Make Sense of Deep Neural Networks using TensorBoard

github.com/PythonWorkshop/tensorboard_demos

Arpan Chakraborty

@runoffthemill
github.com/napratin

UDACITY

Getting Started

Note: This notebook is written in Python 3. You'll need Jupyter/iPython to run it.

Local installation

Fetch the repo: github.com/PythonWorkshop/tensorboard_demos

```
git clone git@github.com:PythonWorkshop/tensorboard_demos
cd tensorboard_demos/
```

Option A: Conda install

```
conda env create
source activate tensorflow
jupyter notebook tensorboard_basics.ipynb
```

Option B: Pip install

```
pip3 install numpy matplotlib scikit-learn
pip3 install --upgrade <binary URL for your system>
jupyter notebook tensorboard_basics.ipynb
```

See TensorFlow instructions to pick the correct binary URL for your system.

Binder

launch binder

If you have trouble getting TensorFlow to work, hit the **launch binder** badge to run in the cloud. Note that this is an experimental feature.

Neural Networks Primer

A brief introduction to neural networks, with an eye on data structures.

Simple linear model

Express output (y) as a linear combination of inputs (x):

$$y = x_0 w_0 + x_1 w_1 + \ldots + x_n w_n + b$$

Learning goal: Given enough examples of x and y, find w and b that best capture the mapping.

Simple linear model

Express output (y) as a linear combination of inputs (x):

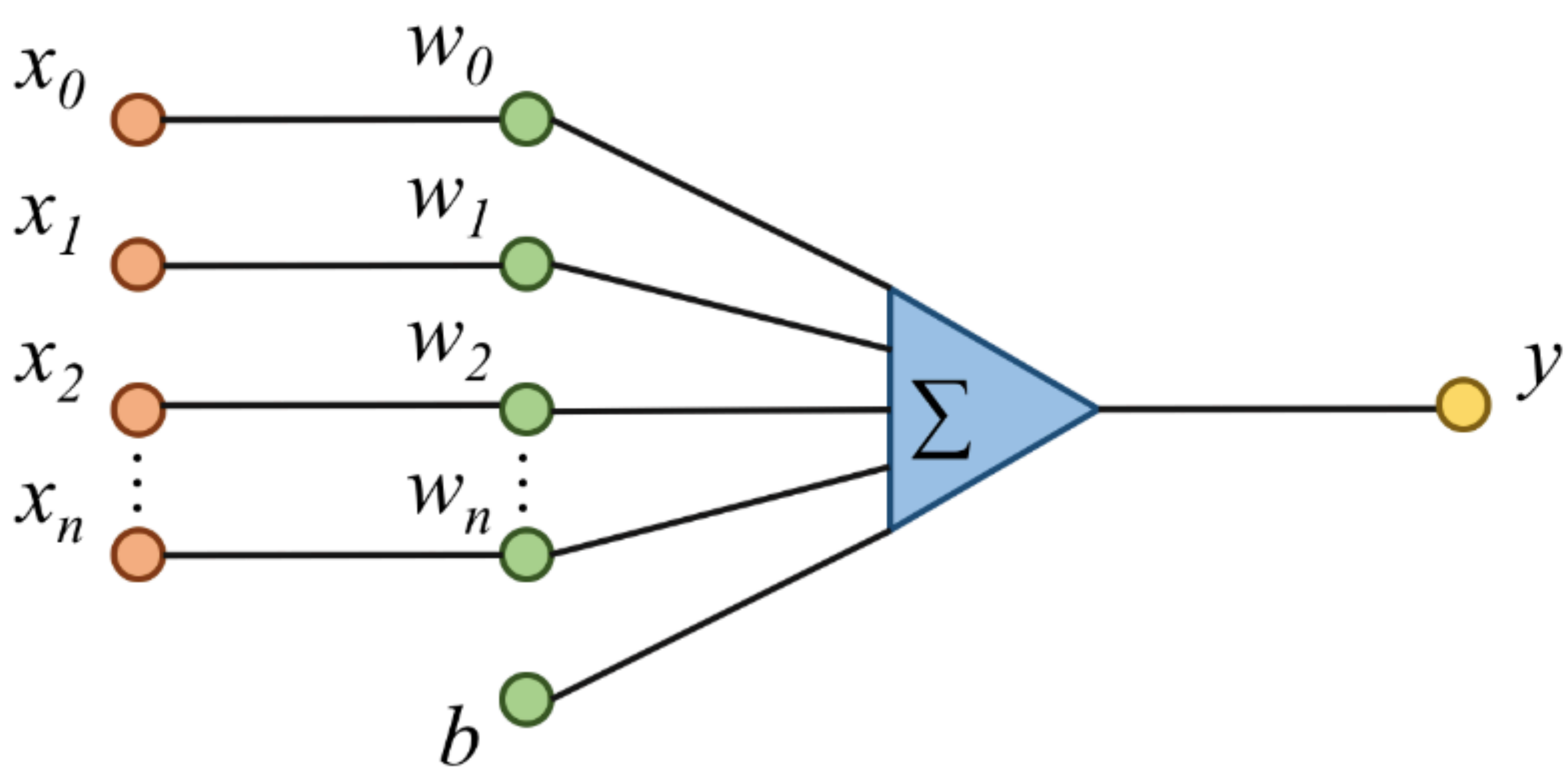
$$y = x_0 w_0 + x_1 w_1 + \ldots + x_n w_n + b$$

Learning goal: Given enough examples of x and y, find w and b that best capture the mapping.

