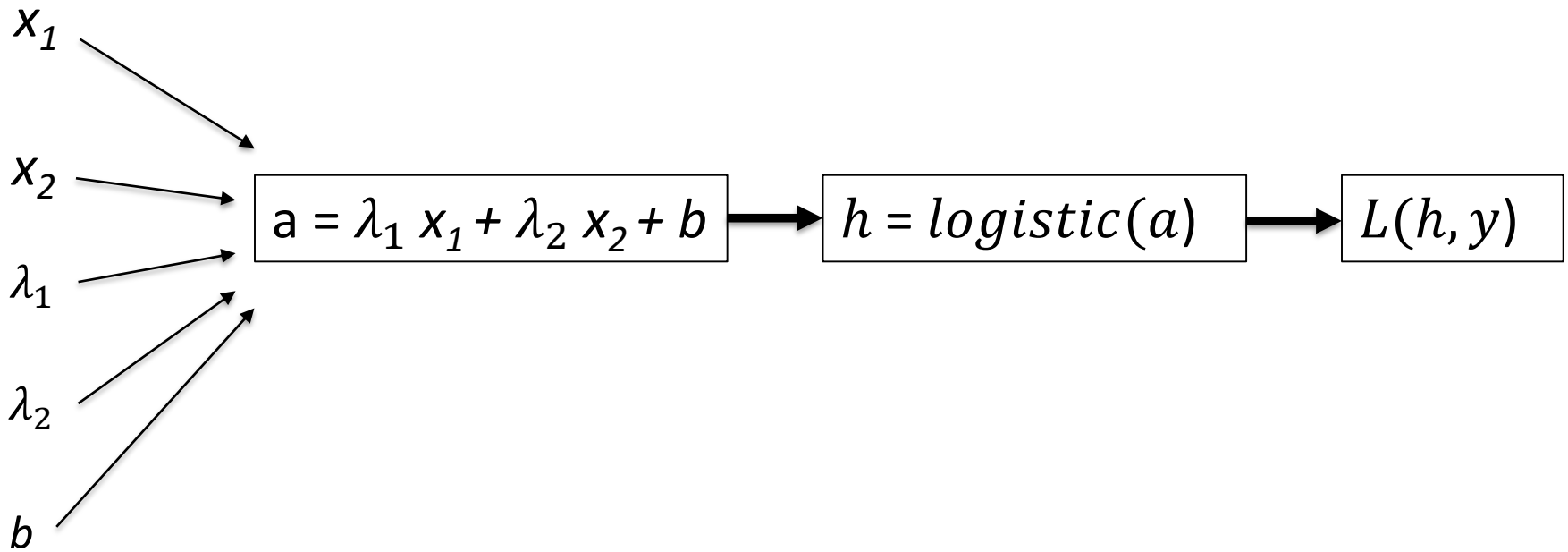
```
    B_val_2 = sess.run(B, feed_dict={ A: [[1, 2, 3]] })  
    print (B_val_2)
```


Automatic Differentiation



Automatic Differentiation

- ▶ TensorFlow supports automatic differentiation, in other words it allows us to automatically compute the partial derivatives of variables in our graph.
- ▶ A number of different optimization techniques are provided in TensorFlow, which use the partial derivatives to optimize an objective function.



Automatic Differentiation

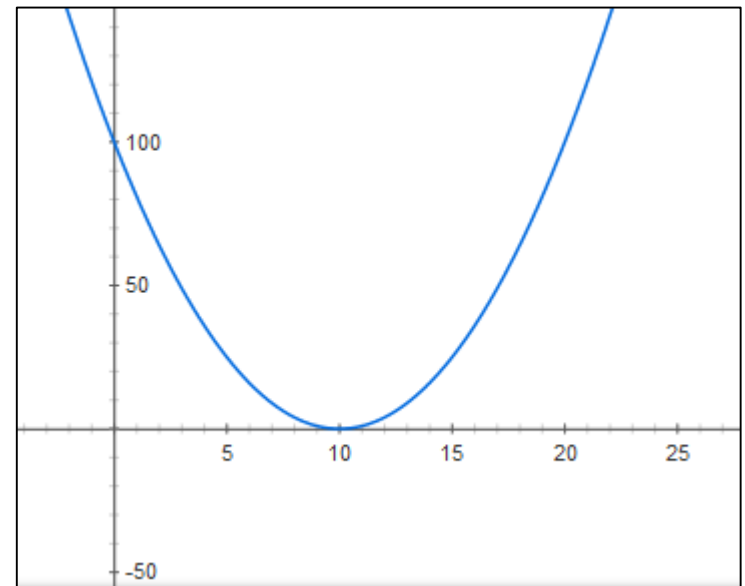
- ▶ For example, TF supplies a Gradient Descent optimization function that **updates the variables in a graph according to the gradient descent update rule.**
- ▶ In the code below we create an instance of **GradientDescentOptimization** with a specific learning rate and configure it to minimize a specific graph node. This code adds a gradient descent optimizer node to our graph. We can then run this in a session (as we would any node) and it will update all variables using the gradient descent update rule.

```
# In the construction phase we have the following:  
# Notice it is pointed at the node whose cost function we want to minimize  
# In this example that node is called graphNode  
  
learning_rate = 0.05  
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(graphNode)
```

- ▶ Many other optimizers are also provided in [tf.train](#) such as MomentumOptimizer, Adam, AdaGrad etc

Using TensorFlow to Minimize a Function

- ▶ In the following example, we are going to write a TensorFlow program to find the minimum value of x in the following function:
- ▶ $x^2 - 20x + 100$
- ▶ The minimum value of x is obvious but we will illustrate the process of building a simple TensorFlow program to:
 - ▶ Model the equation using a graph (construction)
 - ▶ Create a session and solve using gradient descent optimizer (execution).



```
learning_rate = 0.01
```

```
x = tf.Variable(50.0, tf.float32)
```

```
#  $X^2 - 20X + 100$ 
```

```
firstTerm = tf.square(x)
```

```
secondTerm = tf.multiply(-20.0, x)
```

```
quadratic = tf.add(tf.add(firstTerm, secondTerm), 100.0)
```

```
learning_rate = 0.01
```

```
x = tf.Variable(50.0, tf.float32)
```

```
#  $X^2 - 20X + 100$ 
```

```
firstTerm = x**2
```

```
secondTerm = -20.0 * x
```

```
quadratic = firstTerm + secondTerm + 100.0
```

```
import tensorflow as tf
```

```
learning_rate = 0.01
```

```
x = tf.Variable(50.0, tf.float32)
```

```
#  $X^2 - 20X + 100$ 
```

```
firstTerm = x**2
```

```
secondTerm = -20.0 * x
```

```
quadratic = firstTerm + secondTerm + 100.0
```

```
optimizer =
```

```
tf.train.GradientDescentOptimizer(learning_rate).minimize(quadratic)
```

```
with tf.Session() as sess:
```

```
    # Initialize our variable x
```

```
    sess.run(tf.global_variables_initializer())
```

```
    sess.run(optimizer)
```

```
    print (sess.run(x))
```

$$X^2 - 20X + 100$$

49.2

Here we run our Gradient Descent Optimizer node. This performs a step of gradient descent on our function node called quadratic.

```
import tensorflow as tf
```

```
learning_rate = 0.01
```

```
x = tf.Variable(50.0, tf.float32)
```

10.000024

```
#  $X^2 - 20X + 100$ 
```

```
firstTerm = x**2
```

```
secondTerm = -20.0 * x
```

```
quadratic = firstTerm + secondTerm + 100.0
```

```
optimizer =
```

```
tf.train.GradientDescentOptimizer(learning_rate).minimize(quadratic)
```

```
with tf.Session() as sess:
```

```
    # Initialize our variable x
```

```
    sess.run(tf.global_variables_initializer())
```

```
    for num in range(1000):
```

```
        sess.run(optimizer)
```

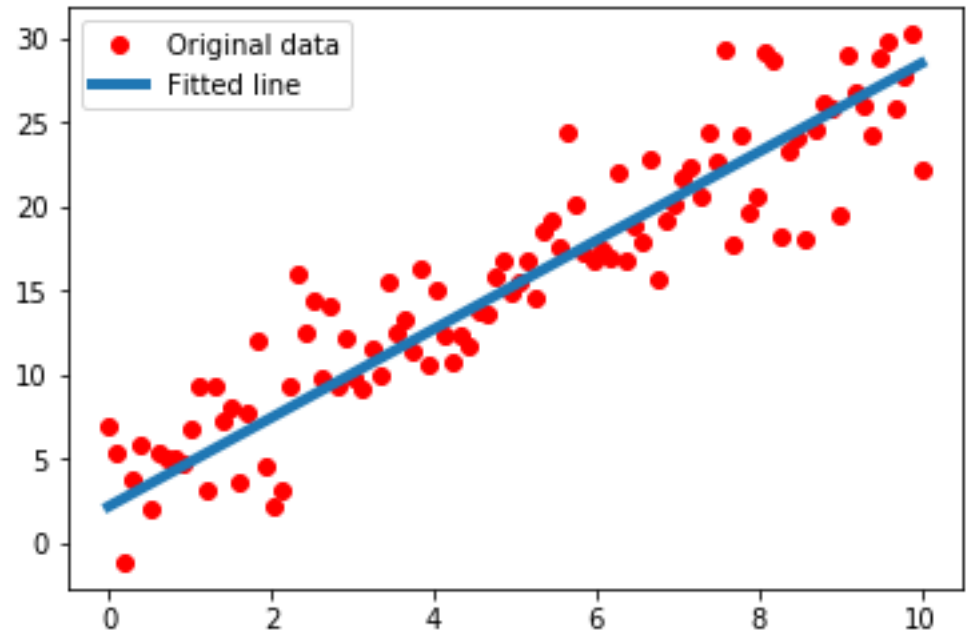
```
    print (sess.run(x))
```

Notice we now run the graph 1000 times. Each time it runs it runs one step of gradient descent.

Linear Regression Using TensorFlow

- ▶ We have now seen how to feed data into a graph as well as optimize variables in a graph. In the following example, we are going to build a Linear Regression algorithm using TensorFlow and put both of these components together.
- ▶ Remember the first step is to build the computational graph. We then execute the graph in a Session.

