UiO : Department of Informatics
University of Oslo

INF 5860 Machine learning for image classification

Lecture 5 : Introduction to TensorFlow

Tollef Jahren

February 14, 2018

OUTLINE

- Deep learning frameworks
- TensorFlow
  - TensorFlow graphs
  - TensorFlow session
  - TensorFlow constants
  - TensorFlow variables
  - TensorFlow feeding data to the graph
  - Tensorboard
  - TensorFlow save/restore models
  - TensorFlow example

ABOUT TODAY

- You will get an introduction to one of the most widely used deep learning frameworks
- The goal is for you to be familiar with TensorFlow's computational graph
- Understand how to use basic tensors and operator

- TensorFlow version: 1.4
- Python: 3.6

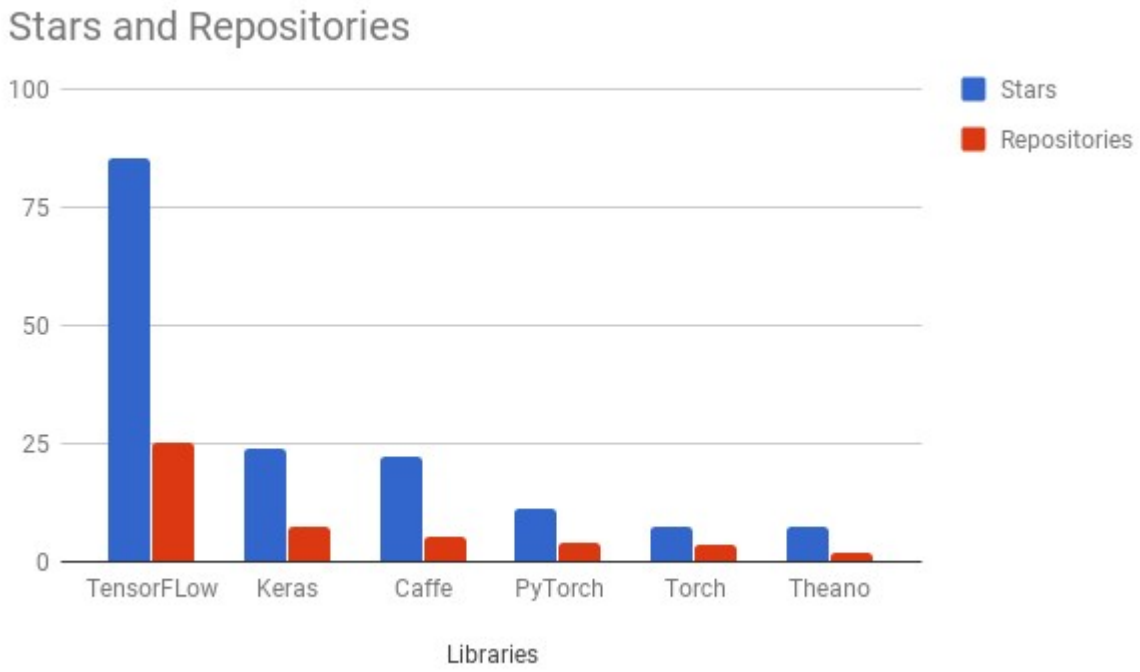
Readings

- https://www.tensorflow.org/
- https://web.stanford.edu/class/cs20si/2017/syllabus.html [lecture notes/slides 1-5]
- https://www.youtube.com/channel/UCMq6IdbXar_KtYixMS_wHcQ/videos

Progress

- **Deep learning frameworks**
- TensorFlow
  - TensorFlow graphs
  - TensorFlow session
  - TensorFlow constants
  - TensorFlow variables
  - TensorFlow feeding data to the graph
  - Tensorboard
  - TensorFlow save/restore models
  - TensorFlow example

Popularity


Stars and Repositories
100
75
50
25
0
Stars
Repositories
TensorFLow
Keras
Caffe
PyTorch
Torch
Theano
Libraries

Why do we need Deep learning frameworks?

- **Speed:**
  - Fast GPU/CPU implementation of matrix multiplication, convolutions and backpropagation
- **Automatic differentiations**:
  - Pre-implementation of the most common functions and it's gradients.
- **Reuse:**
  - Easy to reuse other people's models
- **Less error prone:**
  - The more code you write yourself, the more errors

Deep learning frameworks

Deep learning frameworks does a lot of the complicated computation, remember last week….

$$z_k^{[L]} = \sum_{j=1}^{n^{[L-1]}} w_{jk}^{[L]} a_j^{[L-1]} + b_k^{[L]}$$

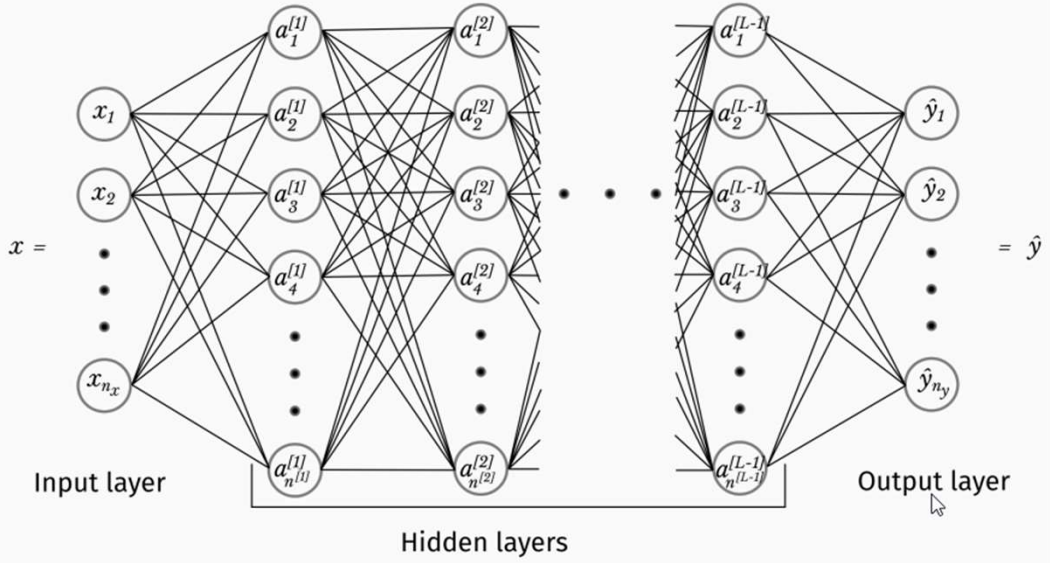
$$a_k^{[L]} = s(z_k^{[L]})$$

$$= \hat{y}_k$$

for

$$k = 1, \ldots, n_y,$$

$$= 1, \ldots, n^{[L]}.$$



Progress

- Deep learning frameworks
- **TensorFlow**
  - TensorFlow graphs
  - TensorFlow session
  - TensorFlow constants
  - TensorFlow variables
  - TensorFlow feeding data to the graph
  - Tensorboard
  - TensorFlow Save/restore models
  - TensorFlow example

Why TensorFlow

- Python API
- Can use CPU, GPU
- Supports many platforms:
  - Raspberry Pi, Android, Windows, iOS, Linux

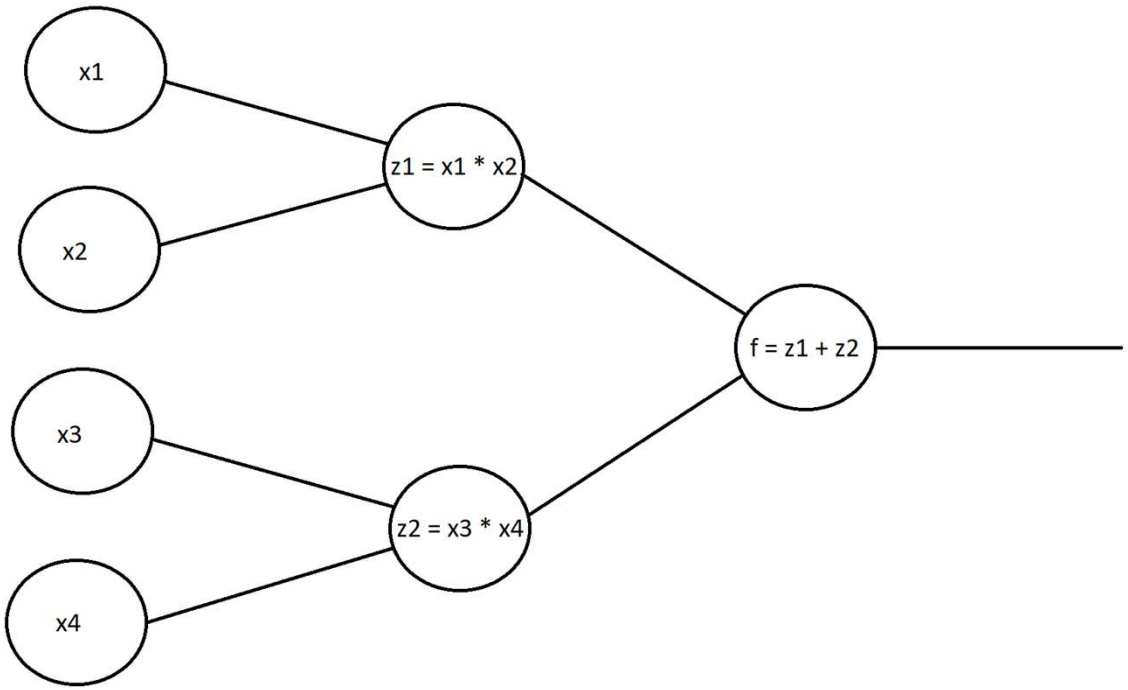
- Companies using TensorFlow:
  - Nvidia, Uber, ebay, snapchat, google, Airbnb, twitter and many more

Progress

- Deep learning frameworks
- TensorFlow
  - **TensorFlow graphs**
  - TensorFlow session
  - TensorFlow constants
  - TensorFlow variables
  - TensorFlow feeding data to the graph
  - Tensorboard
  - TensorFlow Save/restore models
  - TensorFlow example

What is a computational graph?