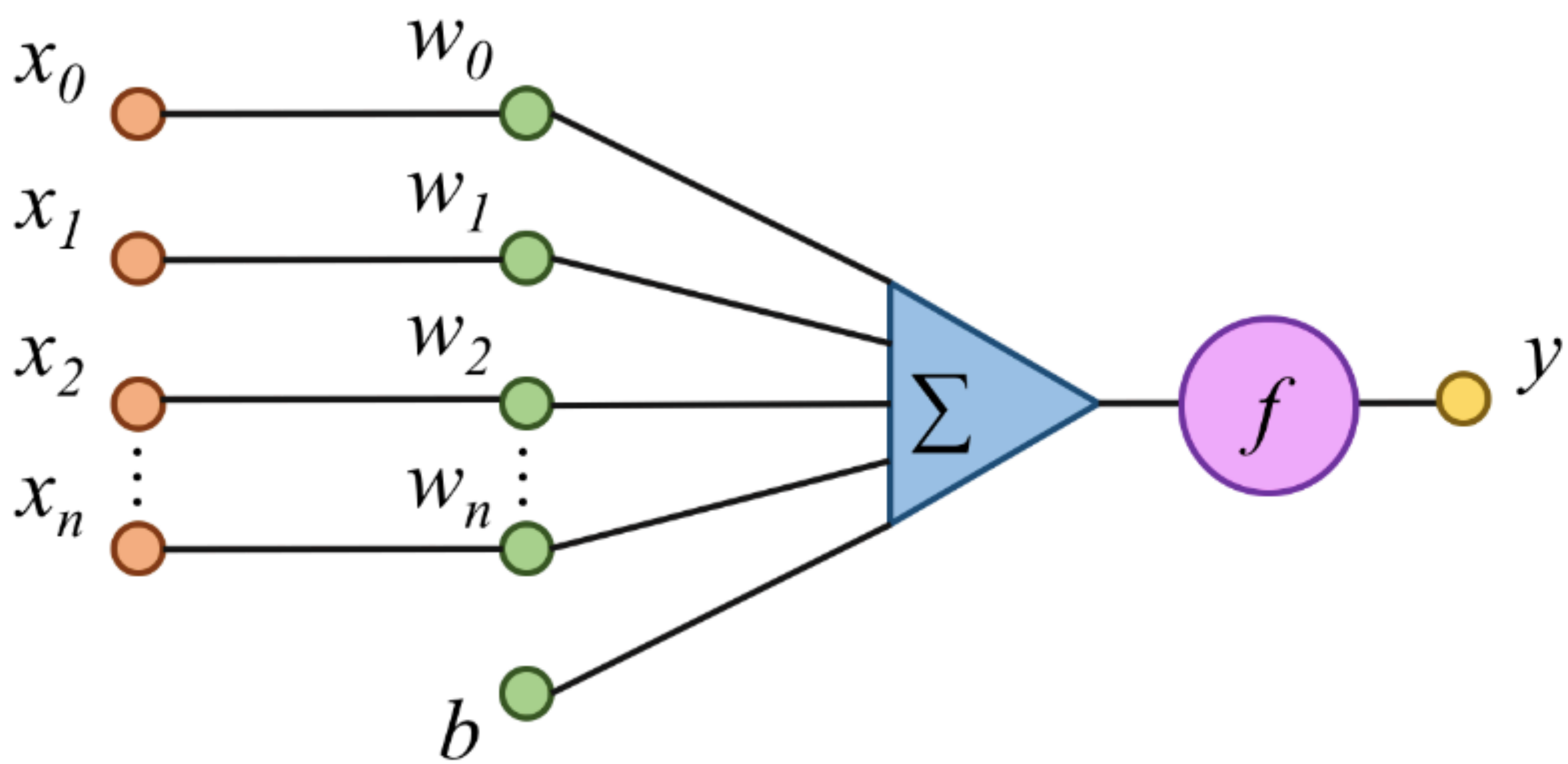
Linear model: Vector notation

$$y = \begin{bmatrix} x_0 & x_1 & \ldots & x_n \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{bmatrix} + b$$

Linear model: Graphical representation



Single neuron model



Neuron model: Vector notation

$$y = f(\begin{bmatrix} x_0 & x_1 & \dots & x_n \end{bmatrix} \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{bmatrix} + b)$$

Here $f()$ is a *non-linear* activation function, that maps real-valued output y to some target space, e.g. the sigmoid function maps to $[0, 1]$.

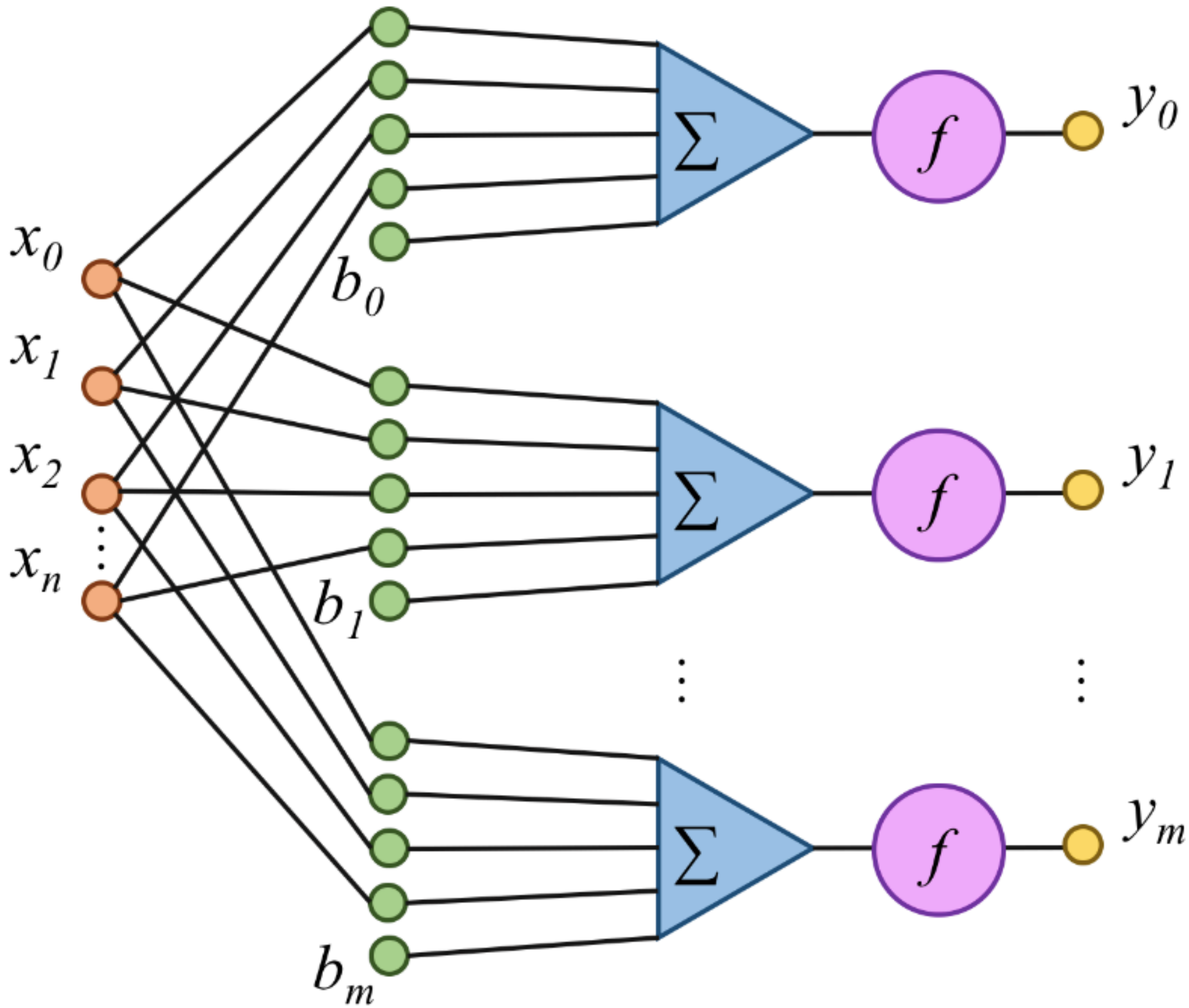
Multiple neurons

Instead of a single neuron, now we have m of them.

$$\begin{bmatrix} y_0 & y_1 & \dots & y_m \end{bmatrix} = f(\begin{bmatrix} x_0 & x_1 & \dots & x_n \end{bmatrix} \begin{bmatrix} w_{00} & w_{01} & \dots & w_{0m} \\ w_{10} & w_{11} & \dots & w_{1m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n0} & w_{n1} & \dots & w_{nm} \end{bmatrix} \\ + \begin{bmatrix} b_0 & b_1 & \dots & b_m \end{bmatrix})$$

This is equivalent to a *layer* of neurons.

Multiple neurons: Graphical representation



Process multiple data samples at once

Instead of a single data sample, now we are dealing with k samples in parallel.

$$\begin{bmatrix} y_{00} & y_{01} & \dots & y_{0m} \\ y_{10} & y_{11} & \dots & y_{1m} \\ \vdots & \vdots & \ddots & \vdots \\ y_{k0} & y_{k1} & \dots & y_{km} \end{bmatrix} = f(\begin{bmatrix} x_{00} & x_{01} & \dots & x_{0n} \\ x_{10} & x_{11} & \dots & x_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{k0} & x_{k1} & \dots & x_{kn} \end{bmatrix}\begin{bmatrix} w_{00} & w_{01} & \dots & w_{0m} \\ w_{10} & w_{11} & \dots & w_{1m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n0} & w_{n1} & \dots & w_{nm} \end{bmatrix} + \begin{bmatrix} b_0 & b_1 & \dots & b_m \end{bmatrix})$$

Note: Here the bias values b are repeated for each sample (row).

