CS291K – Advanced Data Mining

Instructor: Xifeng Yan
Computer Science
University of California at Santa Barbara

TensorFlow

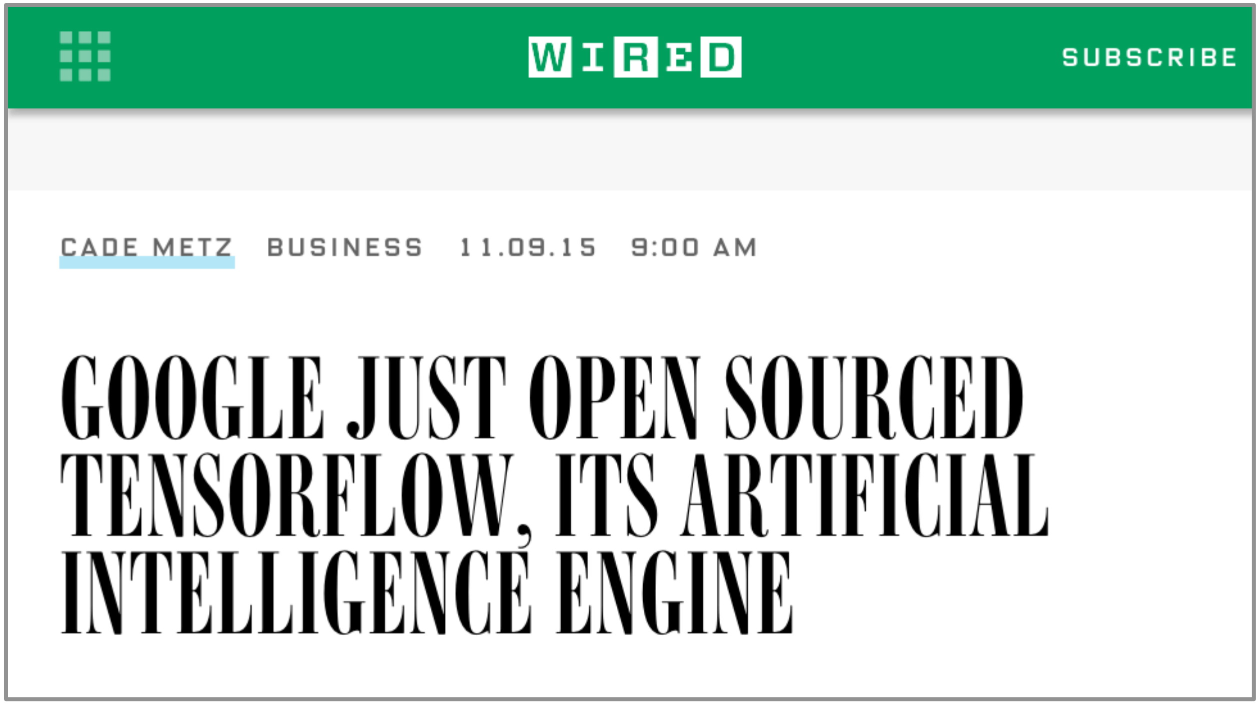
Lecturer: Honglei Liu
Computer Science
University of California at Santa Barbara

- The slides are adapted from:
  - “**Large-Scale Distributed Systems for Training Neural Networks”,** Jeff Dean and Oriol Vinyals, NIPS 2015 tutorial
  - **“Introduction to TensorFlow”**, Jon Gauthier (Stanford NLP Group), 12 November 2015
  - **“TensorFlow: neural networks lab”**, Gianluca Corrado and Andrea Passerini

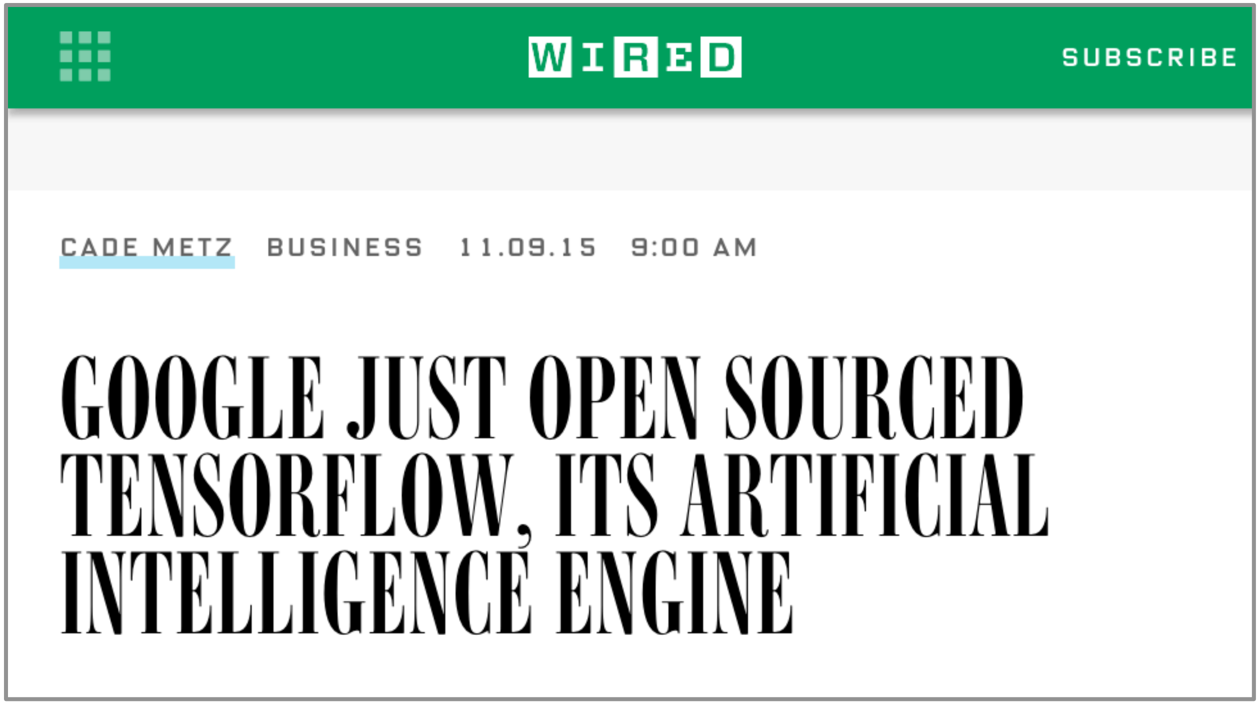
Outline

- History and Introduction
- Basics
- A complete example
- Demo
- Convolutional Neural Network

What is TensorFlow?



What is TensorFlow?

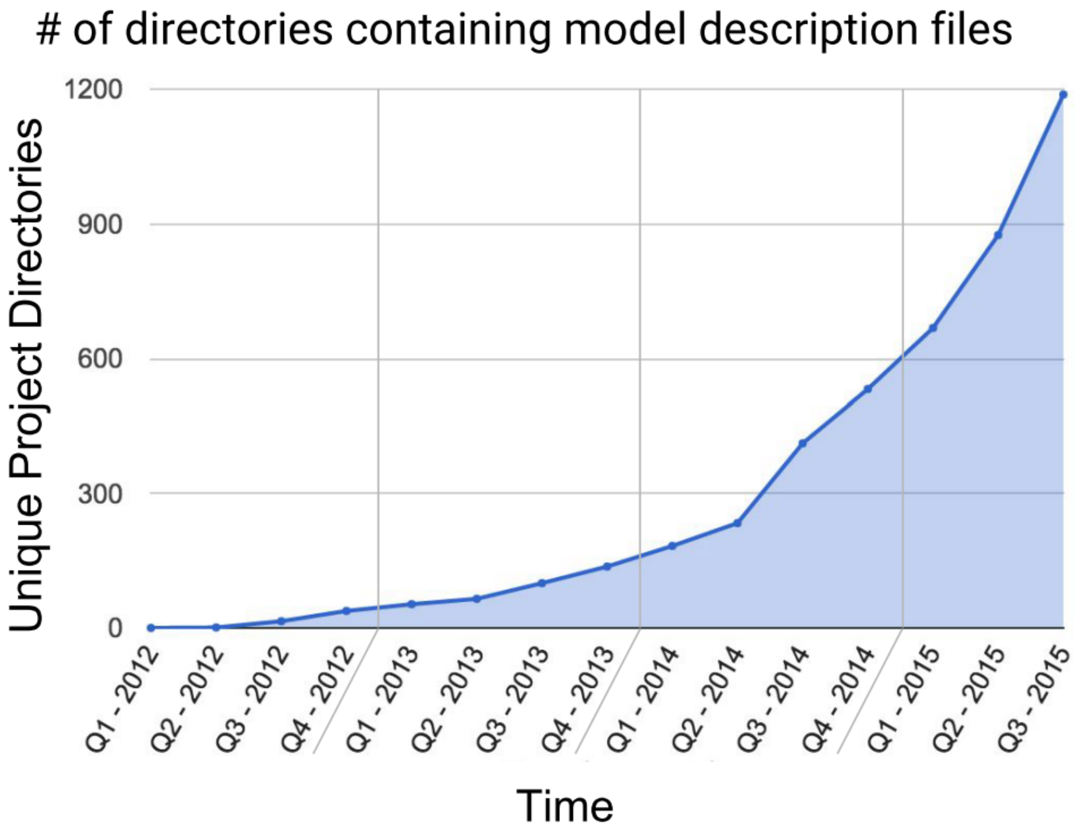

WIRED SUBSCRIBE

CADE METZ BUSINESS 11.09.15 9:00 AM

GOOGLE JUST OPEN SOURCED TENSORFLOW, ITS ARTIFICIAL INTELLIGENCE ENGINE

"TensorFlow is an interface for expressing machine learning algorithms, and an implementation for executing such algorithms."

