CSC 498R: Internet of Things

Lecture 09: TensorFlow

Instructor: Haidar M. Harmanani

Fall 2017

IoT Components

- Things we connect: **Hardware, sensors *and* actuators**

- Connectivity
  - Medium we use to connect things

- Platform
  - Processing and storing collected data
    - Receive and send data via standardized interfaces or API
    - Store the data
    - Process the data.

- Analytics
  - Get insights from gathered data

- User Interface

What's TensorFlow™?

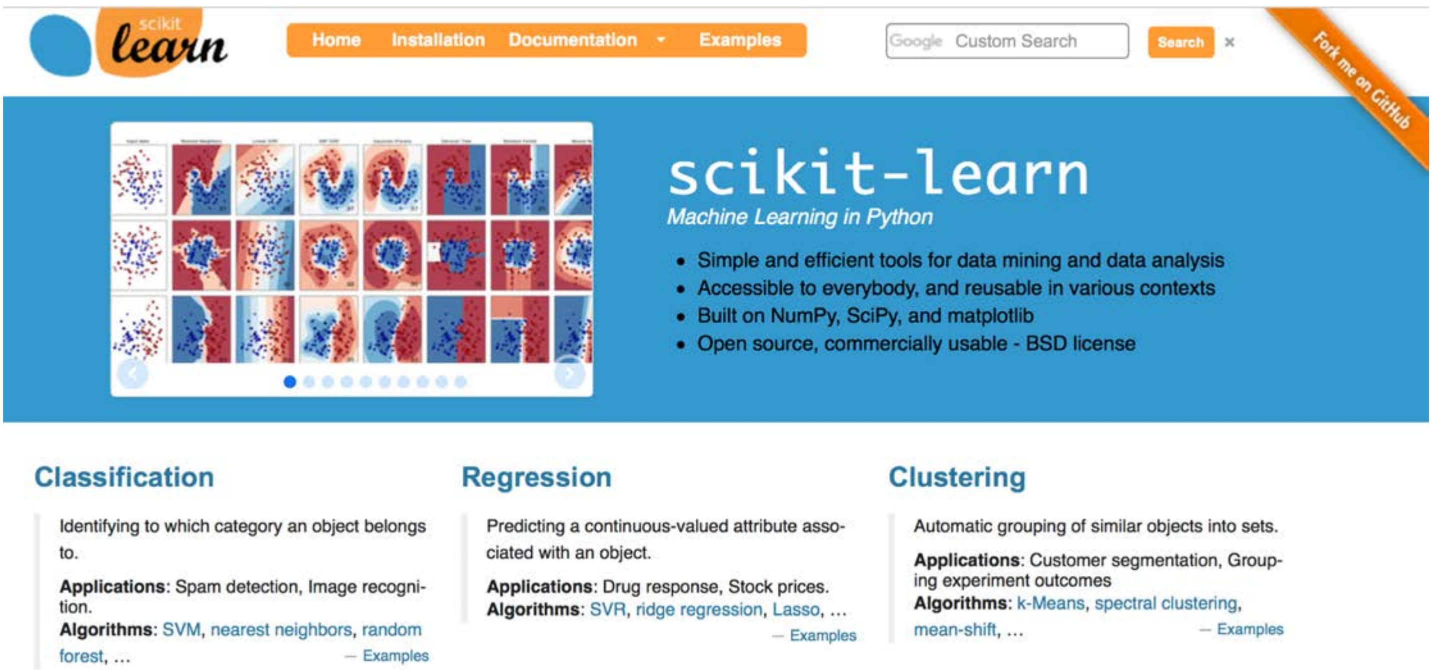
- Open source software library for numerical computation using data flow graphs
- Originally developed by *Google Brain Team* to conduct machine learning and deep neural networks research
- General enough to be applicable in a wide variety of other domains as well
- TensorFlow provides an extensive suite of functions and classes that allow users to build various models from scratch

Not the Only Deep Learning Library

- Other interesting deep/machine learning libraries
  - Theano [UoM]
  - scikit-learn [started as Google Summer of Code]
  - Torch
  - Caffe
  - CNTK [Miscrosoft]
  - DisBelief [Google]
  - cuDNN
- For comparison see:
  - https://en.wikipedia.org/wiki/Comparison_of_deep_learning_software

TensorFlow vs. scikit-learn

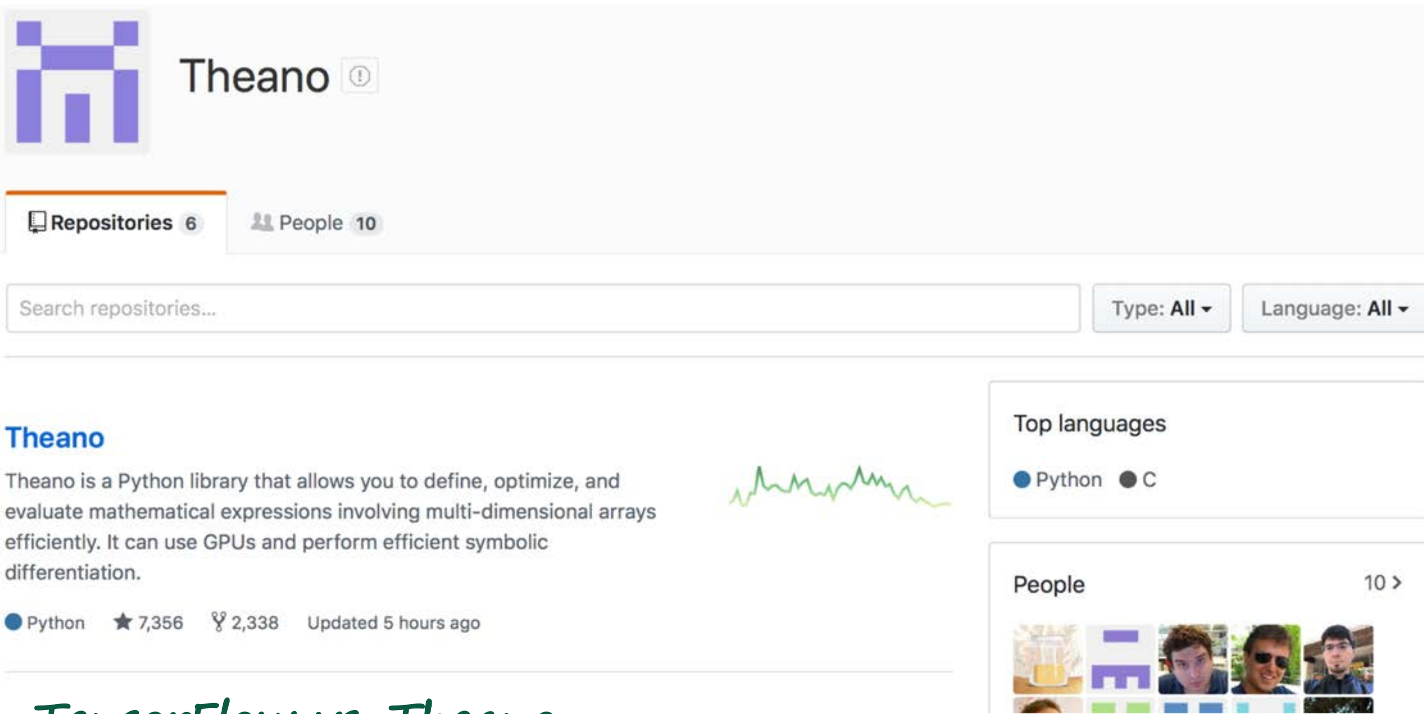
- scikit-learn
  - Model already built; "off-the-shelf"'
  - Fit/ predict style
- TensorFlow
  - Have to build model up
  - Should be able to describe your model in the form of a datagraph with functions like gradient descent, add, max, etc.



TensorFlow vs. Scikit-learn

TensorFlow vs. Theano

- Theano is a deep-learning library with python wrapper
- Very similar systems.
- TensorFlow has better support for distributed systems though, and has development funded by Google, while Theano is an academic project.