Process multiple data samples at once

Instead of a single data sample, now we are dealing with k samples in parallel.

$$\begin{bmatrix} y_{00} & y_{01} & \dots & y_{0m} \\ y_{10} & y_{11} & \dots & y_{1m} \\ \vdots & \vdots & \ddots & \vdots \\ y_{k0} & y_{k1} & \dots & y_{km} \end{bmatrix} = f(\begin{bmatrix} x_{00} & x_{01} & \dots & x_{0n} \\ x_{10} & x_{11} & \dots & x_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{k0} & x_{k1} & \dots & x_{kn} \end{bmatrix}\begin{bmatrix} w_{00} & w_{01} & \dots & w_{0m} \\ w_{10} & w_{11} & \dots & w_{1m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n0} & w_{n1} & \dots & w_{nm} \end{bmatrix} + \begin{bmatrix} b_0 & b_1 & \dots & b_m \end{bmatrix})$$

Note: Here the bias values b are repeated for each sample (row).

What does this all look like?

Process multiple data samples at once

Instead of a single data sample, now we are dealing with k samples in parallel.

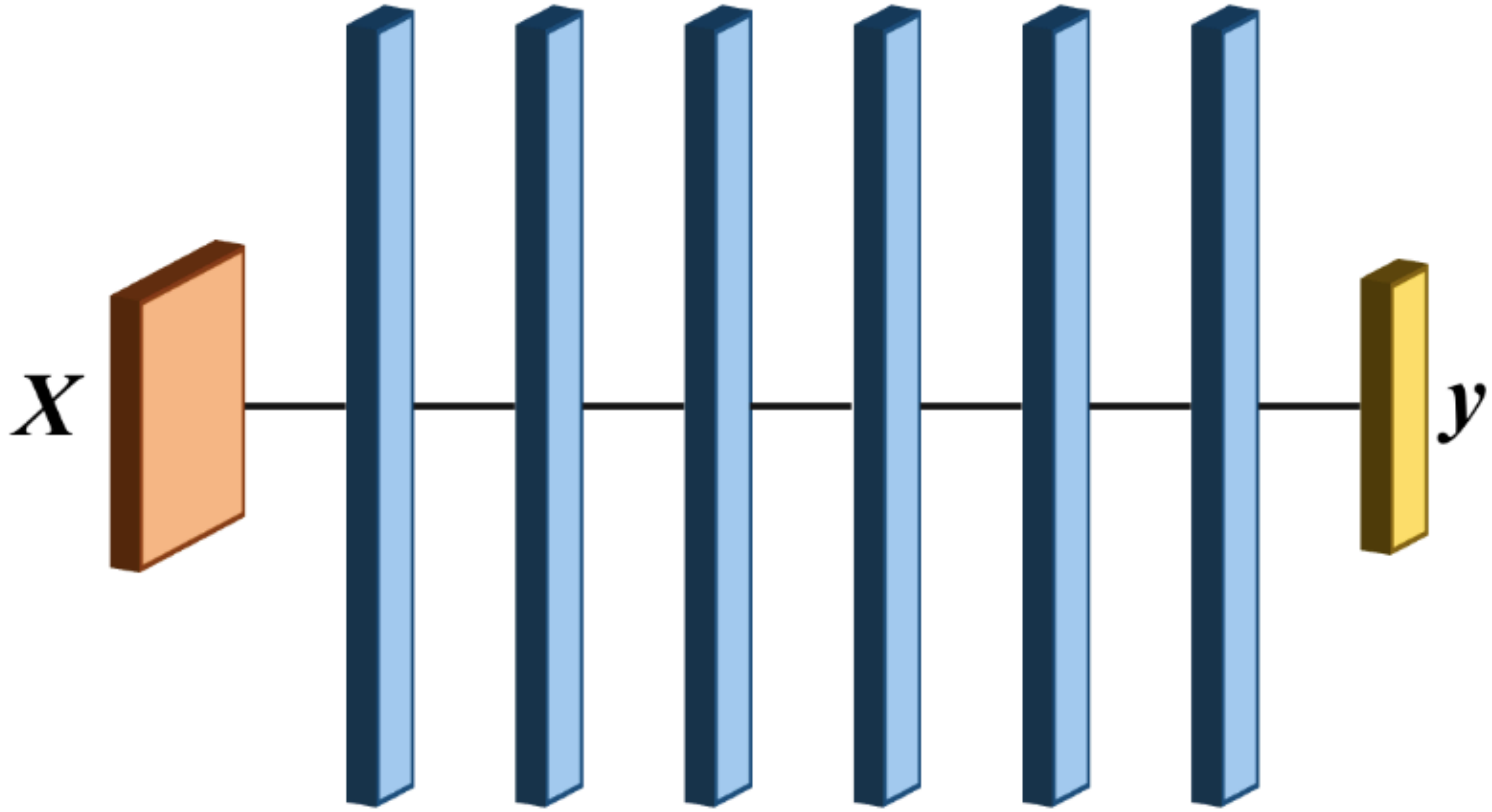
$$\begin{bmatrix} y_{00} & y_{01} & \dots & y_{0m} \\ y_{10} & y_{11} & \dots & y_{1m} \\ \vdots & \vdots & \ddots & \vdots \\ y_{k0} & y_{k1} & \dots & y_{km} \end{bmatrix} = f(\begin{bmatrix} x_{00} & x_{01} & \dots & x_{0n} \\ x_{10} & x_{11} & \dots & x_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{k0} & x_{k1} & \dots & x_{kn} \end{bmatrix}\begin{bmatrix} w_{00} & w_{01} & \dots & w_{0m} \\ w_{10} & w_{11} & \dots & w_{1m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n0} & w_{n1} & \dots & w_{nm} \end{bmatrix} + \begin{bmatrix} b_0 & b_1 & \dots & b_m \end{bmatrix})$$

Note: Here the bias values b are repeated for each sample (row).

What does this all look like?

TENSORS!

Deep Neural Networks



Ready for some Machine Learning?

Generate Some Data to Classify

In [1]:

```
import numpy as np

num_examples, num_features = (1000, 2)  # dataset size
num_classes = 2  # binary classification task

X = np.random.random((num_examples, num_features))
y = np.int_(X[:, 0] * X[:, 0] + X[:, 1] >= 1).reshape(-1, 1)
```

In [2]:
```python
print("Features (X): {shape[0]}x{shape[1]}".format(shape=X.shape))
print(X[:10])
```

```
Features (X): 1000x2
[[ 0.58578979  0.96048674]
 [ 0.41171871  0.52405548]
 [ 0.50849822  0.75811416]
 [ 0.6386604   0.23468108]
 [ 0.14959901  0.54418769]
 [ 0.29773577  0.72048556]
 [ 0.67330963  0.31292161]
 [ 0.87784032  0.74873901]
 [ 0.60013032  0.35332369]
 [ 0.9033483   0.74016079]]
```

In [3]:
```
print("Labels (y): {shape[0]}".format(shape=y.shape))
print(y[:10])
```

```
Labels (y): 1000
[[1]
 [0]
 [1]
 [0]
 [0]
 [0]
 [0]
 [1]
 [0]
 [1]]
```