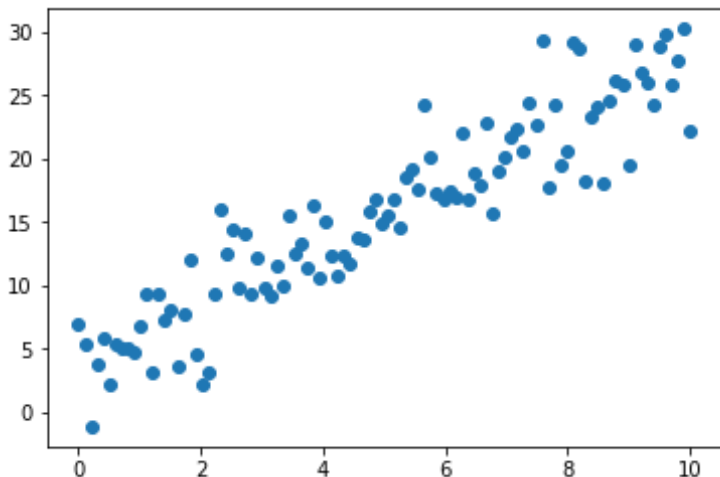
- ▶
$$h(x) = \lambda_1 x + b$$



Which of the above (λ_1 , b or x), should be a placeholder and which should be a variable? Is there anything else we should consider?

Linear Regression – Preparing Data

- ▶ In the code we create some sample data for our linear regression model.
- ▶ Notice we also specify some of the parameters for our linear regression model.



```
import matplotlib.pyplot as plt
import numpy as np

np.random.seed(10)
train_X=np.linspace(0,10,100)
noise=np.random.normal(0,1.5,100)

train_Y=((2.5*train_X)+3) +noise

plt.scatter(train_X,train_Y)
plt.legend()
plt.title('Noise data points')
plt.show()

# Parameters
learning_rate = 0.01
num_Itérations = 5000
display_step = 100
n_samples = train_X.shape[0]
```

Linear Regression – Building the Graph

- ▶ In the segment of code we set up our computational graph

```
# The placeholders that will hold the input training data provided to the graph
```

```
X = tf.placeholder(tf.float32)
```

```
Y = tf.placeholder(tf.float32)
```

```
# The parameters for the model are stored as variables in the graph
```

```
w = tf.Variable(np.random.randn(), name="weight")
```

```
b = tf.Variable(np.random.randn(), name="bias")
```

```
# The node that applies the linear regression model
```

```
pred = X*w+b
```

We create two placeholders for the single column of feature data and the corresponding target data.

We also create two variables. One for the coefficient and one for the bias.

$$\frac{1}{2m} \sum_{i=0}^m ((h(x^i) - y^i))^2$$

- Building the Graph

The placeholders that will hold the input training data

```
X = tf.placeholder(tf.float32)
```

```
Y = tf.placeholder(tf.float32)
```

The parameters for the model are stored as variables

```
w = tf.Variable(np.random.randn(), name="weight")
```

```
b = tf.Variable(np.random.randn(), name="bias")
```

The node that applies the linear regression model

```
pred = X*w+b
```

Calculate the mean squared error

```
error = pred - Y
```

```
sumSqErrors = tf.reduce_sum(tf.pow(error, 2))
```

```
cost = sumSqErrors/(2*n_samples)
```

Next we create our optimizer node. The minimize() function will modify the

variables W and b according to the gradient descent update rule

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
```

The next step is to add a node to our graph for our **cost function**. The cost function we use here is our squared error cost function. Notice we are comparing the predicted output with the actual labels Y.

Linear Regression – Building the Graph

- ▶ In the segment of code we set up our computational graph

```
# The placeholders that will hold the input tra
```

```
X = tf.placeholder(tf.float32)
```

```
Y = tf.placeholder(tf.float32)
```

```
# The parameters for the model are stored as
```

```
w = tf.Variable(np.random.randn(), name="we
```

```
b = tf.Variable(np.random.randn(), name="bias",
```

```
# The node that applies the linear regression model
```

```
pred = X*w+b
```

```
# Calculate the mean squared error
```

```
error = pred - Y
```

```
sumSqErrors = tf.reduce_sum(tf.pow(error, 2))
```

```
cost = sumSqErrors/(2*n_samples)
```

```
# Next we create our optimizer node. The minimize() function will modify the
```

```
# variables W and b according to the gradient descent update rule
```

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
```

The final step our graph is to add a gradient descent node. The node is pointed at our cost function (the cost node). It calculate the gradients of all variables in the graphs and updates them in order to decrease the cost function.

Linear Regression – Create a Session

- ▶ We have built the graph so now lets run the session.

```
# Start training  
with tf.Session() as sess:
```

```
# Initialize all variables  
sess.run(tf.global_variables_initializer())
```

Initialize all variables in the graph.

```
# Fit all training data  
for epoch in range(num_Itérations):
```

```
    sess.run(optimizer, feed_dict={X: train_X, Y: train_Y})
```

```
    if (epoch)%display_step == 0:  
        c = sess.run(cost, feed_dict={X: train_X, Y:train_Y})  
        print("Epoch Number: ", epoch, "cost=", c, " w=", sess.run(w), "b=", sess.run(b))
```

```
training_cost = sess.run(cost, feed_dict={X: train_X, Y: train_Y})  
print("Training cost=", training_cost, "w=", sess.run(w), "b=", sess.run(b), '\n')
```

Linear Regression – Create a Session

- ▶ We have built the graph so now lets run the session

```
# Start training  
with tf.Session() as sess:
```

```
# Initialize all variables  
sess.run(tf.global_variables_initializer())
```

```
# Fit all training data  
for epoch in range(num_Iterations):
```

```
    sess.run(optimizer, feed_dict={X: train_X, Y: train_Y})
```

```
    if (epoch)%display_step == 0:  
        c = sess.run(cost, feed_dict={X: train_X, Y:train_Y})  
        print("Epoch Number: ", epoch, "cost=", c, " w=", sess.run(w), "b=", sess.run(b))
```

```
training_cost = sess.run(cost, feed_dict={X: train_X, Y: train_Y})  
print("Training cost=", training_cost, "w=", sess.run(w), "b=", sess.run(b), '\n')
```

Notice we have a for loop that runs the optimizer node `num_Iterations` times. Each time it iterates it applies the gradient descent update rule to all variables in the graph (the weight `W` and the bias `b`). Notice we feed the train feature and label data into the graph which populates the `X` and `Y` tensors with values

Linear Regression – Create a Session

- ▶ We have built the graph so now lets run the s

```
# Start training  
with tf.Session() as sess:
```

```
# Initialize all variables  
sess.run(tf.global_variables_initializer())
```

```
# Fit all training data  
for epoch in range(num_Itérations):
```

```
    sess.run(optimizer, feed_dict={X: train_X, Y: train_Y})
```

```
    if (epoch)%display_step == 0:
```

```
        c = sess.run(cost, feed_dict={X: train_X, Y:train_Y})
```

```
        print("Epoch Number: ", epoch, "cost=", c, " w=", sess.run(w), "b=", sess.run(b))
```

```
training_cost = sess.run(cost, feed_dict={X: train_X, Y: train_Y})
```

```
print("Training cost=", training_cost, "w=", sess.run(w), "b=", sess.run(b), '\n')
```

Periodically, we calculate the current cost (every 10 iterations). We print out the current cost and the value of the W and b variables.

Linear Regression – Create a Session

- ▶ We have built the graph so now lets run the session.

```
# Start training  
with tf.Session() as sess:
```

```
# Initialize all variables  
sess.run(tf.global_variables_initializer())
```

```
# Fit all training data  
for epoch in range(num_Itérations):
```

```
    sess.run(optimizer, feed_dict={X: train_X, Y: train_Y})
```

```
    if (epoch)%display_step == 0:
```

```
        c = sess.run(cost, feed_dict={X: train_X, Y:train_Y})
```

```
        print("Epoch Number: ", epoch, "cost=", c, " w=", sess.run(w), "b=", sess.run(b))
```

```
training_cost = sess.run(cost, feed_dict={X: train_X, Y: train_Y})
```

```
print("Training cost=", training_cost, "w=", sess.run(w), "b=", sess.run(b), '\n')
```

When the gradient descent loop has finished iterating we calculate the final cost and output the best values of W and b .

Linear Regression – Building the Graph

- ▶ We have built the graph so now lets run the session

with `tf.Session()` as `sess`:

```
# Initialize all variables
```

```
sess.run(tf.global_variables_initializer())
```

```
# Fit all training data
```

```
for epoch in range(num_Itérations):
```

```
    _, c, wt, bias = sess.run([optimizer, cost, w, b], feed_dict={X: train_X, Y: train_Y})
```

```
    if (epoch)%display_step == 0:
```

```
        print("Epoch Number: ", epoch, "cost=", c, " w=", wt, "b=", bias)
```

```
training_cost = sess.run(cost, feed_dict={X: train_X, Y: train_Y})
```

```
print("Training cost=", training_cost, "w=", sess.run(w), "b=", sess.run(b), '\n')
```

The following is a slight variation of the code presented in the previous slide. Notice for each run of the graph we run the optimizer and cost node. This is slightly more efficient than the previous solution as in order to run the optimizer node it would have to have solved the cost node. It has already been solved so we are just capturing it.

Linear Regression – Building the Graph

- ▶ Finally, we create the Session and run the graph.
- ▶ The full working code for this example can be found [here](#).

continued from previous slide

```
predictedYValues = sess.run(w) * train_X + sess.run(b)
```

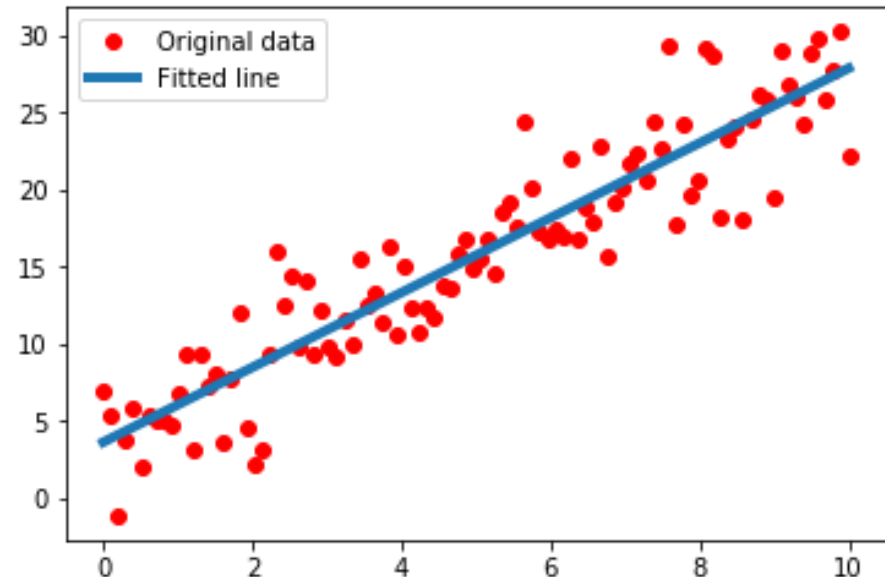
```
# Graphic display
```

```
plt.plot(train_X, train_Y, 'ro', label='Original data')
```

```
plt.plot(train_X, predictedYValues, label='Fitted line', linewidth=4.0)
```

```
plt.legend()
```

```
plt.show()
```



Visualization using TensorBoard

- ▶ TF provides a visualization tool called TensorBoard that allows you to visualize and inspect the graph you have created but also visualize useful information about your model.
- ▶ TensorBoard operates by reading **TensorFlow event files**, which contain summary data that you can generate when running TensorFlow.
- ▶ Please note the following slides will use Datalab to illustrate the use of TensorBoard.



