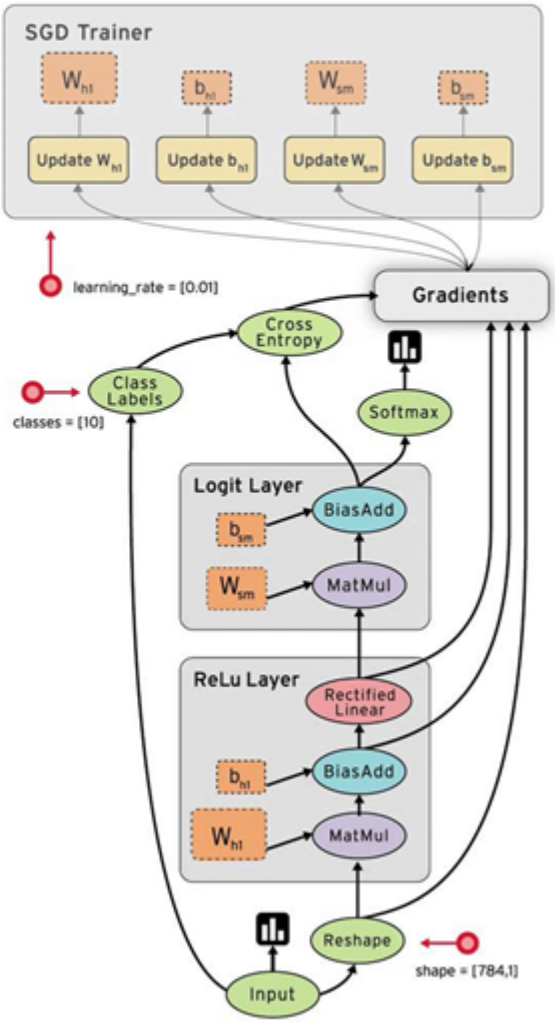
TensorFlow and Deep Learning

About TensorFlow

- TensorFlow was originally developed by the Google Brain Team within Google's Machine Intelligence research organization.
- Purpose: Conducting machine learning and deep neural networks research.
- Not only for Deep learning: The system is general enough to be applicable in a wide variety of other domains as well.
- Open source software library for numerical computation using data flow graphs.
- The flexible architecture allows you to deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API.

Data Flow Graph

- Describe mathematical computation with a directed graph of **nodes** & **edges**.
  - Nodes in the graph represent mathematical operations
  - Edges describe the i/o relationships between nodes
  - Data edges carry dynamically-sized multidimensional data arrays, or **tensors**
- The flow of tensors through the graph is where TensorFlow gets its name. Nodes are assigned to computational devices and execute asynchronously and in parallel once all the tensors on their incoming edges becomes available.



TensorFlow 1.0 Hightlights

- February, 2017
- Python APIs have been changed to resemble NumPy more closely
- Not backwards-compatible $\Longrightarrow$ migration guide and conversion script
- Experimental APIs for Java and Go
- Higher-level API modules tf.layers, tf.metrics, and tf.losses
- Experimental release of XLA, a domain-specific compiler for TensorFlow graphs, that targets CPUs and GPUs
- Introduction of the TensorFlow Debugger (tfdbg), a command-line interface and API for debugging live TensorFlow programs
- New Android demos for object detection and localization, and camera-based image stylization.
- Installation improvements: Python 3 docker images have been added, and TensorFlow's pip packages are now PyPI compliant.

Installing Tensorflow on Windows

(For the platforms other than Windows, refer to TensorFlow homepage)

- CUDA 8.0 & cuDNN 5.1

  (If you are to use CPU only, then skip this step)

  - Check whether your graphic card support CUDA 8.0
  - Go to Nvidia's homepage, and install the library according to your Windows version
  - [Note] cnDNN 5.1 is just files in a folder(usually 'cuda')

  $\Longrightarrow$ Don't forget to add the pathname to personal PATH variable.

- Python 3.5 (I prefer **Anaconda** python because many packages are pre-installed)
  - Current version of Anaconda python is 3.6
  - You must install "**Anaconda3-4.2.0-Windows-x86_64**" (64-bit case) (https://repo.continuum.io/archive/index.html)
  - After installing Anaconda, upgrade pip:

```
C:\kikim>python -m pip install -U pip
```

- Create virtual environment with "**conda**"

```
C:\kikim>conda create -n tensorflow
C:\kikim>activate tensorflow
(tensorflow) C:\kikim>pip install --ignore-installed --upgrade
https://storage.googleapis.com/tensorflow/windows/gpu/tensorflow_gpu-1.1.0-cp35-
cp35m-win_amd64.whl
```