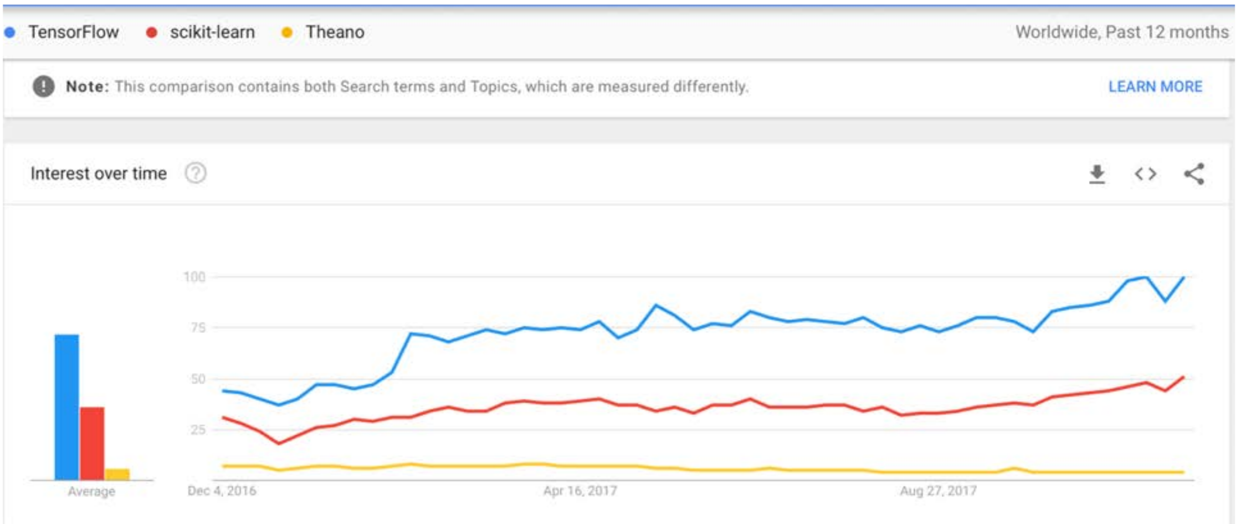


TensorFlow vs. Theano

TensorFlow vs. Numpy

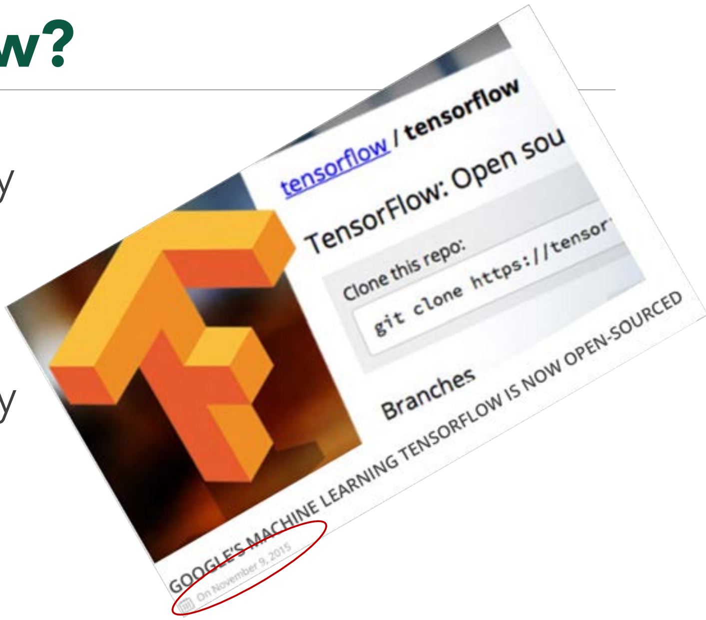
- Few people make this comparison, but TensorFlow and Numpy are quite similar.
- Numpy has Ndarray support, but doesn't offer methods to create tensor functions and automatically compute derivatives (+ no GPU support).

Google Trends to the Rescue



What is TensorFlow?

- A deep learning library recently open-sourced by Google.
- Provides primitives for defining functions on tensors and automatically computing their derivatives



What is TensorFlow?

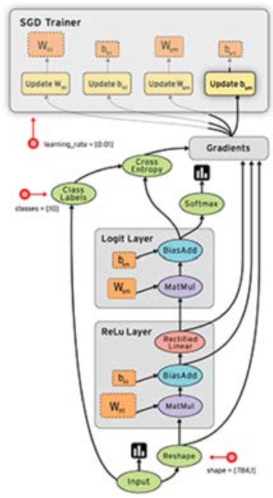
- Python API
- Portability: deploy computation to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API
- Flexibility: from Raspberry Pi, Android, Windows, iOS, Linux to server farms
- Visualization (TensorBoard)
- Checkpoints (for managing experiments)
- Auto-differentiation *autodiff* (no more taking derivatives by hand. Yay)
- Large community (> 10,000 commits and > 3000 TF-related repos in 1 year)
- Awesome projects already using TensorFlow

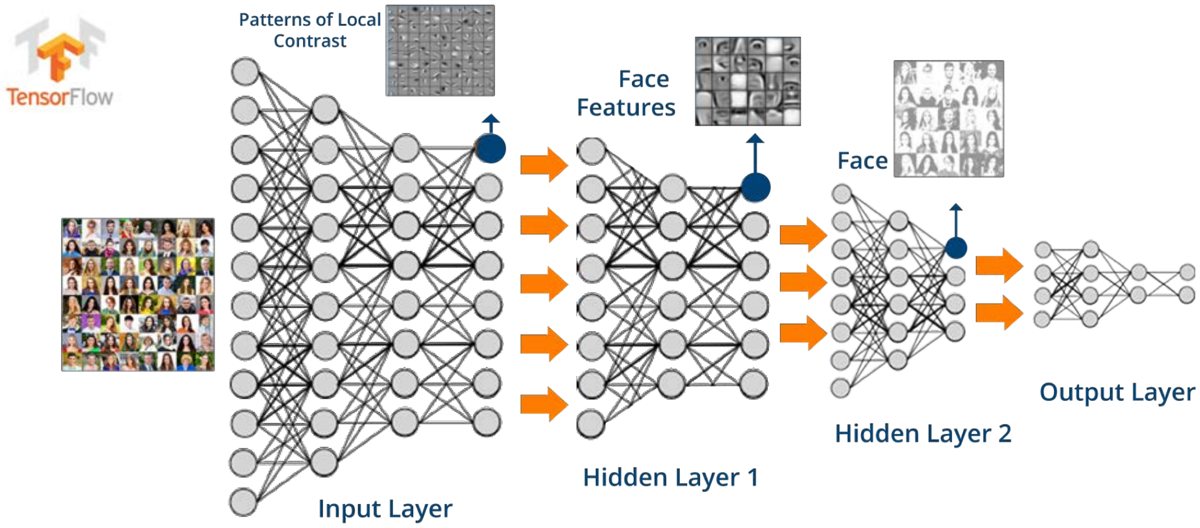
Companies using Tensorflow

- Google
- OpenAI
- DeepMind
- Snapchat
- Uber
- Airbus
- eBay
- Dropbox
- … and of course many startups

How Does it Work?

- Uses data flow graphs to represent a learning model
  - Comprise of nodes and edges
  - Nodes represent mathematical operations
  - Edges represent multi-dimensional data arrays (tensors)
  - “TensorFlow”
- Core is written in a combination of highly-optimized C++ and CUDA
  - Using Eigen and cuDNN





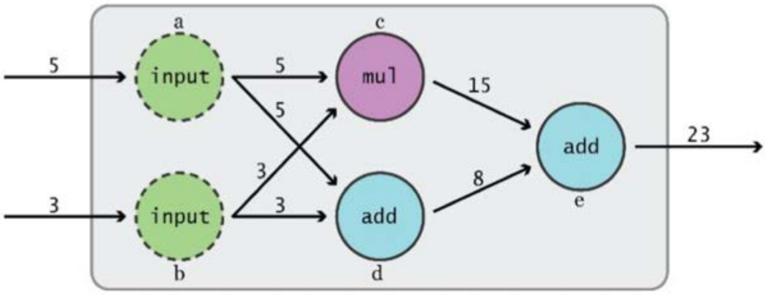
TensorFlow

Getting Started…

```
import tensorflow as tf
```

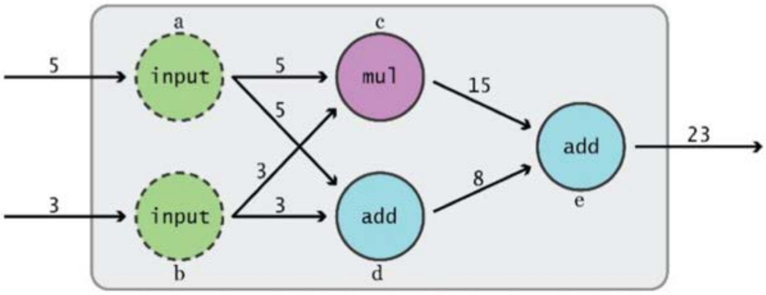
Data Flow Graphs

- TensorFlow separates definition of computations from their execution



Data Flow Graphs

- Phase 1: assemble a graph
- Phase 2: use a session to execute operations in the graph.

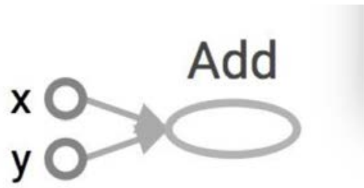


What's a Tensor?