Fit to screen



Download PNG

Run

C:\...

(1)

Session

runs (0)

Upload

Choose File



Trace inputs

Color

Structure

Device

▼ Close legend.

Graph (* = expandable)



Namespace* 2



OpNode 2



Unconnected series* 2



Connected series* 2



Constant 2



Summary 2



Dataflow edge 2

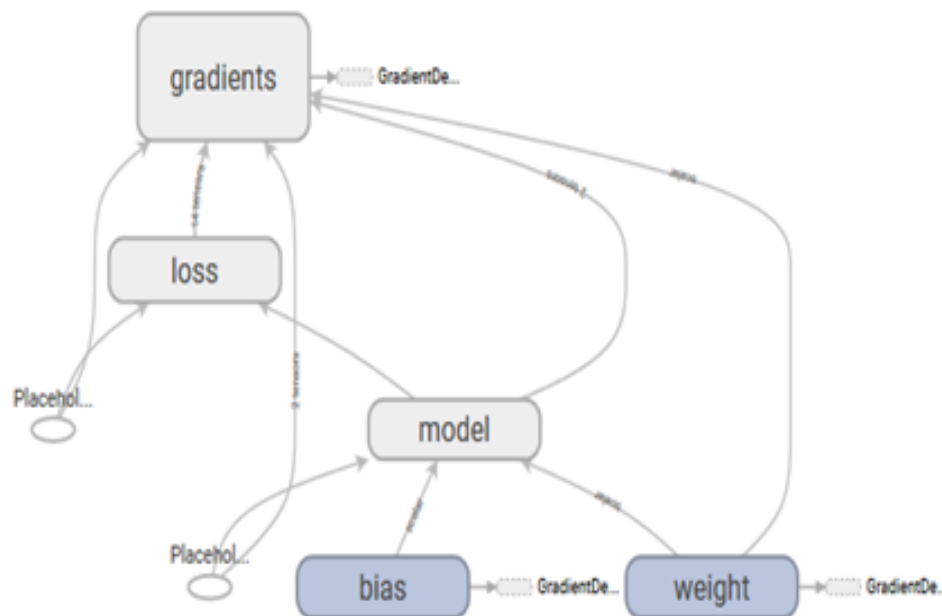


Control dependency edge 2



Reference edge 2

Main Graph



Auxiliary Nodes



Starting TensorBoard

- ▶ To get TensorBoard up and running for Datalab you simply have to import TensorBoard (first line below)
- ▶ The second line starts TensorBoard and points it at the ./logs folder as it's root folder.

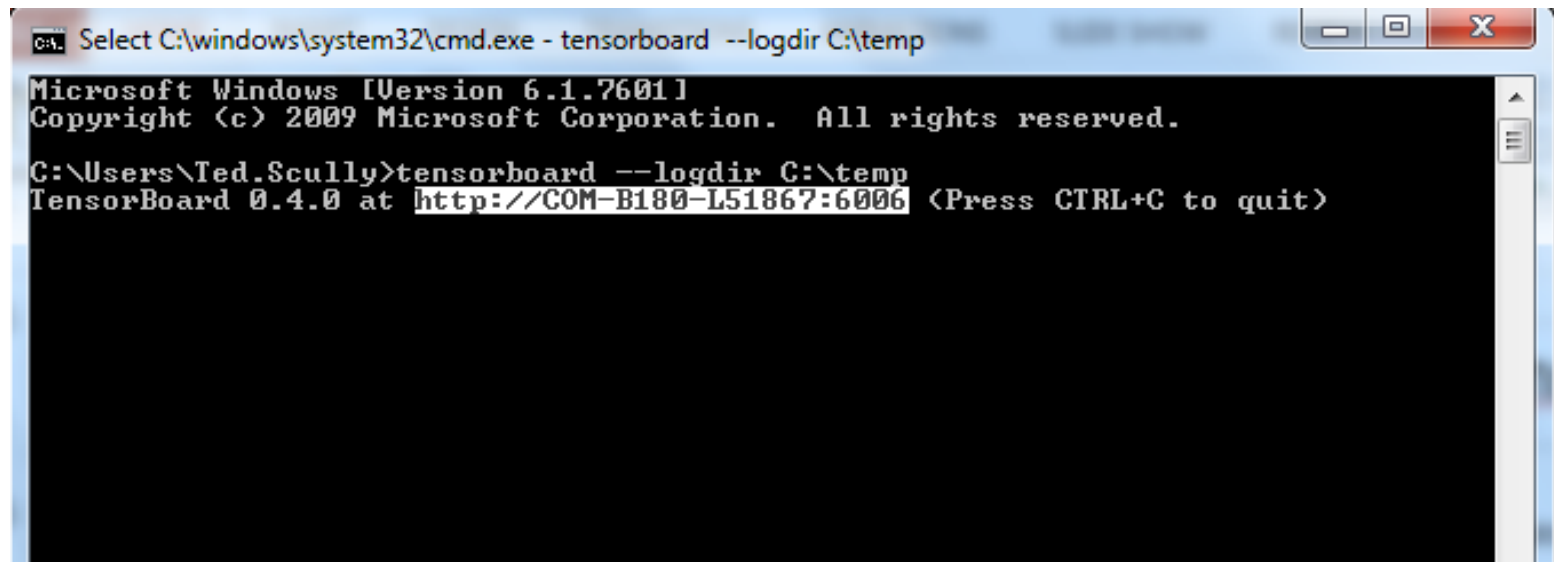
```
from google.datalab.ml import TensorBoard as tb  
tb.start('./logs')
```

- ▶ When you run the code below you should get the following output:
- ▶ TensorBoard was started successfully with pid 3635. Click [here](#) to access it.

Getting Tensorboard working with Colab is a little more involved but there is a good tutorial [here](#).

Starting TensorBoard From a Windows Console

- ▶ If you have TensorFlow installed locally then you should be able to call TensorBoard from the command line as follows:
- ▶ Once you have some log files generated then you can use TensorBoard by pointing at the log directory:
 - ▶ First open up a command terminal and type the following (in this example we assume the log files are written to C:\temp):
 - ▶ **tensorboard --logdir C:\temp**
- ▶ The output will provide you with a URL (highlighted below) that you can use to access Tensorboard.



```
CA: Select C:\windows\system32\cmd.exe - tensorboard --logdir C:\temp
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Ted.Scully>tensorboard --logdir C:\temp
TensorBoard 0.4.0 at http://COM-B180-L51867:6006 (Press CTRL+C to quit)
```

Using TensorBoard with Datalab

- ▶ To help organize the various event files that are generated from your code I suggest you use the following code.
- ▶ This will define a timestamped directory within the ./logs folder.
- ▶ Each time you run your code generating the TensorBoard output you should point the output at this timestamped directory.
- ▶ This keeps everything well organized into separate folders (otherwise TensorBoard can end up collating information from multiple different runs)

```
from datetime import datetime

currentTime =
datetime.utcnow().strftime("%Y%m%d%H%M%S")
root_logdir = "./logs"
logdir = "{}{}run-{}{}".format(root_logdir, currentTime)
```


No dashboards are active for the current data set.

Probable causes:

- You haven't written any data to your event files.
- TensorBoard can't find your event files.

If you're new to using TensorBoard, and want to find out how to add data and set up your event files, check out the [README](#) and perhaps the [TensorBoard tutorial](#).

If you think TensorBoard is configured properly, please see [the section of the README devoted to missing data problems](#) and consider filing an issue on GitHub.

Last reload: Fri Mar 15 2019 14:17:29 GMT+0000 (Greenwich Mean Time)

Data location: ./logs

When you run TensorBoard first you may get the following output. The reason is that you have generated any event files yet. Let's now look at how to do generate event files.