Growing Use of Deep Learning at Google



Across many products/areas:

Android
Apps
drug discovery
Gmail
Image understanding
Maps
Natural language understanding
Photos
Robotics research
Speech
Translation
YouTube
... many others ...

Two Generations

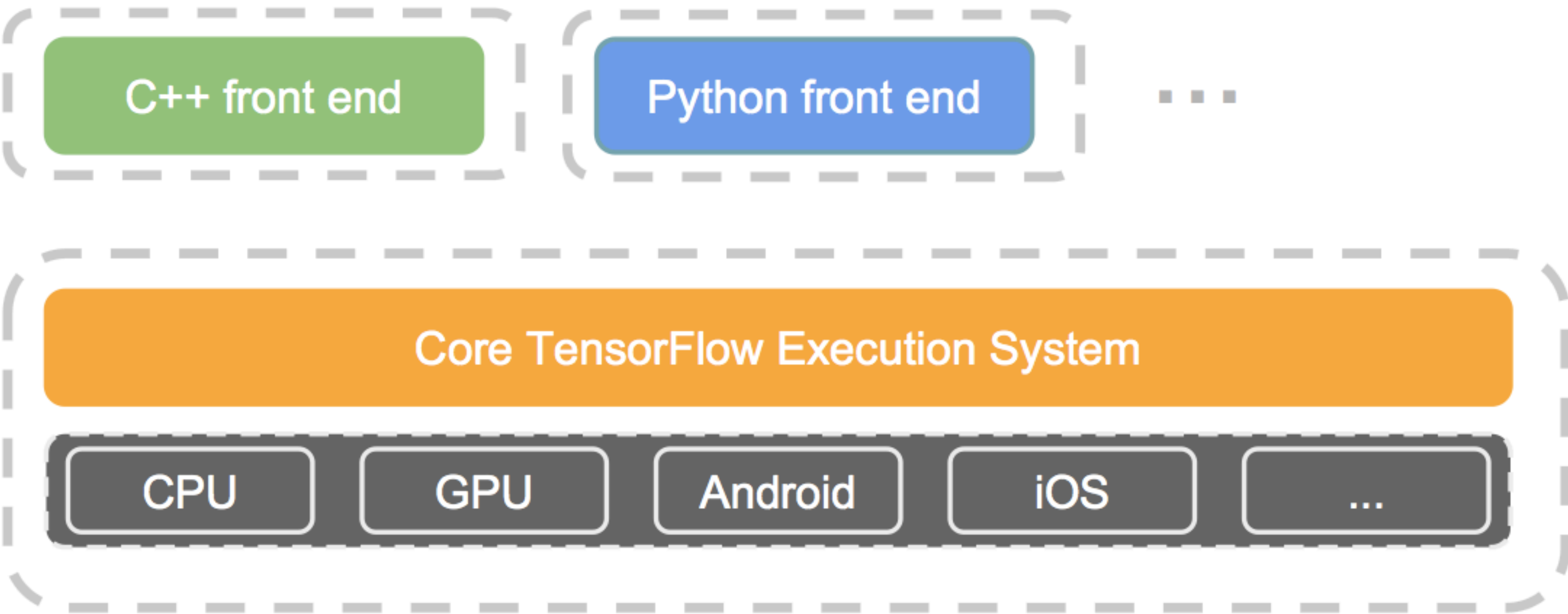
- 1st generation – DistBelief (Dean et al., NIPS 2012)
  - Scalable, good for production, but not very flexible for research

Two Generations

- 1st generation – DistBelief (Dean et al., NIPS 2012)
  - Scalable, good for production, but not very flexible for research
- 2nd generation – TensorFlow
  - Scalable, good for production, but also flexible for variety of research uses
  - Portable across range of platforms
  - Open sourced single-machine TensorFlow on Nov. 9th, 2015
  - Updates for distributed implementation (version 0.8) on April 13, 2016

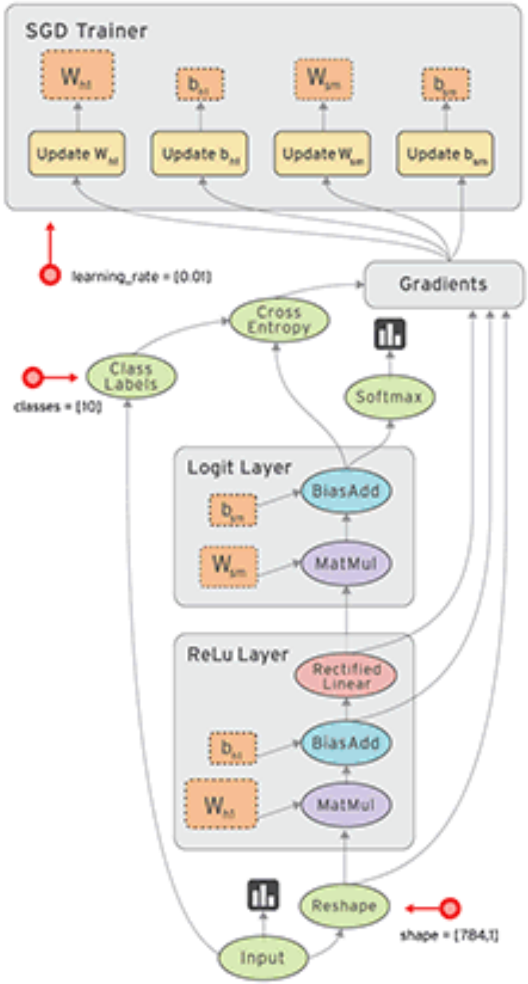
TensorFlow: Expressing High-Level ML Computations

- Core in C++
- Different front ends for specifying/driving the computation
- Automatically runs models on range of platforms



Big idea

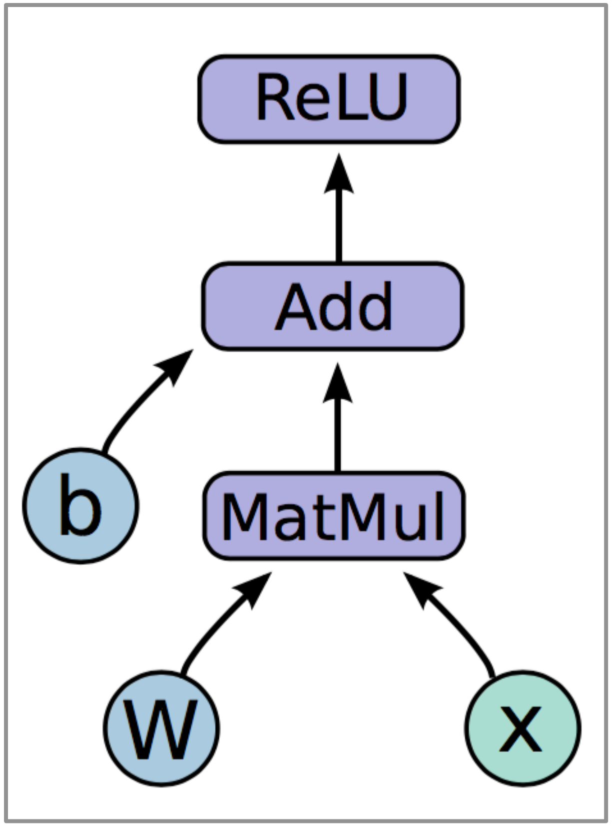
- Express a numeric computation as a **graph**.
- Graph nodes are **operations** which have any number of inputs and outputs
- Graph edges are **tensors** which flow between nodes



Basics

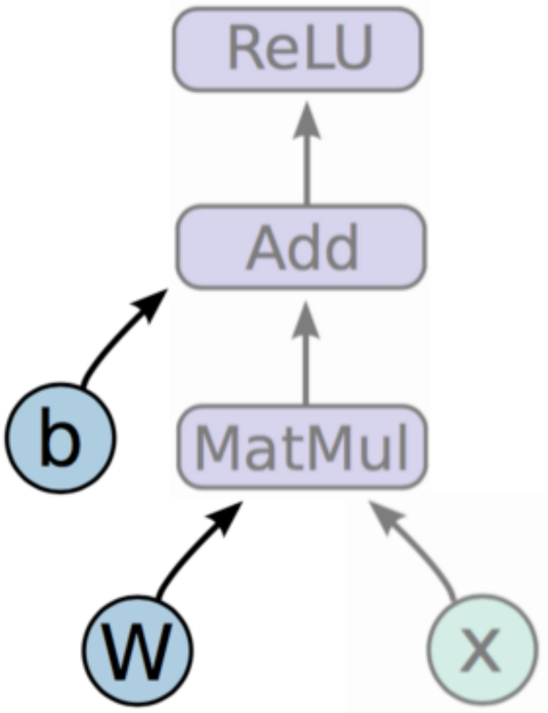
One-Layer neural network

$$h_i = \mathrm{ReLU}(Wx + b)$$



One-Layer neural network

$$h_i = \mathrm{ReLU}(Wx + b)$$

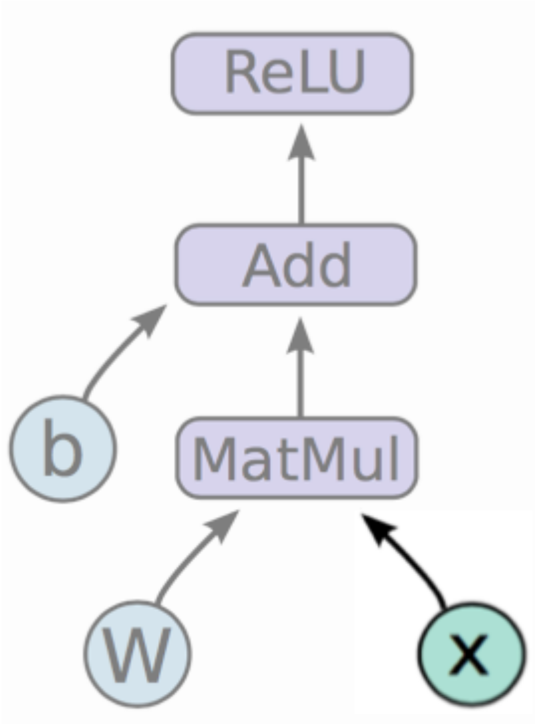


Variables: nodes whose value can be used and modified by the computation.