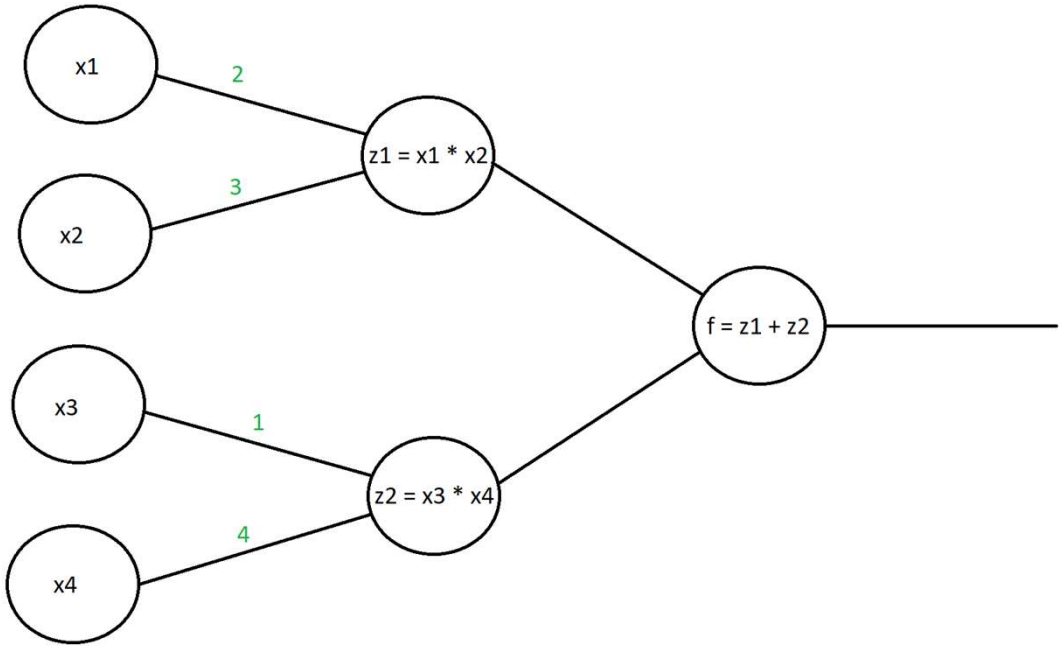
$$f(\vec{x}) = x_1 * x_2 + x_3 * x_4 \qquad f(\vec{x}) = z_1 + z_2$$



Forward propagation

$$f(\vec{x}) = x_1 * x_2 + x_3 * x_4$$

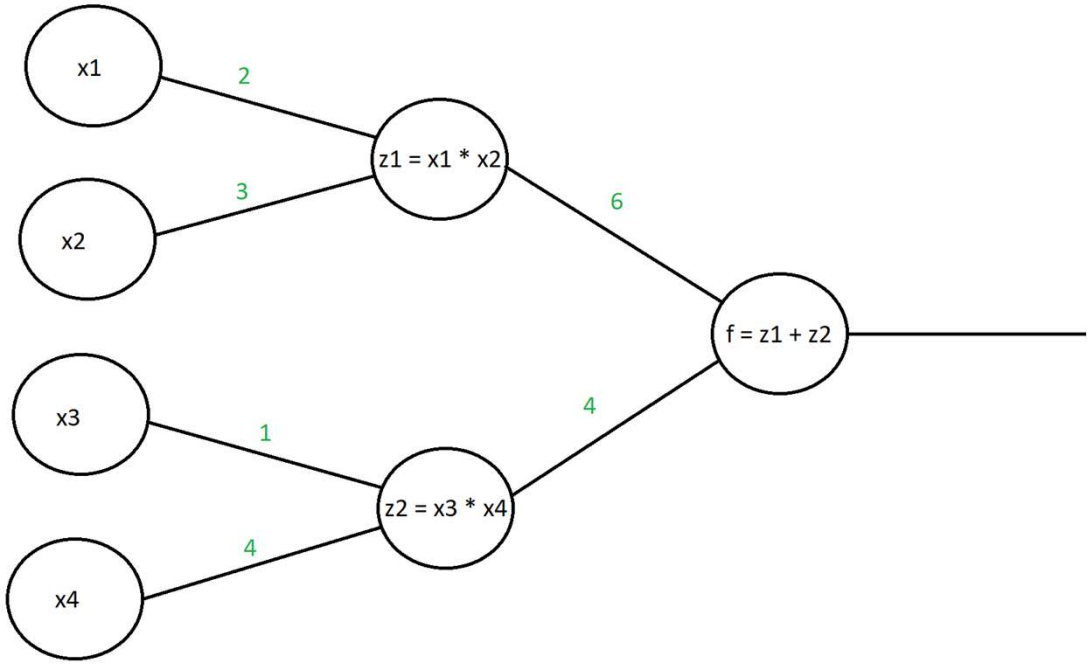
$$f(\vec{x}) = z_1 + z_2$$



Forward propagation

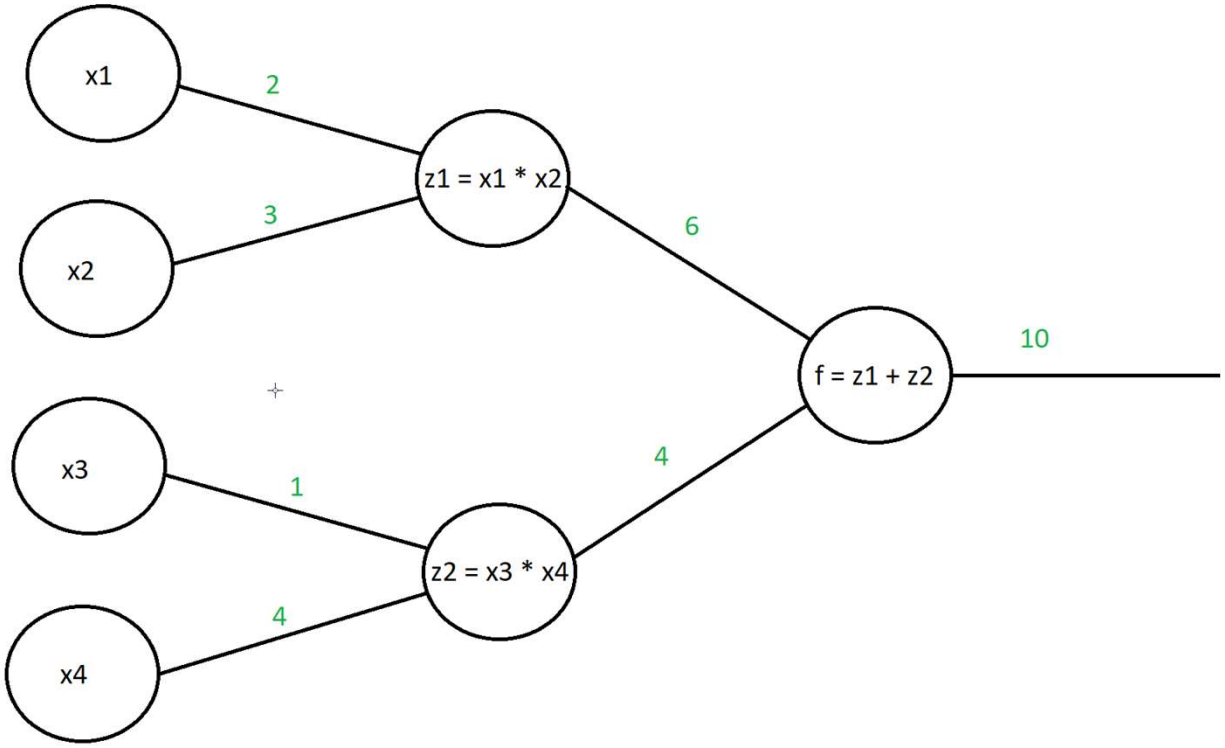
$$f(\vec{x}) = x_1 * x_2 + x_3 * x_4$$

$$f(\vec{x}) = z_1 + z_2$$



Forward propagation

$f(\vec{x}) = x_1 * x_2 + x_3 * x_4$ $f(\vec{x}) = z_1 + z_2$



Backward propagation

- What if we want to get the derivative of f with respect to the different x values?

$$f(\vec{x}) = x_1 * x_2 + x_3 * x_4 \qquad f(\vec{x}) = z_1 + z_2$$

$$\frac{\partial f(\vec{x})}{\partial x_1} = \frac{\partial f}{\partial z_1}\frac{\partial z_1}{\partial x_1} = x_2$$

$$\frac{\partial f(\vec{x})}{\partial x_3} = \frac{\partial f}{\partial z_2}\frac{\partial z_2}{\partial x_3} = x_4$$