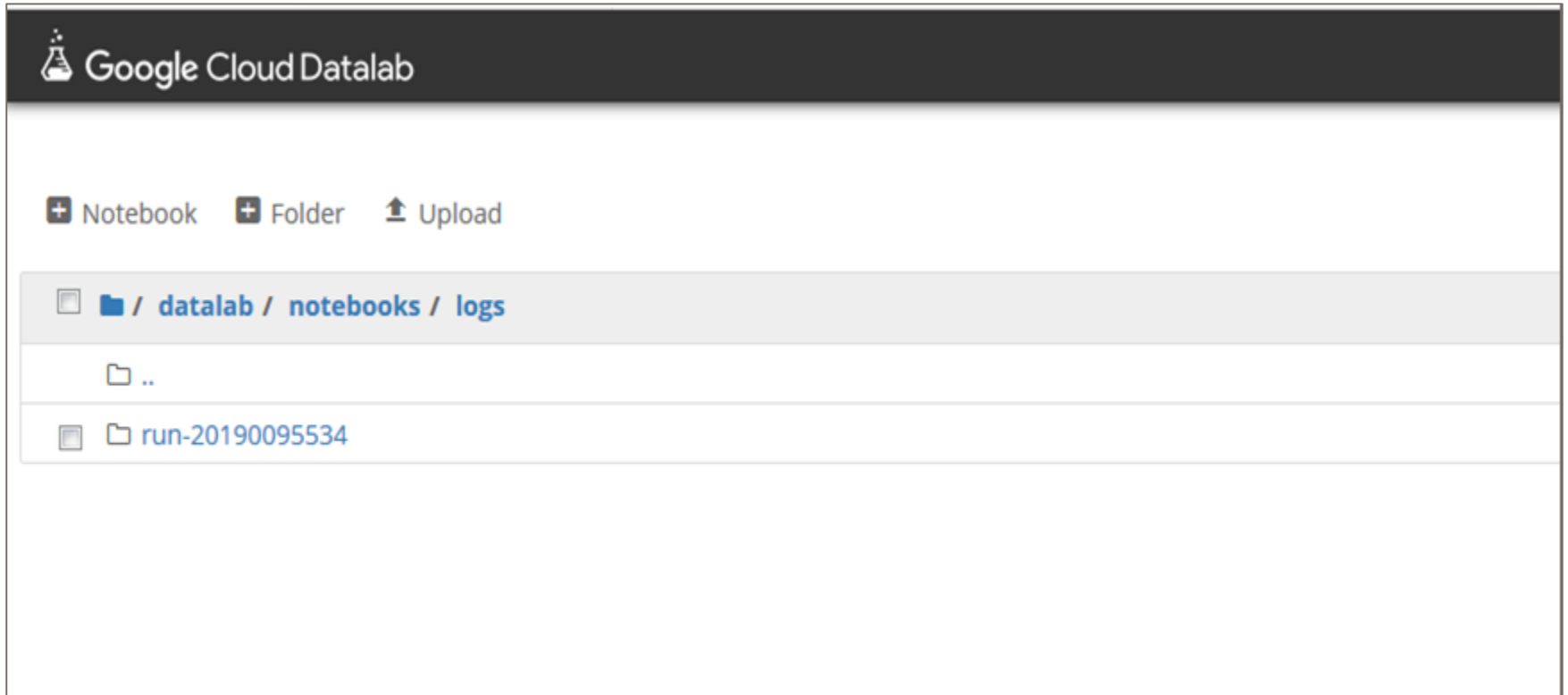
Visualization using TensorBoard

- ▶ TensorFlow provides a **tf.summary.FileWriter** class that allows us to write data (event files), which can then be read by TensorBoard.
- ▶ For example, if we want to **visualise the graph** for the linear regression example we can do the following (within our Session block).
- ▶ Note the first argument to the FileWriter is the location that you want it to store the log files, which creates the timestamped folder in our log directory.

with tf.Session() as sess:

```
# op to write data to Tensorboard
summary_writer = tf.summary.FileWriter(logdir, graph=tf.get_default_graph())
summary_writer.add_graph(sess.graph)
```

After running the session outlined in the previous slide you should now notice a new directory appear in your root directory (./logs). If you return to TensorBoard and refresh the page you should now be able to see the graph you built previously (see next slide).



- Fit to screen
- Download PNG

Run run-201903090...

Session runs (0)

Upload Choose File

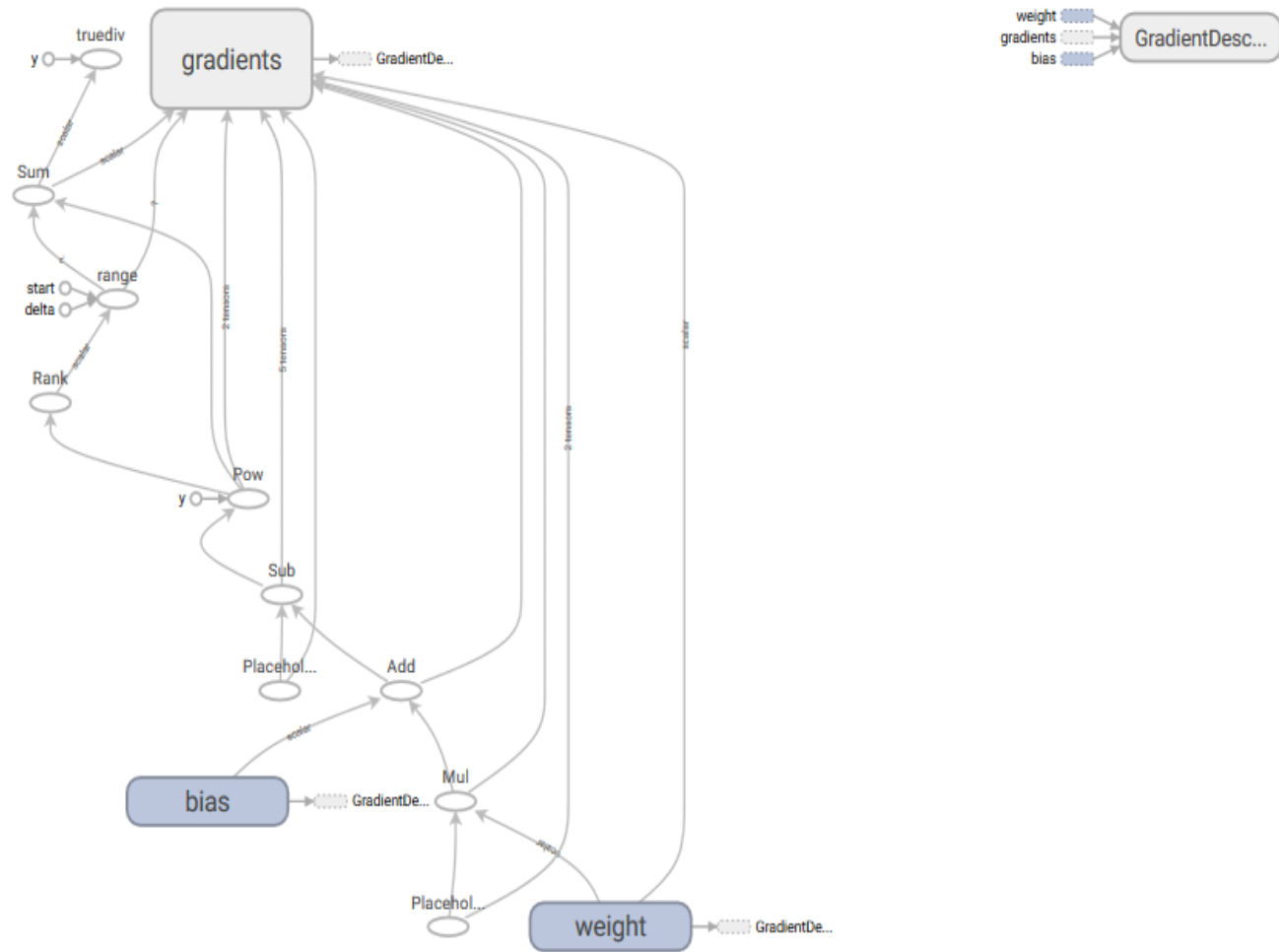
Trace inputs

Color Structure Device

Close legend.

Graph (* = expandable)

- Namespace* ?
- OpNode ?
- Unconnected series* ?
- Connected series* ?
- Constant ?
- Summary ?
- Dataflow edge ?
- Control dependency edge ?
- Reference edge ?



Visualization using TensorBoard

- ▶ Each time you run your TensorFlow code you should be generating a new value for logdir variable.
- ▶ Therefore, the event data for each run is being written to a different underlying directory.
- ▶ Therefore, the structure might look like
 - ▶ ->logs
 - ▶ -----> run-20190119082635
 - ▶ -----> run-20190119082896
- ▶ Over the next few slides we will look at how to effectively organize the components of a graph.

.

```
from datetime import datetime
currentTime = datetime.utcnow().strftime("%Y%m%d%H%M%S")
root_logdir = "./logs"
logdir = "{} /run-{} /".format(root_logdir, currentTime)
```

```
tf.reset_default_graph()
```

```
X = tf.placeholder(tf.float32)
Y = tf.placeholder(tf.float32)
```

```
w = tf.Variable(np.random.randn(), name="weight")
b = tf.Variable(np.random.randn(), name="bias")
```

```
# The node that applies the linear regression model
pred = X*w+b
```

```
# Start training
with tf.Session() as sess:
```

```
# op to write logs to Tensorboard
summary_writer = tf.summary.FileWriter(logdir, graph=tf.get_default_graph())
summary_writer.add_graph(sess.graph)
```

In this example we just include a portion of the code for linear regression and generate the graph.

```
from datetime import datetime
currentTime = datetime.utcnow().strftime("%Y%m%d%H%M%S")
root_logdir = "./logs"
logdir = "{} /run-{}".format(root_logdir, currentTime)
```

```
tf.reset_default_graph()
```

```
X = tf.placeholder(tf.float32)
```

```
Y = tf.placeholder(tf.float32)
```

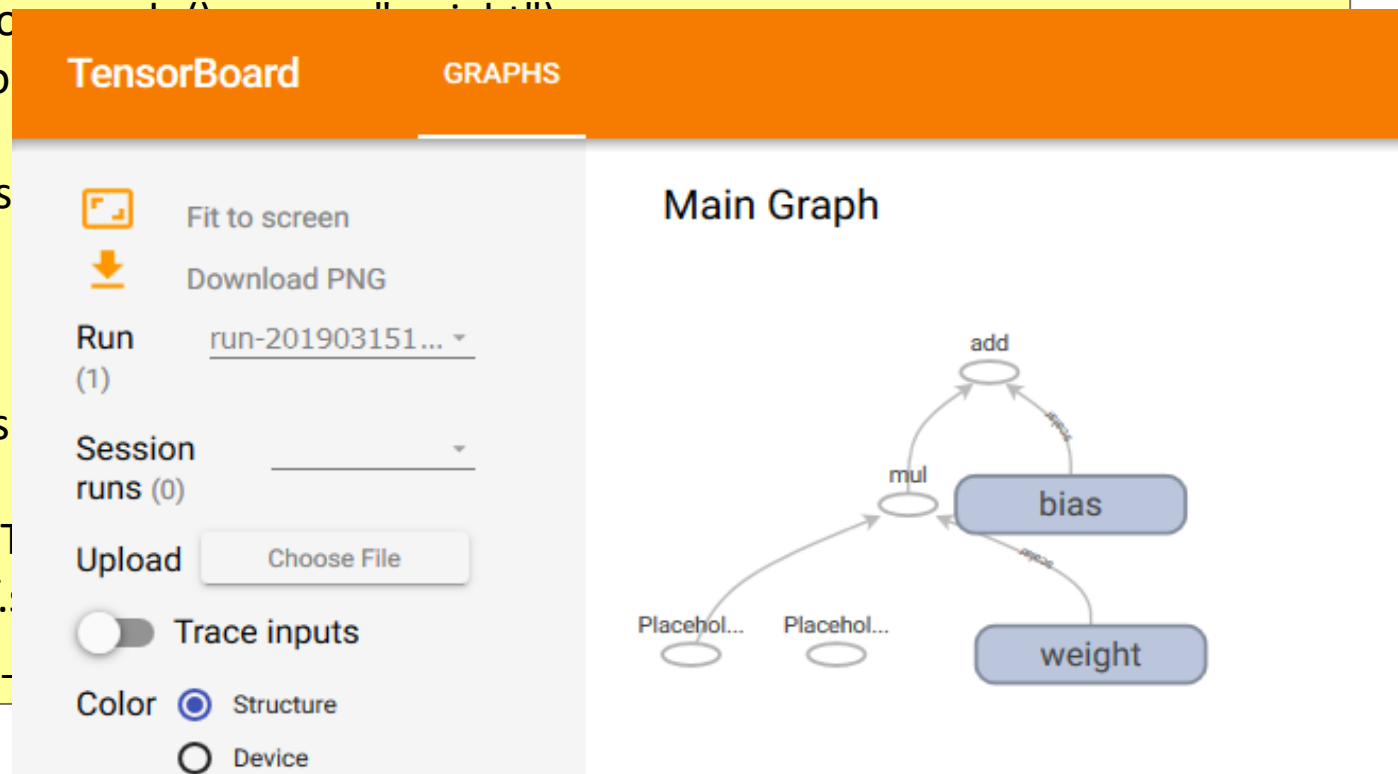
```
w = tf.Variable(np.random.randn(1000, 1000))
```

```
b = tf.Variable(np.random.randn(1000, 1))
```

```
# The node that applies the softmax operation
pred = X*w+b
```

```
# Start training
with tf.Session() as sess:
```

```
# op to write logs to TensorBoard
summary_writer = tf.summary.FileWriter(logdir)
summary_writer.add_summary(pred, 0)
```



```
from datetime import datetime
```

```
currentTime = datetime.utcnow().strftime("%Y%m%d%H%M%S")
```

```
root_logdir = "./logs"
```

```
logdir = "{}{}run-{}".format(root_logdir, currentTime)
```

```
tf.reset_default_graph()
```

```
X = tf.placeholder(tf.float32)
```

```
Y = tf.placeholder(tf.float32)
```

```
w = tf.Variable(np.random.randn(), name="weight")
```

```
b = tf.Variable(np.random.randn(), name="bias")
```

```
# The node that applies the linear regression model
```

```
pred = X*w+b
```

```
# Calculate the mean squared error
```

```
error = pred - Y
```

```
sqError = tf.pow(error, 2)
```