parameters

One-Layer neural network

$$h_i = \mathrm{ReLU}(Wx + b)$$

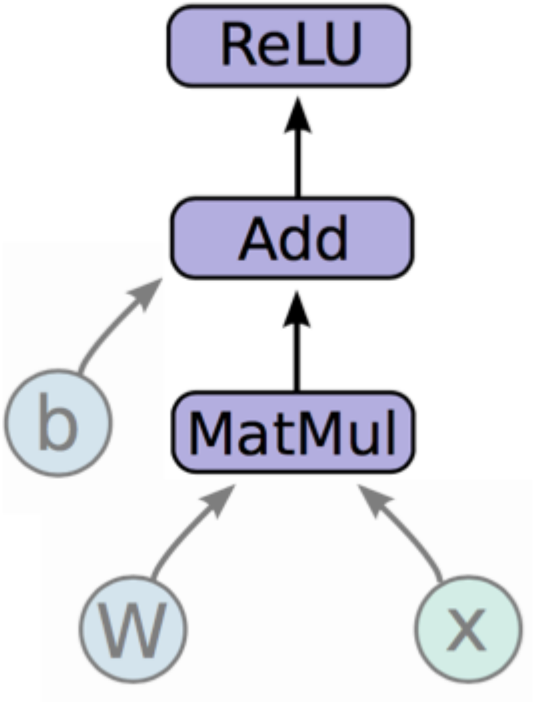


Placeholders: nodes whose value is fed in at execution time.

inputs, outputs…

One-Layer neural network

$$h_i = \text{ReLU}(Wx + b)$$



Mathematical operations:

MatMul: Multiply two matrix values.

Add: Add elementwise

ReLU: Rectified linear unit function.

Build a graph

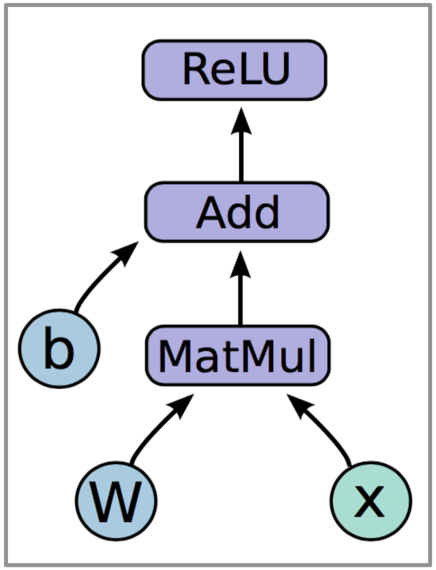
1. Create model weights, including initialization
   W ~ *Uniform*(-1, 1); b = **0**
2. Create input placeholder x
   m x 784 input matrix
3. Create computation graph

```
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, (None, 784))
h_i = tf.nn.relu(tf.matmul(x, W) + b)
```

$$h_i = \text{ReLU}(Wx + b)$$



Run the graph

So far we have defined a **graph**.

We can deploy this graph with a **session**: a binding to a particular execution context (e.g. CPU, GPU)

Run the graph

```
sess.run(fetches, feeds)
```

Fetches: List of graph nodes. Return the outputs of these nodes.

Feeds: Dictionary mapping from placeholders to concrete values. Specifies the value of each placeholders given in the dictionary.

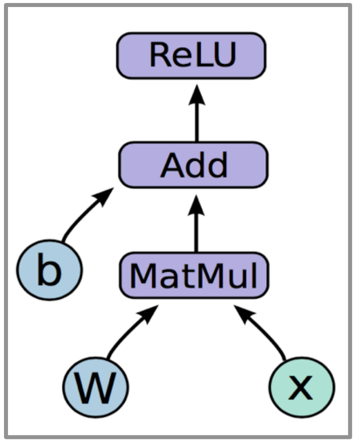
$$h_i = \mathrm{ReLU}(Wx + b)$$

```
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, (None, 784))
h_i = tf.nn.relu(tf.matmul(x, W) + b)

sess = tf.Session()
sess.run(tf.initialize_all_variables())
sess.run(h_i, {x: np.random.random(64, 784}})
```



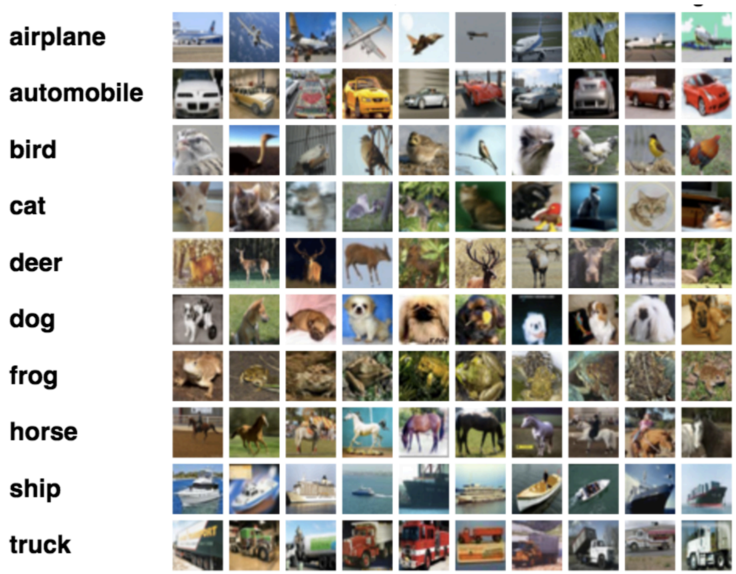
Basic flow

1. Build a graph
2. Initialize a session
3. Fetch and feed data with `Session.run`

A complete example

CIFAR-10 dataset

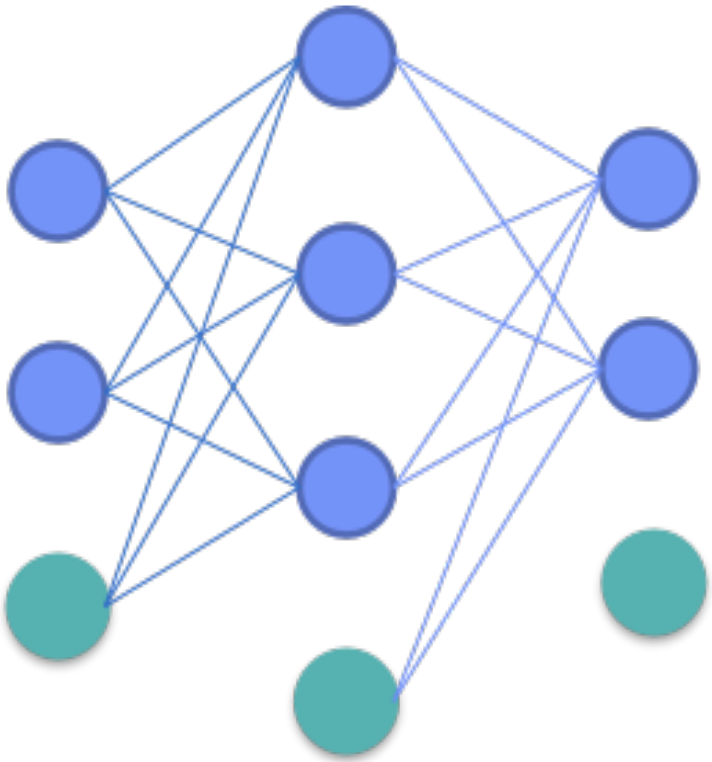
- Colored images in 10 classes



- Each image is a 32x32x3 dimensional array
- Training: 50k, Testing: 10k

Neural Network

Input Hidden(ReLU) Output(Softmax)



Load the dataset

- Use `load_CIFAR10`

```
from data_utils import load_CIFAR10
X_train, y_train, X_test, y_test = load_CIFAR10('./cifar-10-batches-py/')
```

```
X_train: (50000, 32, 32, 3) y_train: (50000,)
X_test: (10000, 32, 32, 3) y_test (10000,)
```

Load the dataset

- Use `load_CIFAR10`

```
from data_utils import load_CIFAR10
X_train, y_train, X_test, y_test = load_CIFAR10('./cifar-10-batches-py/')
```

```
X_train: (50000, 32, 32, 3) y_train: (50000,)
X_test: (10000, 32, 32, 3) y_test (10000,)
```

- Resample the dataset: train: 49k, validation: 1k, test: 1k

Load the dataset

- Use `load_CIFAR10`

```
from data_utils import load_CIFAR10
X_train, y_train, X_test, y_test = load_CIFAR10('./cifar-10-batches-py/')
```

```
X_train: (50000, 32, 32, 3) y_train: (50000,)
X_test: (10000, 32, 32, 3) y_test (10000,)
```