- Validate your installation

```
(tensorflow) C:\kikim>python
>>> import tensorflow as tf
>>> hello = tf.constant('Hello, TensorFlow!')
>>> sess = tf.Session()
>>> print(sess.run(hello))
```

  - If you see warning messages like "The TensorFlow library wasn't compiled to use SSE instructions …." , ignore them. You have No problem. Following code will helps.

```
>>> import os
>>> os.environ['TF_CPP_MIN_LOG_LEVEL']='2'
>>> import tensorflow as tf
```

TensorFlow requires explicit evaluation!

```
In [37]: a = np.zeros((2,2))

In [38]: ta = tf.zeros((2,2))

In [39]: print(a)
[[ 0. 0.]
 [ 0. 0.]]

In [40]: print(ta)
Tensor("zeros_1:0", shape=(2, 2), dtype=float32)

In [41]: print(ta.eval())
[[ 0. 0.]
 [ 0. 0.]]
```

TensorFlow computations define a **computation graph** that has no numerical value until evaluated!

Core TensorFlow Data Structures

- **Constants**
- **Variables**: a modificable tensor that lives in TensorFlow's graph of interacting operations
- **Placeholders**: “symbolic variables”, must be fed with data on execution
- **Session**: encapsulates the environment in which operation objects are executed, and Tensor objects are evaluated

TensorFlow Session Object

- Regular close() method

```
# Build a graph.
import tensorflow as tf
a = tf.placeholder("float")
b = tf.placeholder("float")
c = tf.multiply(a, b)

# Launch the graph in a session.
sess = tf.Session()

# Evaluate the tensor `c`
print(sess.run(c, feed_dict={a: 5, b: 6}))
sess.close()
```

- Context manager

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

- Interactive Session

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

Tensorflow Computational Graph

- Computational Graph: A series of TensorFlow operations arranged into a graph of nodes. Each node takes zero(= constant) or more tensors(=variables) as inputs and produces a tensor as an output.
- Constant:

```
node1 = tf.constant(3.0, tf.float32)
node2 = tf.constant(4.0)  # also tf.float32 implicitly
print(node1, node2)
Tensor("Const:0", shape=(), dtype=float32) Tensor("Const_1:0", shape=(),
dtype=float32)
sess = tf.Session()
print(sess.run([node1, node2]))
[3.0, 4.0]
node3 = tf.add(node1, node2)
print("node3: ", node3)
print("sess.run(node3): ",sess.run(node3))
node3:  Tensor("Add_2:0", shape=(), dtype=float32)
sess.run(node3):  7.0
```

TensorFlow Variables

```
W1 = tf.ones((2,2))
W2 = tf.Variable(tf.zeros((2,2)), name="weights")
with tf.Session() as sess:
    print(sess.run(W1))
    print(sess.run(W2))
[[ 1.  1.]
 [ 1.  1.]]
. . . Whole bunch of ERROR messages . . .
```

```
W1 = tf.ones((2,2))
W2 = tf.Variable(tf.zeros((2,2)), name="weights")
with tf.Session() as sess:
    print(sess.run(W1))
    sess.run(tf.global_variables_initializer())
    print(sess.run(W2))
[[ 1.  1.]
 [ 1.  1.]]
[[ 0.  0.]
 [ 0.  0.]]
```

TensorFlow variables must be initialized before they have values!
Contrast with constant tensors.

Initialization of Variables

```
W = tf.Variable(tf.zeros((2,2)), name="weights")
R = tf.Variable(tf.random_normal((2,2)), name="random_weights")
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    print(sess.run(W))
    print(sess.run(R))
```

Variable objects can be initialized from constants or random values

Initializes all variables with specified values.

Updating Variable State

```
state = tf.Variable(0, name="counter")
new_value = tf.add(state, tf.constant(1))   # Roughly new_value = state + 1
update = tf.assign(state, new_value)        # Roughly state = new_value
with tf.Session() as sess:
    sess.run(tf.global_variables_initializer())
    print(sess.run(state))
    for _ in range(3):
        sess.run(update)
        print(sess.run(state))
```