- An n-dimensional matrix
  - 0-d tensor: scalar (number)
  - 1-d tensor: vector
  - 2-d tensor: matrix
  - and so on

Data Flow Graphs

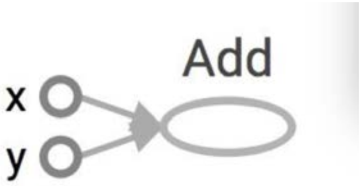
```
import tensorflow as tf
a = tf.add(2, 3)
```



- Why x, y?
  - TF automatically names the nodes when you don't explicitly name them.
  - For now:
    - x = 3
    - y = 5

Data Flow Graphs

```
import tensorflow as tf
a = tf.add(2, 3)
```



- Nodes: operators, variables, and constants
- Edges: tensors

- Tensors are data.
  - Data Flow ->Tensor Flow

Data Flow Graphs

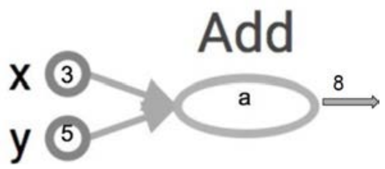
```
import tensorflow as tf
a = tf.add(2, 3)
print a
```


```
>> Tensor("Add:0", shape=(), dtype=int32)
(Not 5)
```

How to get the value of a?

- Create a session, assign it to variable `sess` so we can call it later
- Within the session, evaluate the graph to fetch the value of a

```
import tensorflow as tf
a = tf.add(3, 5)
sess = tf.Session()
print sess.run(a)          # >> 8
sess.close()
```

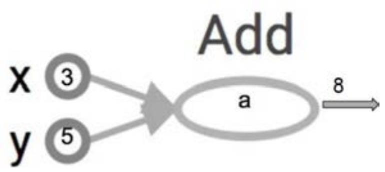


The session will look at the graph, trying to think: hmm, how can I get the value of a, then it computes all the nodes that leads to a.

How to get the value of a?

- Create a session, within the session, evaluate the graph to fetch the value of a

```
import tensorflow as tf
a = tf.add(3, 5)
# with clause takes care of sess.close()
with tf.Session() as sess:
    print (sess.run(a))
```



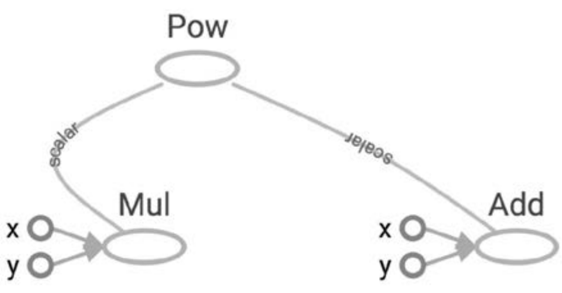
The session will look at the graph, trying to think: hmm, how can I get the value of a, then it computes all the nodes that leads to a.

tf.Session()

- A Session object encapsulates the environment in which Operation objects are executed, and Tensor objects are evaluated.

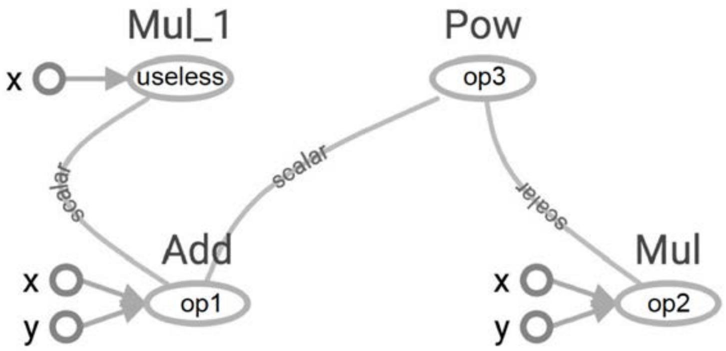
More Graphs

```
import tensorflow as tf
x = 2
y = 3
op1 = tf.add(x, y)
op2 = tf.multiply(x, y)
op3 = tf.pow(op2, op1)
with tf.Session() as sess:
   sess.run(op3)
```



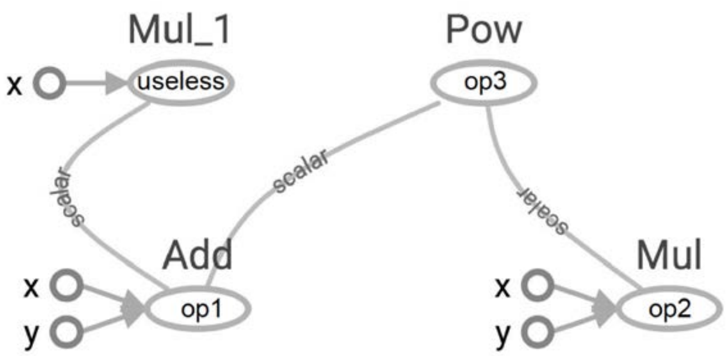
Subgraphs

```
import tensorflow as tf
x = 2
y = 3
op1 = tf.add(x, y)
op2 = tf.multiply(x, y)
useless = tf.multiply(x, op1)
op3 = tf.pow(op2, op1)
with tf.Session() as sess:
   op3 = sess.run(op3)
```



Because we only want the value of op3 and op3 doesn't depend on useless, session won't compute values of useless → save computation

Subgraphs

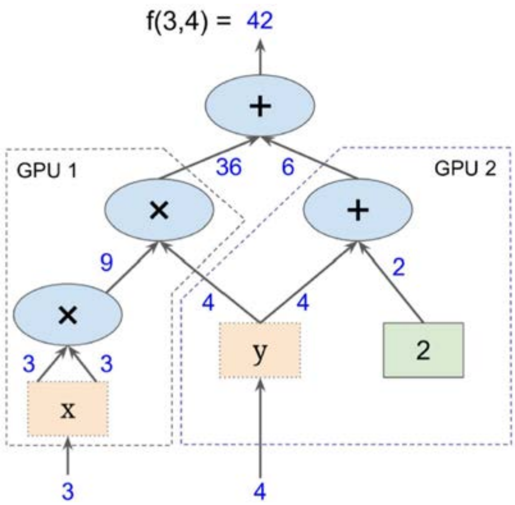



```
import tensorflow as tf
x = 2
y = 3
op1 = tf.add(x, y)
op2 = tf.multiply(x, y)
useless = tf.multiply(x, op1)
op3 = tf.pow(op2, op1)
with tf.Session() as sess:
    op3, not_useless = sess.run([op3, useless])
```

tf.Session.run(fetches, feed_dict=None, options=None, run_metadata=None)
Pass all variables whose values you want to a list in fetches

Subgraphs

- Possible to break graphs into several chunks and run them in parallel across multiple CPUs, GPUs, or devices



Distributed Computation

- To put part of a graph on a specific CPU or GPU:

```
import tensorflow as tf

# Creates a graph.
with tf.device('/gpu:2'):
    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.matmul(a, b)

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

# Runs the op.
print sess.run(c)
```

Building More Than One Graph

- You can but you don’t need more than one graph
  - The session runs the default graph
- But what if I really want to?
  - Multiple graphs require multiple sessions, each will try to use all available resources by default
  - Can't pass data between them without passing them through python/numpy, which doesn't work in distributed
  - It’s better to have disconnected subgraphs within one graph

Example

```
g = tf.Graph()
with g.as_default():
    a = 3
    b = 5
    x = tf.add(a, b)
sess = tf.Session(graph=g) # session is run on graph g
# run session
sess.close()
```

Example

- To handle the default graph:

```
g = tf.get_default_graph()
```


Why Graphs?