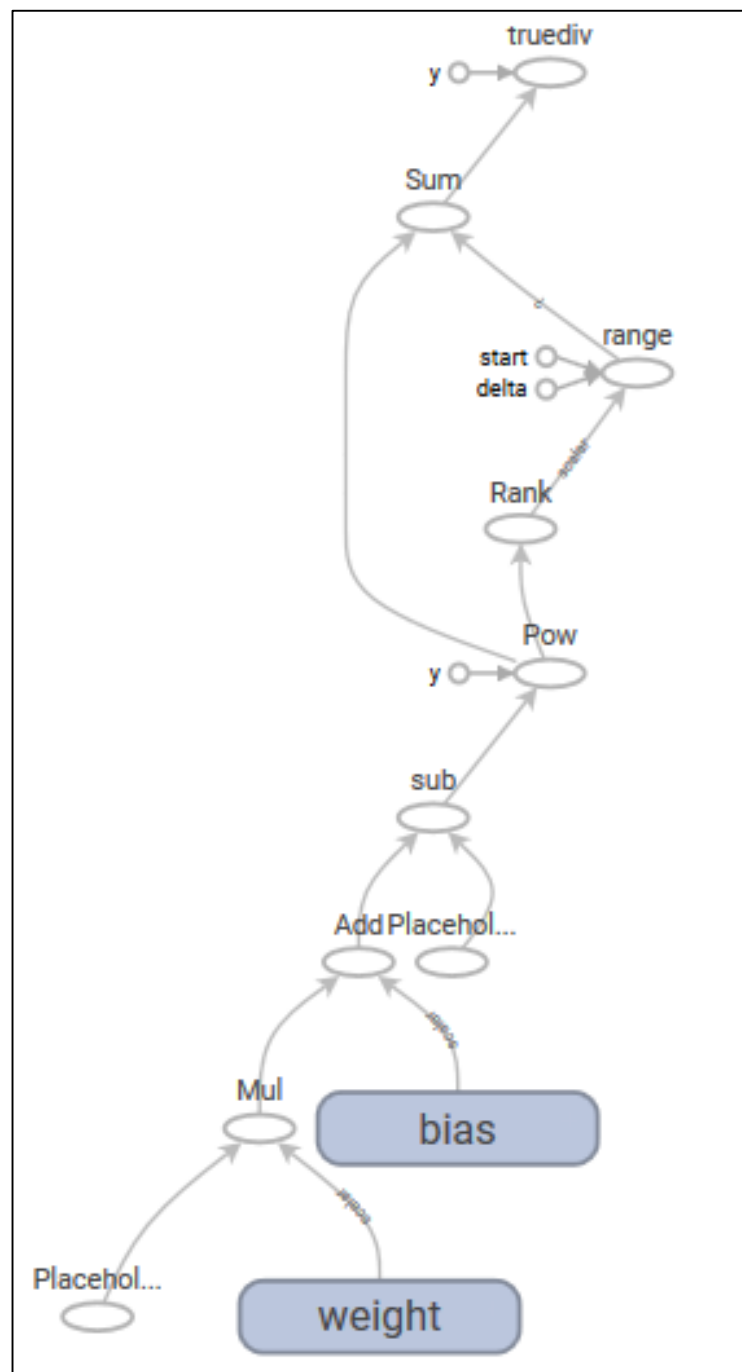
```
sumSqErrors = tf.reduce_sum(sqError)
```

```
cost = sumSqErrors/(2*n_samples)
```

```
# Start training (same as previous slide)
```

```
with tf.Session() as sess:
```

Next insert the code to calculate the mean squared error. Let's see how that impacts our graph



```
tf.reset_default_graph()
```

```
X = tf.placeholder(tf.float32)
```

```
Y = tf.placeholder(tf.float32)
```

```
w = tf.Variable(np.random.randn(), name="weight")
```

```
b = tf.Variable(np.random.randn(), name="bias")
```

```
# The node that applies the linear regression model
```

```
pred = X*w+b
```

```
# Calculate the mean squared error
```

```
error = pred - Y
```

```
sqError = tf.pow(error, 2)
```

```
sumSqErrors = tf.reduce_sum(sqError)
```

```
cost = sumSqErrors/(2*n_samples)
```

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
```

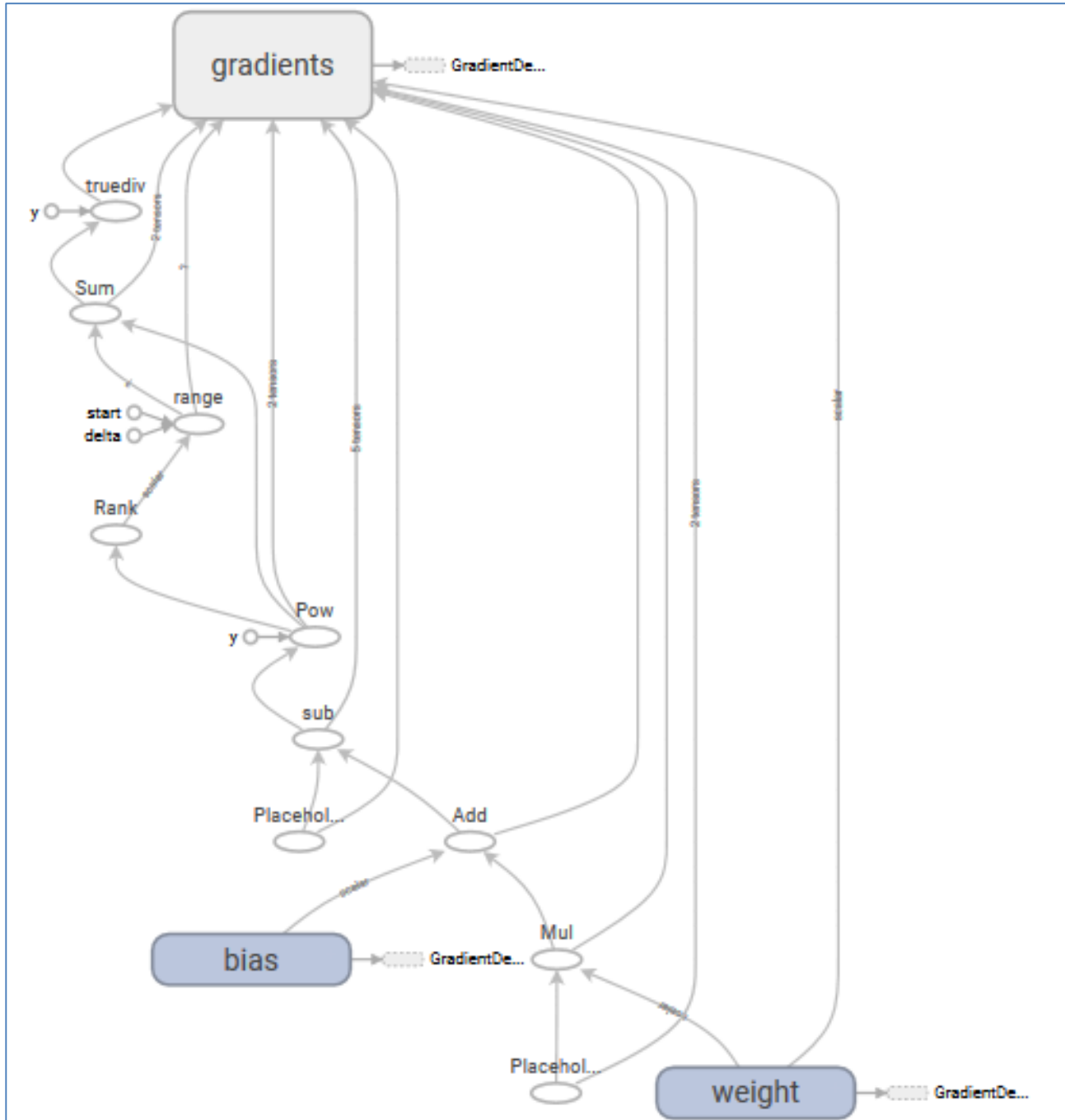
```
# Start training
```

```
with tf.Session() as sess:
```

```
# op to write logs to Tensorboard
```

```
summary_writer = tf.summary.FileWriter(logdir, graph=tf.get_default_graph())
```

Next add the gradient descent
node.



One option is that I could add a name to each of the operations that I define to improve readability. However, this is not going to lessen the complexity of the graph.

Name Scopes

- ▶ As can see the graph can quickly become cluttered. This is just for linear regression! There can be thousands of *nodes* for neural network models.
- ▶ To avoid this, TF allows you to create **name scopes**, which allows you to **group related nodes** under a single heading.
- ▶ In the example below we create a name scope called **loss** and group all operations involved in calculating the mean squared error.

