Learning Systems 2018:
Lecture 4 – Tensorflow Overview

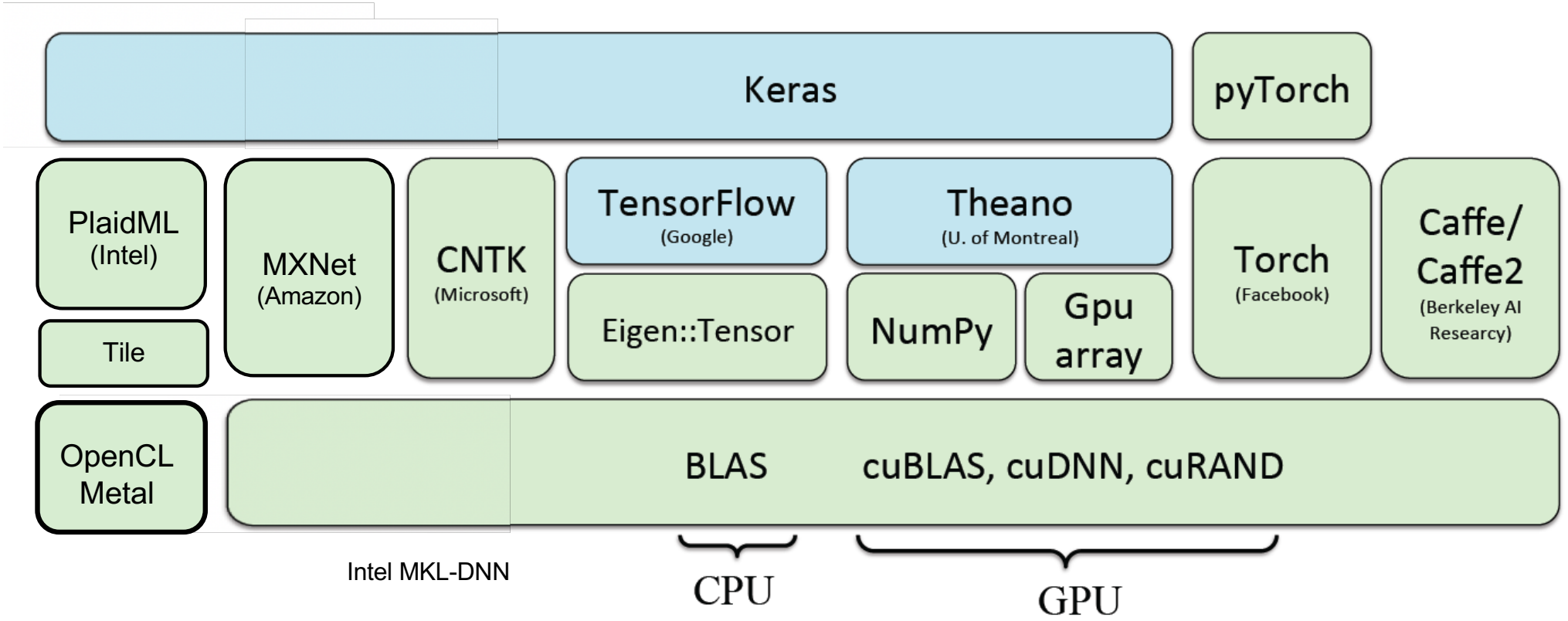
Crescat scientia; vita excolatur

Ian Foster and Rick Stevens
Argonne National Laboratory
The University of Chicago

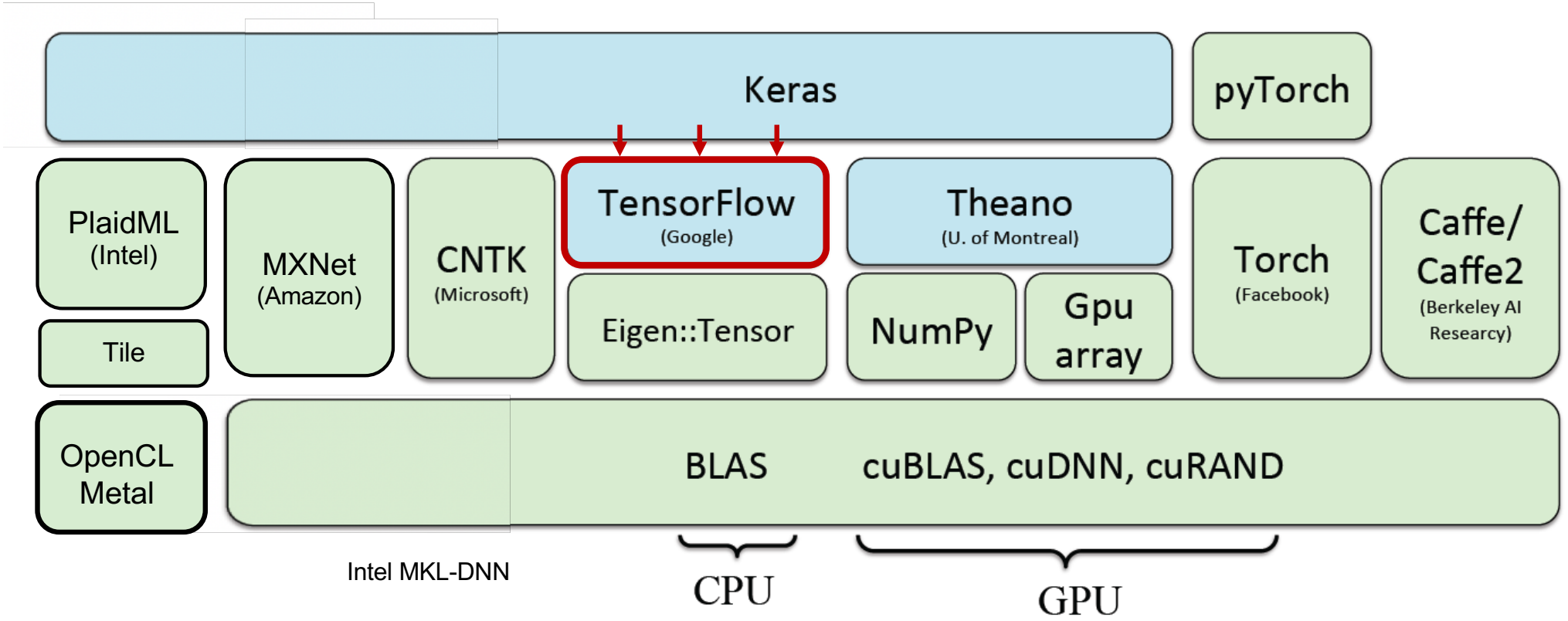
Resources

- https://medium.com/@the1ju/simple-logistic-regression-using-keras-249e0cc9a970
- https://github.com/pbhatnagar3/cs224s-tensorflow-tutorial/blob/master/tensorflow%20MNIST.ipynb
- https://web.stanford.edu/class/cs224s/lectures/Tensorflow-tutorial.pdf
- https://www.tensorflow.org/guide/summaries_and_tensorboard