- Resample the dataset: train: 49k, validation: 1k, test: 1k
- Zero-center data

```
mean_image = np.mean(X_train, axis=0)
X_train -= mean_image
X_val -= mean_image
X_test -= mean_image
```

Load the dataset

- Use `load_CIFAR10`

```
from data_utils import load_CIFAR10
X_train, y_train, X_test, y_test = load_CIFAR10('./cifar-10-batches-py/')
```

```
X_train: (50000, 32, 32, 3) y_train: (50000,)
X_test: (10000, 32, 32, 3) y_test (10000,)
```

- Resample the dataset: train: 49k, validation: 1k, test: 1k
- Zero-center data

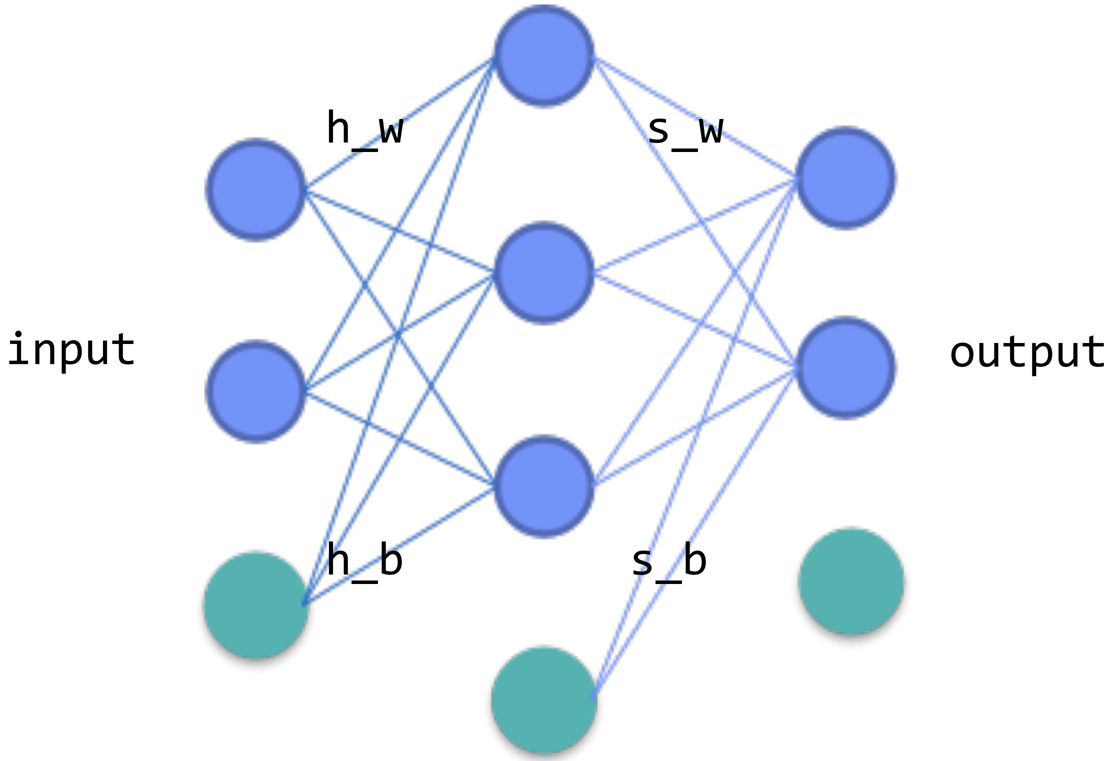
```
mean_image = np.mean(X_train, axis=0)
X_train -= mean_image
X_val -= mean_image
X_test -= mean_image
```

- Reshape images to row vectors

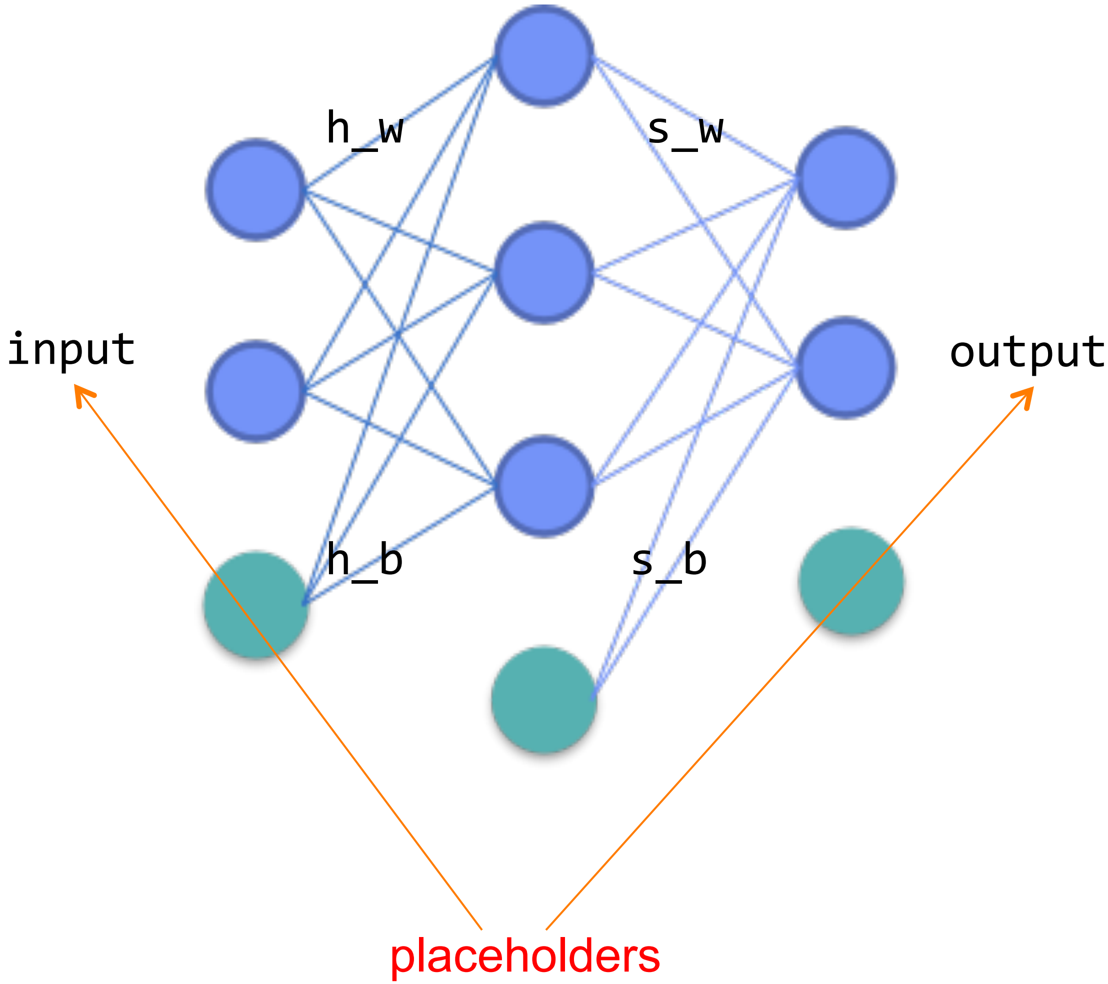
```
X_train = X_train.reshape(num_training, -1)
X_val = X_val.reshape(num_validation, -1)
X_test = X_test.reshape(num_test, -1)
```

```
training: (49000, 3072) (49000,)
validation: (1000, 3072) (1000,)
testing: (1000, 3072) (1000,)
```