```
.....  
# The node that applies the linear regression model  
pred = X*w+b  
  
# Calculate the mean squared error  
  
with tf.name_scope("loss") as scope:  
  
    error = pred - Y  
    sqError = tf.pow(error, 2, name="power")  
    sumSqErrors = tf.reduce_sum(sqError)  
    cost = sumSqErrors/(2*n_samples)  
    print (cost)  
.....
```

Name Scopes

- ▶ The name of every operation defined within the scope is now prefixed with "loss/".
- ▶ For example if I was to print out cost below it would print the following
- ▶ `Tensor("loss/truediv:0", shape=(), dtype=float32)`
- ▶ Of course I can also give each op a specific name if I wish.

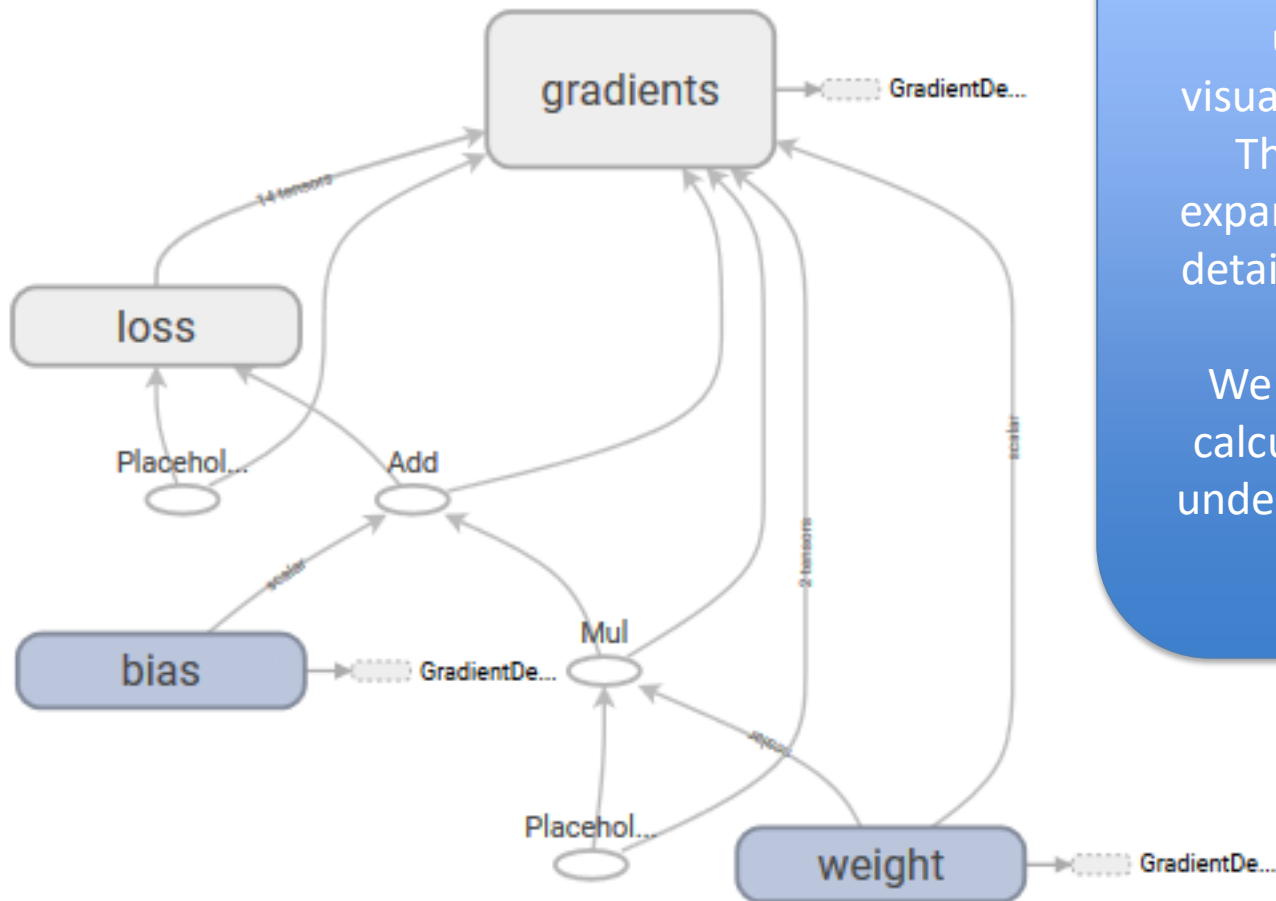
```
.....  
# The node that applies the linear regression model  
pred = X*w+b  
  
# Calculate the mean squared error  
  
with tf.name_scope("loss") as scope:  
  
    error = pred - Y  
    sqError = tf.pow(error, 2, name="power")  
    sumSqErrors = tf.reduce_sum(sqError)  
    cost = sumSqErrors/(2*n_samples)  
    print(cost)  
.....
```

Name Scopes

- ▶ You will have noticed in the previous slides that TensorFlow will automatically name each of the ops. In the code below we provide our own name for each of the nodes.
- ▶ In TensorBoard, the error, sumSquareErrors and cost now appear inside the loss namespace, which appears collapsed by default

with `tf.name_scope("loss")` as scope:

```
error = tf.subtract(pred, Y, name="error")
sqError = tf.pow(error, 2, name="power")
sumSqErrors = tf.reduce_sum(sqError, name="sumError")
cost = tf.div(sumSqErrors, (2*n_samples), name="meanSquaredError")
```



Notice the name scope helps us to simplify the visualization of our model.

The loss node can be expanded to provide more detail just by clicking on it.

We could also place the calculations of the model under its own namespace.

The placeholders that will hold the input training data provided to the graph

```
X = tf.placeholder(tf.float32)
```

```
Y = tf.placeholder(tf.float32)
```

The parameters for the model are stored as variables in the graph

```
w = tf.Variable(np.random.randn(), name="weight")
```

```
b = tf.Variable(np.random.randn(), name="bias")
```

The node that applies the linear regression model

```
with tf.name_scope("model") as scope:
```

```
    pred = X*w+b
```

Let's tidy it up further by
adding a name scope for
calculating the
predictions

```
with tf.name_scope("loss") as scope:
```

```
    error = tf.subtract(pred, Y, name="error")
```

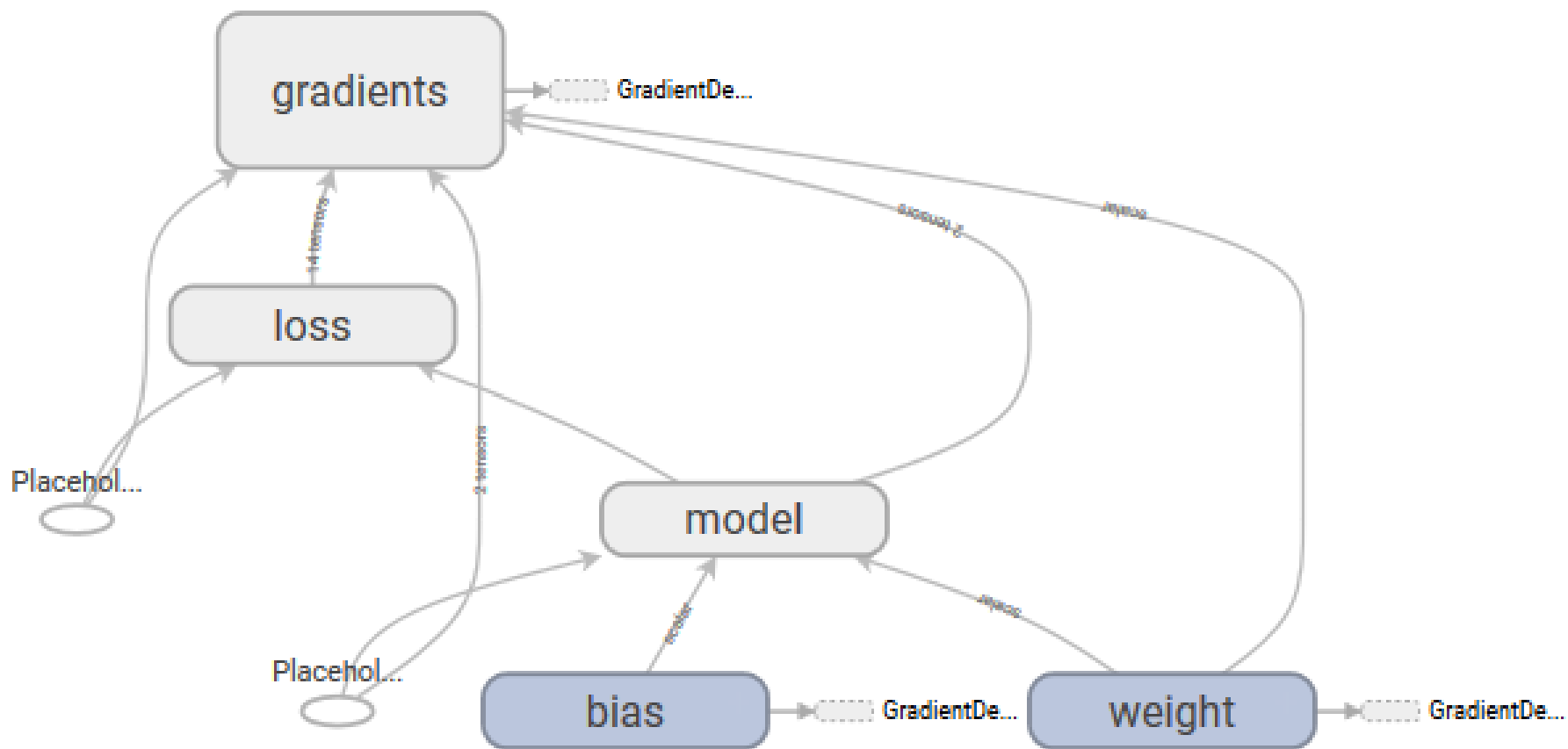
```
    sqError = tf.pow(error, 2, name="power")
```

```
    sumSqErrors = tf.reduce_sum(sqError, name="sumError")
```

```
    cost = tf.div(sumSqErrors,(2*n_samples), name="meanSquaredError")
```

```
    print (cost)
```

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
```