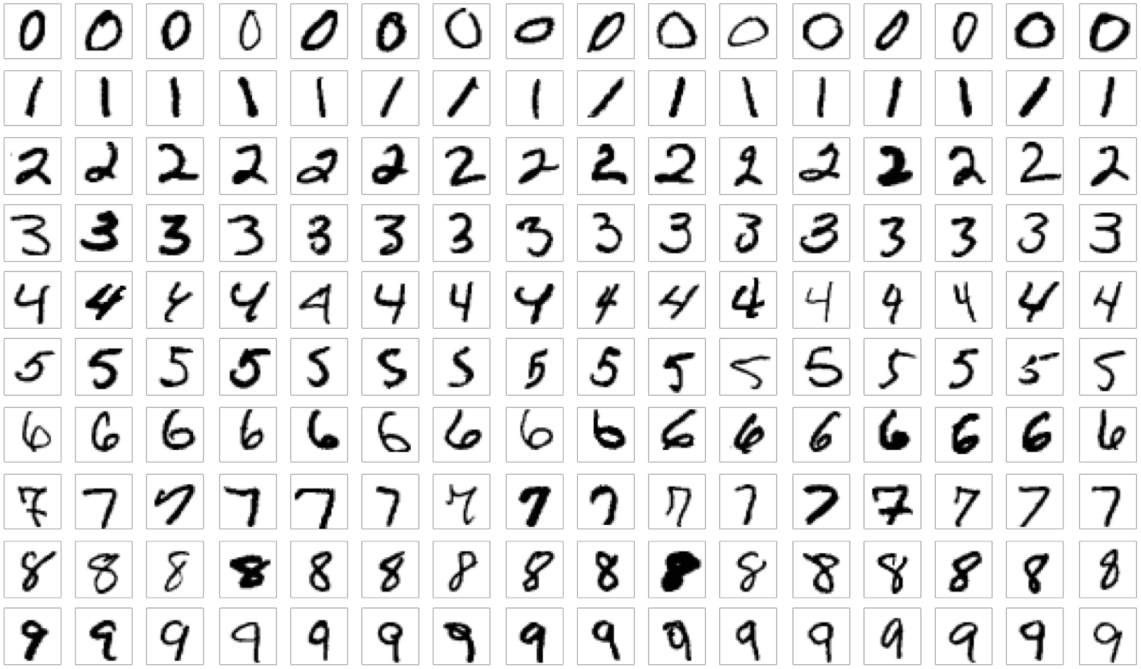
Deep learning software stacks



Deep learning software stacks



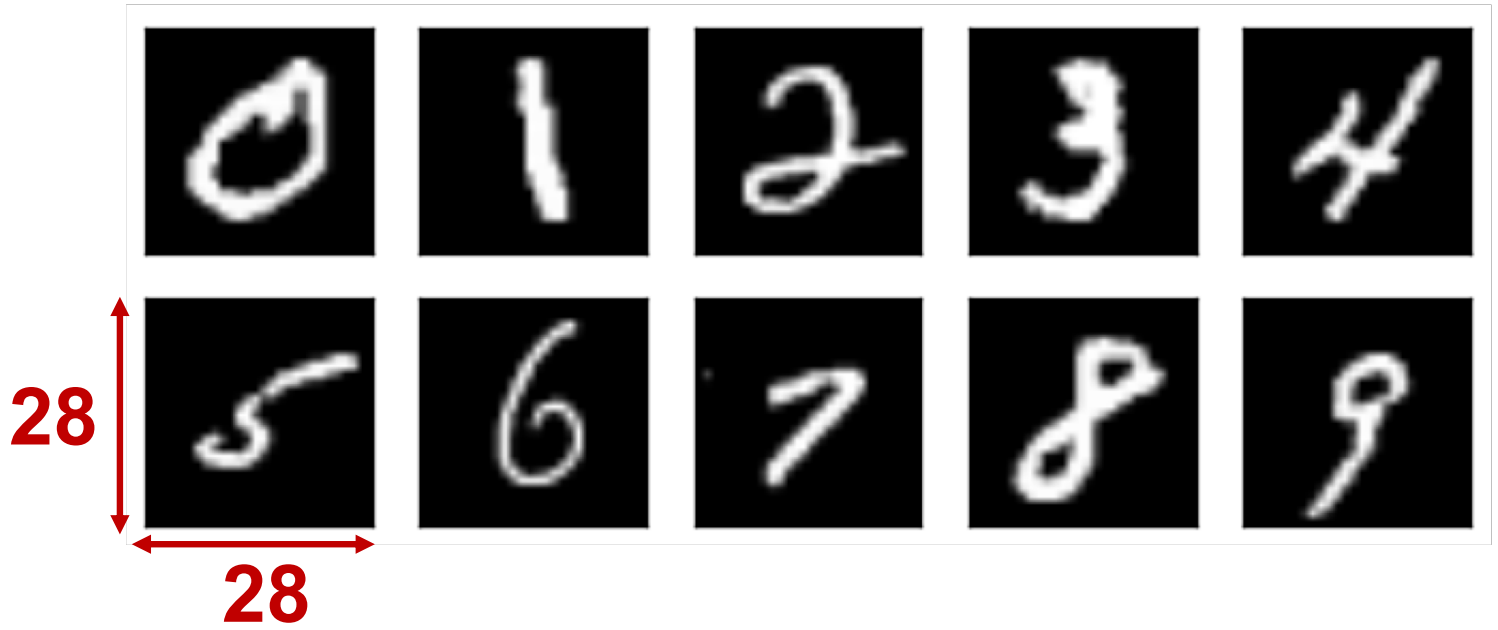
The MNIST test suite



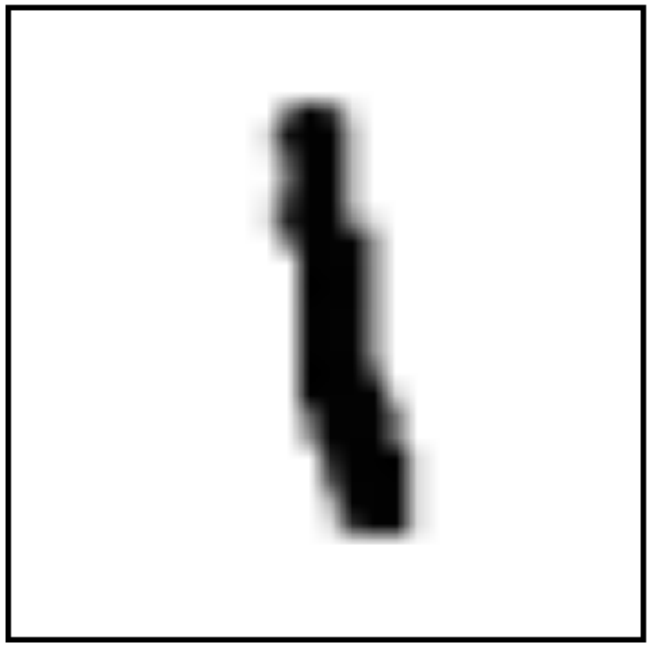
60,000 training examples
10,000 testing examples
Size-normalized
28 x 28 pixels

http://yann.lecun.com/exdb/mnist/

The MNIST test suite



The MNIST test suite



≈

0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	.6	.8	0	0	0	0	0	0
0	0	0	0	0	0	.7	1	0	0	0	0	0	0
0	0	0	0	0	0	.7	1	0	0	0	0	0	0
0	0	0	0	0	0	.5	1	.4	0	0	0	0	0
0	0	0	0	0	0	0	1	.4	0	0	0	0	0
0	0	0	0	0	0	0	1	.4	0	0	0	0	0
0	0	0	0	0	0	0	1	.7	0	0	0	0	0
0	0	0	0	0	0	0	1	1	0	0	0	0	0
0	0	0	0	0	0	0	.9	1	.1	0	0	0	0
0	0	0	0	0	0	0	.3	1	.1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0

(showing 14 x 14 for brevity)

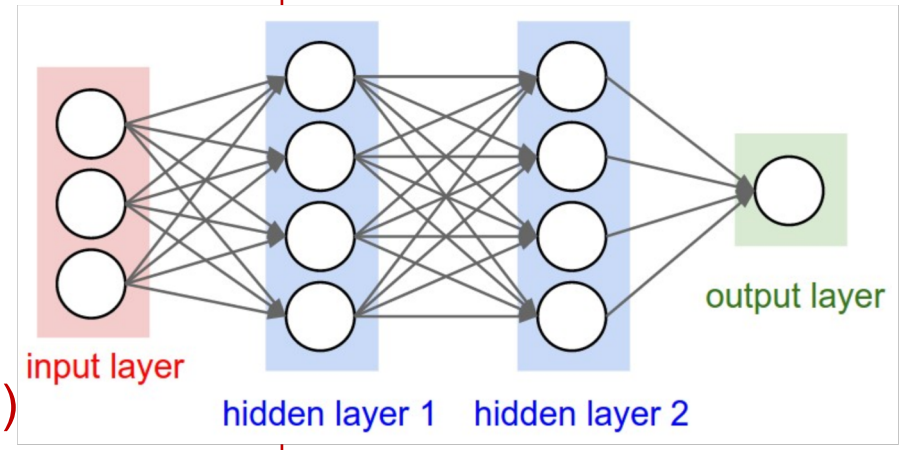
Flattened:

[0 0 0 … 0 **.6 .8** 0 … 0 **.7 1** 0 … 0 **.5 1 .4** 0 … 0 1 .4 0 … 0 1 .4 0 … 0 1 .7 0 … 0 1 1 0 … 0 .9 1 .1 0 …]

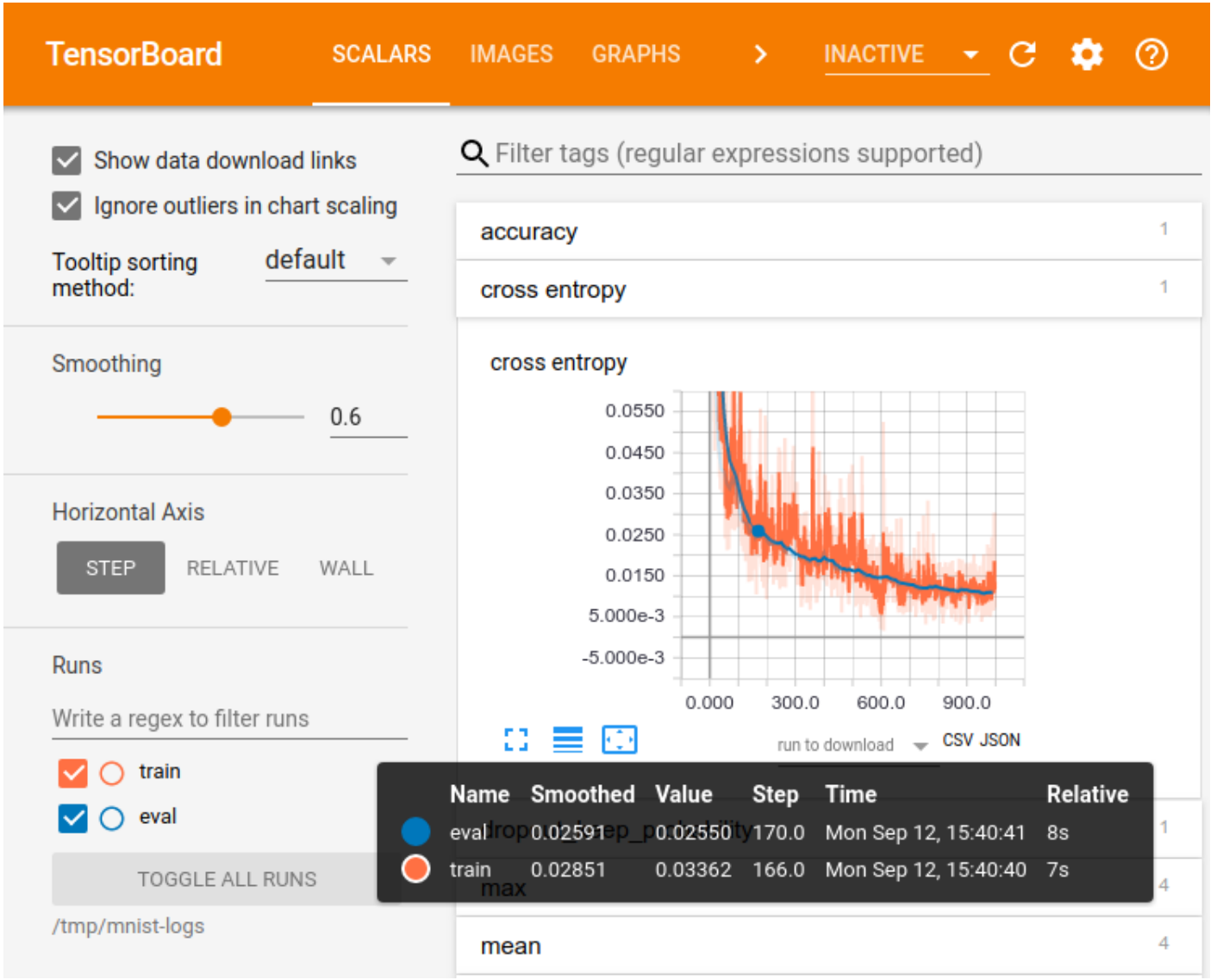
https://tensorflow.rstudio.com/tensorflow/articles/tutorial_mnist_beginners.html