1) Save computation (only run subgraphs that lead to the values you want to fetch)
2) Break computation into small, differential pieces to facilitates auto-differentiation
3) Facilitate distributed computation, spread the work across multiple CPUs, GPUs, or devices
4) Many common machine learning models are commonly taught and visualized as directed graphs already

Back to Our First TensorFlow Program

```
import tensorflow as tf
a = tf.constant(2)
b = tf.constant(3)
x = tf.add(a, b)
with tf.Session() as sess:
   print sess.run(x)
```


Visualize Our First TensorFlow Program

```
import tensorflow as tf
a = tf.constant(2)
b = tf.constant(3)
x = tf.add(a, b)
with tf.Session() as sess:
   # add this line to use TensorBoard
   writer = tf.summary.FileWriter('./graphs', sess.graph)
   print (sess.run(x))
writer.close() # close the writer when you’re done using it
```

Run it

- Go to terminal, run:

```
$ python [yourprogram].py
$ tensorboard --logdir="./graphs" --port 6006
```

- Then open your browser and go to:

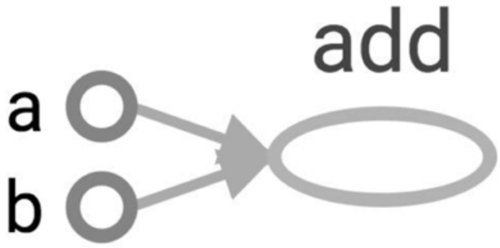
```
http://localhost:6006/
```


Visualize Our First TensorFlow Program

```
import tensorflow as tf
a = tf.constant(2)
b = tf.constant(3)
x = tf.add(a, b)
with tf.Session() as sess:
   # add this line to use TensorBoard
   writer = tf.summary.FileWriter('./graphs, sess.graph)
   print sess.run(x)
writer.close() # close the writer when you're done using it
```

Change Const, Const_1 to the names we give the variables

```
import tensorflow as tf
a = tf.constant(2, name="a")
b = tf.constant(3, name="b")
x = tf.add(a, b, name="add")
writer = tf.summary.FileWriter("./graphs", sess.graph)
with tf.Session() as sess:
   print sess.run(x) #>>5
```



TensorBoard helps when building complicated models.

More Constants

```
import tensorflow as tf
a = tf.constant([2, 2], name="a")
b = tf.constant([[0, 1], [2, 3]], name="b")
x = tf.add(a, b, name="add")
y = tf.multiply(a, b, name="mul")
with tf.Session() as sess:
    x, y = sess.run([x, y])
    print x, y
```

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

Tensors filled with a specific value

tf.zeros(shape, dtype=tf.float32, name=None)

- Creates a tensor of shape and all elements will be zeros (when ran in session)

```
tf.zeros([2, 3], tf.int32) ==>[[0, 0, 0], [0, 0, 0]] # Similar to numpy.zeros
```

more compact than other constants in the graph def → faster startup (esp. in distributed)

Tensors filled with a specific value

tf.zeros_like(input_tensor, dtype=None, name=None, optimize=True)

- Create a tensor of shape and type (unless type is specified) as the input_tensor but all elements are zeros

```
# input_tensor is [0, 1], [2, 3], [4, 5]]
tf.zeros_like(input_tensor) ==> [[0, 0], [0, 0], [0, 0]]
```


Tensors filled with a specific value

- Same:

```
tf.ones(shape, dtype=tf.float32, name=None)

tf.ones_like(input_tensor, dtype=None, name=None, optimize=True)
```

Similar to:
numpy.ones,
numpy.ones_like

Tensors filled with a specific value

- Same:

```
tf.fill(dims, value, name=None)
```

- creates a tensor filled with a scalar value.

```
tf.fill([2, 3], 8) ==>[[8, 8, 8], [8, 8, 8]]
```

In numpy, this takes two step:
1. Create a numpy array a
2. a.fill(value)

Constants as Sequences

```
tf.linspace(start, stop, num, name=None) # slightly different from np.linspace
tf.linspace(10.0, 13.0, 4) ==>[10.0 11.0 12.0 13.0]

tf.range(start, limit=None, delta=1, dtype=None, name='range')
```

- # 'start' is 3, 'limit' is 18, 'delta' is 3

```
tf.range(start, limit, delta) ==>[3, 6, 9, 12, 15]
```

- # 'limit' is 5

```
tf.range(limit) ==>[0, 1, 2, 3, 4]
```

- Tensor objects are not iterable

```
for _ in tf.range(4): # TypeError
```

Randomly Generated Constants

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

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

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

tf.random_shuffle(value, seed=None, name=None)

tf.random_crop(value, size, seed=None, name=None)

tf.multinomial(logits, num_samples, seed=None, name=None)

tf.random_gamma(shape, alpha, beta=None, dtype=tf.float32, seed=None, name=None)
```


Randomly Generated Constants

```
tf.set_random_seed(seed)
```

Operations

```
a = tf.constant([3, 6])
b = tf.constant([2, 2])

tf.add(a, b) #>>[5 8]
tf.add_n([a, b, b]) #>>[7 10]. Equivalent to a + b + b

tf.multiply(a, b) #>>[6 12] because mul is element wise

tf.matmul(a, b) #>>ValueError
tf.matmul(tf.reshape(a, [1, 2]), tf.reshape(b, [2, 1])) #>>[[18]]

tf.div(a, b) #>>[1 3]
tf.mod(a, b) #>>[1 0]
```


TensorFlow Data Types

- TensorFlow takes Python natives types: boolean, numeric (int, float), strings

```
# 0-d tensor, or "scalar"
t_0 = 19
tf.zeros_like(t_0) # ==> 0
tf.ones_like(t_0) # ==> 1

# 1-d tensor, or "vector"
t_1 = ['apple', 'peach', 'banana']
tf.zeros_like(t_1) # ==> ['' '' '']
tf.ones_like(t_1) # ==> TypeError: Expected string, got 1 of type 'int' instead.

# 2x2 tensor, or "matrix"
t_2 = [[True, False, False],
       [False, False, True],
       [False, True, False]]

tf.zeros_like(t_2) # ==> 2x2 tensor, all elements are False
tf.ones_like(t_2) # ==> 2x2 tensor, all elements are True
TensorFlow Data Types
```

TF vs NP Data Types

- TensorFlow integrates seamlessly with NumPy

```
tf.int32 == np.int32 # True
```

- Can pass numpy types to TensorFlow ops

```
tf.ones([2, 2], np.float32) # ⇒ [[1.0 1.0], [1.0 1.0]]
```

- For `tf.Session.run(fetches):`
  – If the requested fetch is a Tensor , then the output of will be a NumPy ndarray.

Notes

- Constants are stored in the graph definition
  – This makes loading graphs expensive when constants are big
  – Only use constants for primitive types.
- Use variables or readers for more data that requires more memory

Variables

- # create variable a with scalar value

```
a = tf.Variable(2, name="scalar")
```

Note that tf.Variable is a class, but tf.constant is an op

- # create variable b as a vector

```
b = tf.Variable([2, 3], name="vector")
```

- # create variable c as a 2x2 matrix

```
c = tf.Variable([[0, 1], [2, 3]], name="matrix")
```

- # create variable W as 784 x 10 tensor, filled with zeros

```
W = tf.Variable(tf.zeros([784,10]))
```


You have to initialize your variables

- The easiest way is initializing all variables at once:

```
init = tf.global_variables_initializer()
with tf.Session() as sess:
        sess.run(init)
```

- Initialize only a subset of variables:

```
init_ab = tf.variables_initializer([a, b], name="init_ab")
with tf.Session() as sess:
        sess.run(init_ab)
```

- Initialize a single variable

```
W = tf.Variable(tf.zeros([784,10]))
with tf.Session() as sess:
        sess.run(W.initializer)
```

Eval() a variable

```
# W is a random 700 x 100 variable object
W = tf.Variable(tf.truncated_normal([700, 10]))
with tf.Session() as sess:
   sess.run(W.initializer)
   print W

>>Tensor("Variable/read:0", shape=(700, 10), dtype=float32)
```


eval() a variable

```
# W is a random 700 x 100 variable object
W = tf.Variable(tf.truncated_normal([700, 10]))
with tf.Session() as sess:
   sess.run(W.initializer)
   print W

>>>> [[-0.76781619 -0.67020458 1.15333688 ..., -0.98434633 -1.25692499 -0.90904623]
[-0.36763489 -0.65037876 -1.52936983 ..., 0.19320194 -0.38379928 0.44387451]
[ 0.12510735 -0.82649058 0.4321366 ..., -0.3816964 0.70466036 1.33211911]
...,
[ 0.9203397 -0.99590844 0.76853162 ..., -0.74290705 0.37568584 0.64072722]
[-0.12753558 0.52571583 1.03265858 ..., 0.59978199 -0.91293705 -0.02646019]
[ 0.19076447 -0.62968266 -1.97970271 ..., -1.48389161 0.68170643 1.46369624]]
```

tf.Variable.assign()

```
tf.Variable.assign()
W = tf.Variable(10)
W.assign(100)
with tf.Session() as sess:
    sess.run(W.initializer)
    print W.eval()          #>>10
```

W.assign(100) doesn't assign the value 100 to W. It creates an assign op, and that op needs to be run to take effect.

tf.Variable.assign()

```
tf.Variable.assign()
W = tf.Variable(10)
W.assign(100)
with tf.Session() as sess:
    sess.run(W.initializer)
    print W.eval()          #>>10
```

```
W = tf.Variable(10)
assign_op = W.assign(100)
with tf.Session() as sess:
    sess.run(W.initializer)
    sess.run(assign_op)
print W.eval()          # >> 100
```

W.assign(100) doesn't assign the value 100 to W. It creates an assign op, and that op needs to be run to take effect.

assign_add() and assign_sub()