Roughly,

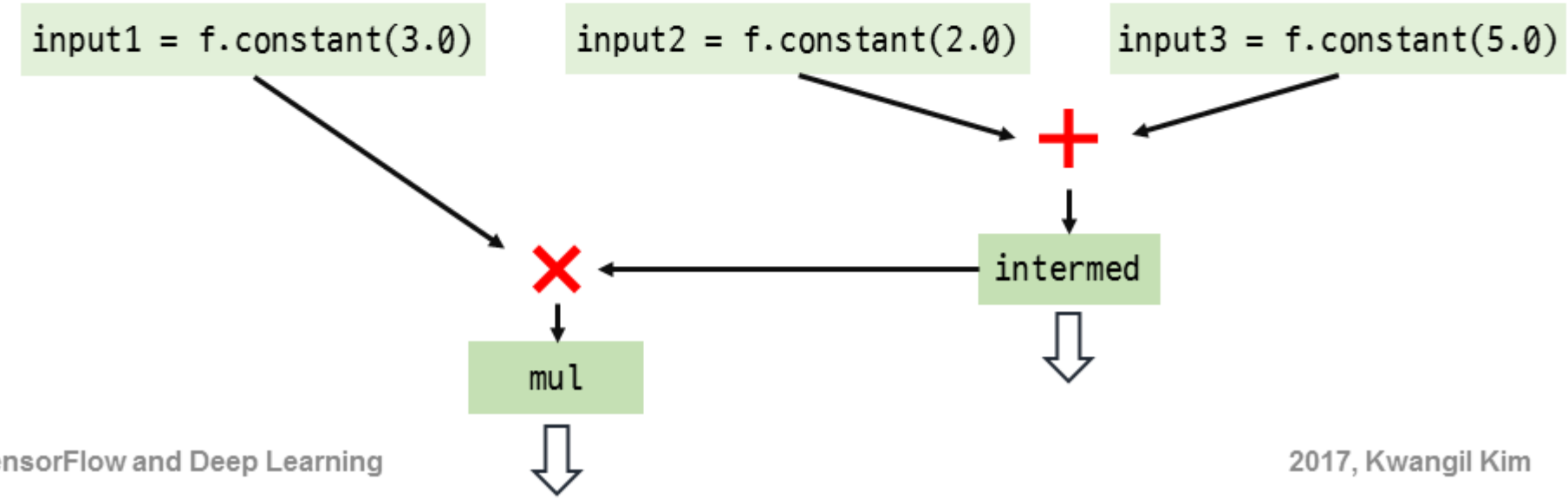
```
state = 0
print(state)
for _ in range(3):
    state = state + 1
    print(state)
```

```
0
1
2
3
```

Fetching Variable State

```
input1 = tf.constant(3.0)
input2 = tf.constant(2.0)
input3 = tf.constant(5.0)
intermed = tf.add(input2, input3)
mul = tf.multiply(input1, intermed)
with tf.Session() as sess:
result = sess.run([mul, intermed])
print(result)
[21.0, 7.0]
```

- Calling `sess.run(var)` on a `tf.Session()` object retrieves its value.
- Can retrieve multiple variables simultaneously with `sess.run([var1, var2])`
  (See Fetches in TF docs)



Inputting Data

- `tf.placeholder` variables: dummy nodes that provide entry points for data to computational graph.
- `feed_dict`: a python dictionary mapping from `tf.placeholder` vars (or their names) to data (numpy arrays, lists, etc.)

```
input1 = tf.placeholder(tf.float32)
input2 = tf.placeholder(tf.float32)
output = tf.multiply(input1, input2)
with tf.Session() as sess:
    print(sess.run([output], feed_dict={input1:[7.], input2:[2.]}))

[array([ 14.], dtype=float32)]
```

Fetch value of output from computation graph

Feed data into computation graph

Variable Scope

- Complicated TensorFlow models can have hundreds of variables.
  - Variable scope is a simple type of namespacing that adds prefixes to variable names within scope
  - `tf.variable_scope()` provides simple name-spacing to avoid clashes
  - `tf.get_variable()` creates/accesses variables from within a variable scope
- **`v = tf.get_variable(name, shape, dtype, initializer)`**
  - `tf.get_variable_scope()[.reuse == False]`

```
with tf.variable_scope("foo"):
    with tf.variable_scope("bar"):
        v = tf.get_variable("v", [1])
assert v.name == "foo/bar/v:0"
```

  - `tf.get_variable_scope().reuse == True`

```
with tf.variable_scope("foo"):
    v = tf.get_variable("v", [1])
with tf.variable_scope("foo", reuse=True):
    v1 = tf.get_variable("v", [1])
assert v1 == v
```

- tf.variable_scope(<scope_name>)

```
with tf.variable_scope("foo"):
    with tf.variable_scope("bar"):
        v = tf.get_variable("v", [1])
assert v.name == "foo/bar/v:0"
```

- **Capturing**

```
with tf.variable_scope("foo") as foo_scope:
    v = tf.get_variable("v", [1])
with tf.variable_scope(foo_scope):
    w = tf.get_variable("w", [1])
with tf.variable_scope(foo_scope, reuse=True):
    v1 = tf.get_variable("v", [1])
    w1 = tf.get_variable("w", [1])
assert v1 == v
assert w1 == w
```