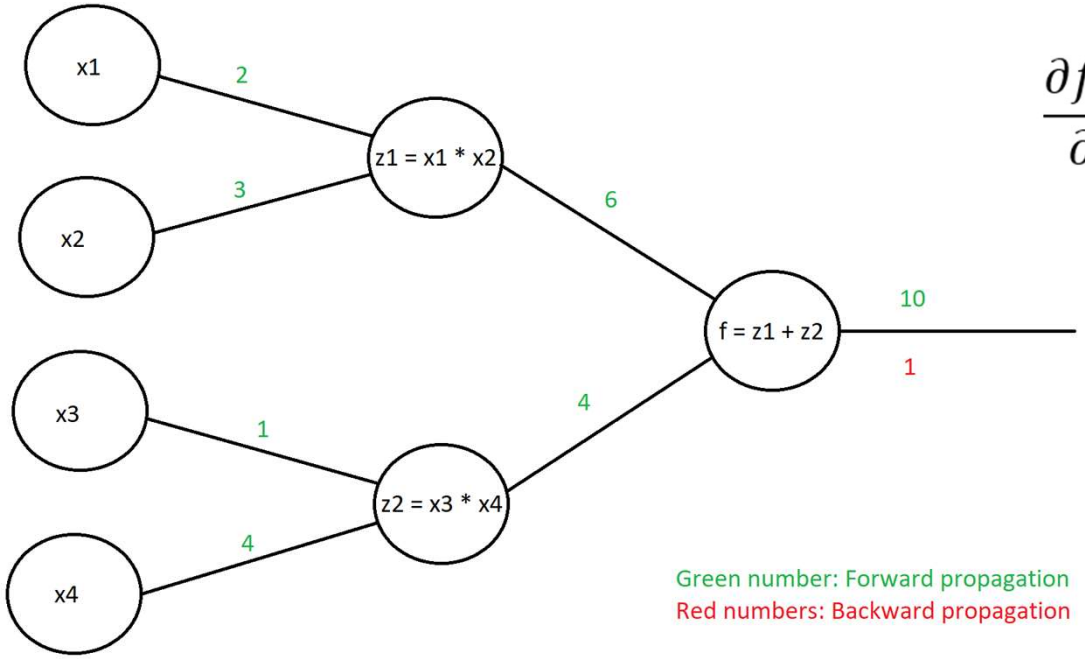
Backward propagation

- Lets take the derivative of f with respect to x1

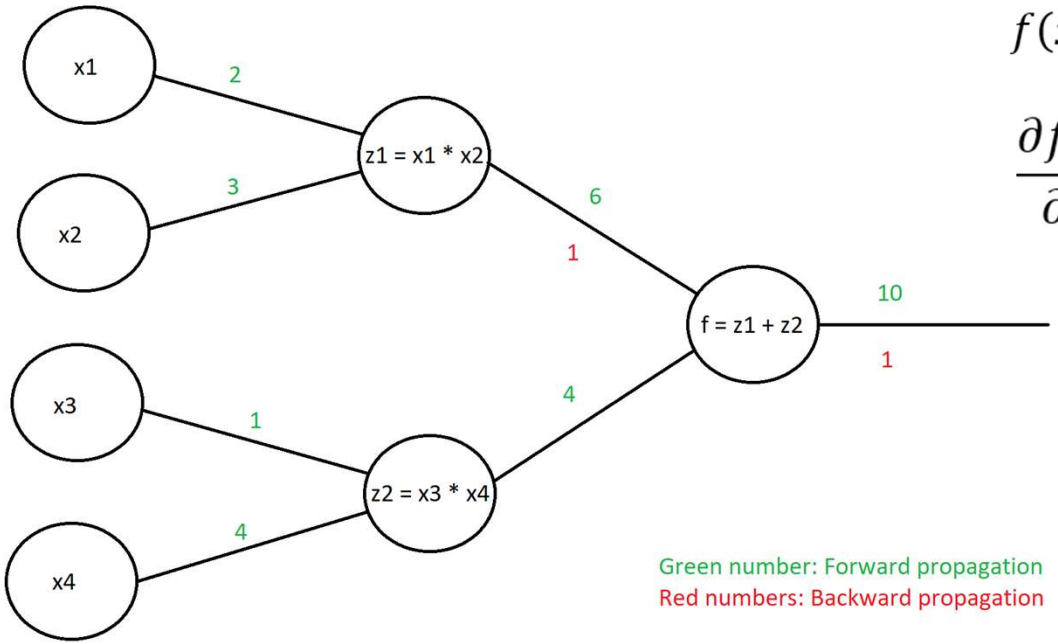
$$f(\vec{x}) = x_1 * x_2 + x_3 * x_4$$

$$f(\vec{x}) = z_1 + z_2$$

$$\frac{\partial f(\vec{x})}{\partial x_1} = \frac{\partial f}{\partial z_1}\frac{\partial z_1}{\partial x_1} = x_2$$



Backward propagation



$$f(\vec{x}) = x_1 * x_2 + x_3 * x_4$$

$$f(\vec{x}) = z_1 + z_2$$

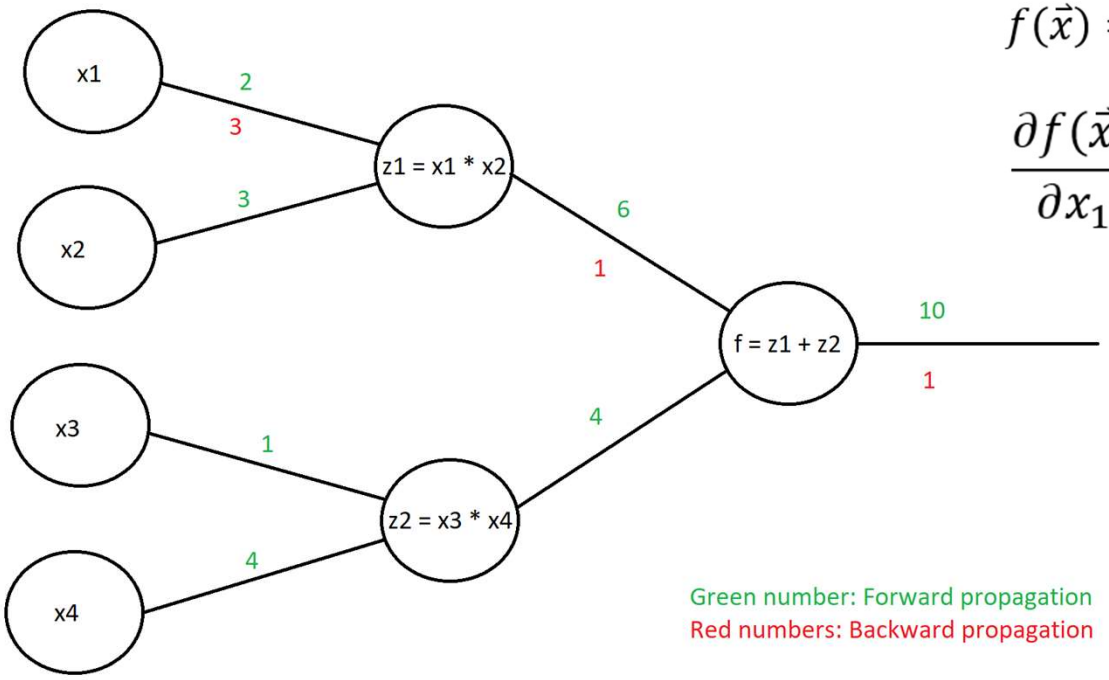
$$\frac{\partial f(\vec{x})}{\partial x_1} = \frac{\partial f}{\partial z_1}\frac{\partial z_1}{\partial x_1} = x_2$$

Backward propagation

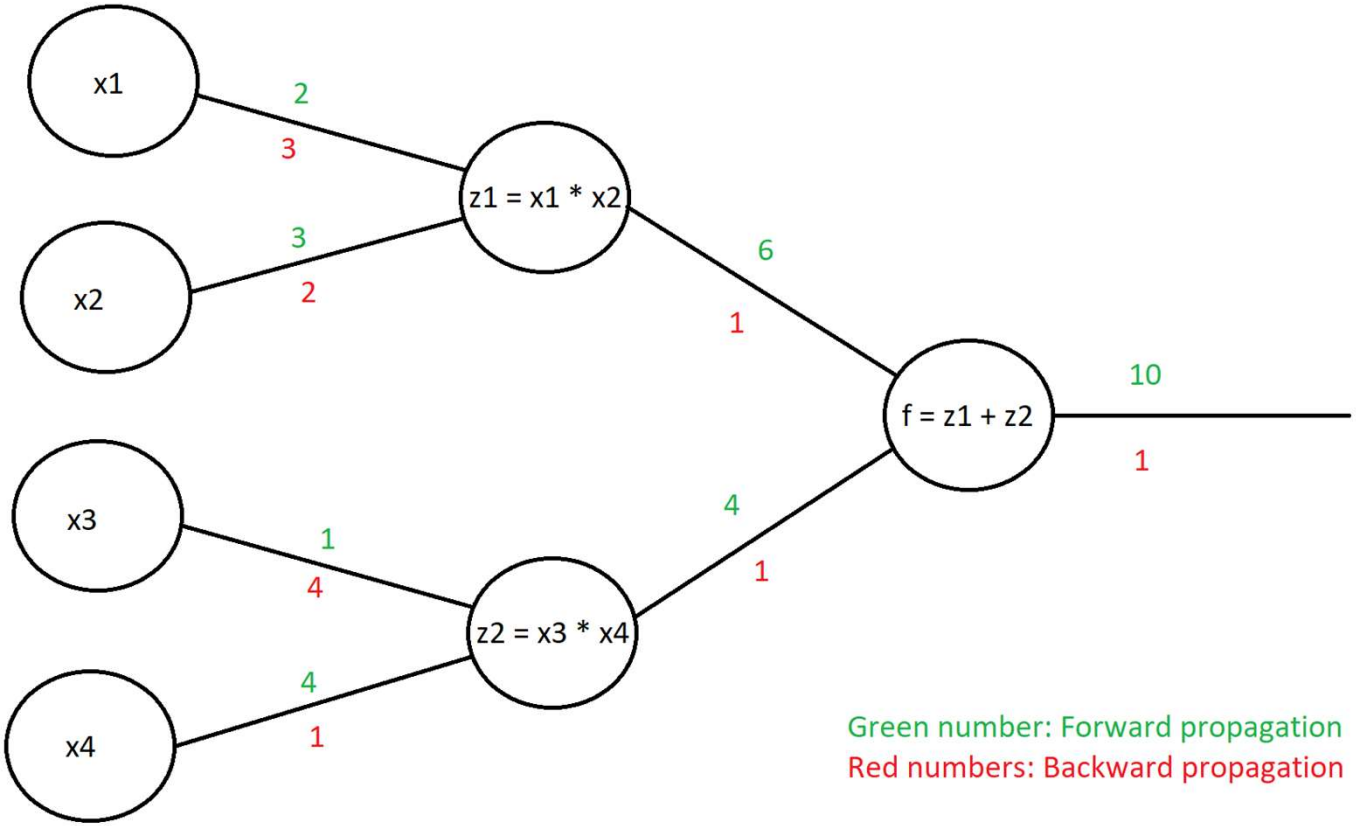
$$f(\vec{x}) = x_1 * x_2 + x_3 * x_4$$

$$f(\vec{x}) = z_1 + z_2$$

$$\frac{\partial f(\vec{x})}{\partial x_1} = \frac{\partial f}{\partial z_1}\frac{\partial z_1}{\partial x_1} = x_2$$

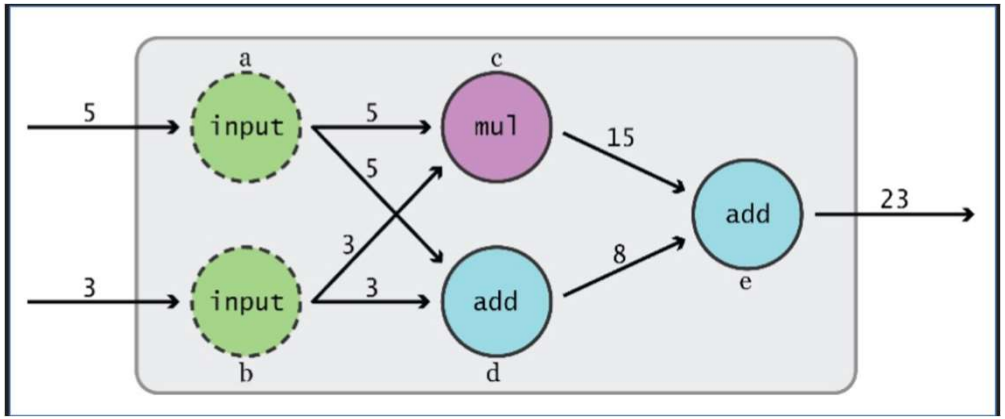


Backward propagation



TensorFlow: Computational graph

- You need to define the computational graph before you can use it.