Performance Graphs in TensorBoard

- ▶ You can also easily generate performance graphs in TensorBoard.
- ▶ TB provides a number of **ops** that allow us to write data about our model and various aspects of its performance to the event logs, which can in turn be depicted as graphs.
- ▶ The most widely used is **tf.summary.scalar** .
- ▶ You can attach a summary.scalar node to any node that you want to collect data from. For example, in linear regression we might be interested in looking at the behaviour of our cost function as gradient descent iterations.
- ▶ The final line creates a node in the graph. When this summary node is run in a Session it will evaluate the value of the cost node and write it to a **TensorBoard-compatible binary log string**.

with tf.name_scope("loss") as scope:

```
error = tf.subtract(pred, Y, name="error")
sqError = tf.pow(error, 2, name="power")
sumSqErrors = tf.reduce_sum(sqError, name="sumError")
cost = tf.div(sumSqErrors,(2*n_samples), name="meanError")
```

```
mseSummary = tf.summary.scalar("Mean_Error", cost)
```

Graphs in TensorBoard

- ▶ There are two changes to note below. Notice we are running the scalar summary node `mseSummary` and capturing what it returns in `currentMse`.
- ▶ The `FileWriter` writes this summary to the log file in the log directory.

```
# Start training
```

```
with tf.Session() as sess:
```

```
    # op to write logs to Tensorboard
```

```
    summary_writer = tf.summary.FileWriter(logdir, graph=tf.get_default_graph())
```

```
    summary_writer.add_graph(sess.graph)
```

```
    # Initialize all variables
```

```
    sess.run(tf.global_variables_initializer())
```

```
    # Fit all training data
```

```
    for epoch in range(num_Itérations):
```

```
        _, c, currentMse = sess.run([optimizer, cost, mseSummary], feed_dict={X: train_X, Y: train_Y})
```

```
        summary_writer.add_summary(currentMse, epoch)
```

Graphs in TensorBoard

- ▶ There are two changes to note below. Notice we are running the scalar summary node `mseSummary` and capturing what it returns in `currentMse`.
- ▶ The `FileWriter` writes this summary to the log file in the log directory.

```
# Start training  
with tf.Session() as sess:
```

```
    # op to write logs to Tensorboard  
    summary_writer = tf.summary.FileWriter(log_dir)  
    summary_writer.add_graph(sess.graph)
```

```
    # Initialize all variables  
    sess.run(tf.global_variables_initializer())
```

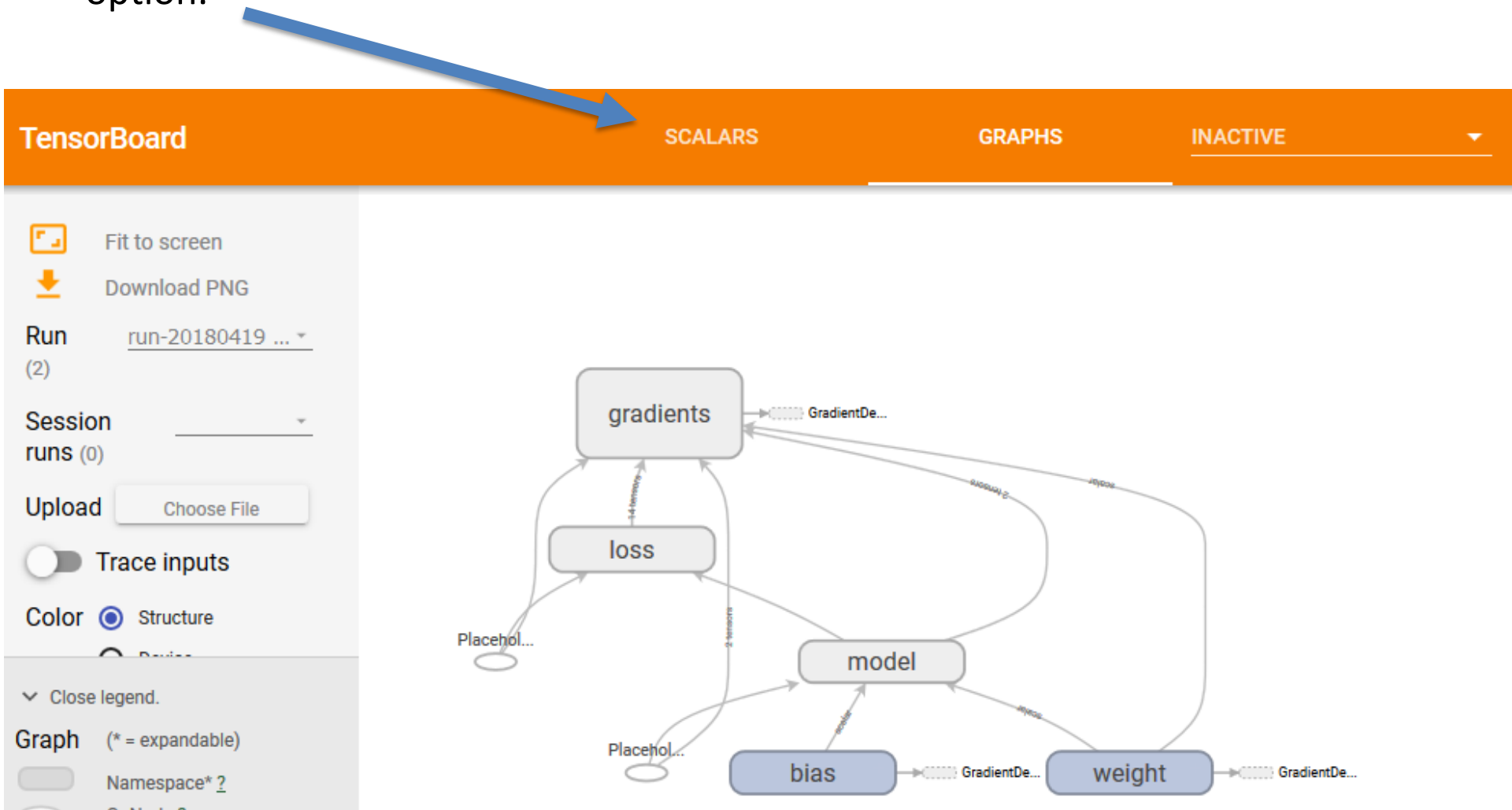
```
    # Fit all training data  
    for epoch in range(num_Itérations):
```

```
        _, c, currentMse = sess.run([optimizer, cost, mseSummary], feed_dict={X: train_X, Y: train_Y})  
        summary_writer.add_summary(currentMse, epoch)
```

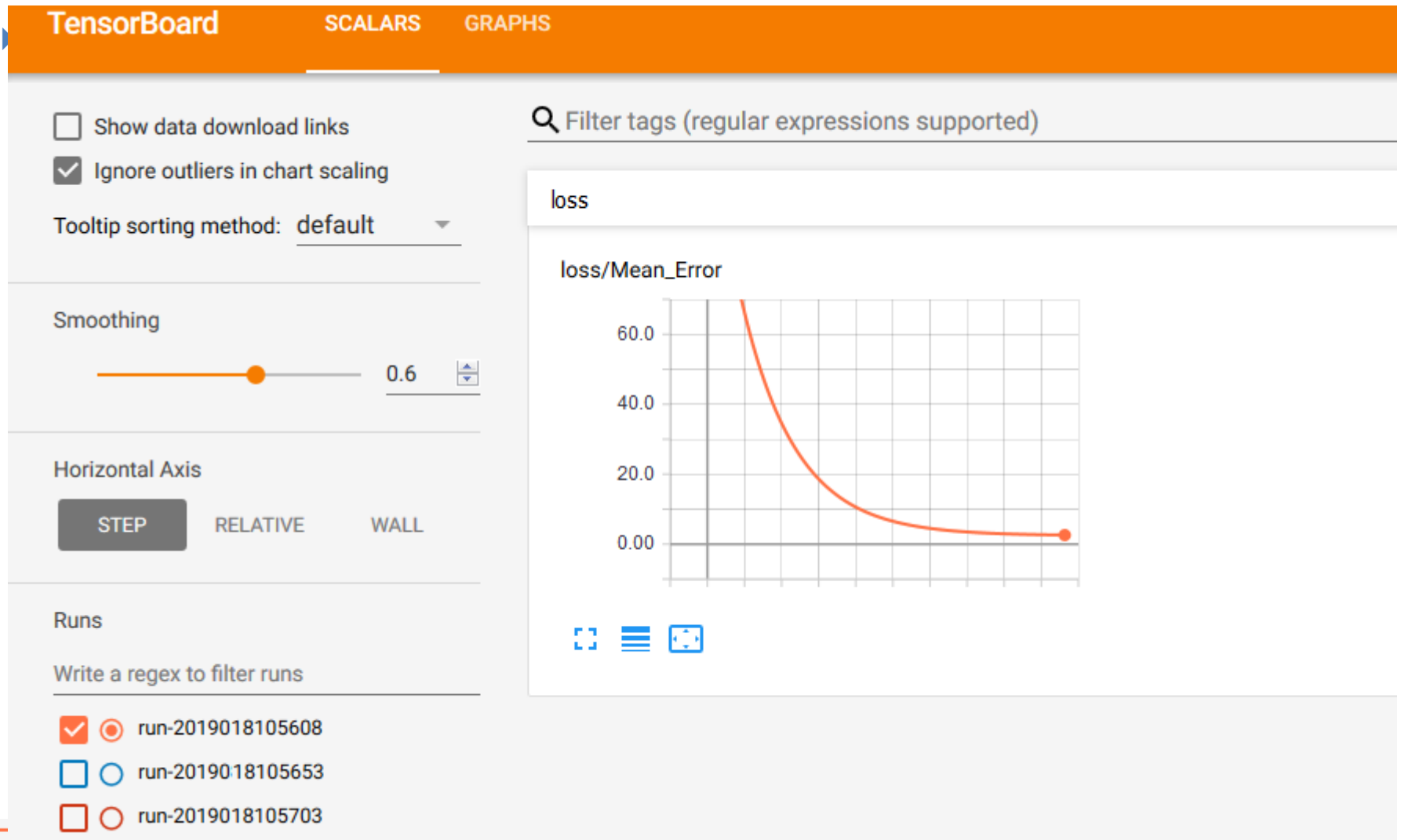
You will notice in the code that we are writing a new value for the MSE for every single iteration of gradient descent. For more computationally intensive models this is not advised as it can slow down training. Instead you should write summary data periodically. (For every 10 or 20 iterations of your optimizer)

Graphs in TensorBoard

- ▶ When you run your Session and reload TensorBoard you should see a SCALARS option.