Initializers

```
with tf.variable_scope("foo", initializer=tf.constant_initializer(0.4)):
    v = tf.get_variable("v", [1])
    assert v.eval() == 0.4  # Default initializer as set above.
    w = tf.get_variable("w", [1], initializer=tf.constant_initializer(0.3)):
    assert w.eval() == 0.3  # Specific initializer overrides the default.
    with tf.variable_scope("bar"):
        v = tf.get_variable("v", [1])
        assert v.eval() == 0.4  # Inherited default initializer.
    with tf.variable_scope("baz", initializer=tf.constant_initializer(0.2)):
        v = tf.get_variable("v", [1])
        assert v.eval() == 0.2  # Changed default initializer.
```

- Names of ops

```
with tf.variable_scope("foo"):
    x = 1.0 + tf.get_variable("v", [1])
assert x.op.name == "foo/add"
```

`tf.variable_scope("name")` implicitly opens a `tf.name_scope("name")`

```
with tf.variable_scope("foo"):
    with tf.name_scope("bar"):
        v = tf.get_variable("v", [1])
        x = 1.0 + v
assert v.name == "foo/v:0"
assert x.op.name == "foo/bar/add"
```

Name scopes can be opened in addition to a variable scope, and then they will only affect the names of the ops, but not of variables.

Linear Regression

- Simple linear regression: $y = \mathbf{w}^{\mathrm{T}}\mathbf{x} + b$
- Example: $y = 0.1x + 0.3$
  - Data generation : `y_data = 0.1 * x_data + 0.3 + noise`

    where

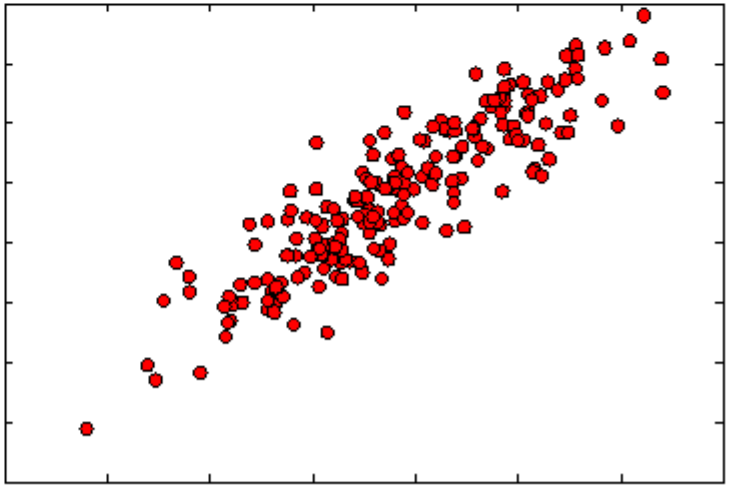
    `x_data` ~ $\mathcal{N}(0.0, 0.55)$, 200 samples

    `noise` ~ $\mathcal{N}(0.0, 0.03)$
  - Loss function(Sum of Squared Errors)

    `loss` $= (y - y\_data)^2$
  - Gradient descent:

    $$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E(\mathbf{w}^{(\tau)})$$

    where the learning rate $\eta = 0.5$



```
%matplotlib inline
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np

# Define input data
n_samples = 200
set_points = []
for i in range(n_samples):
    x1 = np.random.normal(0.0, 0.55)
    y1= x1 * 0.1 + 0.3 + np.random.normal(0.0, 0.03)
    set_points.append([x1, y1])
x_data = [v[0] for v in set_points]
y_data = [v[1] for v in set_points]

# Define variables and model
w = tf.Variable(tf.random_uniform([1], -1.0, 1.0))
b = tf.Variable(tf.zeros([1]))
y = w * x_data + b
```

```
# Define loss function and Optimizer
loss = tf.reduce_mean(tf.square(y - y_data))
optimizer = tf.train.GradientDescentOptimizer(0.5)
train = optimizer.minimize(loss)

init = tf.global_variables_initializer()
sess = tf.Session()
sess.run(init)

# Train the model
for step in range(20):
    sess.run(train)
    if step % 2 == 0:
        plt.plot(x_data, y_data, 'ro')
        plt.plot(x_data, sess.run(w) * x_data + sess.run(b))
        plt.xlabel('x'); plt.ylabel('y')
        plt.xlim(-2,2); plt.ylim(0.1,0.6)
        plt.legend()
        plt.show()
```

- Plots of Results