(Simple) MNIST in Keras

```
import tensorflow as tf
mnist = tf.keras.datasets.mnist

(x_train, y_train),(x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = Sequential()
model.add(Flatten(input_shape=(28, 28)))
model.add(Dense(units=256, activation='relu'))
model.add(Dense(units=256, activation='relu'))
model.add(Dense(units=10, activation='softmax'))

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5)
model.evaluate(x_test, y_test)
```



Keras and Tensorboard



Set up Tensorboard (Source)

```
[ ]  LOG_DIR = '/tmp/log'
     get_ipython().system_raw(
         'tensorboard --logdir {} --host 0.0.0.0 --port 6006 &'
         .format(LOG_DIR)
     )
```

```
[ ]  ! wget https://bin.equinox.io/c/4VmDzA7iaHb/ngrok-stable-linux-amd64.zip > /dev/null 2>&1
     ! yes | unzip ngrok-stable-linux-amd64.zip > /dev/null 2>&1
```

```
[ ]  get_ipython().system_raw('./ngrok http 6006 &')
```

```
[ ]  ! curl -s http://localhost:4040/api/tunnels | python3 -c \
         "import sys, json; print(json.load(sys.stdin)['tunnels'][0]['public_url'])"
```

```
http://8fd226b5.ngrok.io
```

```python
# tf Graph input
X = tf.placeholder("float", [None, num_input])
Y = tf.placeholder("float", [None, num_classes])
```

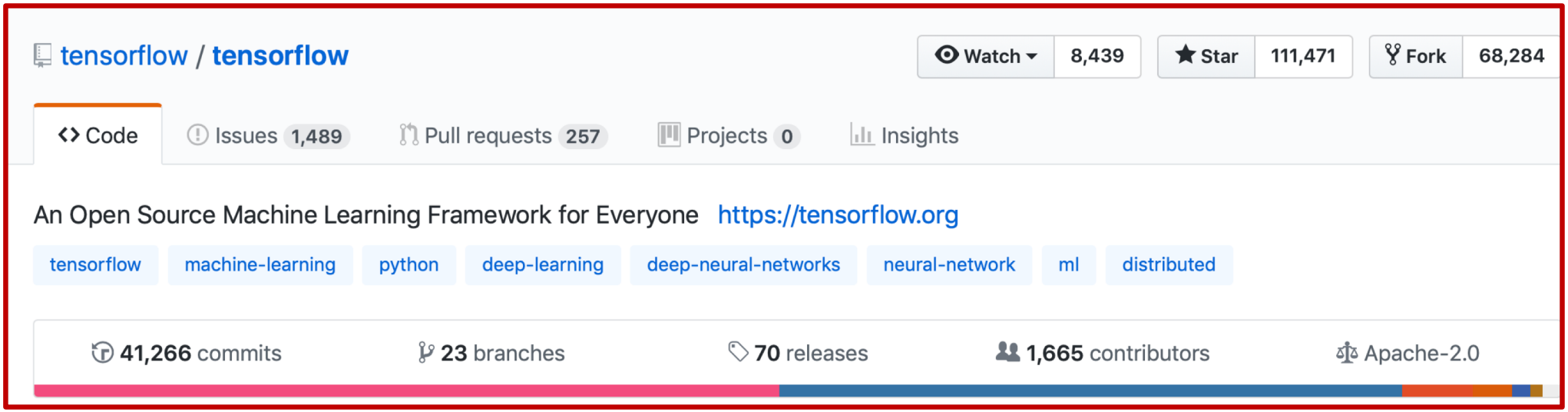
```python
# Store layers weight & bias
weights = {
    'h1': tf.Variable(tf.random_normal([num_input, n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_hidden_2, num_classes]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([num_classes]))
}
```

```python
# Create model
def neural_net(x):
    # Hidden fully connected layer with 256 neurons
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    # Hidden fully connected layer with 256 neurons
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    # Output fully connected layer with a neuron for each class
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
    return out_layer
```

```python
# Construct model
logits = neural_net(X)

# Define loss and optimizer
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
    logits=logits, labels=Y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)

# Evaluate model (with test logits, for dropout to be disabled)
correct_pred = tf.equal(tf.argmax(logits, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()
```

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

    # Run the initializer
    sess.run(init)

    for step in range(1, num_steps+1):
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        # Run optimization op (backprop)
        sess.run(train_op, feed_dict={X: batch_x, Y: batch_y})
        if step % display_step == 0 or step == 1:
            # Calculate batch loss and accuracy
            loss, acc = sess.run([loss_op, accuracy], feed_dict={X: batch_x,
                                                                 Y: batch_y})
            print("Step " + str(step) + ", Minibatch Loss= " + \
                  "{:.4f}".format(loss) + ", Training Accuracy= " + \
                  "{:.3f}".format(acc))

    print("Optimization Finished!")