What is a TensorFlow Tensor

- An n-dimensional array

0-D: A scalar

1-D: An array (vector)

2-D: A matrix

Defining a tensor in TensorFlow

- The main types of tensors are:
  - tf.Variable / tf.get_variable
  - tf.constant
  - tf.placeholder

- Attributes (some of them):
  - Shape
  - dtype
  - name

Example of a tensors

In [4]:
```
import tensorflow as tf
```

In [5]:
```
a = tf.constant(value=3, name='myConstant', dtype=tf.float32, shape=())
print(a)
```

```
Tensor("myConstant_1:0", shape=(), dtype=float32)
```

In [17]:
```
a = tf.Variable(initial_value=3, trainable=True, name='myVariable', dtype=tf.float32)
print(a)
```

```
<tf.Variable 'myVariable_2:0' shape=() dtype=float32_ref>
```

In [124]:
```
a = tf.placeholder(name='myPlaceholder', dtype=tf.float32, shape=())
print(a)
```

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

Tensors need unique names

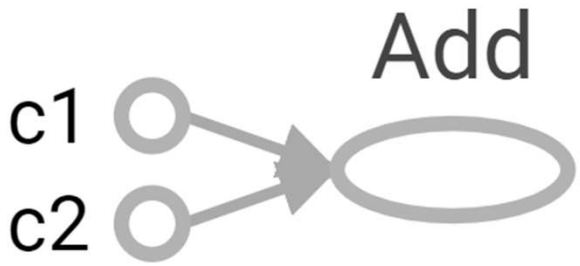
- Every tensor defined within a graph needs to have a unique name
- TensorFlow will automatically add an index to the name which will increment if more tensors with the “same” name gets defined.

```
In [32]: a = tf.constant(value=3, name='const', dtype=tf.float32, shape=())
         b = tf.constant(value=3, name='const', dtype=tf.float32, shape=())
         print(a)
         print(b)

         Tensor("const:0", shape=(), dtype=float32)
         Tensor("const_1:0", shape=(), dtype=float32)
```

Defining an Operator in TensorFlow

In [41]:
```
a = tf.constant(value=6.0, name='c1', dtype=tf.float32, shape=())
b = tf.constant(value=1.0, name='c2', dtype=tf.float32, shape=())
c = tf.add(a,b,name=None)
print(a)
print(b)
print(c)
```

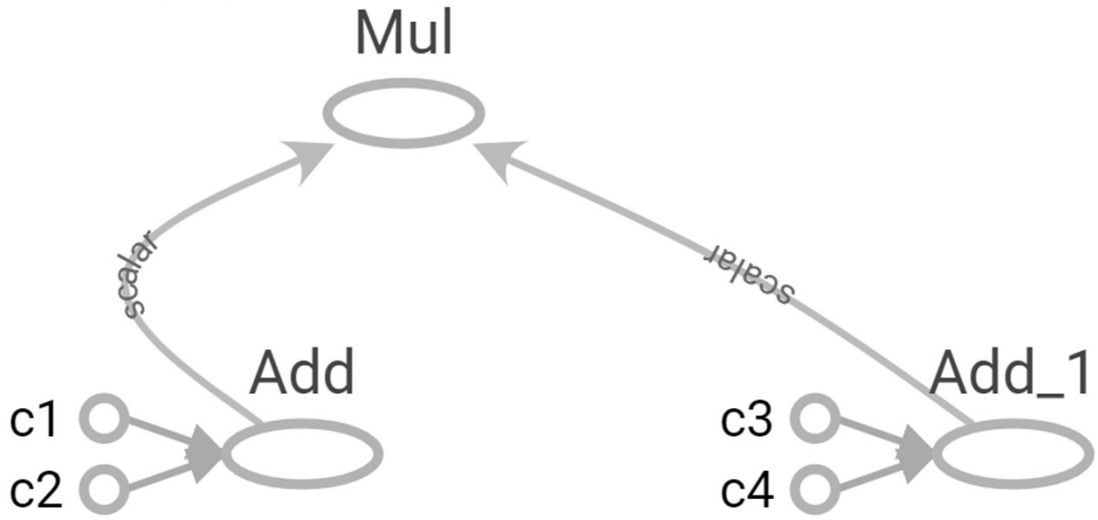
```
Tensor("c1:0", shape=(), dtype=float32)
Tensor("c2:0", shape=(), dtype=float32)
Tensor("Add:0", shape=(), dtype=float32)
```



If no name is given, TensorFlow will automatically give the node a name.

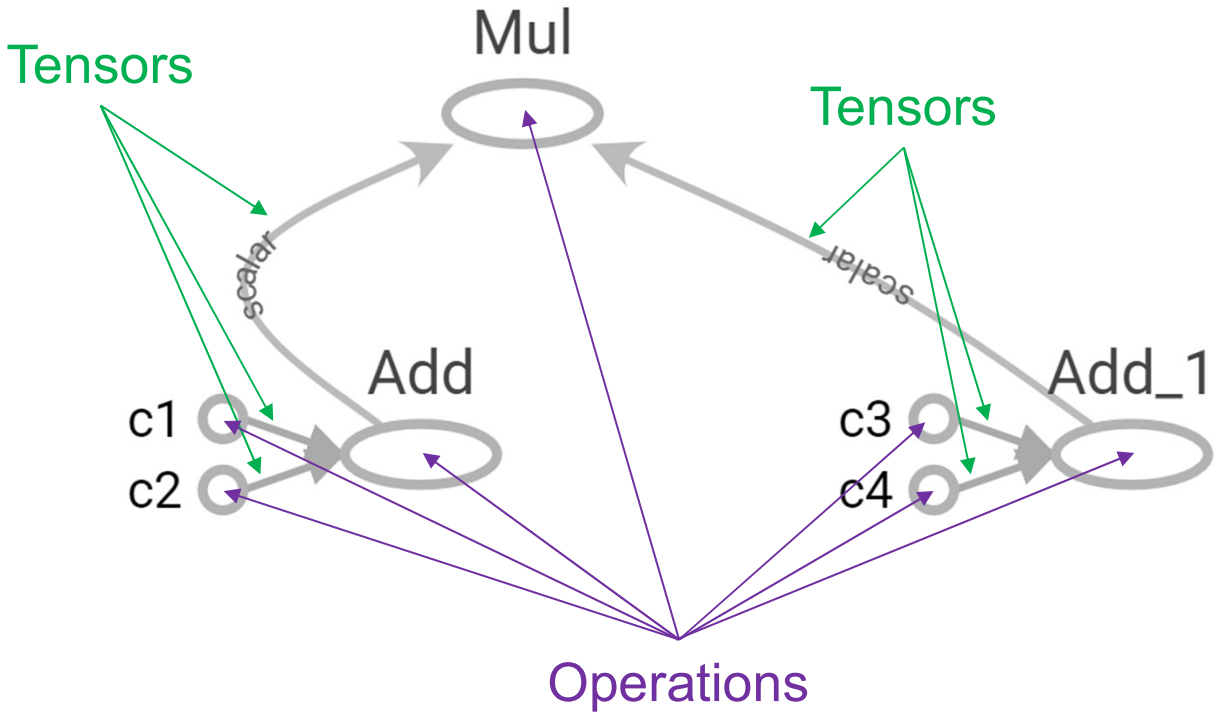
The graph

- The TensorFlow graph is a definition, not any computation.
- The computational graph is a series of TensorFlow operations arranged into a graph. The graph is composed of two types of objects:
  - Operation: Nodes in the graph
  - Tensors: The edges in the graph



The graph

- Operation (“ops”): Nodes in the graph
- Tensors: The edges in the graph



The graph

- The **tf.Graph** holds two types of information:
  - Graph structure
  - Graph collection: Meta data

- Examples of Meta data
  - Global variables
  - Trainable variable
  - Regularization loss
  - Moving average variables
  - Summaries

Multiple graphs

- It is possible to create multiple graph, but we don't do it!
  - Each graph will require it's own session
  - To pass data between the graphs, we need to pass them through Python/NumPy
  - Working with one graph is easier, and its better to have disconnected subgraphs.

Progress

- Deep learning frameworks
- TensorFlow
  - TensorFlow graphs
  - **TensorFlow session**
  - TensorFlow constants
  - TensorFlow variables
  - TensorFlow feeding data to the graph
  - Tensorboard
  - TensorFlow Save/restore models
  - TensorFlow example

Executing the tf.Graph: tf.Session

- We have seen that variables and constants are handles to elements in the computational graph only.
- We execute the graph using a tf.Session

```
In [50]: c = tf.add(3.0, 5.0)
         sess = tf.Session()
         c_val = sess.run(c)
         sess.close()

         print(c)
         print(c_val)

Tensor("Add:0", shape=(), dtype=float32)
8.0
```

Two ways to use tf.Session

In [50]:
```
c = tf.add(3.0, 5.0)
sess = tf.Session()
c_val = sess.run(c)
sess.close()

print(c)
print(c_val)
```

```
Tensor("Add:0", shape=(), dtype=float32)
8.0
```

In [52]:
```
c = tf.add(3.0, 5.0)
with tf.Session() as sess:
    c_val = sess.run(c)

print(c)
print(c_val)
```

```
Tensor("Add:0", shape=(), dtype=float32)
8.0
```

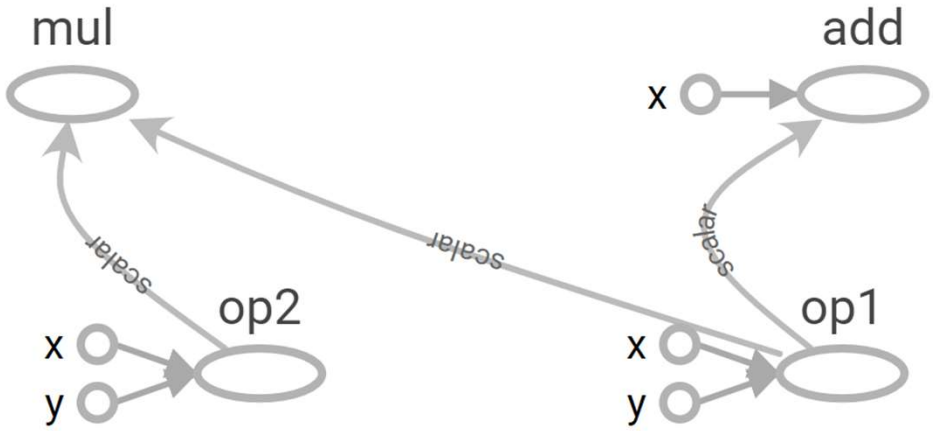
tf.Session

- When a tf.Session is created, the required resources are allocated.
- The tf.Session object can be configured:

```
In [ ]: sess = tf.Session(config=tf.ConfigProto(allow_soft_placement=True,
                                                log_device_placement=True))
```