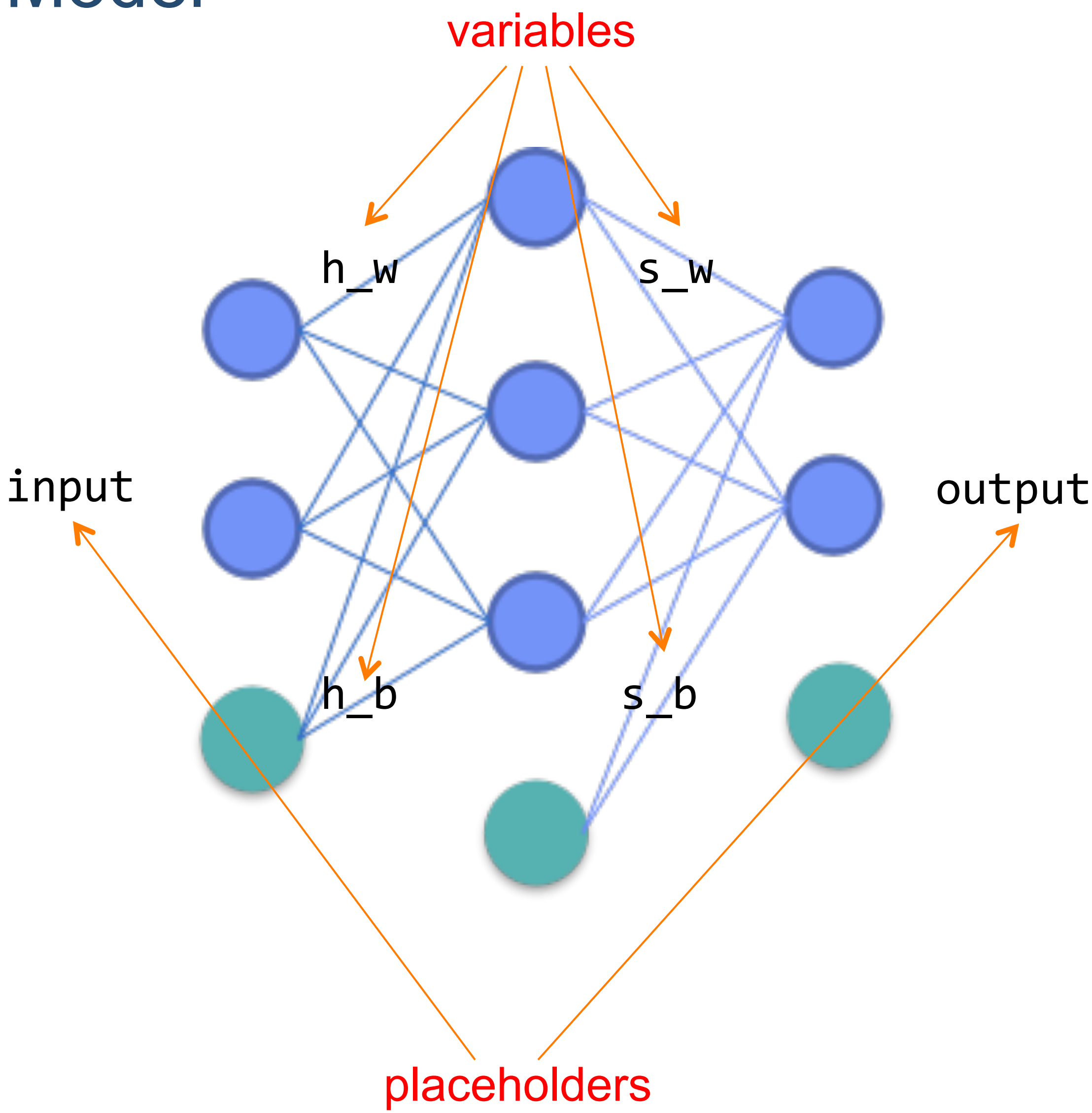
Model



Model



Model



Implementation of the Model

- Import TensorFlow

```
import tensorflow as tf
```

Implementation of the Model

- Import TensorFlow

```
import tensorflow as tf
```

- Define the placeholders

```
# Generate placeholders for the images and labels
images_placeholder = tf.placeholder(tf.float32, shape=(None,input_size),name='images')
labels_placeholder = tf.placeholder(tf.int64, shape=(None),name='labels')
```

Implementation of the Model

- Import TensorFlow

```
import tensorflow as tf
```

- Define the placeholders

```
# Generate placeholders for the images and labels
images_placeholder = tf.placeholder(tf.float32, shape=(None,input_size),name='images')
labels_placeholder = tf.placeholder(tf.int64, shape=(None),name='labels')
```

- Define the first layer

```
# hidden layer
with tf.name_scope('hidden'):
    # define variables (parameters to be trained)
    # initialization: http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf
    h_w = tf.Variable(
        tf.truncated_normal([input_size,hidden_size],
                            stddev=1.0 / math.sqrt(float(input_size))),
        name='hidden_weights')
    h_b = tf.Variable(tf.zeros([hidden_size]),name='hidden_biases')

    # define operations
    hidden = tf.nn.relu(tf.matmul(images_placeholder, h_w) + h_b)
```

define scope to organize variables

hidden/hidden_weights

Implementation of the Model

- Import TensorFlow

```
import tensorflow as tf
```

- Define the placeholders

```
# Generate placeholders for the images and labels
images_placeholder = tf.placeholder(tf.float32, shape=(None,input_size),name='images')
labels_placeholder = tf.placeholder(tf.int64, shape=(None),name='labels')
```

- Define the first layer

```
# hidden layer
with tf.name_scope('hidden'):
    # define variables (parameters to be trained)
    # initialization: http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf
    h_w = tf.Variable(
        tf.truncated_normal([input_size,hidden_size],
                            stddev=1.0 / math.sqrt(float(input_size))),
        name='hidden_weights')
    h_b = tf.Variable(tf.zeros([hidden_size]),name='hidden_biases')

    # define operations
    hidden = tf.nn.relu(tf.matmul(images_placeholder, h_w) + h_b)
```

initializer

Implementation of the Model

- Import TensorFlow

```
import tensorflow as tf
```

- Define the placeholders

```
# Generate placeholders for the images and labels
images_placeholder = tf.placeholder(tf.float32, shape=(None,input_size),name='images')
labels_placeholder = tf.placeholder(tf.int64, shape=(None),name='labels')
```

- Define the first layer

```
# hidden layer
with tf.name_scope('hidden'):
    # define variables (parameters to be trained)
    # initialization: http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf
    h_w = tf.Variable(
        tf.truncated_normal([input_size,hidden_size],
                            stddev=1.0 / math.sqrt(float(input_size))),
        name='hidden_weights')
    h_b = tf.Variable(tf.zeros([hidden_size]),name='hidden_biases')

    # define operations
    hidden = tf.nn.relu(tf.matmul(images_placeholder, h_w) + h_b)
```

shape of the variable

Implementation of the Model

- Import TensorFlow

```
import tensorflow as tf
```

- Define the placeholders

```
# Generate placeholders for the images and labels
images_placeholder = tf.placeholder(tf.float32, shape=(None,input_size),name='images')
labels_placeholder = tf.placeholder(tf.int64, shape=(None),name='labels')
```

- Define the first layer

```
# hidden layer
with tf.name_scope('hidden'):
    # define variables (parameters to be trained)
    # initialization: http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf
    h_w = tf.Variable(
        tf.truncated_normal([input_size,hidden_size],
                            stddev=1.0 / math.sqrt(float(input_size))),
        name='hidden_weights')
    h_b = tf.Variable(tf.zeros([hidden_size]),name='hidden_biases')

    # define operations
    hidden = tf.nn.relu(tf.matmul(images_placeholder, h_w) + h_b)
```

Implementation of the Model

- Define the second layer

```
# Linear
with tf.name_scope('softmax_linear'):
    s_w = tf.Variable(
        tf.truncated_normal([hidden_size, output_size],
                            stddev=1.0 / math.sqrt(float(hidden_size))),
        name='softmax_weights')
    s_b = tf.Variable(tf.zeros([output_size]),
                             name='softmax_biases')
    logits = tf.matmul(hidden, s_w) + s_b
```

Implementation of the Model

- Define the second layer

```
# Linear
with tf.name_scope('softmax_linear'):
    s_w = tf.Variable(
        tf.truncated_normal([hidden_size, output_size],
                            stddev=1.0 / math.sqrt(float(hidden_size))),
        name='softmax_weights')
    s_b = tf.Variable(tf.zeros([output_size]),
                             name='softmax_biases')
    logits = tf.matmul(hidden, s_w) + s_b
```

scores before normalization

Implementation of the Model

- Define the second layer

```
# Linear
with tf.name_scope('softmax_linear'):
    s_w = tf.Variable(
        tf.truncated_normal([hidden_size, output_size],
                            stddev=1.0 / math.sqrt(float(hidden_size))),
        name='softmax_weights')
    s_b = tf.Variable(tf.zeros([output_size]),
                      name='softmax_biases')
    logits = tf.matmul(hidden, s_w) + s_b
```

- Define the loss function

Softmax loss

```
# loss function
with tf.name_scope('loss'):
    cross_entropy = tf.nn.sparse_softmax_cross_entropy_with_logits(logits,
                                                                   labels_placeholder,
                                                                   name='xentropy')
    loss = tf.reduce_mean(cross_entropy, name='xentropy_mean')

    # add L2 regularization
    regularizers=tf.nn.l2_loss(h_w)+tf.nn.l2_loss(h_b)+tf.nn.l2_loss(s_w)+tf.nn.l2_loss(s_b)
    #reg=tf.constant(1.0,dtype=tf.float32)
    loss+=reg*regularizers
```

L2 regularizer

Training

- Define the training operation

optimization method

```
# Create the gradient descent optimizer with the given learning rate
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
# Use the optimizer to apply the gradients that minimize the loss
train_op = optimizer.minimize(loss)
```

Training

- Define the training operation

optimization method

```
# Create the gradient descent optimizer with the given learning rate
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
# Use the optimizer to apply the gradients that minimize the loss
train_op = optimizer.minimize(loss)
```

- Start a session

```
# An interactive session prevents garbage collection
sess=tf.InteractiveSession(graph=graph)
```

Training

- Define the training operation

optimization method

```
# Create the gradient descent optimizer with the given learning rate
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
# Use the optimizer to apply the gradients that minimize the loss
train_op = optimizer.minimize(loss)
```

- Start a session

```
# An interactive session prevents garbage collection
sess=tf.InteractiveSession(graph=graph)
```

- Initialize the variables

```
# Run the Op to initialize the variables
init = tf.initialize_all_variables()
sess.run(init)
```

Training

- Define the training operation

optimization method

```
# Create the gradient descent optimizer with the given learning rate
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
# Use the optimizer to apply the gradients that minimize the loss
train_op = optimizer.minimize(loss)
```

- Start a session

```
# An interactive session prevents garbage collection
sess=tf.InteractiveSession(graph=graph)
```

- Initialize the variables

```
# Run the Op to initialize the variables.
init = tf.initialize_all_variables()
sess.run(init)
```

- Get a batch of training samples (for each step)

```
# Pick an offset within the training data
offset = (step * batch_size) % (y_train.shape[0] - batch_size)
# Generate a minibatch.
batch_data = X_train[offset:(offset + batch_size), :]
batch_labels = y_train[offset:(offset + batch_size)]
```