```
my_var = tf.Variable(10)
With tf.Session() as sess:
   sess.run(my_var.initializer)
   # increment by 10
   sess.run(my_var.assign_add(10)) #>>20
   # decrement by 2
   sess.run(my_var.assign_sub(2)) #>>18
```

assign_add() and assign_sub() can't initialize the variable my_var because these ops need the original value of my_var

Each session maintains its own copy of variable

```
W = tf.Variable(10)
sess1 = tf.Session()
sess2 = tf.Session()
sess1.run(W.initializer)
sess2.run(W.initializer)
print sess1.run(W.assign_add(10)) #>>20
print sess2.run(W.assign_sub(2)) #>> 8
print sess1.run(W.assign_add(100)) # >> 120
print sess2.run(W.assign_sub(50)) # >> -42
sess1.close()
sess2.close()
```

Use a variable to initialize another variable

- Want to declare U = 2 * W

```
# W is a random 700 x 100 tensor
W = tf.Variable(tf.truncated_normal([700, 10]))
U = tf.Variable(2 * W)
```

Not so safe (but quite common)

Use a variable to initialize another variable

- Want to declare U = 2 * W

```
# W is a random 700 x 100 tensor
W = tf.Variable(tf.truncated_normal([700, 10]))
U = tf.Variable(2 * W.intialized_value())

# ensure that W is initialized before its value is used to initialize U
```

Safer

Placeholder

- A TF program often has 2 phases:
  - Assemble a graph
  - Use a session to execute operations in the graph
- Can assemble the graph first without knowing the values needed for computation
- Analogy:
  - Can define the function f(x, y) = x*2 + y without knowing value of x or y.
    - x, y are placeholders for the actual values.

Placeholders

- We, or our clients, can later supply their own data when they need to execute the computation

```
tf.placeholder(dtype, shape=None, name=None)
# create a placeholder of type float 32-bit, shape is a vector of 3 elements
a = tf.placeholder(tf.float32, shape=[3])
# create a constant of type float 32-bit, shape is a vector of 3 elements
b = tf.constant([5, 5, 5], tf.float32)
# use the placeholder as you would a constant or a variable
c = a + b # Short for tf.add(a, b)
with tf.Session() as sess:
    print sess.run(c) # Error because a doesn’t have any value
```

Placeholders

- Feed the values to placeholders using a dictionary

```
# create a placeholder of type float 32-bit, shape is a vector of 3 elements
a = tf.placeholder(tf.float32, shape=[3])

# create a constant of type float 32-bit, shape is a vector of 3 elements
b = tf.constant([5, 5, 5], tf.float32)

# use the placeholder as you would a constant or a variable
c = a + b # Short for tf.add(a, b)
with tf.Session() as sess:
    # feed [1, 2, 3] to placeholder a via the dict {a: [1, 2, 3]}
    # fetch value of c
    print sess.run(c, {a: [1, 2, 3]}) # the tensor a is the key, not the string 'a'

#>>[6, 7, 8]
```


Placeholders

- Placeholders are valid ops
- How about feeding multiple data points in?
- We feed all the values in, one at a time

```
with tf.Session() as sess:
    for a_value in list_of_values_for_a:
        print sess.run(c, {a: a_value})
```

Placeholder is just a way to indicate that something must be fed

Placeholder

Feeding values to TF ops

```
tf.Graph.is_feedable(tensor)
# True if and only if tensor is feedable.
```

Feeding values to TF ops

```
# create operations, tensors, etc (using the default graph)
a = tf.add(2, 5)
b = tf.mul(a, 3)
with tf.Session() as sess:
   # define a dictionary that says to replace the
   # value of 'a' with 15
   replace_dict = {a: 15}
   # Run the session, passing in 'replace_dict' as the value
   # to 'feed_dict’
   sess.run(b, feed_dict=replace_dict) # returns 45
```


Avoid Lazy Loading

- Separate the assembling of graph and executing ops
- Use Python attribute to ensure a function is only loaded the first time it’s called

Linear Regression Using TensorFlow

- Recall: Linear Regression models relationship between a scalar dependent variable y and independent variables X

Linear Regression Using TensorFlow

We often hear insurance companies using factors such as number of fire and theft in a neighborhood to calculate how dangerous the neighborhood is.

Linear Regression Using TensorFlow

Question: is it redundant? Is there a relationship between the number of fire and theft in a neighborhood, and if there is, can we find it?

Can we find a function f so that if X is the number of fires and Y is the number of thefts, then: Y = f(X)?

Linear Regression Using TensorFlow

- The City of Chicago
  - X: number of incidents of fire
  - Y: number of incidents of theft
- Predict Predict Y from X
- Model
  - w * X + b
  - $(Y - Y\_predicted)^2$

Data Set

- Name: Fire and Theft in Chicago
  - X = fires per 1000 housing units
  - Y = thefts per 1000 population within the same Zip code in the Chicago metro area
  - Total number of Zip code areas: 42

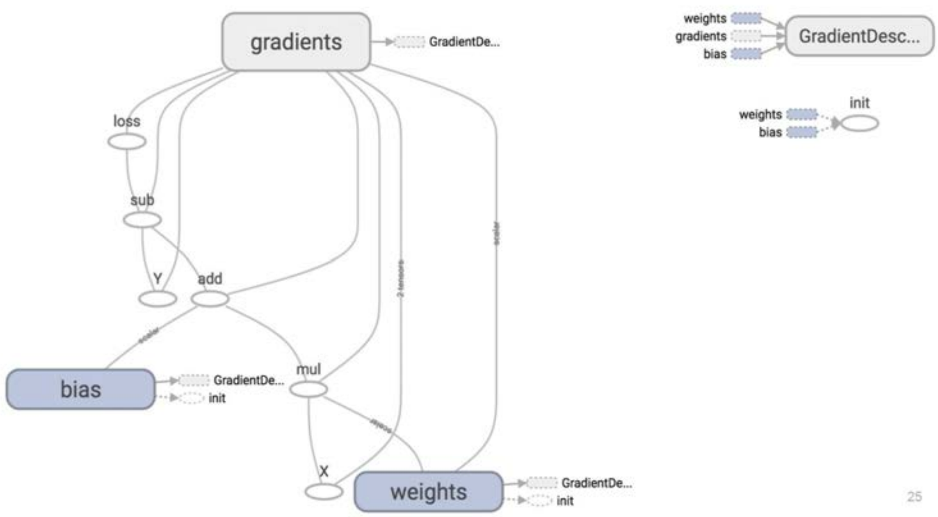
Phase 1: Assemble our graph

- Step 1: Read in data
- Step 2: Create placeholders for inputs and labels
- Step 3: Create weight and bias
- Step 4: Build model to predict Y
- Step 5: Specify loss function
- Step 6: Create optimizer

Phase 2: Train our model

- Initialize variables
- Run optimizer op
  - (with data fed into placeholders for inputs and labels)

Model



Plot the results with matplotlib

- Step 1: Uncomment the plotting code at the end of your program
- Step 2: Run it again

ValueError?

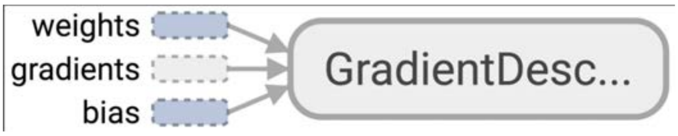
```
w, b = sess.run([w, b])
```

How does TensorFlow know what variables to update?

- Optimizer

```
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001).minimize(loss)
_, l = sess.run([optimizer, loss], feed_dict={X: x, Y:y})
```

- Session looks at all trainable variables that loss depends on and update them



Trainable variables