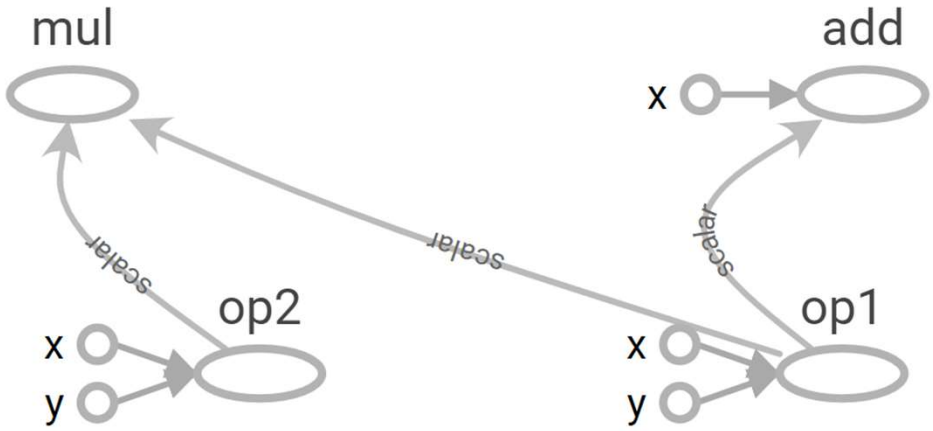
Run parts of the graph only

In [55]:
```
x = tf.constant(value=6.0, name='x', dtype=tf.float32, shape=())
y = tf.constant(value=1.0, name='y', dtype=tf.float32, shape=())

op1 = tf.add(x,y,name='op1')
op2 = tf.add(x,y,name='op2')

mul = tf.multiply(op1,op2, name='mul')
add = tf.add(op1,x,name='add')

with tf.Session() as sess:
    mul_val = sess.run(mul)
```



Run the whole graph

In [55]:
```
x = tf.constant(value=6.0, name='x', dtype=tf.float32, shape=())
y = tf.constant(value=1.0, name='y', dtype=tf.float32, shape=())

op1 = tf.add(x,y,name='op1')
op2 = tf.add(x,y,name='op2')

mul = tf.multiply(op1,op2, name='mul')
add = tf.add(op1,x,name='add')

with tf.Session() as sess:
    mul_val, add_val = sess.run([mul, add])
```



Why graphs

- Save computation, run subgraphs that lead to the values you want to fetch only
- Break computation into small, differential pieces to facilitate auto-differentiation
- Facilitate distributed computation, spread the work across multiple CPUs, GPUs, TPUs, or other devices

Progress

- Deep learning frameworks
- TensorFlow
  - TensorFlow graphs
  - TensorFlow session
  - **TensorFlow constants**
  - TensorFlow variables
  - TensorFlow feeding data to the graph
  - Tensorboard
  - TensorFlow Save/restore models
  - TensorFlow example

tf.constant

In [95]:
```
a = tf.constant(value=6.0, name='scalar', dtype=tf.float32, shape=[])
print(a)
```

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

In [96]:
```
a = tf.constant(value=6.0, name='array', dtype=tf.float32, shape=[2])
print(a)
```

```
Tensor("array:0", shape=(2,), dtype=float32)
```

In [97]:
```
a = tf.constant(value=6.0, name='matrix', dtype=tf.float32, shape=[2,2])
print(a)
```

```
Tensor("matrix:0", shape=(2, 2), dtype=float32)
```

Useful constants

In [101]:
```
a = tf.zeros(shape=[3,2], dtype=tf.float32, name='matrix')
print(a)
with tf.Session() as sess:
    a_val = sess.run(a)
    print(a_val)
```

```
Tensor("matrix_4:0", shape=(3, 2), dtype=float32)
[[ 0.  0.]
 [ 0.  0.]
 [ 0.  0.]]
```

In [102]:
```
a = tf.ones(shape=[3,2], dtype=tf.float32, name='matrix')
print(a)
with tf.Session() as sess:
    a_val = sess.run(a)
    print(a_val)
```

```
Tensor("matrix_5:0", shape=(3, 2), dtype=float32)
[[ 1.  1.]
 [ 1.  1.]
 [ 1.  1.]]
```

Constants as sequences

In [104]:
```
a = tf.range(start=1, limit=9, delta=2, dtype=None, name='range')
print(a)
with tf.Session() as sess:
    a_val = sess.run(a)
    print(a_val)
```

```
Tensor("range:0", shape=(4,), dtype=int32)
[1 3 5 7]
```

Randomly Generated Constants

- tf.set_random_seed(seed)

```
tf.random_normal(shape, mean=0.0, stddev=1.0, dtype=tf.float32, seed=None, name=None)

tf.random_uniform(shape, inval=0, maxval=None, dtype=tf.float32, seed=None, name=None)
```

Operations

Category	Examples
Element-wise mathematical operations	Add, Sub, Mul, Div, Exp, Log, Greater, Less, Equal, ...
Array operations	Concat, Slice, Split, Constant, Rank, Shape, Shuffle, ...
Matrix operations	MatMul, MatrixInverse, MatrixDeterminant, ...
Stateful operations	Variable, Assign, AssignAdd, ...
Neural network building blocks	SoftMax, Sigmoid, ReLU, Convolution2D, MaxPool, ...
Checkpointing operations	Save, Restore
Queue and synchronization operations	Enqueue, Dequeue, MutexAcquire, MutexRelease, ...
Control flow operations	Merge, Switch, Enter, Leave, NextIteration

Arithmetic Operations

- `tf.abs`
- `tf.negative`
- `tf.sign`
- `tf.reciprocal`
- `tf.square`
- `tf.round`
- `tf.sqrt`
- `tf.rsqrt`
- `tf.pow`
- `tf.exp`

TensorFlow Data Types

- `tf.float16` : 16-bit half-precision floating-point.
- `tf.float32` : 32-bit single-precision floating-point.
- `tf.float64` : 64-bit double-precision floating-point.
- `tf.bfloat16` : 16-bit truncated floating-point.
- `tf.complex64` : 64-bit single-precision complex.
- `tf.complex128` : 128-bit double-precision complex.
- `tf.int8` : 8-bit signed integer.
- `tf.uint8` : 8-bit unsigned integer.
- `tf.uint16` : 16-bit unsigned integer.
- `tf.int16` : 16-bit signed integer.
- `tf.int32` : 32-bit signed integer.
- `tf.int64` : 64-bit signed integer.
- `tf.bool` : Boolean.
- `tf.string` : String.
- `tf.qint8` : Quantized 8-bit signed integer.
- `tf.quint8` : Quantized 8-bit unsigned integer.
- `tf.qint16` : Quantized 16-bit signed integer.
- `tf.quint16` : Quantized 16-bit unsigned integer.
- `tf.qint32` : Quantized 32-bit signed integer.
- `tf.resource` : Handle to a mutable resource.

What is wrong with constants?

- Constants are constants and not good for e.g. being a weight matrix
- The data contained in constants are stored in the TensorFlow graph definition.

Constants: The values can be stored in the TensorFlow graph definition

```
In [24]: npConst = np.ones((20,1))
         x = tf.constant(value=npConst, name='myConst', dtype=tf.float32)
         print(tf.get_default_graph().as_graph_def())
```

```
node {
  name: "myConst"
  op: "Const"
  attr {
    key: "dtype"
    value {
      type: DT_FLOAT
    }
  }
  attr {
    key: "value"
    value {
      tensor {
        dtype: DT_FLOAT
        tensor_shape {
          dim {
            size: 20
          }
          dim {
            size: 1
          }
        }
        tensor_content: "\000\000\200?\000\000\200?\000\000\200?\000\000\200?\000\000\200?\000\000\200?\000\000\200?\000\000\20
0?\000\000\200?\000\000\200?\000\000\200?\000\000\200?\000\000\200?\000\000\200?\000\000\200?\000\000\200?\000\000
\200?\000\000\200?\000\000\200?"
      }
    }
  }
}
versions {
  producer: 22
}
```

Constants: The values can be stored in the TensorFlow graph definition

In [30]:
```
x = tf.constant(value=1.0, name='myConst', dtype=tf.float32, shape=[20,1])
print(tf.get_default_graph().as_graph_def())
```

```
node {
  name: "myConst"
  op: "Const"
  attr {
    key: "dtype"
    value {
      type: DT_FLOAT
    }
  }
  attr {
    key: "value"
    value {
      tensor {
        dtype: DT_FLOAT
        tensor_shape {
          dim {
            size: 20
          }
          dim {
            size: 1
          }
        }
        float_val: 1.0
      }
    }
  }
}
versions {
  producer: 22
}
```

Progress

- Deep learning frameworks
- TensorFlow
  - TensorFlow graphs
  - TensorFlow session
  - TensorFlow constants
  - **Tensorflow variables**
  - TensorFlow feeding data to the graph
  - Tensorboard
  - TensorFlow Save/restore models
  - TensorFlow example

Variables

- Operation vs Classes
  - tf.constant is written with lowercase and is an operation (op)
  - tf.Variable is written with uppercase and is a class
    - The tf.Variable class have many operations (ops)

Creating variables

We can define variables two ways:

```
# create variables with tf.Variable
s = tf.Variable(3.0, name="scalar")
```

preferred

```
# create variables with tf.get_variable
s = tf.get_variable("scalar", initializer=tf.constant(3.0))
```

Initializing variables

- The variables to be used in the graph have to be either:
  - Initialized
  - Restored

```
tf.reset_default_graph()
x       = tf.Variable(initial_value=tf.ones(4), name="array")

with tf.Session() as sess:
    sess.run(x.initializer)
    x_val = sess.run(x)
    print(x_val)
```

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

If variables are not initialized

```
tf.reset_default_graph()
x       = tf.Variable(initial_value=tf.ones(4), name="array")

with tf.Session() as sess:
#    sess.run(x.initializer)
    x_val = sess.run(x)
    print(x_val)
```

```
FailedPreconditionError: Attempting to use uninitialized value Variable
```

Initializing all or some variables

In [36]:
```
tf.reset_default_graph()
x       = tf.Variable(initial_value=tf.ones(4), name="array1")
y       = tf.Variable(initial_value=tf.ones(4), name="array2")
z       = tf.add(x,y)

with tf.Session() as sess:
    sess.run(tf.variables_initializer([x, y]))
    z_val = sess.run(z)
    print(z_val)
```

```
[ 2.  2.  2.  2.]
```

```
tf.reset_default_graph()
x       = tf.Variable(initial_value=tf.ones(4), name="array1")
y       = tf.Variable(initial_value=tf.ones(4), name="array2")
z       = tf.add(x,y)

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    z_val = sess.run(z)
    print(z_val)
```

```
[ 2.  2.  2.  2.]
```

tf.Variable.assign()

- The assign operator is handy if you want to update the values in a variable

```
W          = tf.Variable(20)
assign_op = W.assign(W + 200)
with tf.Session() as sess:
    sess.run(W.initializer)
    print(W.eval())
```

```
20
```

```
W          = tf.Variable(20)
assign_op = W.assign(W + 200)
with tf.Session() as sess:
    sess.run(W.initializer)
    sess.run(assign_op)
    print(W.eval())
```

```
220
```

tf.get_variable()

- Why is tf.get_variable preferred over tf.Variable?
  - tf.get_variable makes reusing (sharing) variables easy

```
W = tf.get_variable(name='myVar', initializer=tf.random_normal([2]))
with tf.Session() as sess:
    sess.run(W.initializer)
    print(W)
    print(sess.run(W))
```

```
<tf.Variable 'myVar:0' shape=(2,) dtype=float32_ref>
[-1.40301454  1.01252878]
```

tf.variable_scope()

- tf.variable_scope is used to give a specific prefix to variables names, it is analogous to directory structure.

```
W = tf.get_variable(name='myVar', initializer=tf.random_normal([2]))
print(W)
```

```
<tf.Variable 'myVar:0' shape=(2,) dtype=float32_ref>
```

```
with tf.variable_scope("layer1"):
    W = tf.get_variable(name='myVar', initializer=tf.random_normal([2]))
print(W)
```

```
<tf.Variable 'layer1/myVar:0' shape=(2,) dtype=float32_ref>
```

Reusing variables

- If we try to define two variables with the same name, a valueError is raised.