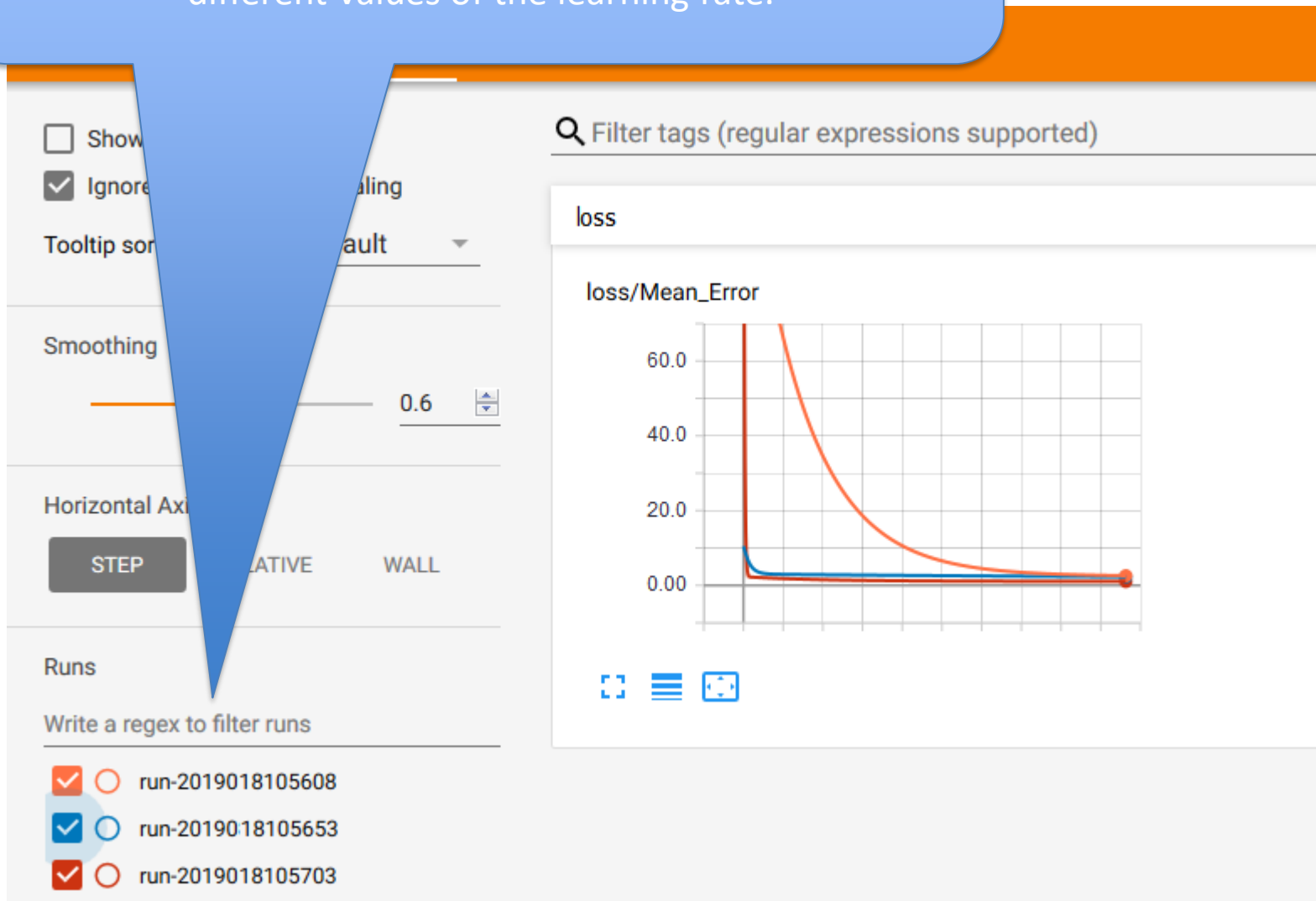
Graphs in TensorBoard



Graphs in TensorBoard

Notice we get a list of the available runs currently contained in the log directory. We can select the specific runs we are interested in. Below I have selected a number of runs using different values of the learning rate.

generated for the MSE.



Multiple Graphs in TensorBoard

- ▶ You can add as many summary ops as you want in your code. For example, if I want to generate a summary graph that depicts the update to weights. I could have the following.

```
W = tf.Variable(np.random.randn(), name="weight")  
  
# .....  
  
mseSummary = tf.summary.scalar("Mean_Squared_Error", cost)  
weightSummary = tf.summary.scalar("weight", w)
```

Please note that this is for linear regression. Therefore, there is only one weight (hence it is a scalar value)

Multiple Graphs in TensorBoard

- ▶ Notice we run both the mse and weight summary nodes during the session. We store what they return in currentMse and currentW.
- ▶ The FileWriter then writes this information to file.

with tf.Session() as sess:

```
# op to write logs to Tensorboard
```

```
summary_writer = tf.summary.FileWriter(logdir, graph=tf.get_default_graph())
```

```
summary_writer.add_graph(sess.graph)
```

```
# Initialize all variables
```

```
sess.run(tf.global_variables_initializer())
```

```
# Fit all training data
```

```
for epoch in range(num_Itérations):
```

```
_, c, currentMse, currentW = sess.run([optimizer, cost, mseSummary, weightSummary],  
feed_dict={X: train_X, Y: train_Y})
```

```
summary_writer.add_summary(currentMse, epoch)
```

```
summary_writer.add_summary(currentW, epoch)
```



- ☐ Show data download links
- ☒ Ignore outliers in chart scaling

Tooltip sorting method: default ▼

Smoothing



Horizontal Axis

STEP

RELATIVE

WALL

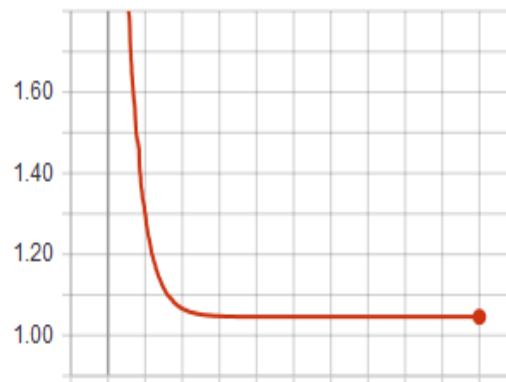
Runs

Write a regex to filter runs

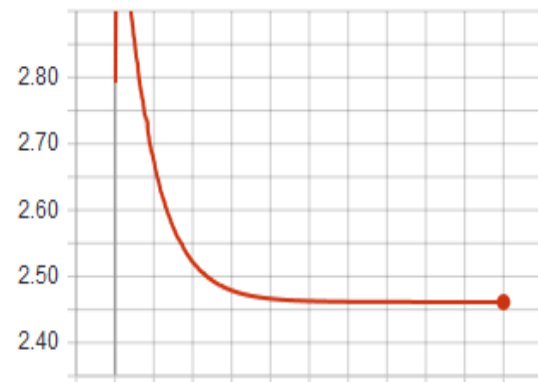
Q.*

Tags matching /.*/ (all tags)

Mean_Squared_Error



Weights



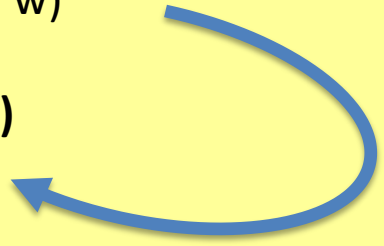
Multiple Graphs in TensorBoard

- ▶ The drawback of what we see in the previous slide is that it can get very repetitive and tedious if we have to add many summary nodes.
- ▶ TF provide **`tf.summary.merge_all()`** which merges all summaries used in the default graph. Now we can run the summary node once.
- ▶ An example is shown on the next slide.

Multiple Graphs in TensorBoard

```
mseSummary = tf.summary.scalar("Mean_Squared_Error", cost)
weightSummary = tf.summary.scalar("weight", w)
```

```
summaryOperations = tf.summary.merge_all()
```



```
# Start training
with tf.Session() as sess:
```

```
    # op to write logs to Tensorboard
    summary_writer = tf.summary.FileWriter(logdir, graph=tf.get_default_graph())
    summary_writer.add_graph(sess.graph)
```

```
    # Initialize all variables
    sess.run(tf.global_variables_initializer())
```

```
    # Fit all training data
    for epoch in range(num_Itérations):
```

```
        _, c, summaryResults = sess.run([optimizer, cost, summaryOperations], feed_dict={X:
train_X, Y: train_Y})
        summary_writer.add_summary(summaryResults, epoch)
```

Multiple Graphs in TensorBoard

```
mseSummary = tf.summary.scalar("Mean_Squared_Error", cost)
```

```
weightSummary = tf.summary.scalar("Weights", W)
```

```
summaryOperations = tf.summary.merge_all()
```

```
# Start training
```

```
with tf.Session() as sess:
```

```
    # op to write logs to Tensorboard
```

```
    summary_writer = tf.summary.FileWriter(logdir, graph=tf.get_default_graph())
```

```
    summary_writer.add_graph(sess.graph)
```

```
    # Initialize all variables
```

```
    sess.run(tf.global_variables_initializer())
```

```
    # Fit all training data
```

```
    for epoch in range(num_Itérations):
```

```
        _, c, summaryResults = sess.run([optimizer, cost, summaryOperations], feed_dict={X:  
train_X, Y: train_Y})
```

```
        summary_writer.add_summary(summaryResults, epoch)
```

Notice we only run the merged
summary node and not each
individual summary node.

Multiple Graphs in TensorBoard

```
mseSummary = tf.summary.scalar("Mean_Squared_Error", cost)
weightSummary = tf.summary.scalar("Weights", W)
```

```
summaryOperations = tf.summary.merge_all()
```

```
# Start training
with tf.Session() as sess:
```

```
    # op to write logs to Tensorboard
    summary_writer = tf.summary.FileWriter(logdir, graph=tf.get_default_graph())
    summary_writer.add_graph(sess.graph)
```

```
    # Initialize all variables
    sess.run(tf.global_variables_initializer())
```

```
    # Fit all training data
    for epoch in range(num_Itérations):
```

```
        _, c, summaryResults = sess.run([optimizer, cost, summaryOperations], feed_dict={X:
train_X, Y: train_Y})
        summary_writer.add_summary(summaryResults, epoch)
```

Full Code for Linear Regression
example available on Canvas.