Training

- Run the training operation (for each step)

```
# Fill a feed dictionary with the actual set of images and labels
# for this particular training step.
feed_dict = {images_placeholder:batch_data,
             labels_placeholder:batch_labels}
# Run one step of the model.  The return values are the activations
# from the `train_op` (which is discarded) and the `loss` Op.  To
# inspect the values of your Ops or variables, you may include them
# in the list passed to sess.run() and the value tensors will be
# returned in the tuple from the call.
_, loss_value = sess.run([train_op, loss],
                         feed_dict=feed_dict)
```

set the palceholders to concrete values

run the training operation with the given batch

Evaluation

- Evaluate accuracy

```
# For a classifier model, we can use the in_top_k Op.
# It returns a bool tensor with shape [batch_size] that is true for
# the examples where the label's is was in the top k (here k=1)
# of all logits for that example.
correct = tf.nn.in_top_k(logits, labels_placeholder, 1)
accuracy=tf.reduce_mean(tf.cast(correct, tf.int32))
feed_dict = {images_placeholder:X_train,
                  labels_placeholder:y_train}
precision=sess.run(accuracy,feed_dict=feed_dict)
```

top-k accuracy

Saving and restoring variables

- Create a Saver

```
# Create a saver for writing training checkpoints.
saver = tf.train.Saver()
```

- Save current variables

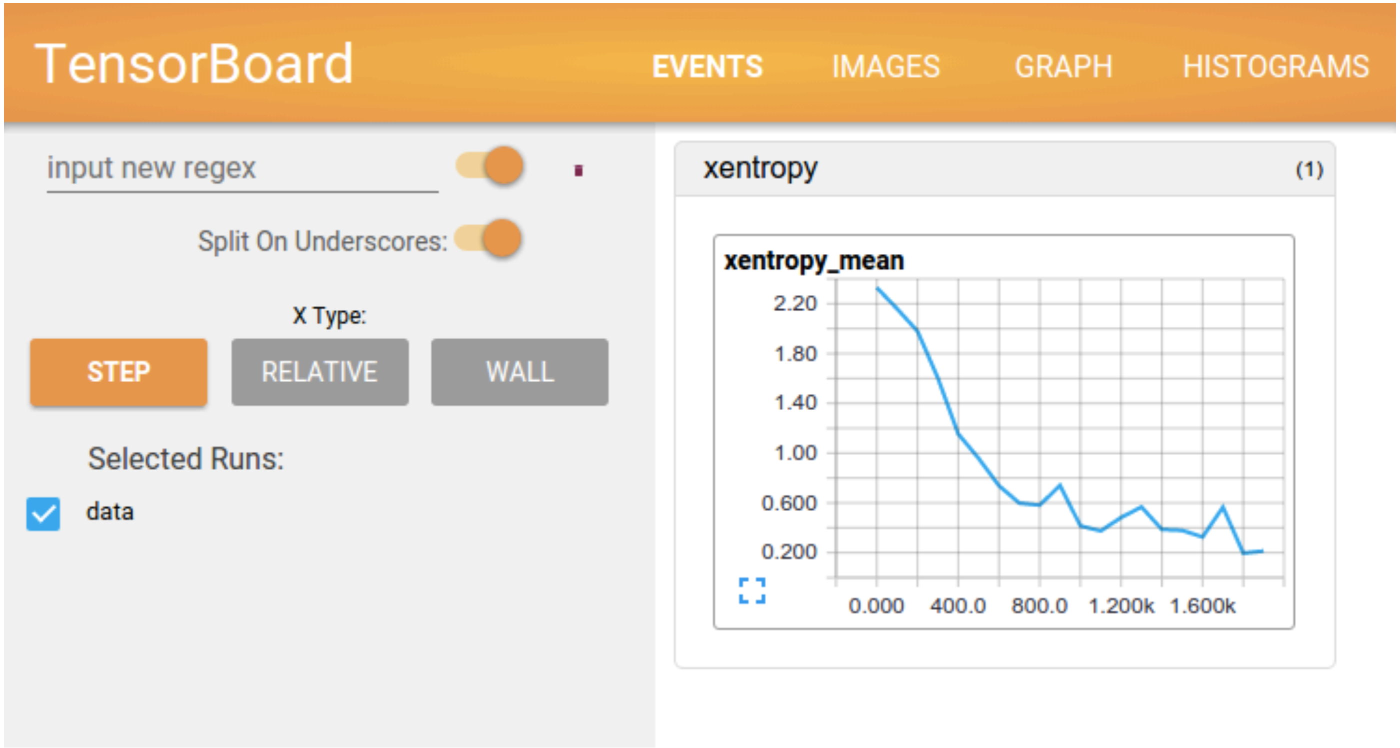
```
# Save the variables to disk
saver.save(sess, os.path.join(log_dir,'checkpoint'), global_step=step+1)
```

- Restore variables

```
# Restore variables from disk.
saver.restore(sess, os.path.join(log_dir,'checkpoint-1500'))
```

TensorBoard: Visualizing Learning

- Read and visualize summary data



TensorBoard: Visualizing Learning

- Collect summary data

```
# add summaries for logging
tf.scalar_summary('loss', loss)
```

- Merge all the summary operations

```
# Build the summary operation based on
# the TF collection of Summaries.
summary_op = tf.merge_all_summaries()
```

- Define a summary writer

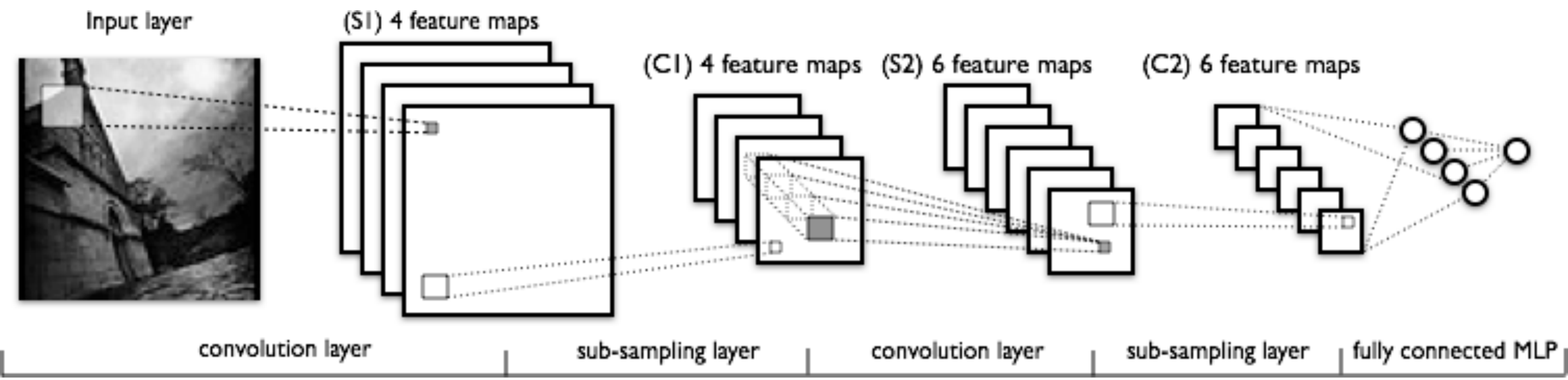
```
# Instantiate a SummaryWriter to output summaries and the Graph.
summary_writer = tf.train.SummaryWriter(log_dir,
                                        graph_def=sess.graph_def)
```

- Run summary operations

```
# Update the events file.
summary_str = sess.run(summary_op, feed_dict=feed_dict)
summary_writer.add_summary(summary_str, step)
```

Demo

Convolutional Neural Network



Convolutional Neural Network

- Images
  - 50000x32x32x3
- 1st layer: convolutional layer
  - 5x5 patch, 4 feature maps
- 2nd layer: max pooling layer
  - 2x2 block
- 3rd layer: convolutional layer
  - 5x5 patch, 6 feature maps
- 4th layer: max pooling layer
  - 2x2 block
- 5th layer: fully connected layer

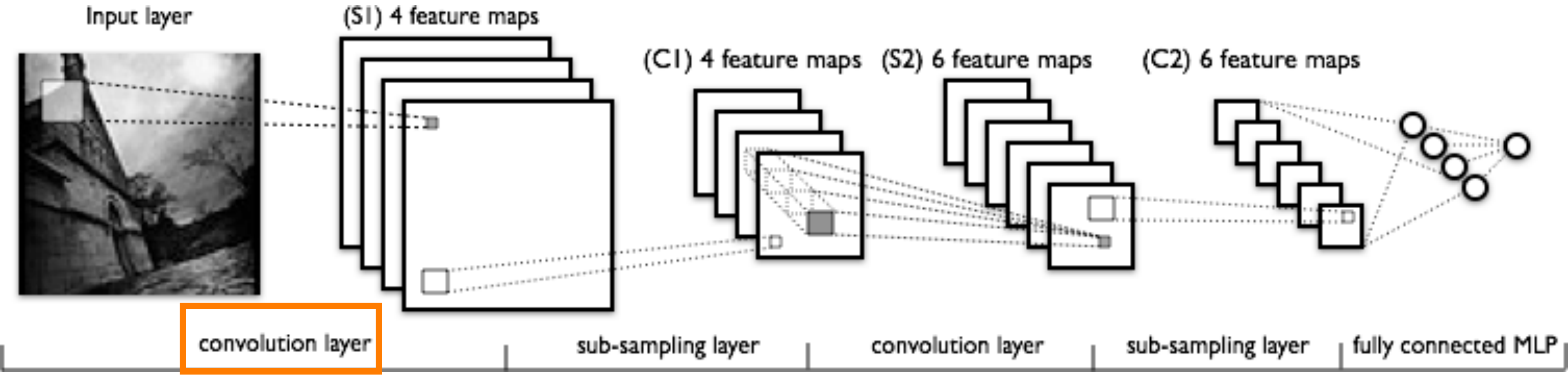
Implementation

- 1st layer: convolutional layer

```
def conv2d(x, W):
  return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')
```

image: batch_size x 32 x 32 x 3

```
W_conv1 = tf.Variable(tf.truncated_normal(shape=[5, 5, 3, 4],stddev=1e-4))
b_conv1 = tf.constant(0.1, [4])
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
```