    # Calculate accuracy for MNIST test images
    print("Testing Accuracy:", \
        sess.run(accuracy, feed_dict={X: mnist.test.images,
                                      Y: mnist.test.labels}))
```

MNIST in Tensorflow

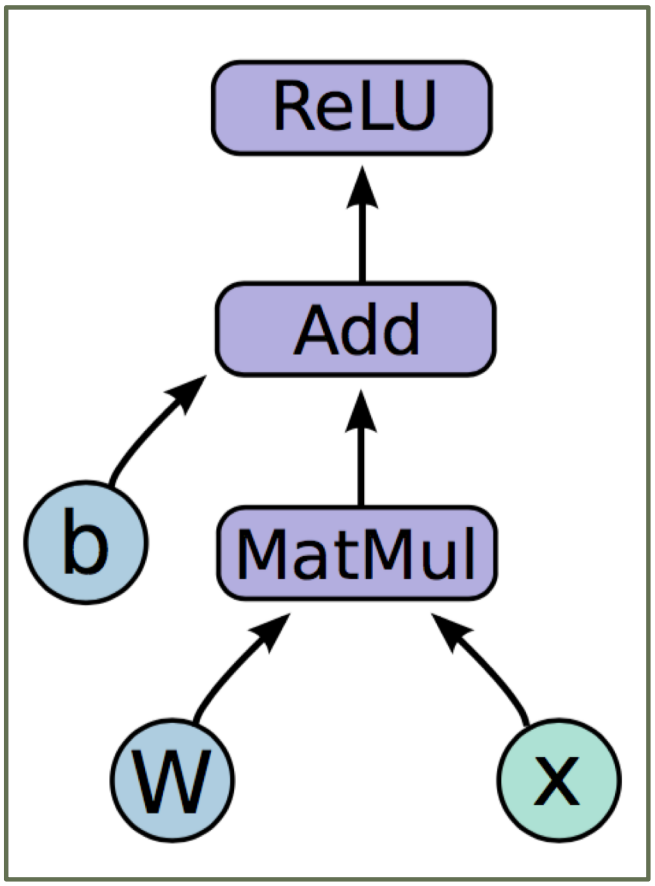


- Open source library for numerical computation using data flow graphs
- Originally developed by Google Brain Team for machine learning research
- “Tensorflow is an interface for expressing machine learning algorithms, and an implementation for executing such algorithms” [Abadi et al., 2015]
- “Tensorflow is inordinately complicated” [Internet commentators]

TensorFlow: The basic idea

1) We express a computation as a **graph**
- Graph nodes are **operations** which have any number of inputs and outputs
- Graph edges are **tensors** [multi-dimensional arrays] which flow between nodes

2) Having defined a graph, we can deploy it with a **session**: a binding to a particular execution context (e.g. CPU, GPU)

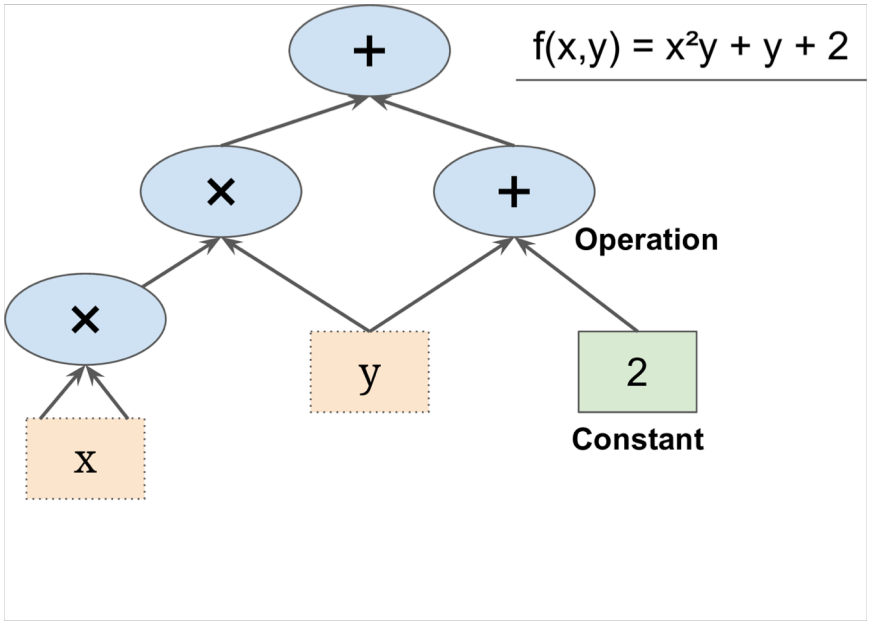


$$h = ReLU(Wx + b)$$

Tensorflow Graphs: A simple example

Graph:

-- Nodes = Operations

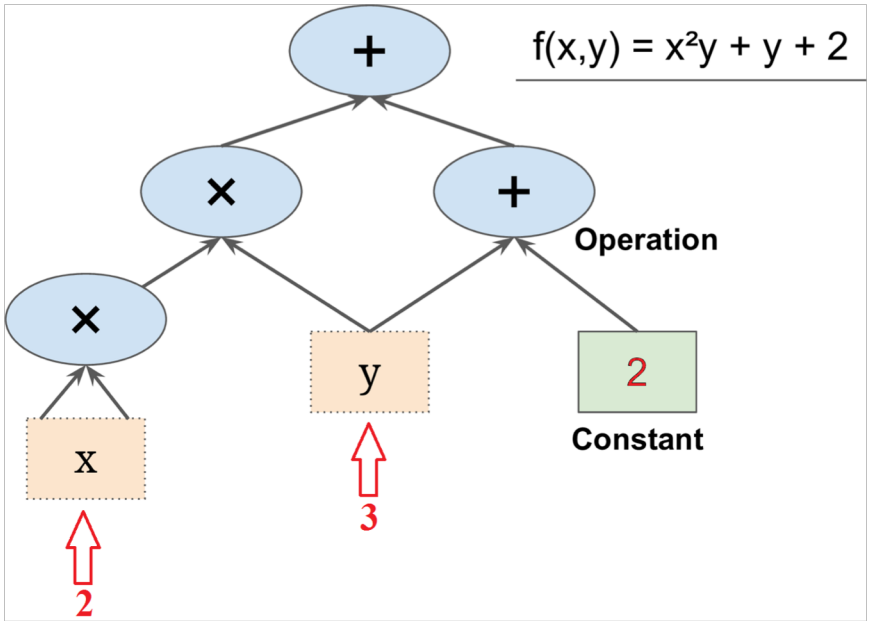


Tensorflow Graphs: A simple example

Graph:

-- Nodes = Operations

-- Edges = Tensors



Tensorflow Graphs: A simple example

Graph:

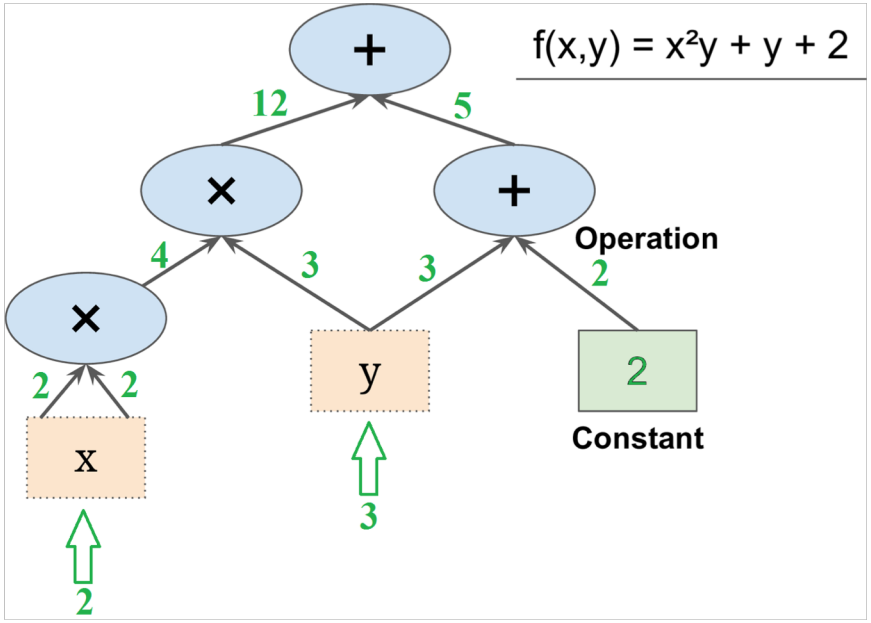
-- Nodes = Operations

-- Edges = Tensors

Session:

-- Tensor = Data

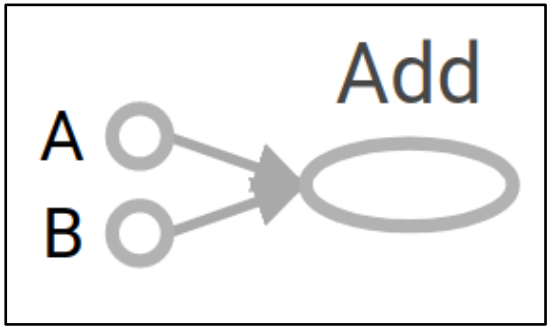
-- Tensor + flow = Data + flow



Tensorflow code

```
import tensorflow as tf
a = tf.constant(2, name='A')
b = tf.constant(3, name='B')
c = tf.add(a, b, name='Add')


print(c)
```

Graph

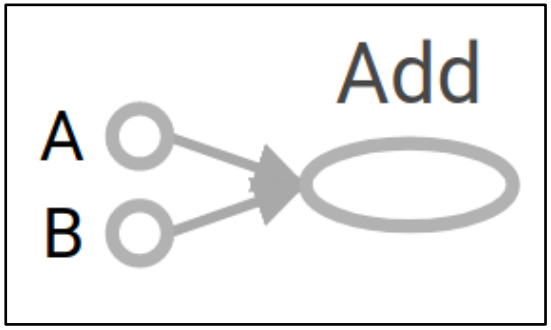


Tensorflow code

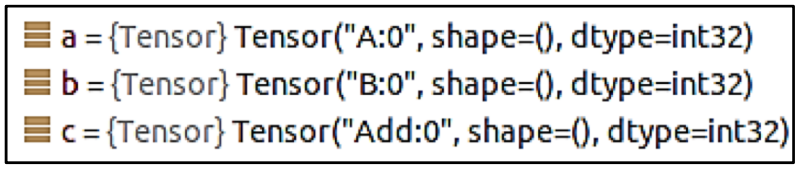
```
import tensorflow as tf
a = tf.constant(2, name='A')
b = tf.constant(3, name='B')
c = tf.add(a, b, name='Add')


print(c)
```

Tensor("Add:0", shape=(), dtype=int32)

Graph



Variables

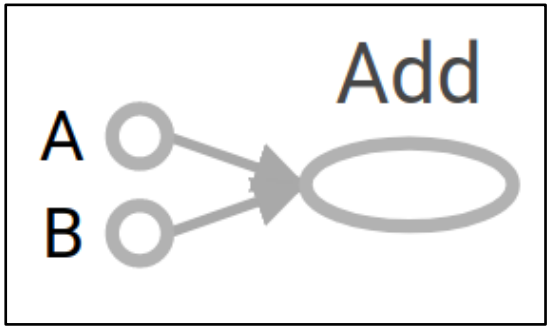


Tensorflow code

```
import tensorflow as tf
a = tf.constant(2, name='A')
b = tf.constant(3, name='B')
c = tf.add(a, b, name='Add')


with tf.Session() as sess:
    print(sess.run(c))
```

5

Graph



Variables

a = {Tensor} Tensor("A:0", shape=(), dtype=int32)
b = {Tensor} Tensor("B:0", shape=(), dtype=int32)
c = {Tensor} Tensor("Add:0", shape=(), dtype=int32)

We:

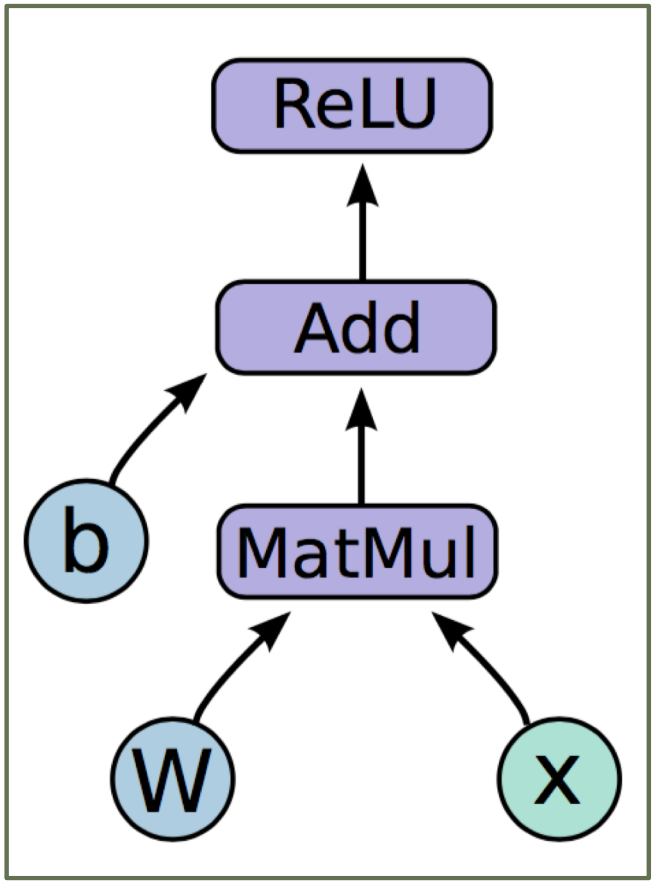
1. Build the GRAPH which represents the data flow of the computation
2. Run the SESSION which executes the operations on the graph

A more complex example

Variables are stateful nodes that output their current value.

State is retained across multiple executions of a graph

Primarily used for parameters



$$h = ReLU(Wx + b)$$

Mathematical operations:

MatMul: Multiply two matrices

Add: Add elementwise

ReLU: Activate with elementwise rectified linear function

Placeholders are nodes whose value is fed in at execution time

(inputs, labels, …)

Data types: Constants

Constants create constant values

For example:

```
s = tf.constant(2, name='scalar')
m = tf.constant([[1, 2], [3, 4]], name='matrix')
```

```
tf.constant( value,
             dtype=None,
             shape=None,
             name='Const',
             verify_shape=False
           )
```

Data types: Variables

Variables are stateful nodes that output their current value

-- They can be saved and restored

-- Gradient updates apply to all variables in the graph

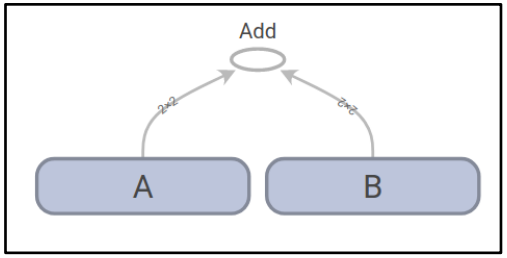
Used for network network parameters
(weights and biases)

```
get_variable(
    name,
    shape=None,
    dtype=None,
    initializer=None,
    regularizer=None,
    trainable=True,
    collections=None,
    caching_device=None,
    partitioner=None,
    validate_shape=True,
    use_resource=None,
    custom_getter=None,
    constraint=None
)
```

```
s = tf.get_variable(name='scalar2', initializer=tf.constant(2))
m = tf.get_variable('matrix', initializer=tf.constant([[0, 1], [2, 3]]))
M = tf.get_variable('big_matrix', shape=(784, 10), initializer=tf.zeros_initializer())
W = tf.get_variable('weight', shape=(784, 10),
                    initializer=tf.truncated_normal_initializer(mean=0.0, stddev=0.01))
```

Using variables

Graph



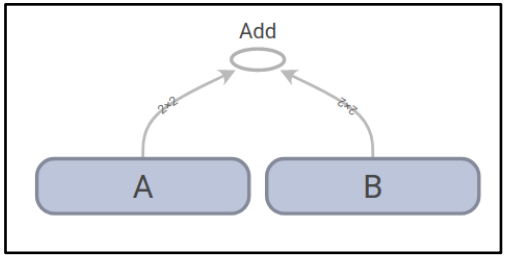

```
# Create graph
a = tf.get_variable(name="A", initializer=tf.constant([[0, 1], [2, 3]]))
b = tf.get_variable(name="B", initializer=tf.constant([[4, 5], [6, 7]]))
c = tf.add(a, b, name="Add")


# Launch the graph in a session
with tf.Session() as sess:


    # Now we can run the desired operation
    print(sess.run(c))
```

Variables

a = {Variable} <tf.Variable 'A:0' shape=(2, 2) dtype=int32_ref>
b = {Variable} <tf.Variable 'B:0' shape=(2, 2) dtype=int32_ref>
c = {Tensor} Tensor("Add:0", shape=(2, 2), dtype=int32)

Using variables

Graph




```
# Create graph
a = tf.get_variable(name="A", initializer=tf.constant([[0, 1], [2, 3]]))
b = tf.get_variable(name="B", initializer=tf.constant([[4, 5], [6, 7]]))
c = tf.add(a, b, name="Add")


# Launch the graph in a session
with tf.Session() as sess:


    # Now we can run the desired operation
    print(sess.run(c))
```

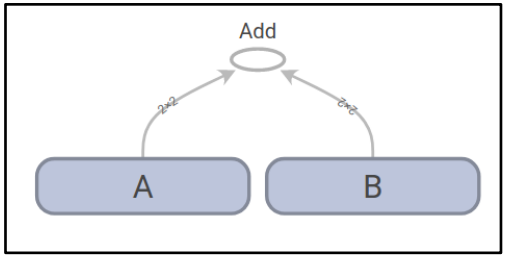
FailedPreconditionError: Attempting to use uninitialized value

Variables

a = {Variable} <tf.Variable 'A:0' shape=(2, 2) dtype=int32_ref>
b = {Variable} <tf.Variable 'B:0' shape=(2, 2) dtype=int32_ref>
c = {Tensor} Tensor("Add:0", shape=(2, 2), dtype=int32)

Using variables

Graph




```
# Create graph
a = tf.get_variable(name="A", initializer=tf.constant([[0, 1], [2, 3]]))
b = tf.get_variable(name="B", initializer=tf.constant([[4, 5], [6, 7]]))
c = tf.add(a, b, name="Add")
# Add an Op to initialize variables
init_op = tf.global_variables_initializer()
# Launch the graph in a session
with tf.Session() as sess:
    # Run the variable initializer
    sess.run(init_op)
    # Now we can run the desired operation
    print(sess.run(c))
```

```
[[ 4,  6], [ 8, 10]]
```

Variables

```
a = {Variable} <tf.Variable 'A:0' shape=(2, 2) dtype=int32_ref>
b = {Variable} <tf.Variable 'B:0' shape=(2, 2) dtype=int32_ref>
c = {Tensor} Tensor("Add:0", shape=(2, 2), dtype=int32)
```

Data types: Placeholder

```
tf.placeholder( dtype,
                shape=None,
                 name=None
               )
```

A Placeholder is a node whose value is fed in at execution time.

-- Assemble the graph without knowing the values needed for computation

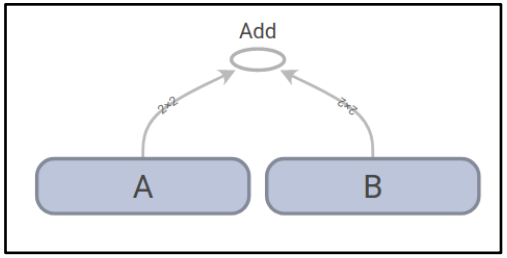
-- Supply the data at execution time.

Used for input data (in classification task: Inputs and labels)

```
a = tf.placeholder(tf.float32, shape=[5])
b = tf.placeholder(dtype=tf.float32, shape=None, name=None)
X = tf.placeholder(tf.float32, shape=[None, 784], name='input')
Y = tf.placeholder(tf.float32, shape=[None, 10], name='label')
```

Using placeholders

Graph



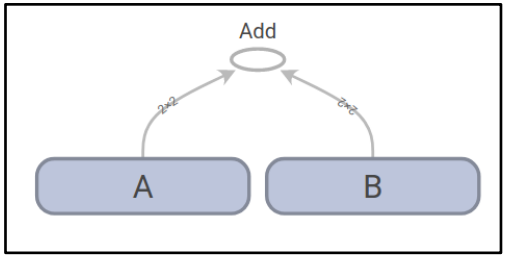

```
import tensorflow as tf
a = tf.constant([5, 5, 5], tf.float32, name='A')
b = tf.placeholder(tf.float32, shape=[3], name='B')
c = tf.add(a, b, name="Add")

with tf.Session() as sess:


    print(sess.run(c))
```

Variables

a = {Tensor} Tensor("A:0", shape=(3,), dtype=float32)
b = {Tensor} Tensor("B:0", shape=(3,), dtype=float32)
c = {Tensor} Tensor("Add:0", shape=(3,), dtype=float32)

Using placeholders

Graph




```
import tensorflow as tf
a = tf.constant([5, 5, 5], tf.float32, name='A')
b = tf.placeholder(tf.float32, shape=[3], name='B')
c = tf.add(a, b, name="Add")

with tf.Session() as sess:



    print(sess.run(c))
```

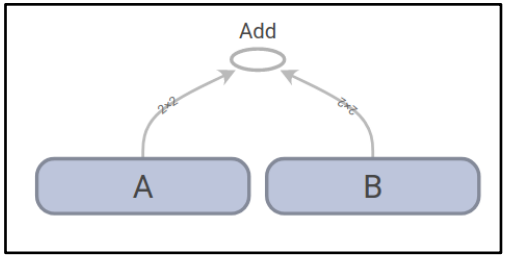
You must feed a value for placeholder tensor 'B' with dtype float and shape [3]

Variables

```
a = {Tensor} Tensor("A:0", shape=(3,), dtype=float32)
b = {Tensor} Tensor("B:0", shape=(3,), dtype=float32)
c = {Tensor} Tensor("Add:0", shape=(3,), dtype=float32)
```

Using placeholders

Graph