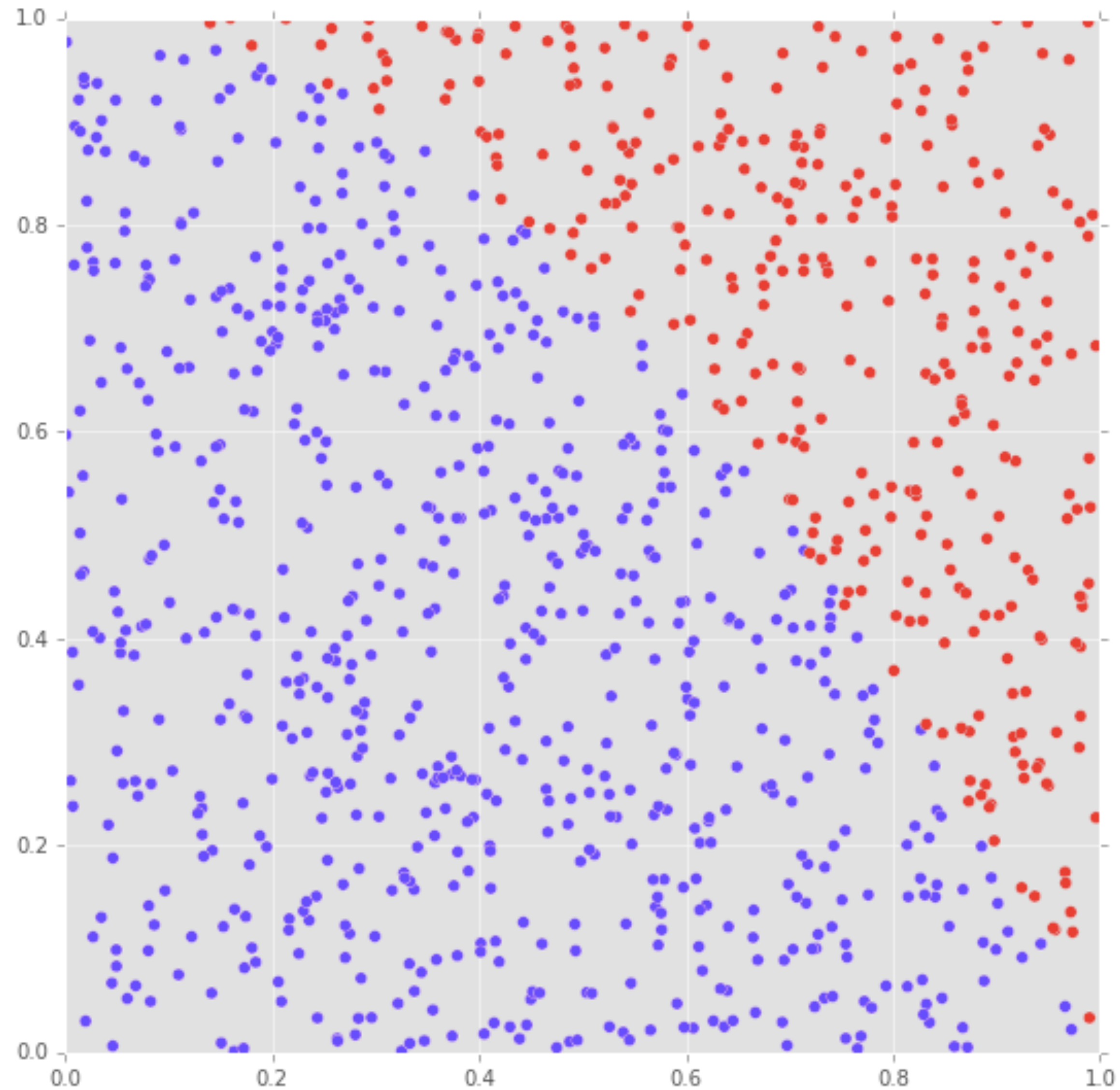
Visualize Data

In [4]:

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use('ggplot')

def plot_data_2D(X, **kwargs):
    """Plot 2D data points in X = np.array([(x0, y0), (x1, y1), ...])."""
    fig, ax = plt.subplots(figsize=(9, 9))
    ax.scatter(X[:, 0], X[:, 1],
               s=35, cmap=plt.cm.get_cmap('rainbow', num_classes), **kwargs)
    ax.set_aspect('equal')
    ax.set_xlim(0, 1)
    ax.set_ylim(0, 1)
    return fig, ax
```

In [5]:
```
plot_data_2D(X, c=y)
```

Out[5]:
```
(<matplotlib.figure.Figure at 0x10986c048>,
 <matplotlib.axes._subplots.AxesSubplot at 0x109863b00>)
```



Prepare Data for Training and Testing

In [6]:

```
# Split data into training and test sets
from sklearn.cross_validation import train_test_split

test_size = 300
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=test_size)
print("Split dataset: {} training, {} test samples".format(len(X_train),
len(X_test)))
```

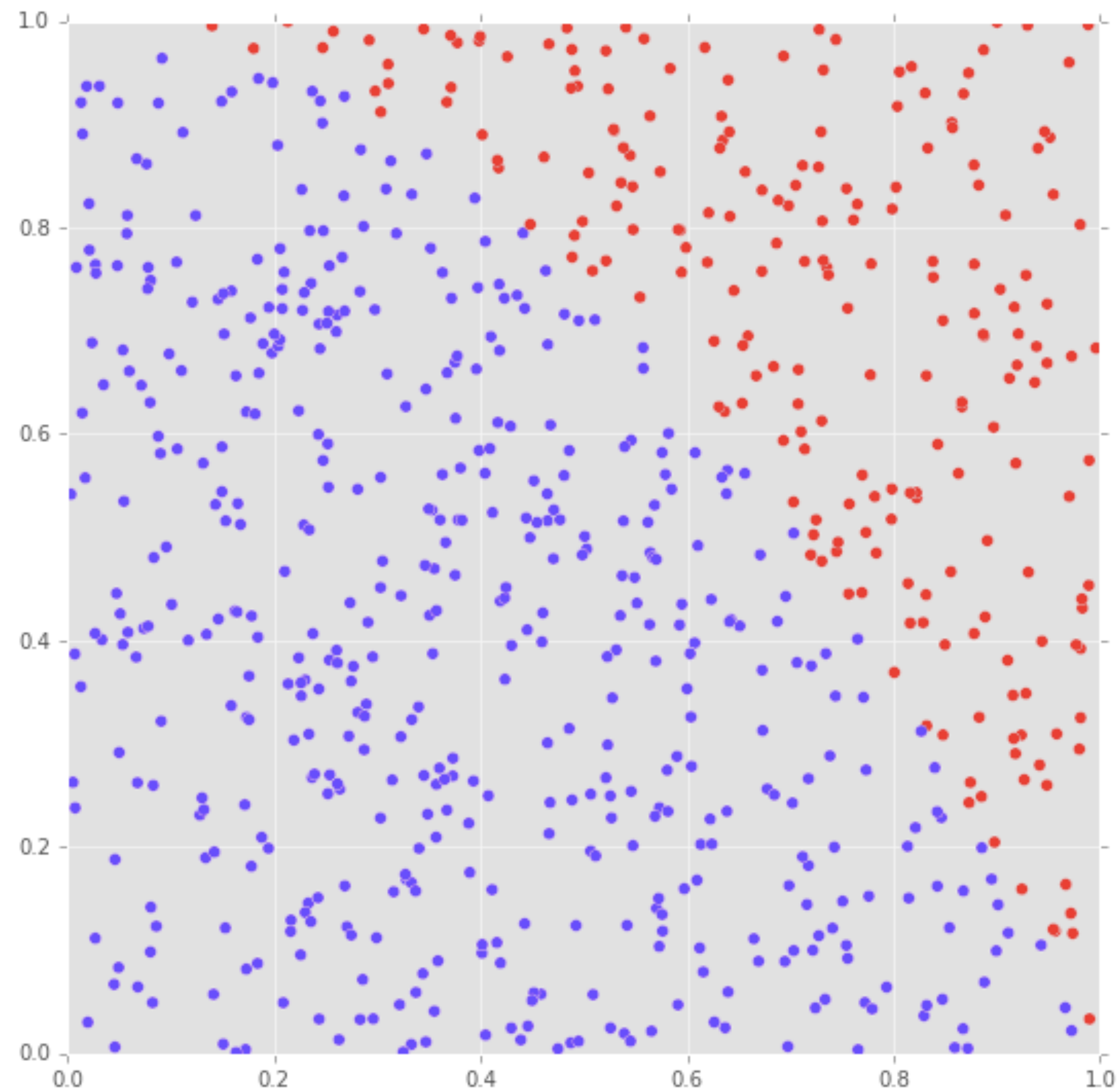
```
Split dataset: 700 training, 300 test samples
```

In [7]:

```python
# Plot training data
plot_data_2D(X_train, c=y_train)
```

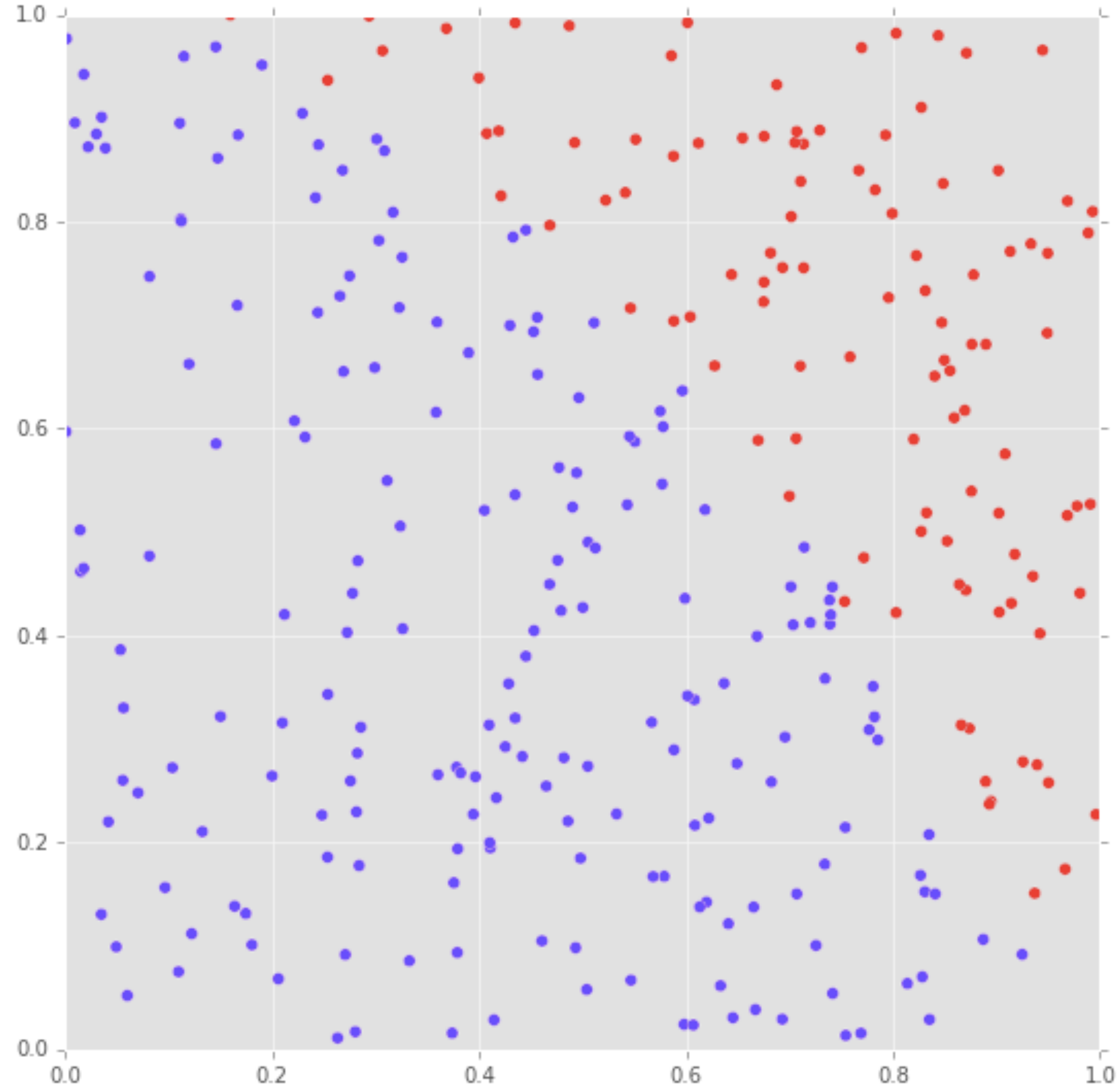
Out[7]:

```
(<matplotlib.figure.Figure at 0x10afc7f28>,
 <matplotlib.axes._subplots.AxesSubplot at 0x10f2a8320>)
```



In [8]:
```
# Plot test data
plot_data_2D(X_test, c=y_test)
```

Out[8]:
```
(<matplotlib.figure.Figure at 0x10f2f7278>,
 <matplotlib.axes._subplots.AxesSubplot at 0x10f30ccf8>)
```



Build Computation Graph

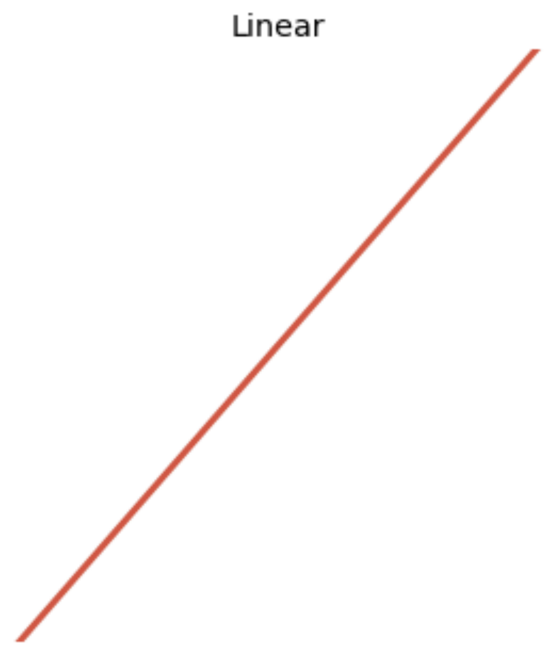
This graph defines the structure of your Neural Network as well as information flow.

In [9]:
```
import tensorflow as tf