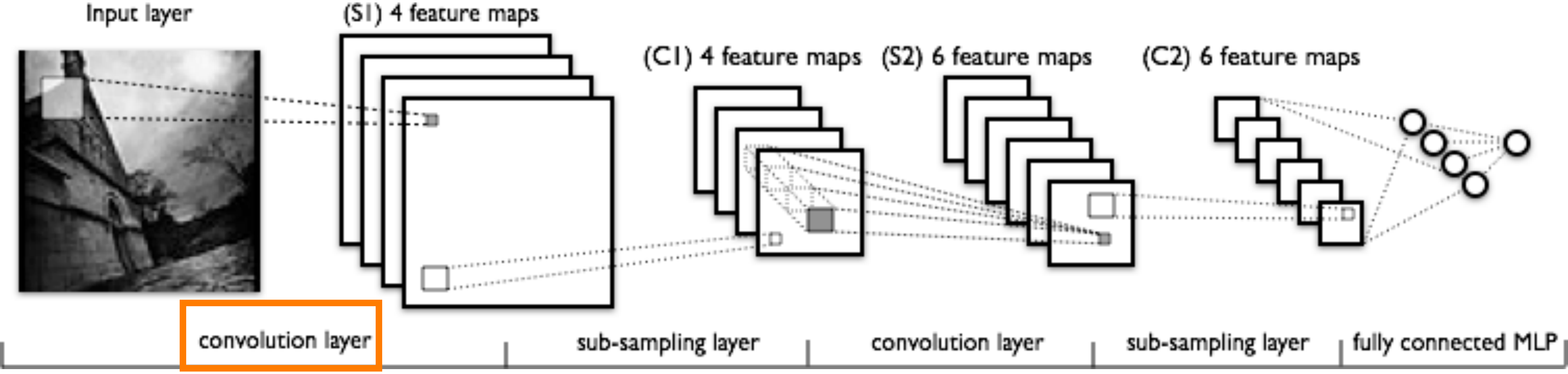
Implementation

- 1st layer: convolutional layer

```
def conv2d(x, W):
  return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')
```

filter: 5 x 5 x 3 x 4

```
W_conv1 = tf.Variable(tf.truncated_normal(shape=[5, 5, 3, 4],stddev=1e-4))
b_conv1 = tf.constant(0.1, [4])
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
```



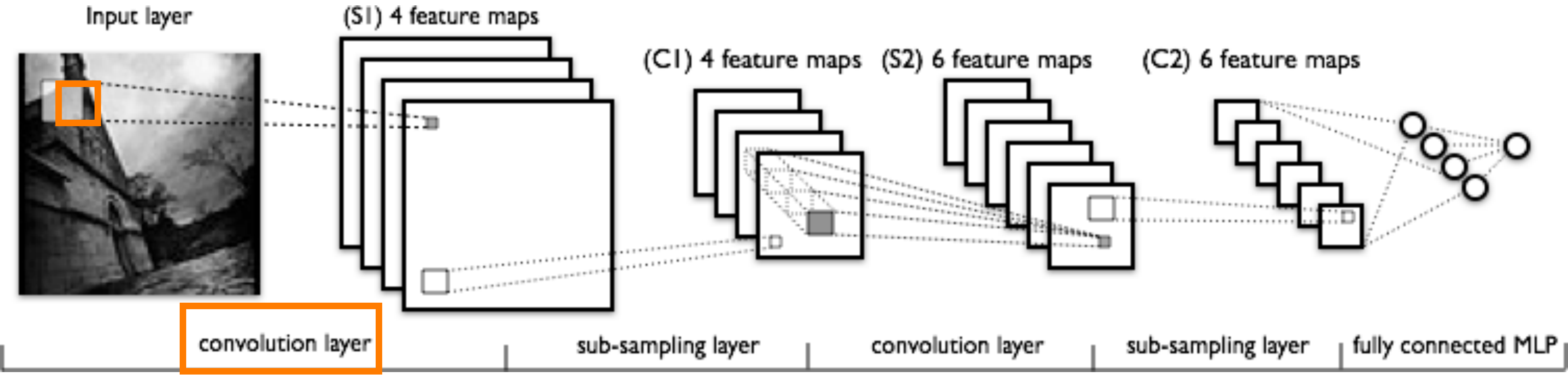
Implementation

- 1st layer: convolutional layer

```
def conv2d(x, W):
  return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')
```

how much to shift the sliding window

```
W_conv1 = tf.Variable(tf.truncated_normal(shape=[5, 5, 3, 4],stddev=1e-4))
b_conv1 = tf.constant(0.1, [4])
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
```



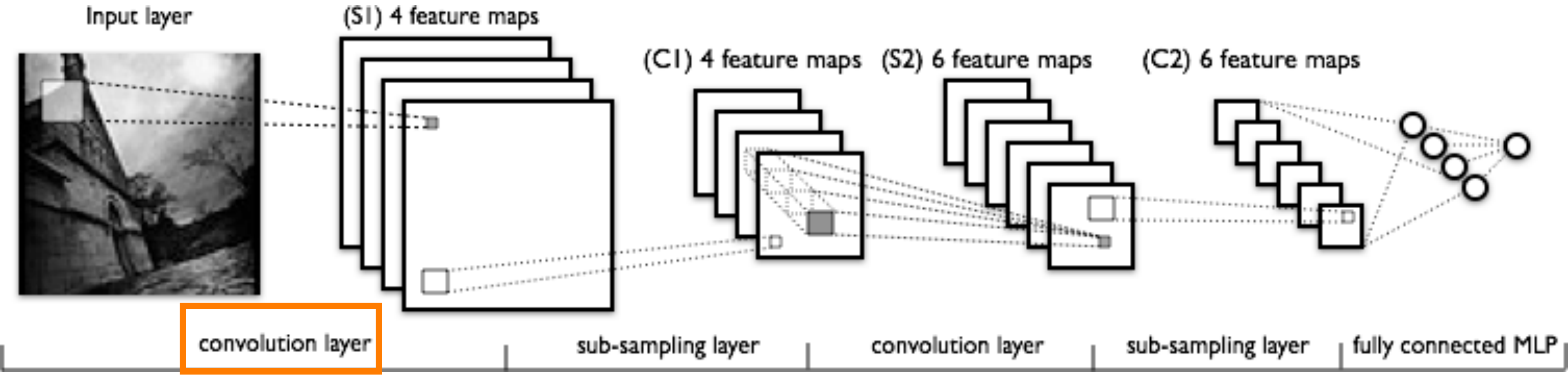
Implementation

- 1st layer: convolutional layer

```
def conv2d(x, W):
  return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

W_conv1 = tf.Variable(tf.truncated_normal(shape=[5, 5, 3, 4],stddev=1e-4))
b_conv1 = tf.constant(0.1, [4])
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
```

32 x 32



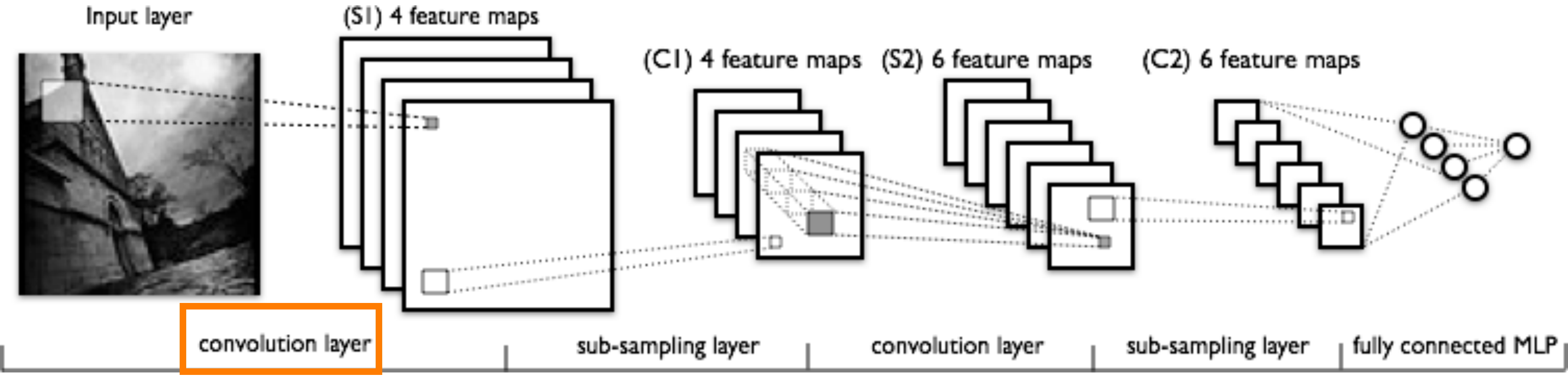
Implementation

- 1st layer: convolutional layer

```
def conv2d(x, W):
  return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')
```

```
W_conv1 = tf.Variable(tf.truncated_normal(shape=[5, 5, 3, 4],stddev=1e-4))
b_conv1 = tf.constant(0.1, [4])
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
```