```
import tensorflow as tf
a = tf.constant([5, 5, 5], tf.float32, name='A')
b = tf.placeholder(tf.float32, shape=[3], name='B')
c = tf.add(a, b, name="Add")

with tf.Session() as sess:
    # Create a dictionary:
    d = {b: [1, 2, 3]}
    # Feed it to the placeholder
    print(sess.run(c, feed_dict=d))
```

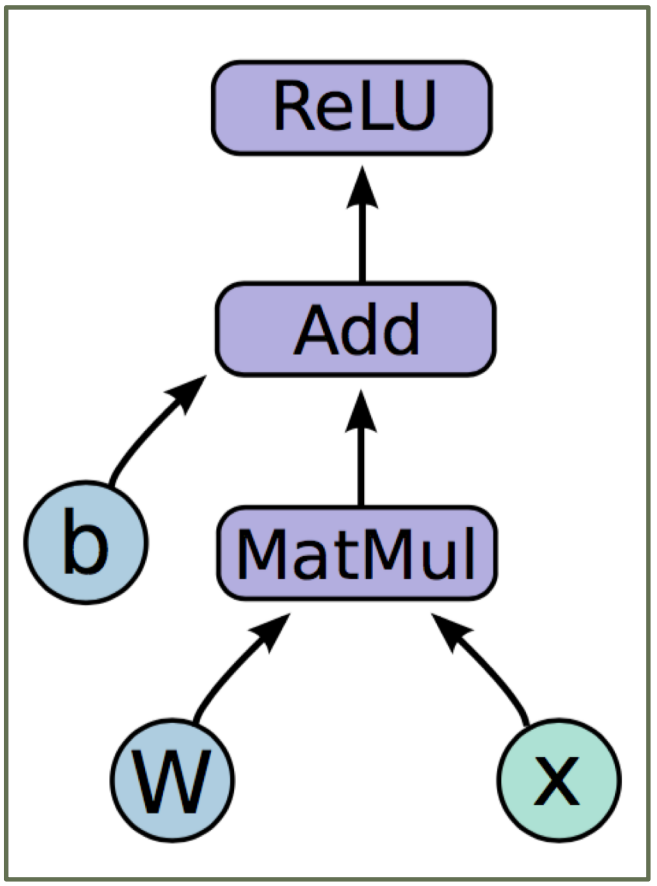
[6, 7, 8]

Variables

a = {Tensor} Tensor("A:0", shape=(3,), dtype=float32)
b = {Tensor} Tensor("B:0", shape=(3,), dtype=float32)
c = {Tensor} Tensor("Add:0", shape=(3,), dtype=float32)
d = {dict} {<tf.Tensor 'B:0' shape=(3,) dtype=float32>: [1, 2, 3]}

Back to our example: Creating a graph

```
import tensorflow as tf

b = tf.Variable(tf.zeros((100,)))
W = tf.Variable(tf.random_uniform((784, 100), -1, 1))

x = tf.placeholder(tf.float32, (100, 784))

h = tf.nn.relu(tf.matmul(x, W) + b)
```



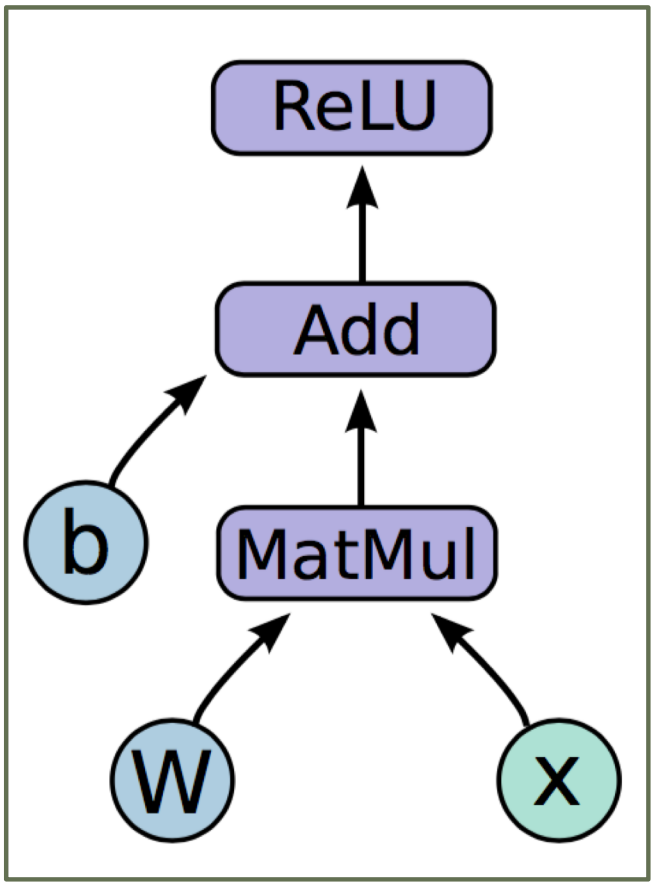
$$h = ReLU(Wx + b)$$

Creating a graph

```
import tensorflow as tf

b = tf.Variable(tf.zeros((100,)))
W = tf.Variable(tf.random_uniform((784, 100), -1, 1))

x = tf.placeholder(tf.float32, (100, 784))

h = tf.nn.relu(tf.matmul(x, W) + b)
```

Deploying the graph into a session

```
with tf.Session() as sess:
    sess.run(tf.initialize_all_variables())
    sess.run(h, {x: np.random.random((100, 784))})
```



$$h = ReLU(Wx + b)$$

```python
# tf Graph input
X = tf.placeholder("float", [None, num_input])
Y = tf.placeholder("float", [None, num_classes])


# Store layers weight & bias
weights = {
    'h1': tf.Variable(tf.random_normal([num_input, n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_hidden_2, num_classes]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([num_classes]))
}


# Create model
def neural_net(x):
    # Hidden fully connected layer with 256 neurons
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    # Hidden fully connected layer with 256 neurons
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    # Output fully connected layer with a neuron for each class
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
    return out_layer


# Construct model
logits = neural_net(X)
```

```python
# Define loss and optimizer
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_w
    logits=logits, labels=Y))
optimizer = tf.train.AdamOptimizer(learning_rate=learn
train_op = optimizer.minimize(loss_op)

# Evaluate model (with test logits, for dropout to be
correct_pred = tf.equal(tf.argmax(logits, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()
```

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

    # Run the initializer
    sess.run(init)

    for step in range(1, num_steps+1):
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        # Run optimization op (backprop)
        sess.run(train_op, feed_dict={X: batch_x, Y: batch_y})
        if step % display_step == 0 or step == 1:
            # Calculate batch loss and accuracy
            loss, acc = sess.run([loss_op, accuracy], feed_dict={X: batch_x,
                                                                 Y: batch_y})
            print("Step " + str(step) + ", Minibatch Loss= " + \
                  "{:.4f}".format(loss) + ", Training Accuracy= " + \
                  "{:.3f}".format(acc))

    print("Optimization Finished!")

    # Calculate accuracy for MNIST test images
    print("Testing Accuracy:", \
        sess.run(accuracy, feed_dict={X: mnist.test.images,
                                      Y: mnist.test.labels}))
```

MNIST in Tensorflow

```
# tf Graph input
X = tf.placeholder("float", [None, num_input])
Y = tf.placeholder("float", [None, num_classes])
```

```
# Store layers weight & bias
weights = {
    'h1': tf.Variable(tf.random_normal([num_input, n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_hidden_2, num_classes]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([num_classes]))
}
```

```
# Create model
def neural_net(x):
    # Hidden fully connected layer with 256 neurons
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    # Hidden fully connected layer with 256 neurons
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    # Output fully connected layer with a neuron for each class
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
    return out_layer
```

```
# Construct model
logits = neural_net(X)
```

Next up, we add a loss node

Use a **placeholder** for **labels**, as they will be supplied later

Build loss node using labels and **prediction**

$$f_i(\vec{x}) = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}} \quad \text{for } i = 1, \ldots, J$$

SoftMax: Converts scores to probabilities that sum to 1

```
prediction = tf.nn.softmax(...)  # Output of neural network
label = tf.placeholder(tf.float32, [100, 10])

cross_entropy = -tf.reduce_sum(label * tf.log(prediction), axis=1)
```

"In information theory, the **cross entropy** between two probability distributions **p** and **q** over the same underlying set of events measures the average number of bits needed to identify an event drawn from the set, if a coding scheme is used that is optimized for an "unnatural" probability distribution **q**, rather than the "true" distribution **p**." [Wikipedia]

$$H(p, q) = -\sum_{x} p(x) \log q(x)$$

We next want to compute gradients

```
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
```

tf.train.GradientDescentOptimizer is an **Optimizer** object

tf.train.GradientDescentOptimizer(lr).**minimize**(cross_entropy) adds optimization **operation** to computation graph

TensorFlow graph **nodes** have **attached gradient operations**

Gradient with respect to **parameters** computed with **backpropagation**

Create the train_step op

```
prediction = tf.nn.softmax(...)
label = tf.placeholder(tf.float32, [None, 10])

cross_entropy = tf.reduce_mean(-tf.reduce_sum(label * tf.log(prediction),
                                              reduction_indices=[1]))

train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)
```

Adding the **train** line adds an optimization node to the graph.

Evaluating this “train_step” Python variable at runtime will automatically apply gradients to all of the variables in our graph.