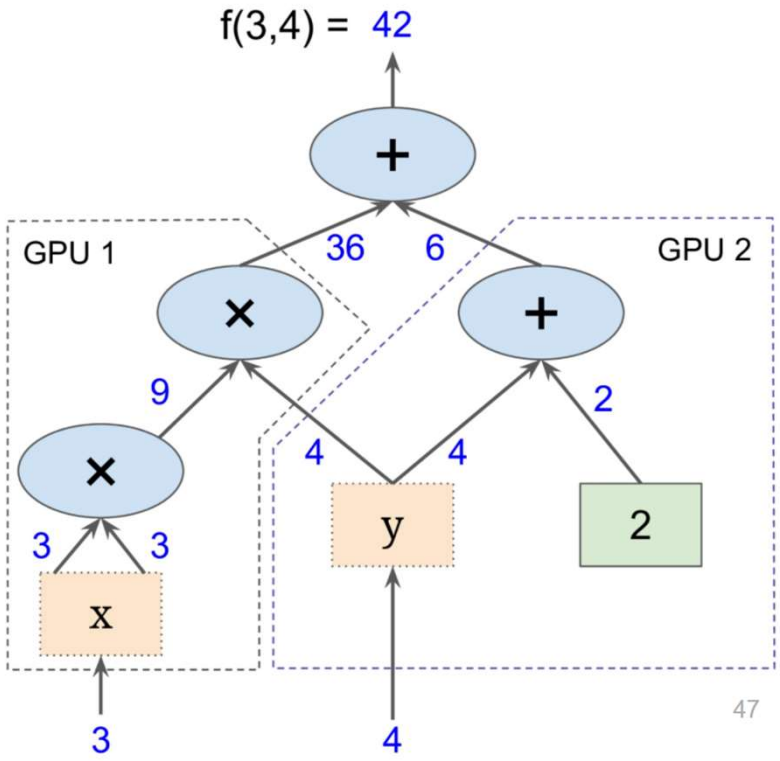
```
with tf.variable_scope("layer1"):
    W1 = tf.get_variable(name='myVar', initializer=tf.random_normal([2]))
    W2 = tf.get_variable(name='myVar', initializer=tf.random_normal([2]))
print(W)
```

```
ValueError: Variable layer1/myVar already exists, disallowed. Did you mean to set reuse=True or reuse=tf.AUTO_REUSE in VarScop
e? Originally defined at:
```

```
with tf.variable_scope("layer1"):
    W1 = tf.get_variable(name='myVar', initializer=tf.random_normal([2]))
    tf.get_variable_scope().reuse_variables()
    W2 = tf.get_variable(name='myVar', initializer=tf.random_normal([2]))
print(W1)
print(W2)
```

```
<tf.Variable 'layer1/myVar:0' shape=(2,) dtype=float32_ref>
<tf.Variable 'layer1/myVar:0' shape=(2,) dtype=float32_ref>
```

Distributed computation (multiple GPU's)



Graph from Hands-On Machine Learning with Scikit-Learn and TensorFlow

Distributed Computation

- tf.device() can be used to put part of the graph to the cpu or gpus

```
# Creates a graph.
with tf.device('/gpu:1'):
    a = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], name='a')
    b = tf.constant([1.0, 2.0, 3.0, 4.0, 5.0, 6.0], name='b')
    c = tf.multiply(a, b)

    #Creates a session with log_device_placement set to True.
sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
print(sess.run(c))
```

```
[  1.   4.   9.  16.  25.  36.]
```

```
Mul: (Mul): /job:localhost/replica:0/task:0/device:GPU:1
b: (Const): /job:localhost/replica:0/task:0/device:GPU:1
a: (Const): /job:localhost/replica:0/task:0/device:GPU:1
```

Progress

- Deep learning frameworks
- TensorFlow
  - TensorFlow graphs
  - TensorFlow session
  - TensorFlow constants
  - TensorFlow variables
  - **Tensorflow feeding data to the graph**
  - Tensorboard
  - TensorFlow Save/restore models
  - TensorFlow example

How to feed data into the graph

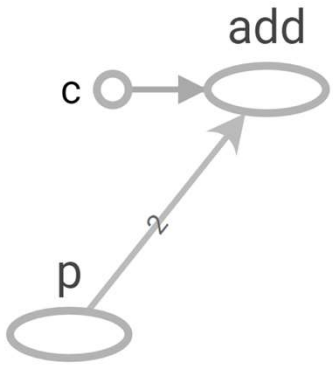
- Often we want to feed data into the graph after it has been defined. For example during training of a neural network, we feed training data and labels into the graphs repeatedly.

- Two methods are recommended:
  - tf.placeholder
  - tf.data.Dataset (we will not go though dataset and iterators)

tf.placeholder

```
# create a placeholder for a vector of 2 elements, type tf.float32
x = tf.placeholder(dtype=tf.float32, shape=[2], name='p')

y = tf.constant(value=[1, 2], dtype=tf.float32, name='c')

z = x + y

with tf.Session() as sess:
    z_val = sess.run(z, feed_dict={x: [3, 4]})
    print(z_val)
```

```
[ 4.  6.]
```



Progress

- Deep learning frameworks
- TensorFlow
  - TensorFlow graphs
  - TensorFlow session
  - TensorFlow constants
  - TensorFlow variables
  - TensorFlow feeding data to the graph
  - **Tensorboard**
  - TensorFlow Save/restore models
  - TensorFlow example

Tensorboard: Visualizing graph

```
x = tf.get_variable(name='x', initializer=tf.random_normal([2]))
y = tf.get_variable(name='y', initializer=tf.random_normal([2]))
z = x+y

writer = tf.summary.FileWriter(logdir='./graphs', graph=tf.get_default_graph())

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    print(sess.run(z))
writer.close()
```