```
tf.Variable(initial_value=None, trainable=True, collections=None,
            validate_shape=True, caching_device=None, name=None,
            variable_def=None, dtype=None,
            expected_shape=None, import_scope=None)
```

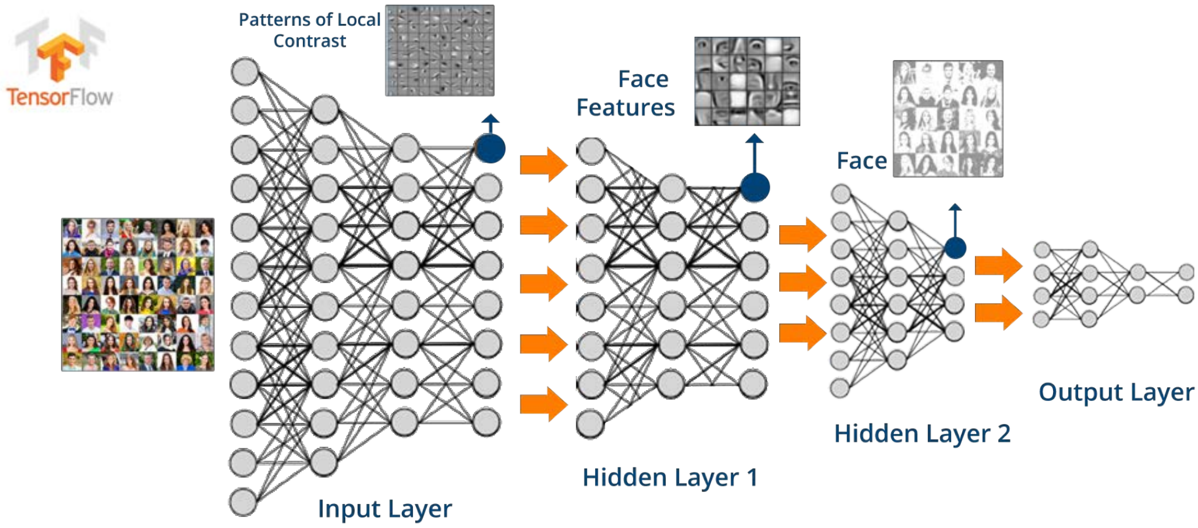
List of optimizers in TF

```
tf.train.GradientDescentOptimizer
tf.train.AdagradOptimizer
tf.train.MomentumOptimizer
tf.train.AdamOptimizer
tf.train.ProximalGradientDescentOptimizer
tf.train.ProximalAdagradOptimizer
tf.train.RMSPropOptimizer
And more
```



```
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
import xlrd

DATA_FILE = "data/fire_theft.xls"

# Step 1: read in data from the .xls file
book = xlrd.open_workbook(DATA_FILE, encoding_override="utf-8")
sheet = book.sheet_by_index(0)
data = np.asarray([sheet.row_values(i) for i in range(1, sheet.nrows)])
n_samples = sheet.nrows - 1

# Step 2: create placeholders for input X (number of fire) and label Y (number of
theft)
X = tf.placeholder(tf.float32, name="X")
Y = tf.placeholder(tf.float32, name="Y")

# Step 3: create weight and bias, initialized to 0
w = tf.Variable(0.0, name="weights")
b = tf.Variable(0.0, name="bias")

# Step 4: construct model to predict Y (number of theft) from the number of fire
Y_predicted = X * w + b

# Step 5: use the square error as the loss function
loss = tf.square(Y - Y_predicted, name="loss")

# Step 6: using gradient descent with learning rate of 0.01 to minimize loss
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.001).minimize(loss)
```

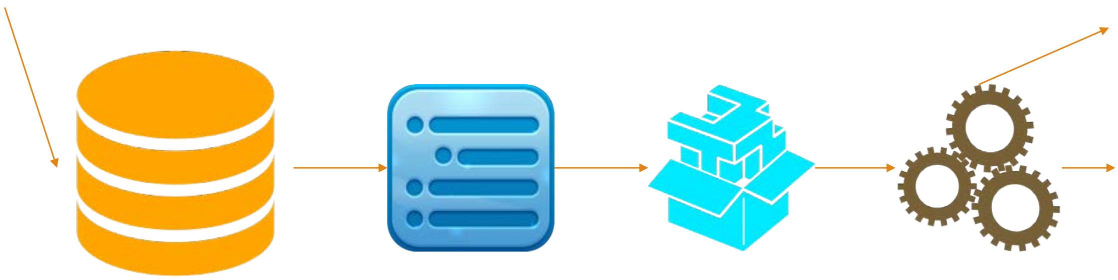
```
with tf.Session() as sess:
        # Step 7: initialize the necessary variables, in this case, w and b
        sess.run(tf.global_variables_initializer())

        # Step 8: train the model
        for i in range(100): # run 100 epochs
                for x, y in data:
                        # Session runs train_op to minimize loss
                        sess.run(optimizer, feed_dict={X: x, Y:y})