Now we can train the model

We can now train the train_step node. To do this we:

1. Create Session
2. Build training schedule (iterative, feeds in data and labels, trains model)
3. Run train_step

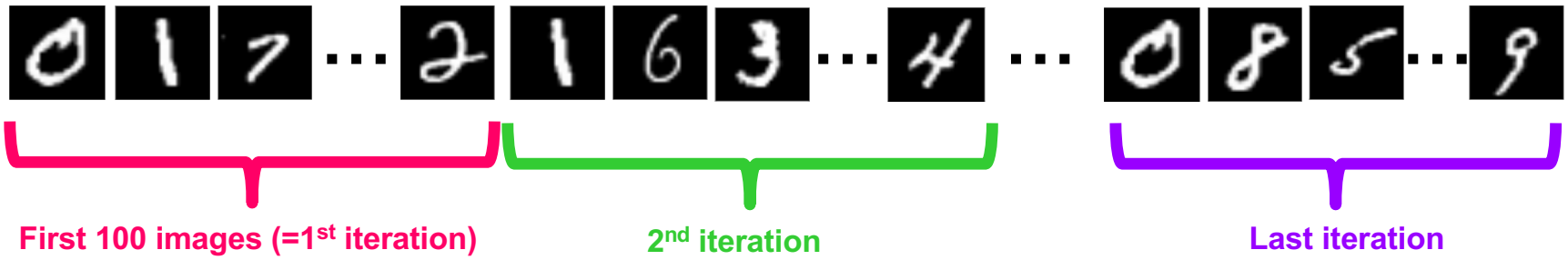
```
sess = tf.Session()
sess.run(tf.initialize_all_variables())

for i in range(1000):
    batch_x, batch_label = data.next_batch()
    sess.run(train_step, feed_dict={x: batch_x, label: batch_label})
```

Batches, epochs, and iteration

Example: MNIST data

-- Number of training data: **N=60,000**

-- Let's take batch size of **B= 100**



How many iterations per epoch? **60,000/100 = 600** **1 epoch = 600 iterations**

MNIST in Tensorflow

```
# tf Graph input
X = tf.placeholder("float", [None, num_input])
Y = tf.placeholder("float", [None, num_classes])
```

Input X and Y: placeholders

```
# Store layers weight & bias
weights = {
    'h1': tf.Variable(tf.random_normal([num_input, n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_hidden_2, num_classes]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([num_classes]))
}
```

Weights and biases: Variables

```
# Create model
def neural_net(x):
    # Hidden fully connected layer with 256 neurons
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    # Hidden fully connected layer with 256 neurons
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    # Output fully connected layer with a neuron for each class
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
    return out_layer
```

Model: Three layers

Define loss and training op

```
# Construct model
logits = neural_net(X)

# Define loss and optimizer
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
    logits=logits, labels=Y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)

# Evaluate model (with test logits, for dropout to be disabled)
correct_pred = tf.equal(tf.argmax(logits, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()
```

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

    # Run the initializer
    sess.run(init)

    for step in range(1, num_steps+1):
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        # Run optimization op (backprop)
        sess.run(train_op, feed_dict={X: batch_x, Y: batch_y})
        if step % display_step == 0 or step == 1:
            # Calculate batch loss and accuracy
            loss, acc = sess.run([loss_op, accuracy], feed_dict={X: batch_x,
                                                                 Y: batch_y})
            print("Step " + str(step) + ", Minibatch Loss= " + \
                  "{:.4f}".format(loss) + ", Training Accuracy= " + \
                  "{:.3f}".format(acc))

    print("Optimization Finished!")

    # Calculate accuracy for MNIST test images
    print("Testing Accuracy:", \
        sess.run(accuracy, feed_dict={X: mnist.test.images,
                                      Y: mnist.test.labels}))
```

Train model

Test accuracy

```
# Define loss and optimizer
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
    logits=logits, labels=Y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)

# Evaluate model (with test logits, for dropout to be disabled)
correct_pred = tf.equal(tf.argmax(logits, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

# Initialize the variables (i.e. assign their default value)
init = tf.global_variables_initializer()
```

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

    # Run the initializer
    sess.run(init)

    for step in range(1, num_steps+1):
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        # Run optimization op (backprop)
        sess.run(train_op, feed_dict={X: batch_x, Y: batch_y})
        if step % display_step == 0 or step == 1:
            # Calculate batch loss and accuracy
            loss, acc = sess.run([loss_op, accuracy], feed_dict={X: batch_x,
                                                                 Y: batch_y})
            print("Step " + str(step) + ", Minibatch Loss= " + \
                  "{:.4f}".format(loss) + ", Training Accuracy= " + \
                  "{:.3f}".format(acc))

    print("Optimization Finished!")

    # Calculate accuracy for MNIST test images
    print("Testing Accuracy:", \
        sess.run(accuracy, feed_dict={X: mnist.test.images,
                                      Y: mnist.test.labels}))
```

Variable sharing

```
variables_dict = {
        "weights": tf.Variable(tf.random_normal([782, 100]), name="weights")
        "biases": tf.Variable(tf.zeros([100]), name="biases")
    }
```

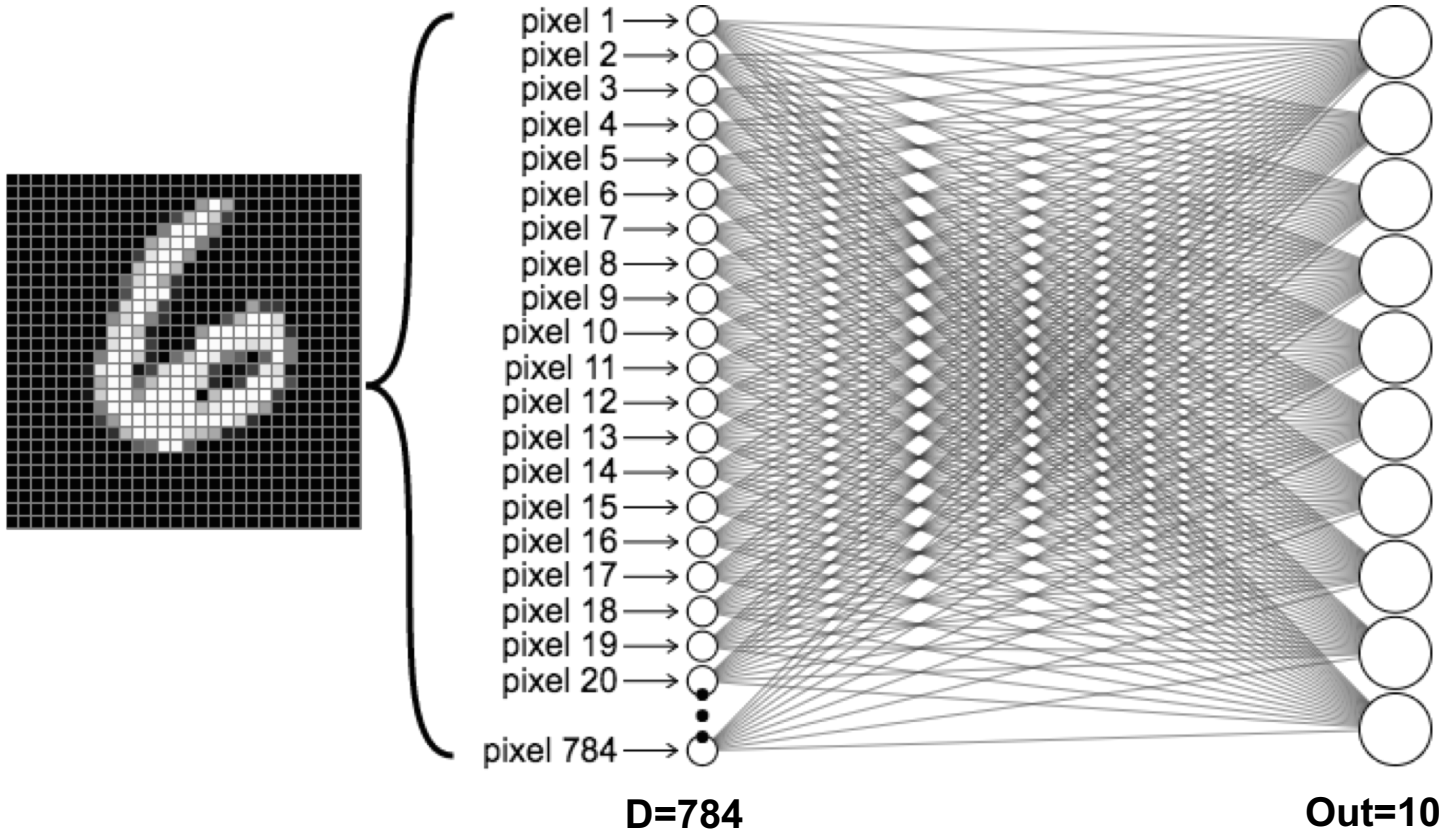
tf.variable_scope() provides simple name-spacing to avoid clashes
tf.get_variable() creates/accesses variables from within a variable scope

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

with tf.variable_scope("foo", reuse=True):
        v1 = tf.get_variable("v")                  # Shared variable found!

with tf.variable_scope("foo", reuse=False):
        v1 = tf.get_variable("v")                  # CRASH foo/v:0 already exists!
```

```python
def nn_layer(input_tensor, input_dim, output_dim, layer_name, act=tf.nn.relu):
  """Reusable code for making a simple neural net layer.