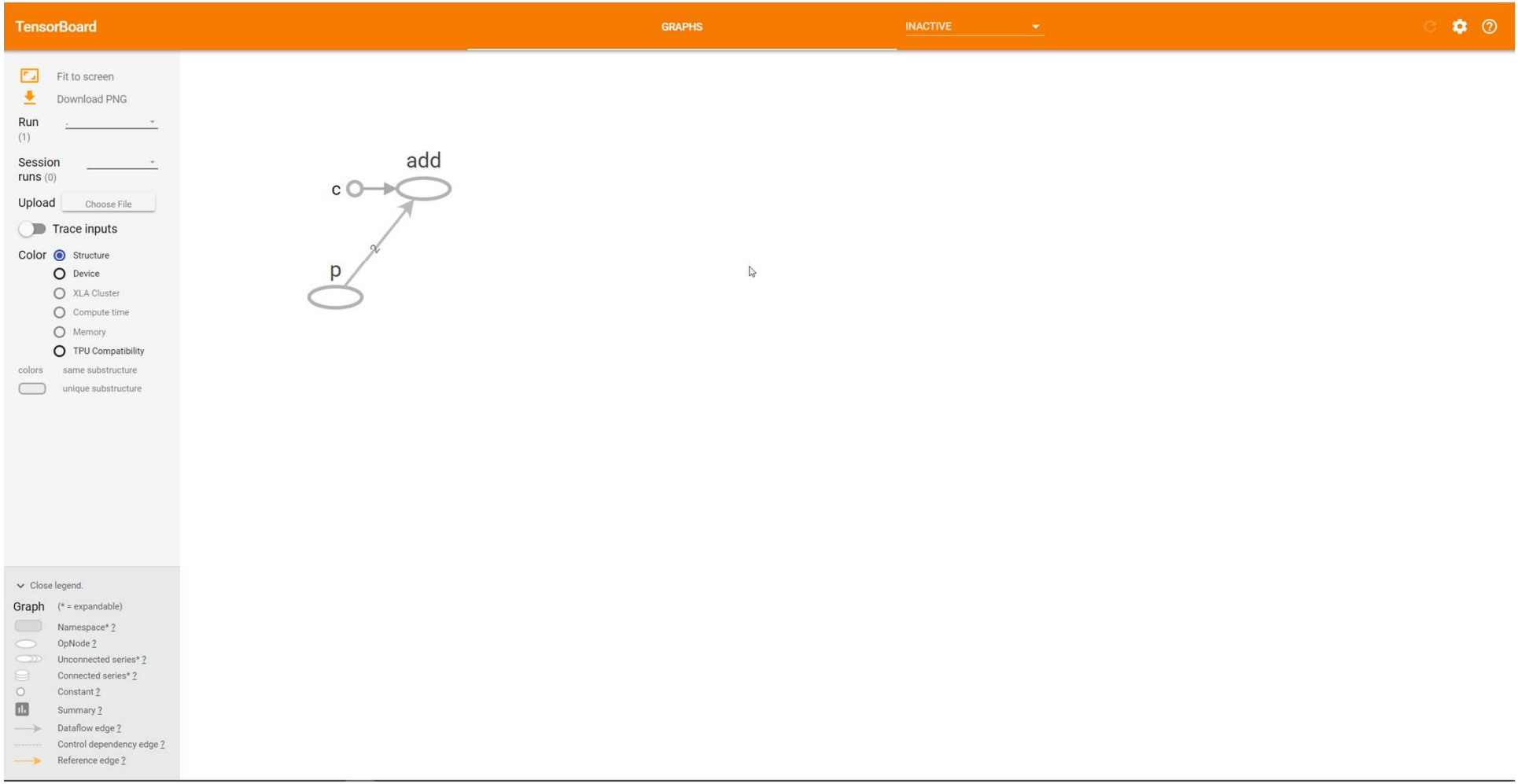
```
[-2.20665431  0.04801518]
```

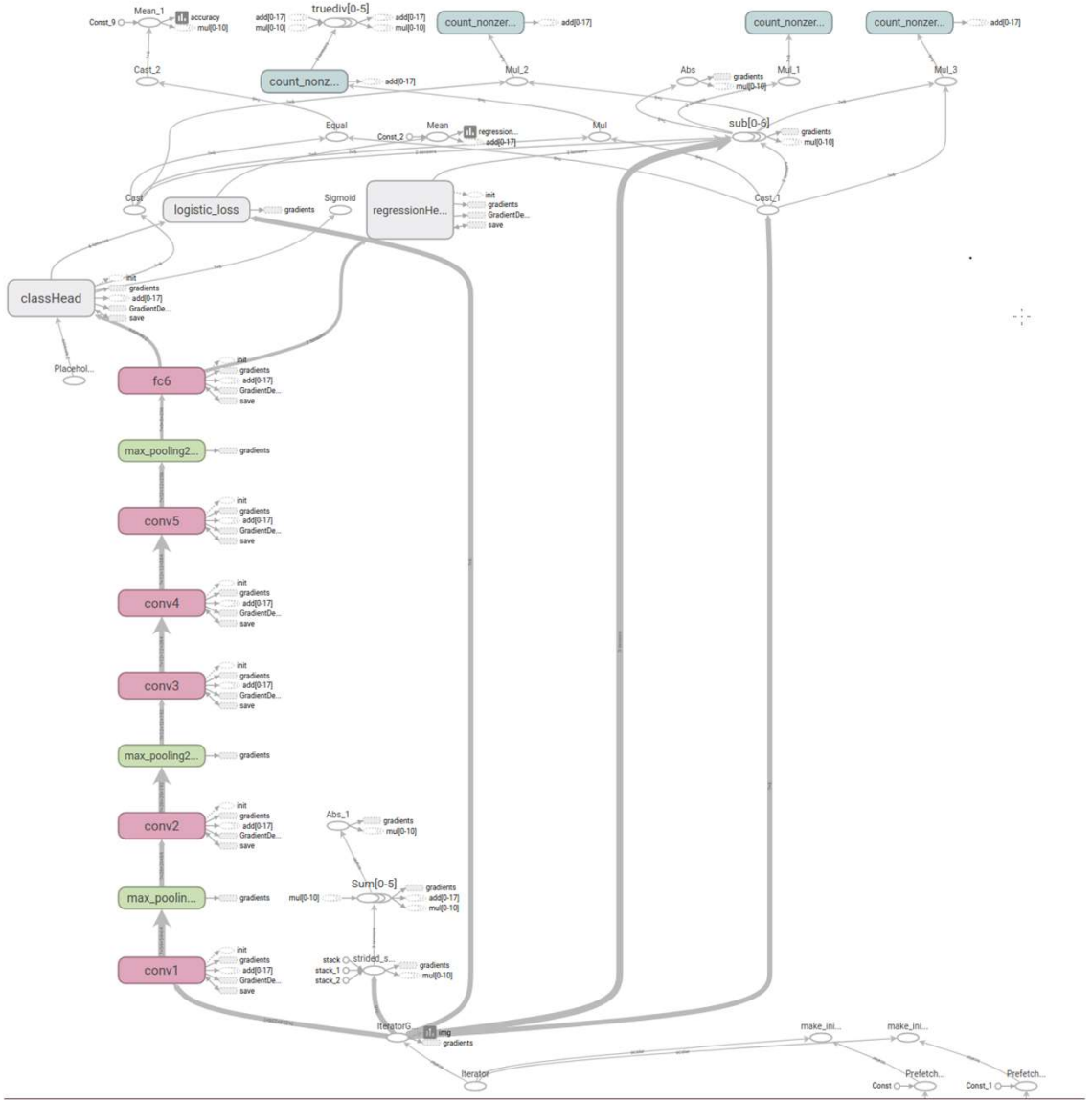
- **In terminal:**

  $ tensorboard --logdir="./graphs" --port 6006

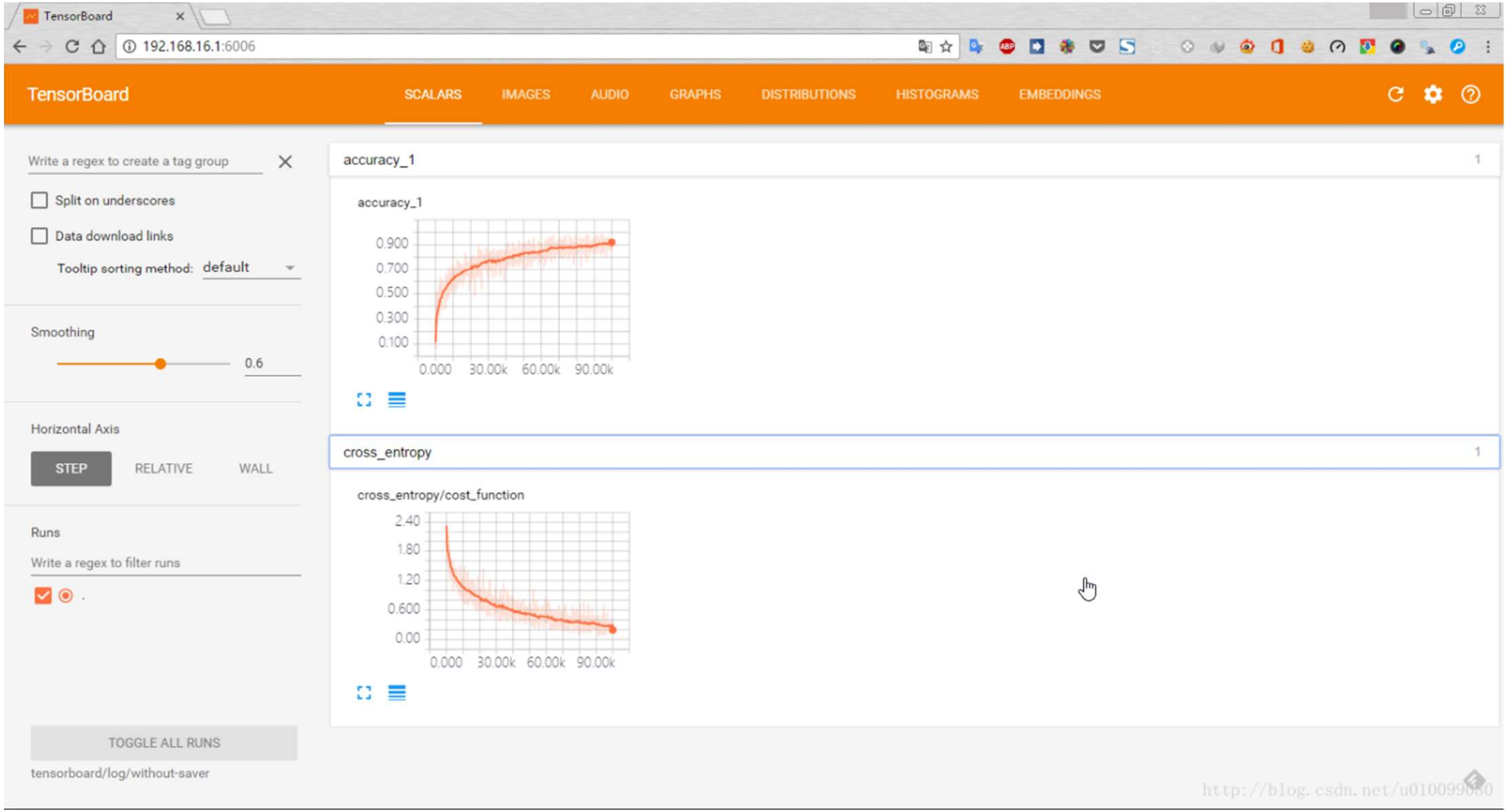
- **In web browser:**

  http://localhost:6006/


TensorBoard
GRAPHS
INACTIVE
Fit to screen
Download PNG
Run (1)
Session runs (0)
Upload
Choose File
Trace inputs
Color
Structure
Device
XLA Cluster
Compute time
Memory
TPU Compatibility
colors
same substructure
unique substructure
add
c
p
Close legend.
Graph
(* = expandable)
Namespace* ?
OpNode ?
Unconnected series* ?
Connected series* ?
Constant ?
Summary ?
Dataflow edge ?
Control dependency edge ?
Reference edge ?


Mean_1
accuracy
truediv[0-5]
count_nonzer...
count_nonzer...
count_nonzer...
Cast_2
count_nonz...
Mul_2
Abs
Mul_1
Mul_3
Equal
Mean
regression...
Mul
sub[0-6]
Cast
logistic_loss
gradients
Sigmoid
regressionHe...
init
gradients
GradientDe...
save
Cast_1
classHead
Placehol...
fc6
max_pooling2...
conv5
conv4
conv3
max_pooling2...
conv2
max_poolin...
conv1
Abs_1
Sum[0-5]
stack
strided_s...
IteratorG...
img
Iterator
make_ini...
make_ini...
Prefetch...
Prefetch...
Const
Const_1

Tensorboard: Visualizing learning



Tensorboard: Visualizing learning

```
x = tf.get_variable(name='x', initializer=tf.random_normal(shape=()))
y = tf.get_variable(name='y', initializer=tf.random_normal(shape=()))
z = x + y

tf.summary.scalar('x', x)
tf.summary.scalar('z', z)
summary_op = tf.summary.merge_all()

tensorboard_writer = tf.summary.FileWriter(logdir='./graphs', graph=tf.get_default_graph())

with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    z_val, summary_str = sess.run([z, summary_op])
    tensorboard_writer.add_summary(summary_str)
```

Progress

- Deep learning frameworks
- TensorFlow
  - TensorFlow graphs
  - TensorFlow session
  - TensorFlow constants
  - TensorFlow variables
  - TensorFlow feeding data to the graph
  - Tensorboard
  - **TensorFlow Save/restore models**
  - TensorfFow example

Save/restore models

- We often want to be able to save and store our models.
  - Unexpected power shoot down
  - Reuse of the already trained network weights
  - Sharing our work e.g. github

tf.train.saver.save()

- How to save our model every 1000 iteration.

```
saver       = tf.train.Saver()
global_step = tf.Variable(0, dtype=tf.int32, trainable=False, name='global_step')

with tf.Session() as sess:
    for step in range(number_of_training_steps):
        # do training of the network

        #Save the model every 1000 training step
        if (step + 1) % 1000==0:
            saver.save(sess, 'checkpoint_directory/model_name', global_step=global_step)
```

tf.train.saver.restore()