        # Step 9: output the values of w and b
        w_value, b_value = sess.run([w, b])
```



TensorFlow Example 1

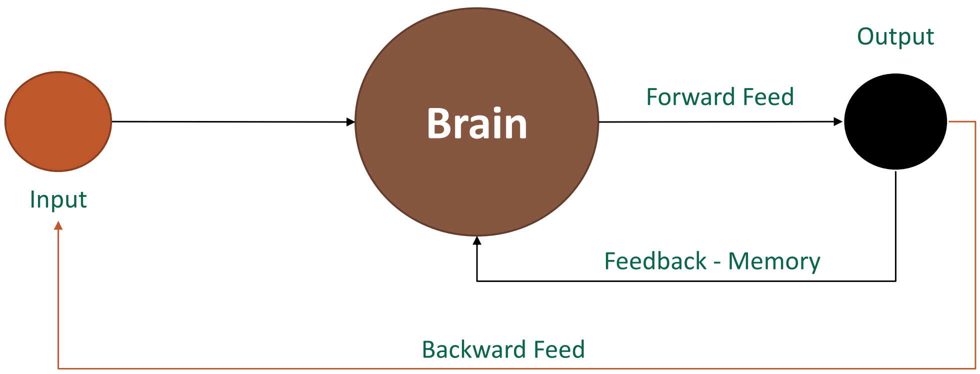
Recall: Machine Learning

- Type of artificial intelligence (AI) that provides computers with the ability to learn without being explicitly programmed.



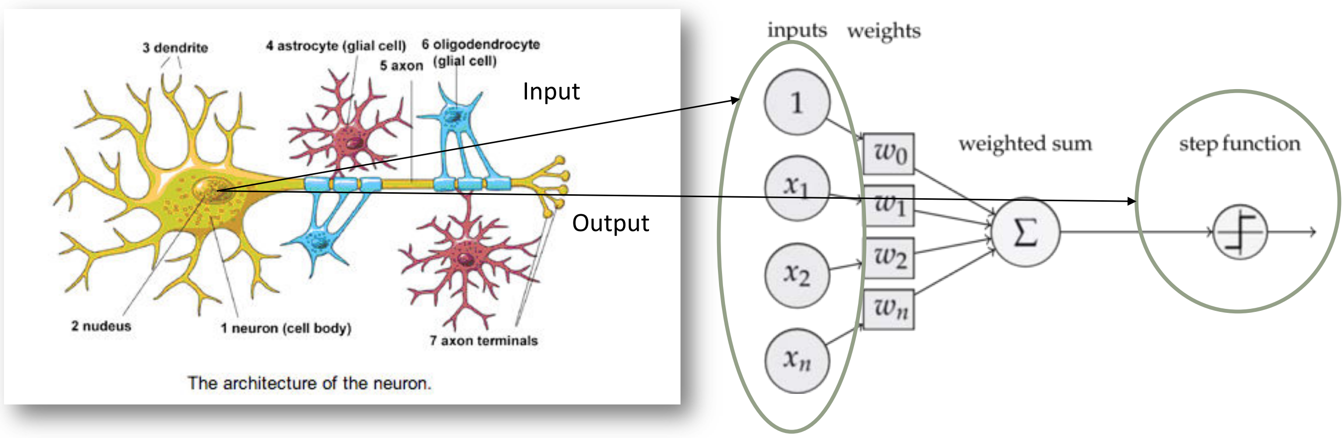
Recall: Artificial Neural Network

Basic Human Nervous System Diagram



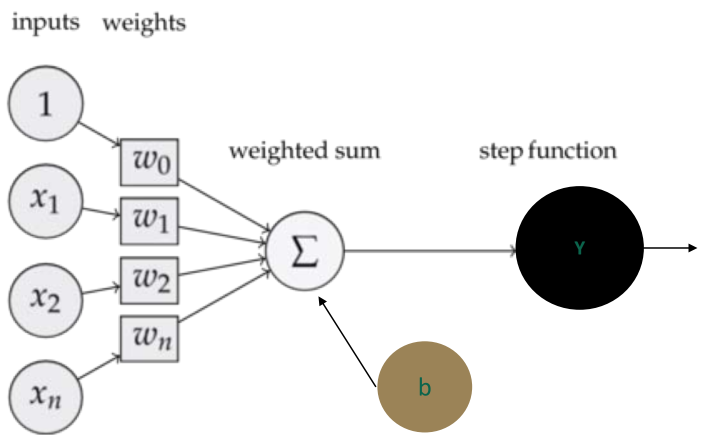
Artificial Neural Network

- Perceptron



NNN Model: Feed Forward

Y_pred = Y (Wx * b)

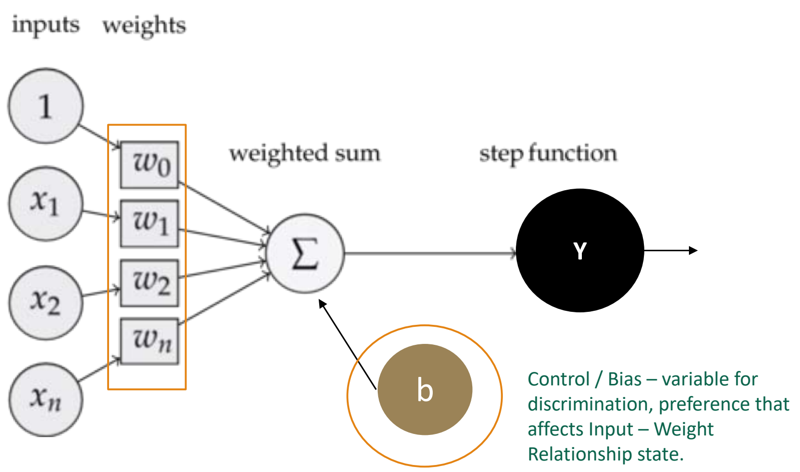


NN Model: Feed Forward

Y_pred = Y (Wx * b)

Variables *are state of nodes which output their current value which is retained across multiple execution.*

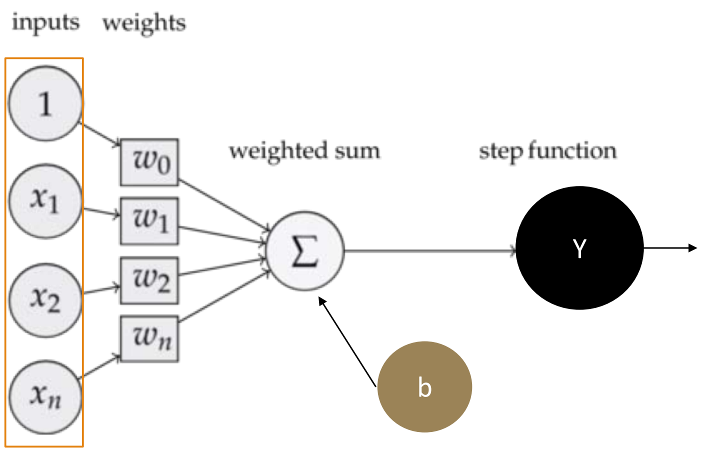
- Gradient Descent, Regression and etc.



NN Model: Feed Forward

Y_pred = Y (Wx * b)

Placeholders are nodes where its value is fed in at execution time.



NN Model: Feed Forward

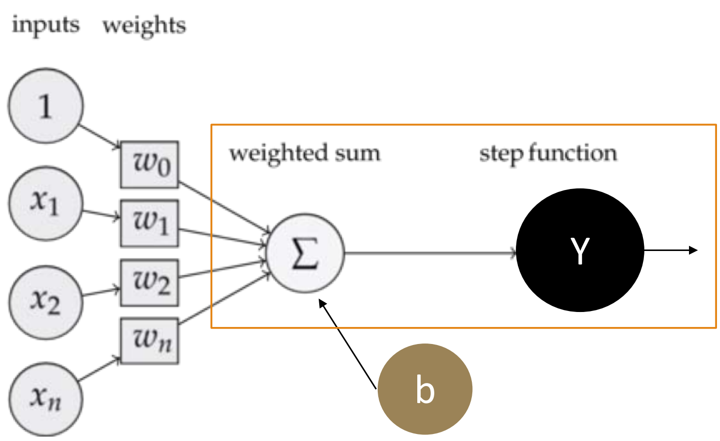
Y_pred = Y (Wx * b)

Mathematical Operation

W(x) *= Multiply Two Matrix or a Weighted Input*

Σ (Add) *= Summation elementwise with broadcasting*

Y *= Step Function with elementwise rectified linear function*



TensorFlow Basic Flow

- Build a graph
  - Graph contains parameter specifications, model architecture, optimization process
- Optimize Predictions, Loss Functions and Learning
- Initialize a session
- Fetch and feed data with Session.run
  - Compilation, optimization, visualization

Back to Our Example…

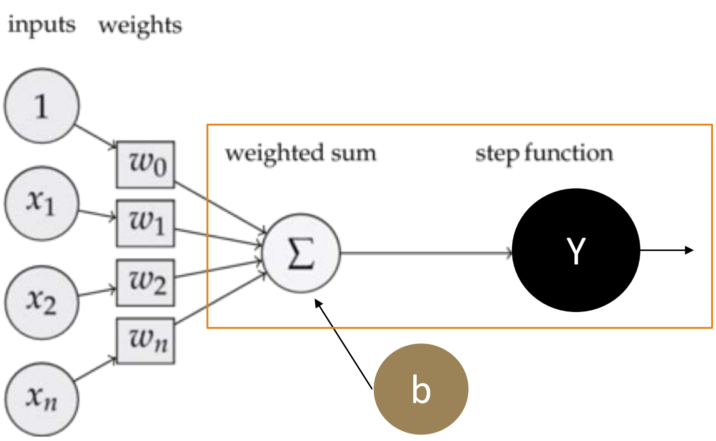
Y_pred = Y (Wx * b)

Mathematical Operation

W(x) *= Multiply Two Matrix or a Weighted Input*

Σ (Add) *= Summation elementwise with broadcasting*

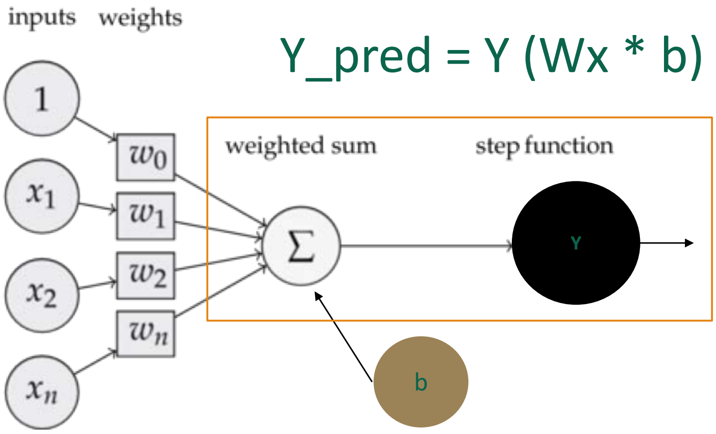
Y *= Step Function with elementwise rectified linear function*



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

# %% Let's create some toy data
plt.ion()
n_observations = 100
fig, ax = plt.subplots(1, 1)
xs = np.linspace(-3, 3, n_observations)
ys = np.sin(xs) + np.random.uniform(-0.5, 0.5, n_observations)
ax.scatter(xs, ys)
fig.show()
plt.draw()

# %% tf.placeholders for the input and output of the network.
# Placeholders are variables which we need to fill in when we
# are ready to compute the graph.
X = tf.placeholder(tf.float32)
Y = tf.placeholder(tf.float32)

# %% We will try to optimize min_(W,b) ||(X*w + b) - y||^2
# The `Variable()` constructor requires an initial value for the
# variable,, which can be a `Tensor` of any type and shape. The
# initial value defines the type and shape of the variable.
# After construction, the type and shape of # the variable are
#fixed. The value can be changed using one of the assign methods.
W = tf.Variable(tf.random_normal([1]), name='weight')
b = tf.Variable(tf.random_normal([1]), name='bias')
Y_pred = tf.add(tf.mul(X, W), b)
```

Implementation of Graph, Plot / Planes, Variables

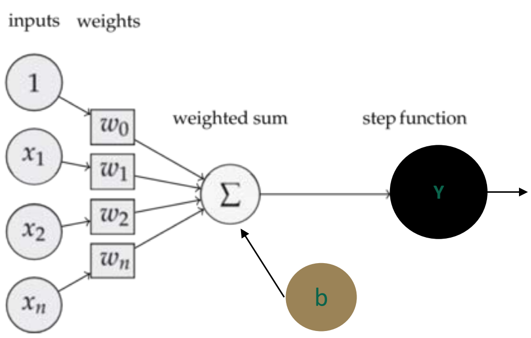
Y_pred = Y (Wx * b)



Codify – Rendering Graph

Y_pred = Y (Wx * b)

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



Codify - Optimization

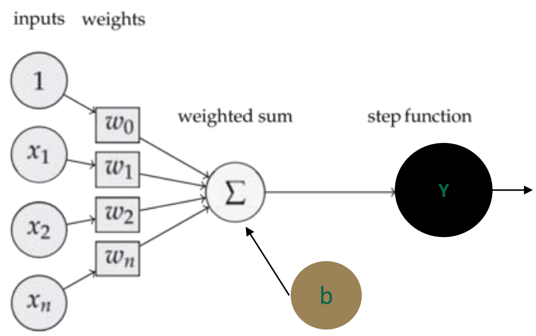
Optimizing Predictions

```
# %% Loss function will measure the distance between our observations
# and predictions and average over them.
cost = tf.reduce_sum(tf.pow(Y_pred - Y, 2)) / (n_observations - 1)
```