define the first layer



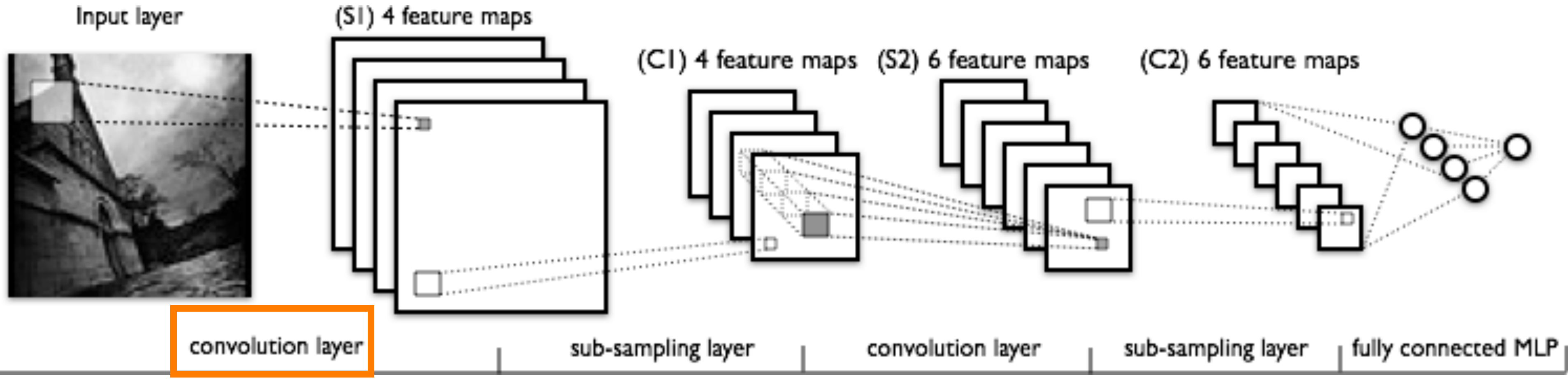
Implementation

- 1st layer: convolutional layer

```
def conv2d(x, W):
  return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

W_conv1 = tf.Variable(tf.truncated_normal(shape=[5, 5, 3, 4],stddev=1e-4))
b_conv1 = tf.constant(0.1, [4])
h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
```

batch_size x 32 x 32 x 4

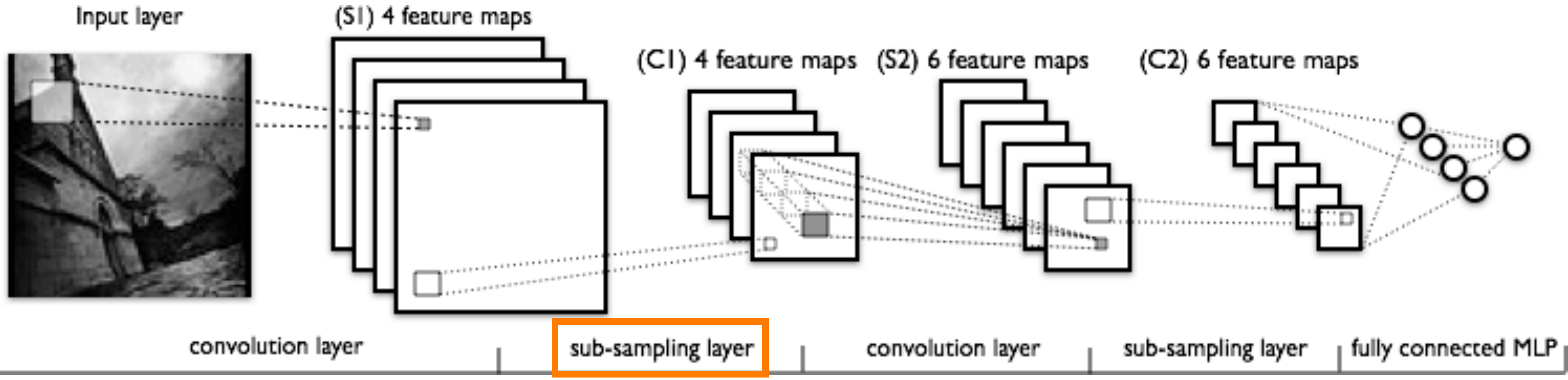


Implementation

- 2nd layer: max pooling layer

feature maps: batch_size x 32 x 32 x 4

```
def max_pool_2x2(x):
  return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
                        strides=[1, 2, 2, 1], padding='SAME')
h_pool2 = max_pool_2x2(h_conv1)
```

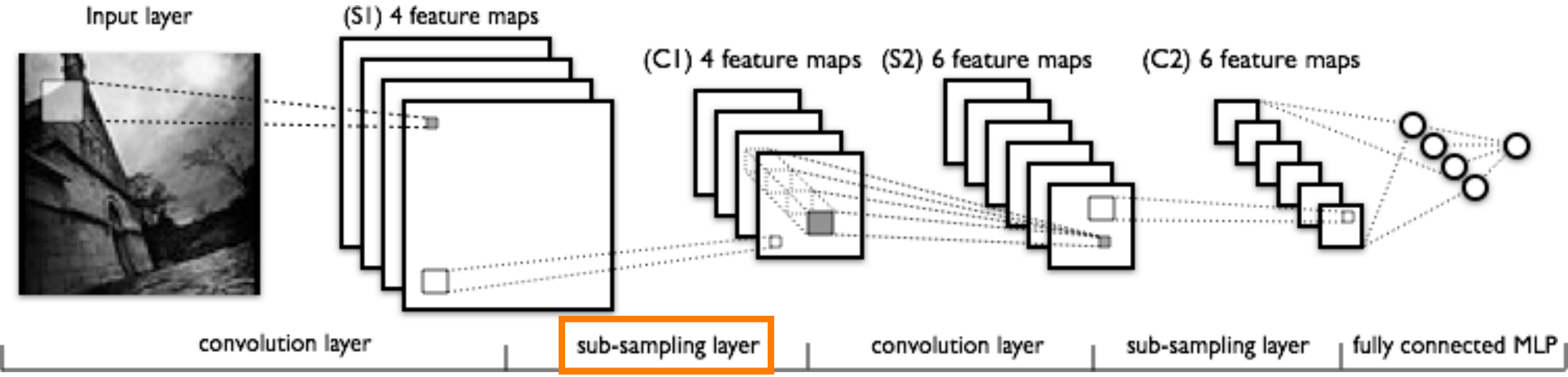


Implementation

- 2nd layer: max pooling layer

pooling block

```
def max_pool_2x2(x):
  return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
                        strides=[1, 2, 2, 1], padding='SAME')
h_pool2 = max_pool_2x2(h_conv1)
```

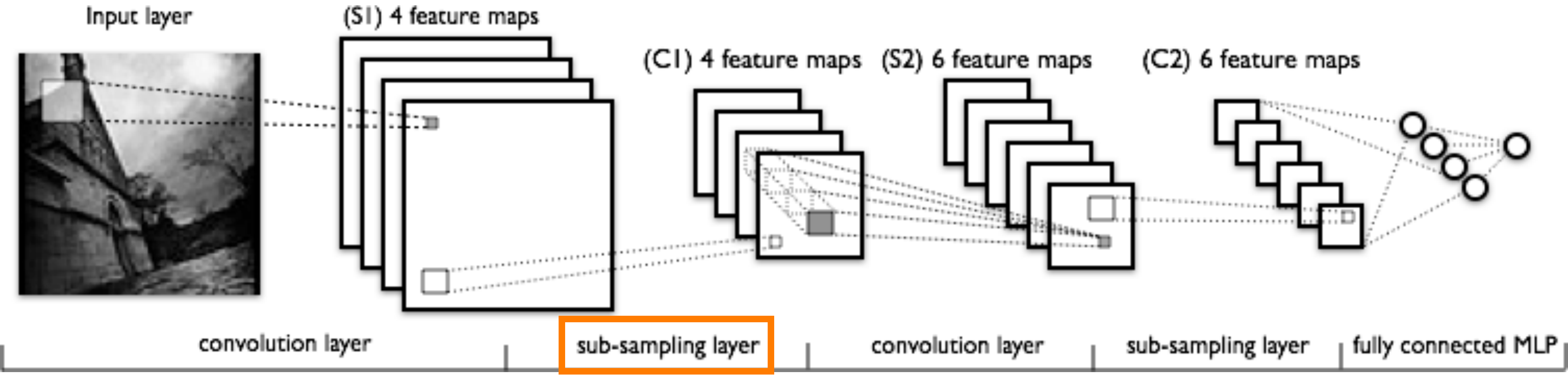


Implementation

- 2nd layer: max pooling layer

```
def max_pool_2x2(x):
  return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
                        strides=[1, 2, 2, 1], padding='SAME')
h_pool2 = max_pool_2x2(h_conv1)
```

batch_size x 16 x 16 x 4



Recommendation readings / videos

- Udacity:
  - Deep Learning: https://www.udacity.com/course/deep-learning--ud730
- Tutorial:
  - https://www.tensorflow.org/
  - TensorFlow WhitePaper: http://download.tensorflow.org/paper/whitepaper2015.pdf
  - Code: https://github.com/tensorflow

Questions?