# Create a TensorFlow session
session = tf.Session()

# Placeholders for input features and ground truth labels
X_placeholder = tf.placeholder(tf.float32,
                               shape=(None, num_features),
                               name="X")
y_placeholder = tf.placeholder(tf.int64,
                               shape=(None, 1),
                               name="y")
```

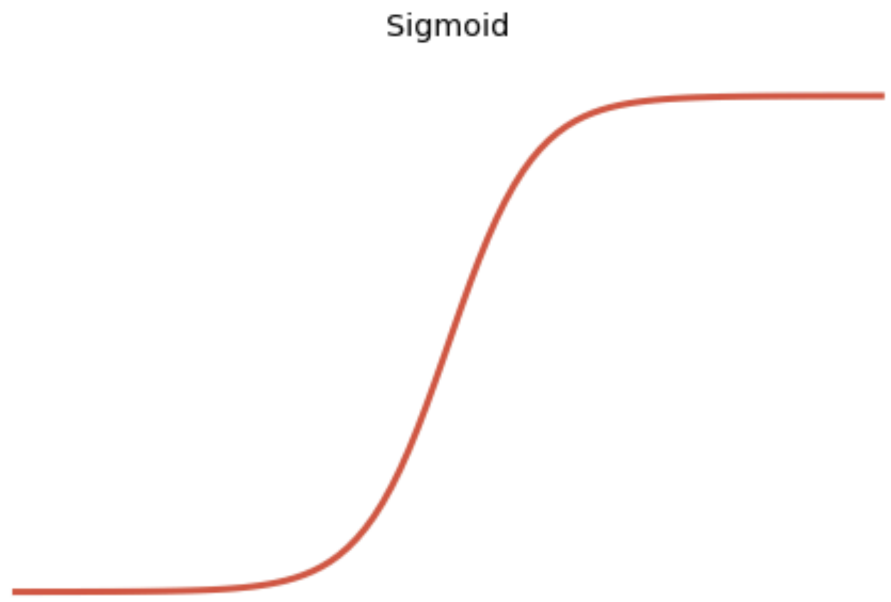
Define layer creation functions

In [11]:
```
def linear_layer(input_tensor, num_units):
    """Linear activation layer: output = inputs * weights + biases"""
    num_inputs = input_tensor.get_shape()[1].value  # inspect tensor size
    weights = tf.Variable(
        tf.truncated_normal([num_inputs, num_units], stddev=1.0),
        name='weights')
    biases = tf.Variable(
        tf.zeros([num_units]),
        name='biases')
    return tf.add(tf.matmul(input_tensor, weights, name='multiply'), biases,
                  name='add')
```

In [12]:
```
plot_activation(lambda x: 0.5 + 0.1*x, title='Linear')  # example
```

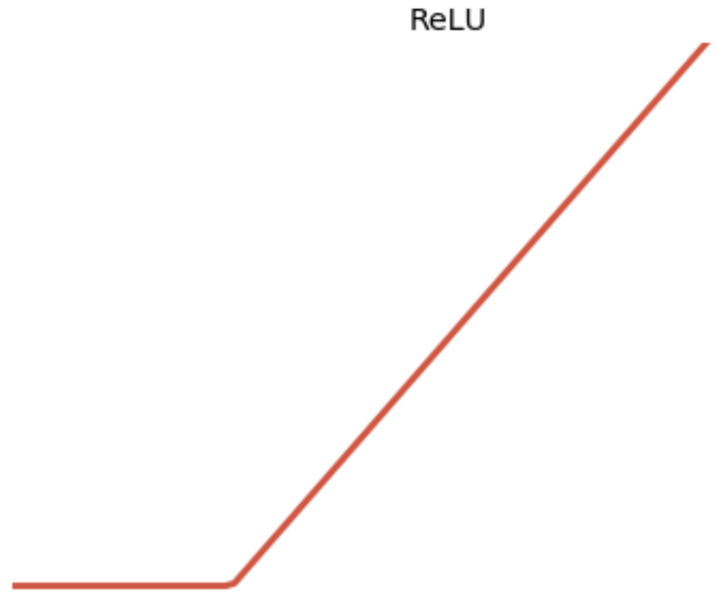


```
In [13]: def sigmoid_layer(input_tensor, num_units):
             """Sigmoid activation layer: output = sigmoid(inputs * weights + biases)"""
             return tf.nn.sigmoid(linear_layer(input_tensor, num_units), name='sigmoid')

         plot_activation(lambda x: 1.0 / (1.0 + np.exp(-x)), title='Sigmoid')  # example
```



In [14]:

```python
def relu_layer(input_tensor, num_units):
    """ReLU activation layer: output = ReLU(inputs * weights + biases)"""
    return tf.nn.relu(linear_layer(input_tensor, num_units), name='relu')

plot_activation(lambda x: (0.5 + 0.1*x).clip(min=0), title='ReLU')  # example
```



Create network structure

In [15]:
```
# Let's make a network with 2 hidden layers (and an output layer)
hidden1_num_units = 10
hidden2_num_units = 3
output_num_units = 1  # binary classification needs only 1 output

with tf.name_scope('hidden1'):
    hidden1 = relu_layer(X_placeholder, hidden1_num_units)

with tf.name_scope('hidden2'):
    hidden2 = relu_layer(hidden1, hidden2_num_units)

with tf.name_scope('output'):
    output = sigmoid_layer(hidden2, output_num_units)
```

Choose an error metric

In [16]:
```
def l2_loss(logits, labels):
    """Euclidean distance or L2-norm: sqrt(sum((logits - labels)^2))"""
    labels = tf.to_float(labels)
    return tf.nn.l2_loss(logits - labels, name='l2_loss')

with tf.name_scope('error'):
    error = l2_loss(output, y_placeholder)  # predicted vs. true labels
    tf.scalar_summary(error.op.name, error)  # write error (loss) to log
```

In [18]:
```
print("Nodes in computation graph:"
    + "\n  Input features: {}".format(X_placeholder)
    + "\n  Hidden layer 1: {}".format(hidden1)
    + "\n  Hidden layer 2: {}".format(hidden2)
    + "\n  Output labels : {}".format(output)
    + "\n  Ground truth  : {}".format(y_placeholder)
    + "\n  Error metric  : {}".format(error))
```

```
Nodes in computation graph:
  Input features: Tensor("X:0", shape=(?, 2), dtype=float32)
  Hidden layer 1: Tensor("hidden1/relu:0", shape=(?, 10), dtype=float32)
  Hidden layer 2: Tensor("hidden2/relu:0", shape=(?, 3), dtype=float32)
  Output labels : Tensor("output/sigmoid:0", shape=(?, 1), dtype=float32)
  Ground truth  : Tensor("y:0", shape=(?, 1), dtype=int64)
  Error metric  : Tensor("error/l2_loss:0", shape=(), dtype=float32)
```

Setup Logging

These logs are later read and visualized by TensorBoard.

In [19]:
```
import time
import os

log_basedir = "logs"
run_label = time.strftime('%Y-%m-%d_%H-%M-%S')  # e.g. 2016-08-18_21-30-45
log_path = os.path.join(log_basedir, run_label)
all_summaries = tf.merge_all_summaries()
summary_writer = tf.train.SummaryWriter(log_path, session.graph)
print("Logging to: {}".format(log_path))
```

```
Logging to: logs/2016-08-20_17-35-19
```

Train your Model

In [20]:
```
# Pick a training algorithm
def sgd_train(error, learning_rate=0.01):
    """Gradient descent optimizer for training.

    Creates an optimizer to compute and apply gradients to all trainable variables.

    Args:
        error: Error (loss) metric.
        learning_rate: Controls the size of each step the optimizer takes.
    Returns:
        training: Training operation, ready to be called with tf.Session.run().
    """
    optimizer = tf.train.GradientDescentOptimizer(learning_rate)
    return optimizer.minimize(error)

with tf.name_scope('training'):
    training = sgd_train(error)
```

In [21]:

```
# Define training parameters
num_steps = 1000  # how many iterations to train for
batch_size = 100  # how many samples in each iteration

# Initialize variables
init_op = tf.initialize_all_variables()
session.run(init_op)
```

In [22]:

```python
# Run training operation for num_steps iterations
for step in range(num_steps):
    # Randomly pick batch_size samples from training set
    sample_idx = np.random.choice(len(X_train), batch_size, replace=False)
    feed_dict = {
        X_placeholder: X_train[sample_idx, :],
        y_placeholder: y_train[sample_idx, :]
    }
    # Note: feed_dict uses placeholder objects as key!

    # Train for one iteration, time it
    start_time = time.time()
    _, error_value = session.run([training, error], feed_dict=feed_dict)
    duration = time.time() - start_time

    # Print an overview and write summaries (logs) every 100 iterations
    if step % 100 == 0 or step == (num_steps - 1):
        print("Step {:4d}: training error = {:5.2f} ({:.3f} sec)"
              .format(step, error_value, duration))
        summary_str = session.run(all_summaries, feed_dict=feed_dict)
        summary_writer.add_summary(summary_str, step)
        summary_writer.flush()
```

```
Step    0: training error = 21.90 (0.016 sec)
Step  100: training error =  2.40 (0.004 sec)
Step  200: training error =  3.04 (0.003 sec)
Step  300: training error =  0.97 (0.003 sec)
Step  400: training error =  1.54 (0.002 sec)
Step  500: training error =  0.73 (0.004 sec)
Step  600: training error =  0.56 (0.001 sec)
Step  700: training error =  0.69 (0.002 sec)
Step  800: training error =  2.18 (0.003 sec)
Step  900: training error =  1.28 (0.003 sec)
Step  999: training error =  1.33 (0.003 sec)
```

Test your Model

In [23]:

```
# Check performance on test set
y_test_pred, test_error = session.run([output, error], feed_dict={
    X_placeholder: X_test,
    y_placeholder: y_test
})
# Note: The placeholder shapes must be compatible with the tensors being supplied!
```

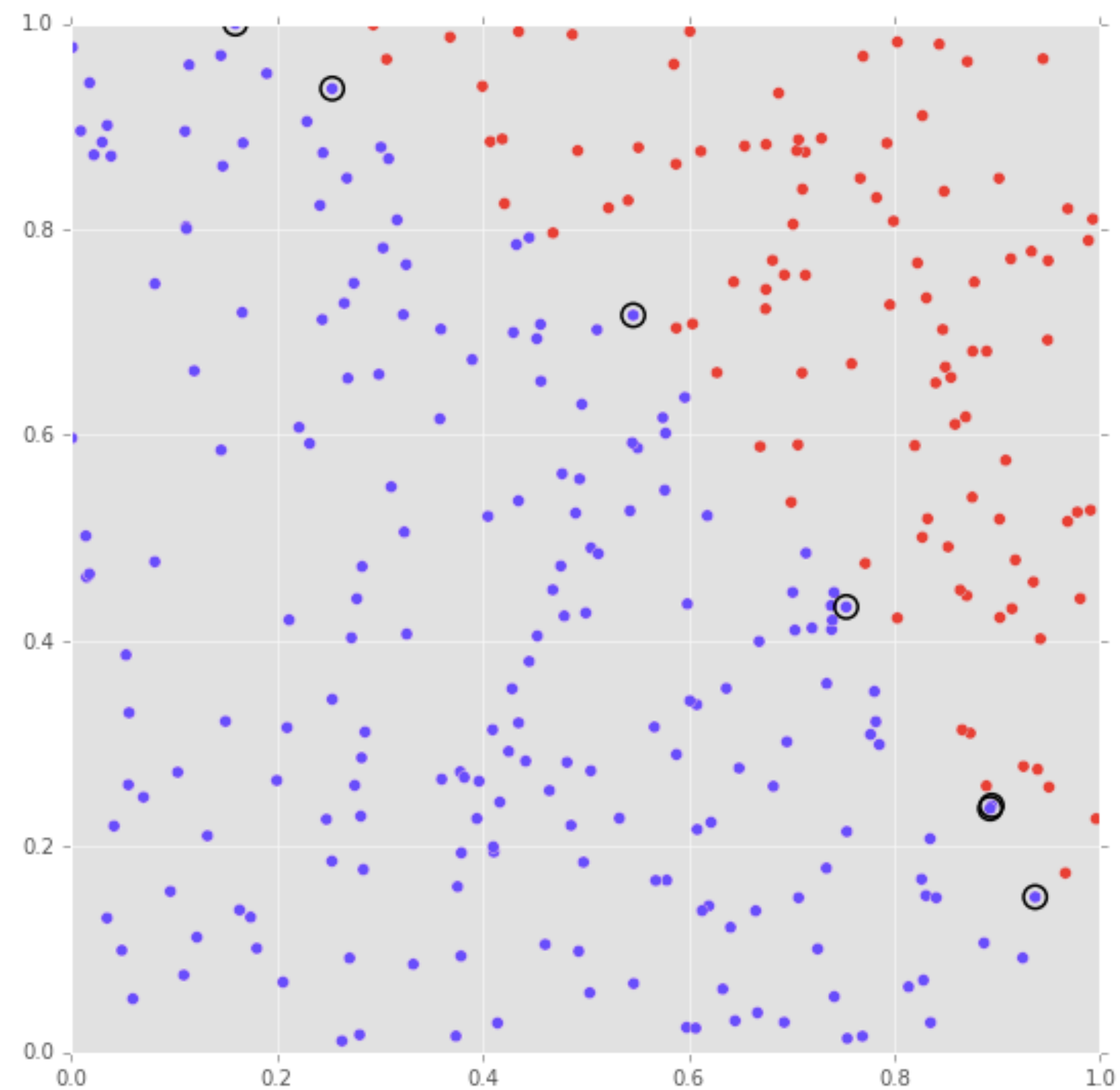
In [24]:

```
y_test_pred = np.int_(np.round_(y_test_pred))
mismatches = (y_test_pred - y_test).flat != 0
print("Test error = {:.2f} ({} mismatches)"
      .format(test_error, sum(np.int_(mismatches))))

_, ax = plot_data_2D(X_test, c=y_test_pred)
ax.scatter(X_test[mismatches, 0], X_test[mismatches, 1],
           s=128, marker='o', facecolors='none', edgecolors='black', linewidths=1.5)
```

```
Test error = 2.59 (7 mismatches)
```

Out[24]:

```
<matplotlib.collections.PathCollection at 0x1149c2710>
```



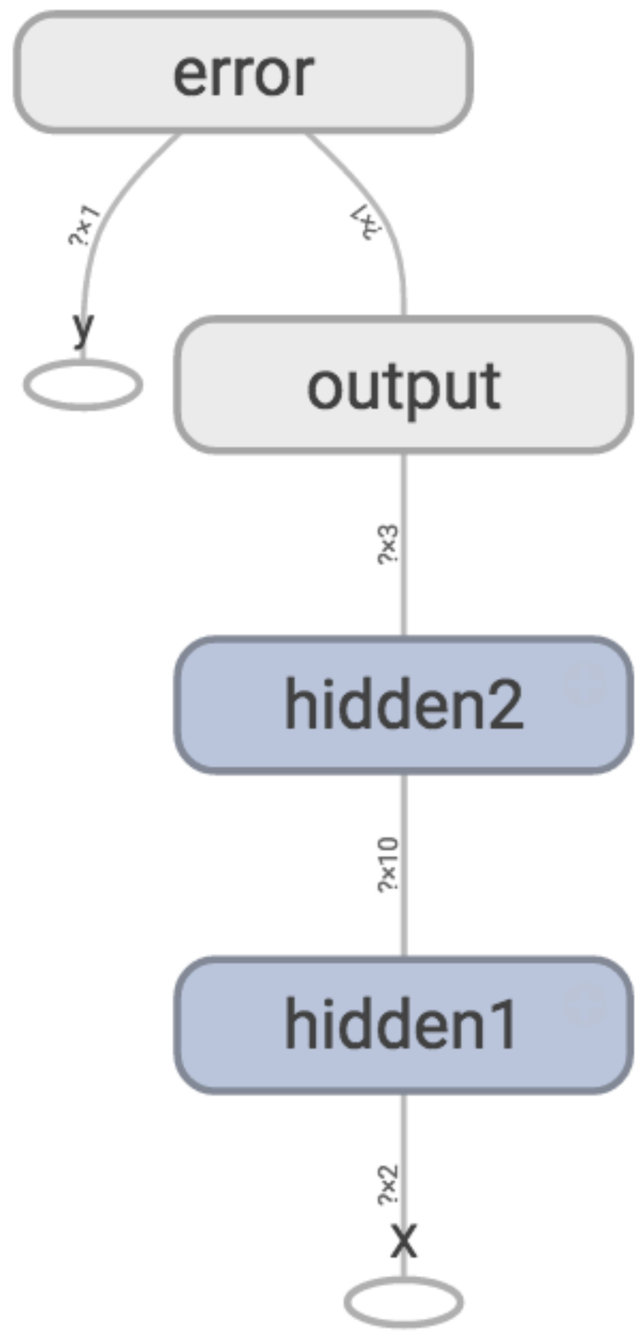
Visualize using TensorBoard

Run TensorBoard with the following command, passing in the appropriate log directory:

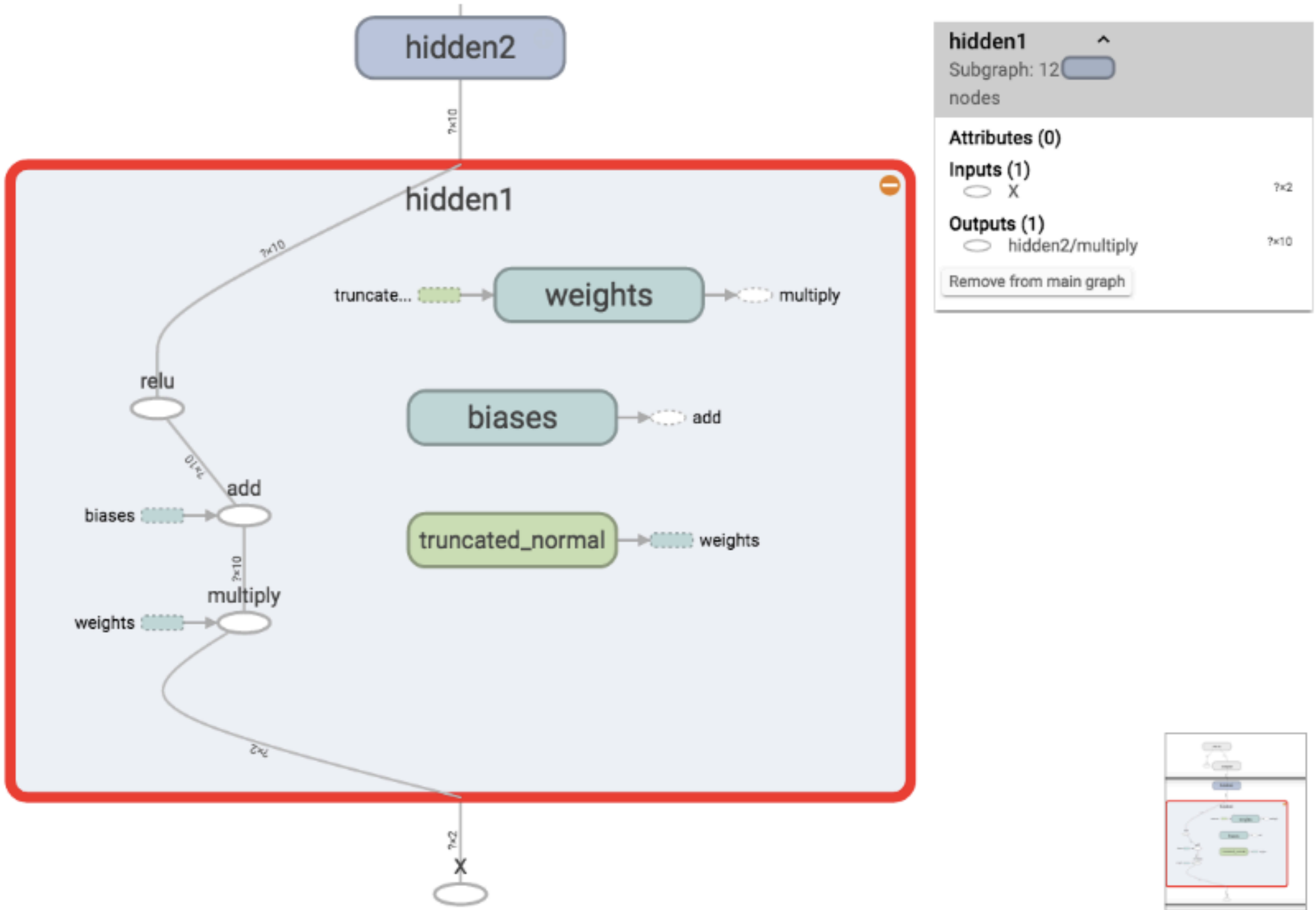
```
tensorboard --logdir=logs
```

And then open the URL that gets printed, in your browser (typically: http://0.0.0.0:6006).

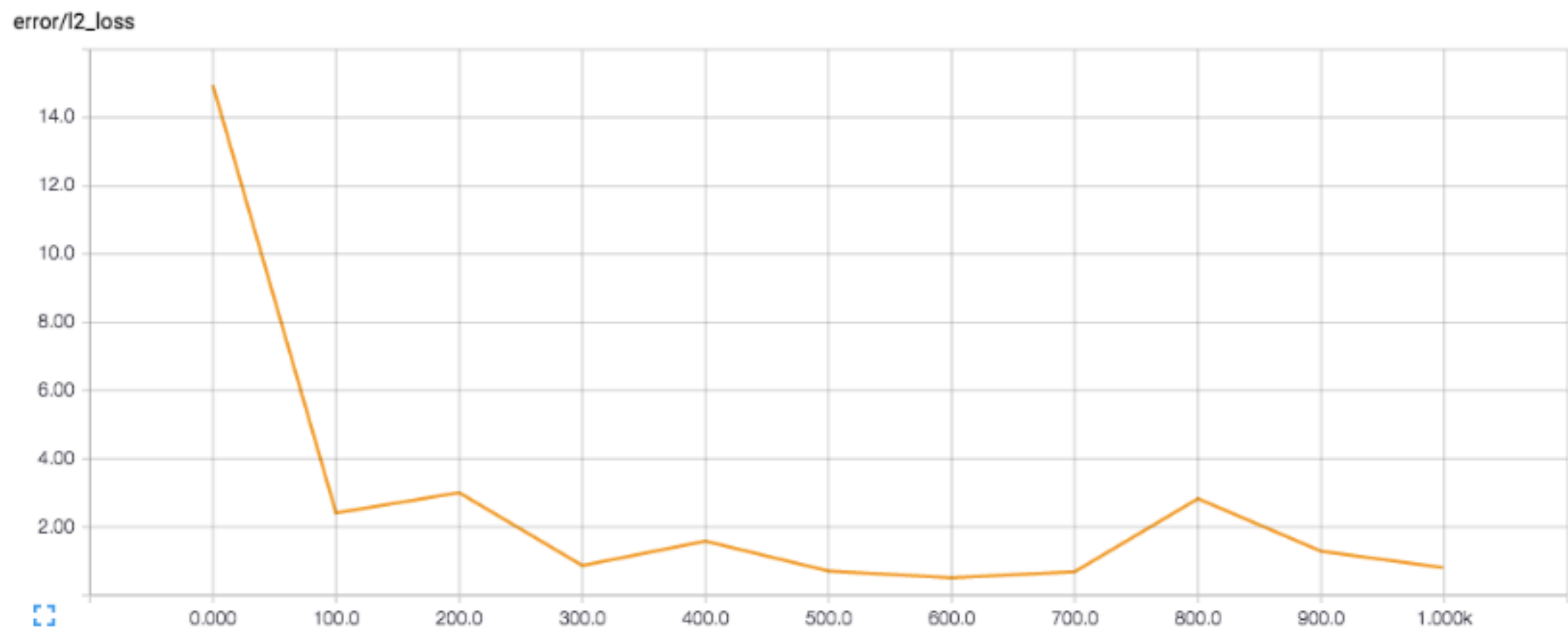
Computation graph



Layer detail: `hidden1`



Error plot


error/l2_loss
14.0
12.0
10.0
8.00
6.00
4.00
2.00
0.000
100.0
200.0
300.0
400.0
500.0
600.0
700.0
800.0
900.0
1.000k

Summary

- Define a neural network model as nodes of a computation graph in TensorFlow.
- Train your model by presenting labeled data and applying an optimization algorithm.
- Log important values like error during training.
- Test your resulting model on unseen data.
- Visualize computation graph and logged data using TensorBoard.

***Note**: You can convert this notebook into slides (HTML + reveal.js) and serve them using:*

```
jupyter nbconvert tensorboard_basics.ipynb --to slides --post serve
```

Make Sense of Deep Neural Networks using TensorBoard

github.com/PythonWorkshop/tensorboard_demos

Arpan Chakraborty

@runoffthemill
github.com/napratin

UDACITY