  It does a matrix multiply, bias add, and then uses ReLU to nonlinearize.
  It also sets up name scoping so that the resultant graph is easy to read,
  and adds a number of summary ops.
  """
  # Adding a name scope ensures logical grouping of the layers in the graph.
  with tf.name_scope(layer_name):
    # This Variable will hold the state of the weights for the layer
    with tf.name_scope('weights'):
      weights = weight_variable([input_dim, output_dim])
      variable_summaries(weights)
    with tf.name_scope('biases'):
      biases = bias_variable([output_dim])
      variable_summaries(biases)
    with tf.name_scope('Wx_plus_b'):
      preactivate = tf.matmul(input_tensor, weights) + biases
      tf.summary.histogram('pre_activations', preactivate)
    activations = act(preactivate, name='activation')
    tf.summary.histogram('activations', activations)
    return activations

hidden1 = nn_layer(x, 784, 500, 'layer1')
```

Back to MNIST

Logistic (i.e., linear) classifier

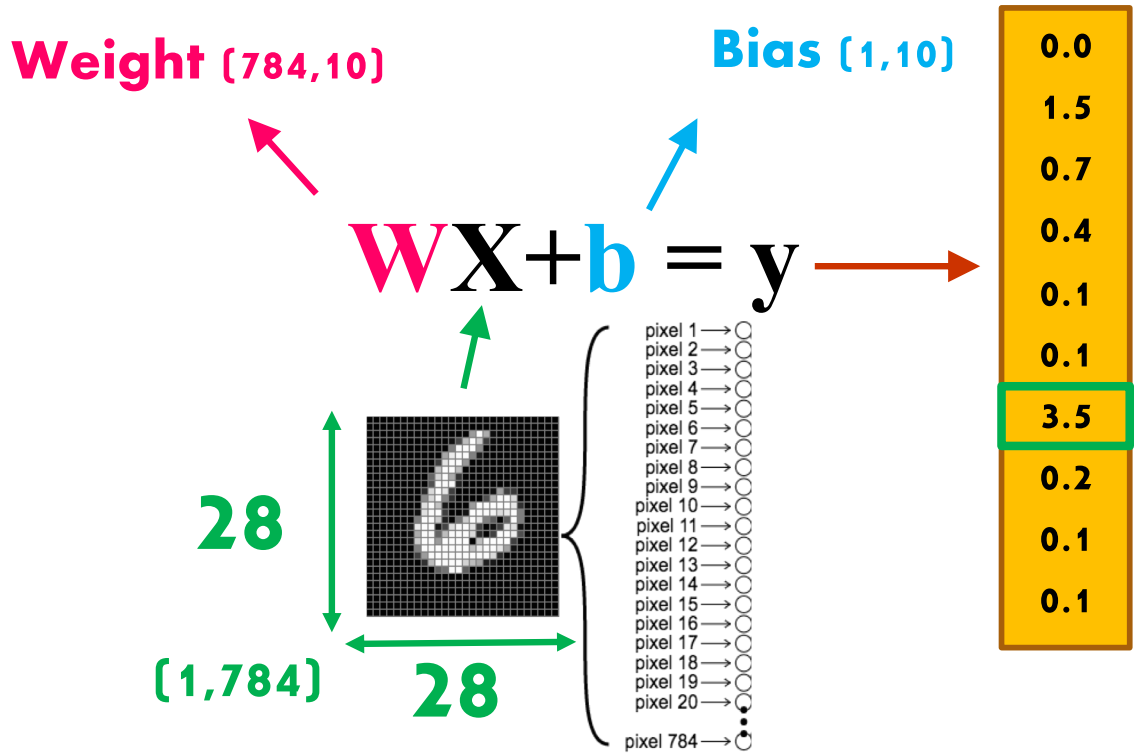


Logistic classifier

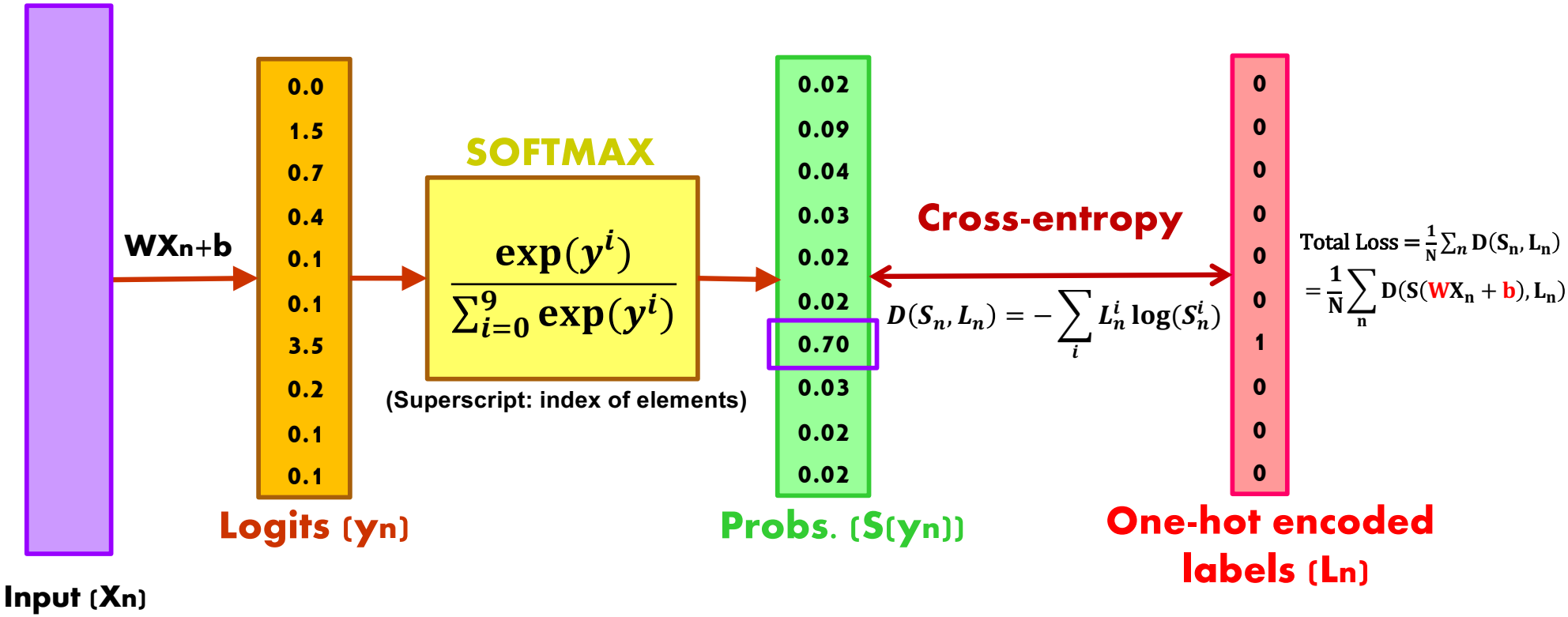
```
input_dim = 28*28
output_dim = nb_classes = 10

model = Sequential()
model.add(Flatten(input_shape=(28, 28)))
model.add(Dense(output_dim, activation='softmax'))

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=5, batch_size=128)
model.evaluate(x_test, y_test)
```

Logistic (i.e., linear) classifier

Set of **N** labeled inputs: **D = {(X_1, y_1), …, (X_N, y_N)}** $\begin{cases} X = input\ data \\ y = input\ label \end{cases}$



Logistic Classifier (linear classifier)



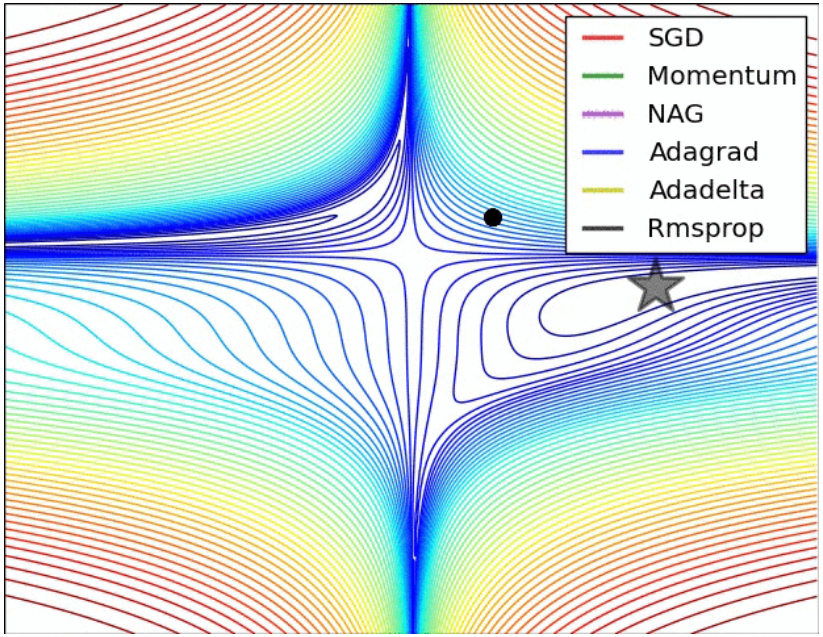
Optimizer

$$\mathbf{L} = \frac{\mathbf{1}}{\mathbf{N}} \sum_{\mathbf{n}} \mathbf{D}(\mathbf{S}(\mathbf{W}\mathbf{X}_{\mathbf{n}} + \mathbf{b}), \mathbf{L}_{\mathbf{n}})$$

Gradient descent:

$$\boldsymbol{w} := \boldsymbol{w} - \boldsymbol{\alpha} \frac{\boldsymbol{\partial L}}{\boldsymbol{\partial w}}$$

Learning rate



An overview of gradient descent optimization algorithms

Logistic Classifier (linear classifier)

- Disadvantages:
  - Few parameters: #parameters = 10x784 + 10 = 7850
  - At the end, it's a linear model!

- Advantages:
  - Linear models are STABLE!