- How to restore a model.

```
#Define your graph

# Restore variable values
all_variables = tf.global_variables()
restorer      = tf.train.Saver(all_variables)
ckpt          = tf.train.latest_checkpoint('checkpoint_directory/model_name')
restorer.restore(sess, ckpt)

#Continue to train your network
```

Progress

- Deep learning frameworks
- TensorFlow
  - TensorFlow graphs
  - TensorFlow session
  - TensorFlow constants
  - TensorFlow variables
  - TensorFlow feeding data to the graph
  - Tensorboard
  - TensorFlow Save/restore models
  - **TensorFlow Cifar10 example**

TensorFlow example: Cifar10

- Define a “dataClass”
- Define a structure of the network (graph)
- Define a loss function
- Define an optimizer
- Train the neural network

Define a “dataClass”

- Properties of "dataClass“:
  - Read in the cifar10 images
  - Perform preprocessing
  - Have a "next_training_batch" function
  - have a "get_test_data" function

```python
from six.moves import cPickle as pickle
import numpy as np
import os
import platform


class dataClass:
    def __init__(self, cifar10_dir):
        X_train, y_train, X_test, y_test = load_CIFAR10(cifar10_dir)

        # Preprocessing: reshape the image data into rows
        X_train = np.reshape(X_train, (X_train.shape[0], -1))
        X_test = np.reshape(X_test, (X_test.shape[0], -1))

        # Normalize the data: subtract the mean image
        mean_image = np.mean(X_train, axis=0)
        X_train -= mean_image
        X_test  -= mean_image

        # add bias dimension and transform into columns
        X_train = np.hstack([X_train, np.ones((X_train.shape[0], 1))])
        X_test  = np.hstack([X_test, np.ones((X_test.shape[0], 1))])

        self.X_train = X_train
        self.X_test  = X_test
        self.y_train = y_train
        self.y_test  = y_test
        self.numbOfTrainSamples = self.X_train.shape[0]
        self.numbOfFeatures     = self.X_train.shape[1]
        self.numbOfClasses      = 10
        return

    def next_training_batch(self, batch_size):
        ind      = np.random.randint(self.numbOfTrainSamples, size=batch_size)
        y_onehot = np.zeros((batch_size, self.numbOfClasses))
        y_onehot[np.arange(batch_size), self.y_train[ind]] = 1
        return self.X_train[ind, :], y_onehot

    def get_test_data(self):
        batch_size = self.X_test.shape[0]
        y_onehot = np.zeros((batch_size, self.numbOfClasses))
        y_onehot[np.arange(batch_size), self.y_test] = 1
        return self.X_test, y_onehot

    def load_CIFAR10(ROOT):
        #.................
```

Import

```
import tensorflow as tf
from utils import dataClass
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
import os
os.environ["CUDA_VISIBLE_DEVICES"] = '0'

#Load cifar10 data
cifar10_dir = 'inf5860/datasets/cifar-10-batches-py'
myData      = dataClass.dataClass(cifar10_dir)
```

Define placeholder's

```
#Define placeholders for being able to feed data to the tensorflow graph
data          = tf.placeholder(shape=(None, myData.numbOfFeatures), dtype=tf.float32, name='data')
labels_onehot = tf.placeholder(shape=(None, myData.numbOfClasses),  dtype=tf.int32,   name='labels_onehot')
global_step   = tf.Variable(initial_value=0, trainable=False, name='global_step')
```

Define the network structure (graph)

```
# N -> number of training samples
# D1 -> number of input features
# D2 -> number of output features
# C -> number of output features

# - W: A array of shape (D1, D2) containing weights.
# - data: A array of shape (N, D1) containing a minibatch of data.
# - labels_onehot: A array of shape (N, C) containing training labels

# Lets define a fully connected neural network
hiddenLayerSizes = [myData.numbOfFeatures, 1024, 265, myData.numbOfClasses]
a = data
for ii in range(len(hiddenLayerSizes)-1):
    layerName = f'layer%s' % ii
    with tf.variable_scope(layerName):
        ny = hiddenLayerSizes[ii]
        nx = hiddenLayerSizes[ii+1]
        W  = tf.get_variable(name='W', shape=(ny, nx), initializer=tf.contrib.layers.xavier_initializer())
        z  = tf.matmul(a, W, name='matmul')
        a  = tf.tanh(z, name='activation_function')
```

Define the loss

```
#Define your loss function
logits = a
loss   = tf.losses.softmax_cross_entropy(onehot_labels=labels_onehot, logits=logits)
```

Define an optimizer

```
#Define an optimizer
all_variables = tf.trainable_variables()
optimizer     = tf.train.GradientDescentOptimizer(learning_rate=0.05)
train_op      = optimizer.minimize(loss, global_step=global_step, var_list=all_variables)
```

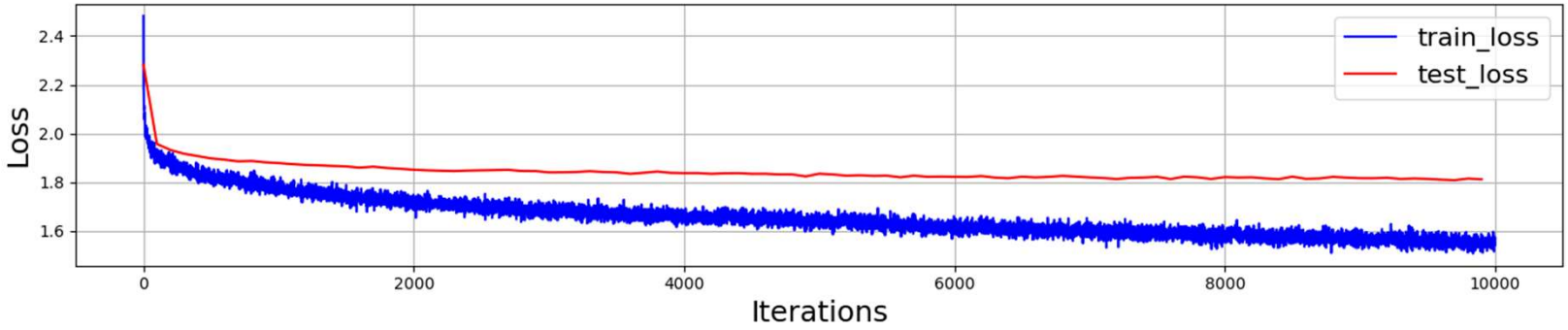
Define an accuracy measure

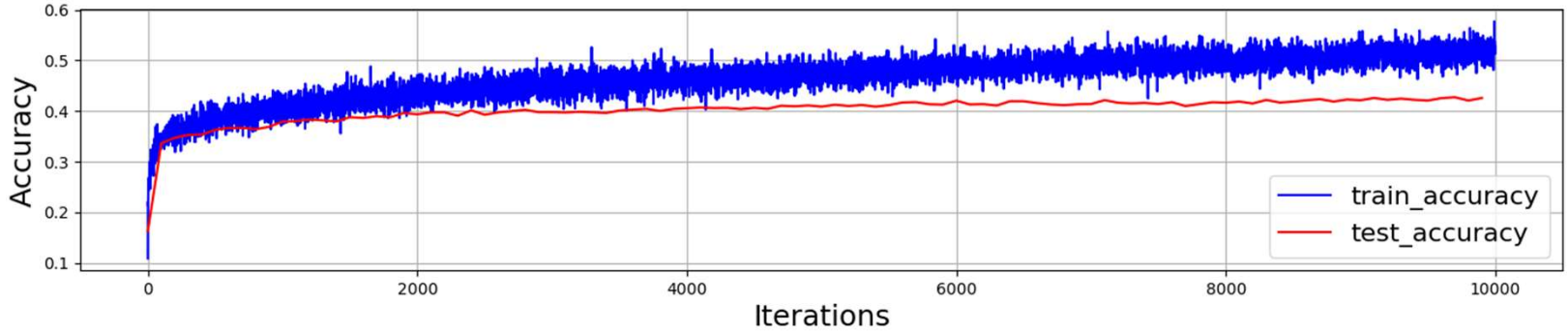
```
#Calculate the accuracy
estimated_class = tf.arg_max(logits, dimension=1)
labels          = tf.arg_max(labels_onehot, dimension=1)
accuracy        = tf.reduce_mean(tf.cast(tf.equal(estimated_class, labels), tf.float32), name='accuracy')
```

Train the network

```
# Hyperparameters
numbOfTrainingSteps = 10000
batch_size          = 1000

#Log train loss/accuracy and test loss/accuracy
train_loss     = np.zeros(numbOfTrainingSteps)
train_accuracy = np.zeros(numbOfTrainingSteps)
test_loss     = []
test_accuracy = []
test_inds     = []

t = tqdm(range(numbOfTrainingSteps), desc='', leave=True, mininterval=2, miniters=2)
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    for ii in t:
        npData, npLabels_onehot   = myData.next_training_batch(batch_size)
        loss_val, accuracy_val, _ = sess.run([loss, accuracy, train_op],
                                                    feed_dict={data: npData, labels_onehot: npLabels_onehot})
        train_loss[ii]         = loss_val
        train_accuracy[ii]     = accuracy_val
        printStr = 'Train Loss: %0.5f   | Train  Accuracy: %0.3f ' % (loss_val, accuracy_val)
        t.set_description(printStr)
        t.refresh()


        if ii % 100 == 0:
            npData, npLabels_onehot = myData.get_test_data()
            loss_val, accuracy_val, _ = sess.run([loss, accuracy, train_op],
                                                 feed_dict={data: npData, labels_onehot: npLabels_onehot})
            test_loss.append(loss_val)
            test_accuracy.append(accuracy_val)
            test_inds.append(ii)
```

Result


train_loss
test_loss
Loss
Iterations
2.4
2.2
2.0
1.8
1.6
0
2000
4000
6000
8000
10000


train_accuracy
test_accuracy
Accuracy
Iterations
0.6
0.5
0.4
0.3
0.2
0.1
0
2000
4000
6000
8000
10000

TensorFlow integrates well with NumPy

- tf.int32 == np.int32 ⇒ True
- tf.ones([2, 2], np.float32)
- The output of a tf.Session.run(ops) ⇒ will be a n-D NumPy array if “ops” is a n-D tensor
- But use tf datatypes if possible

Higher level of abstraction wrappers

- Open source software libraries for numerical computation using data flow graphs:
  - High level wrappers
    - Kears
    - TF-Slim
    - Tf-layers
    - Pretty Tensor
    - TFLearn
  - We will use TensorFlow “core” as it is the fundamental library and it is then easier to learn the higher level wrappers

Next week

- Convolutional neural networks