Optimizing Learning Rate

```
# %% Use gradient descent to optimize W,b
# Performs a single step in the negative gradient
learning_rate = 0.01
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
```

Y_pred = Y (Wx * b)

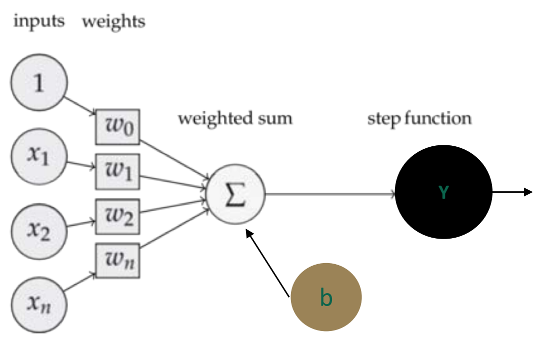



```
# %% We create a session to use the graph
n_epochs = 1000
with tf.Session() as sess:
    # Here we tell tensorflow that we want to initialize all
    # the variables in the graph so we can use them
    sess.run(tf.initialize_all_variables())

    # Fit all training data
    prev_training_cost = 0.0
    for epoch_i in range(n_epochs):
        for (x, y) in zip(xs, ys):
            sess.run(optimizer, feed_dict={X: x, Y: y})

        training_cost = sess.run(
            cost, feed_dict={X: xs, Y: ys})
        print(training_cost)

        if epoch_i % 20 == 0:
            ax.plot(xs, Y_pred.eval(
                feed_dict={X: xs}, session=sess),
                    'k', alpha=epoch_i / n_epochs)
            fig.show()
            plt.draw()

        # Allow the training to quit if we've reached a minimum
        if np.abs(prev_training_cost - training_cost) < 0.000001:
            break
        prev_training_cost = training_cost
fig.show()
```

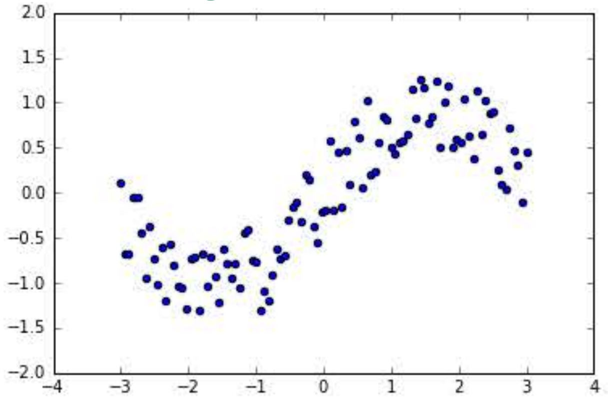
Codify - Optimization

Implementation of Session to make the model ready to be fed with data and show results

Y_pred = Y (Wx * b)

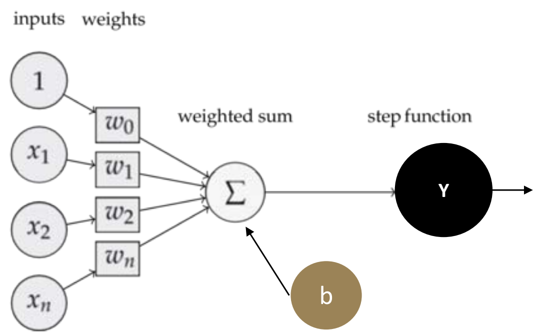


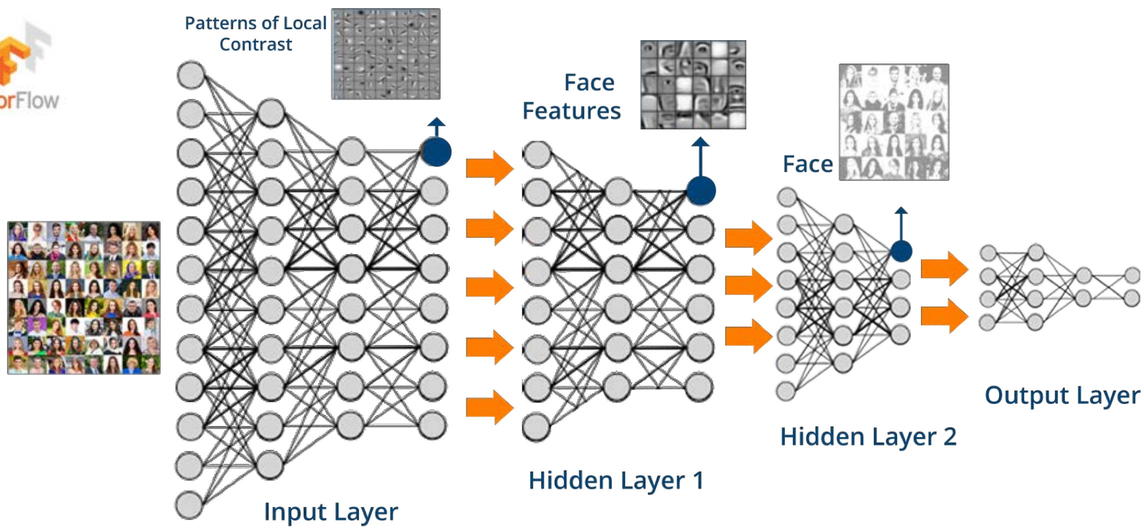
Codify - Result



Gradient Descent is used to optimize W, b which resulted to this Decision Vector Plot

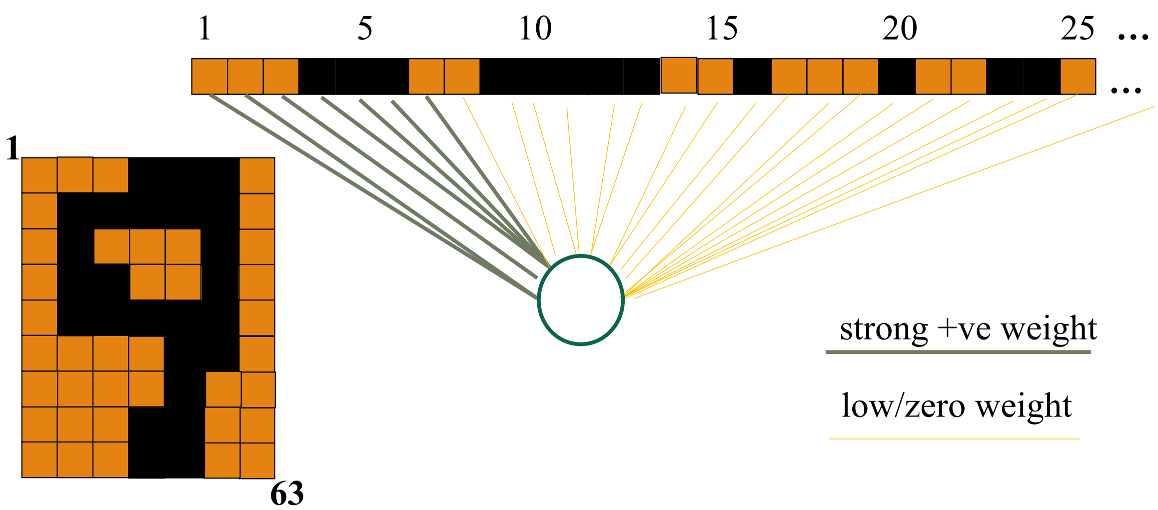
Y_pred = Y (Wx * b)





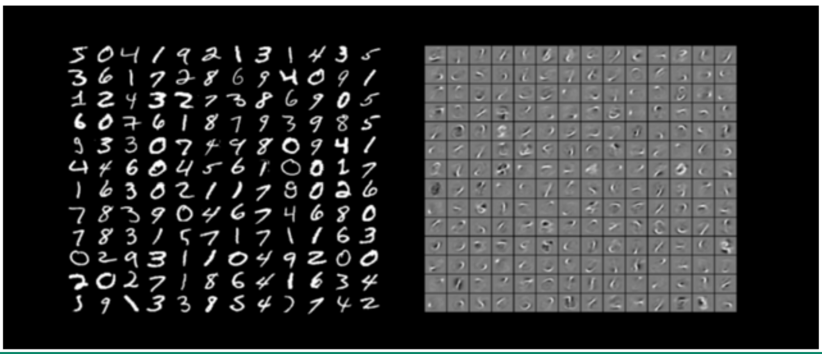
TensorFlow Example 2

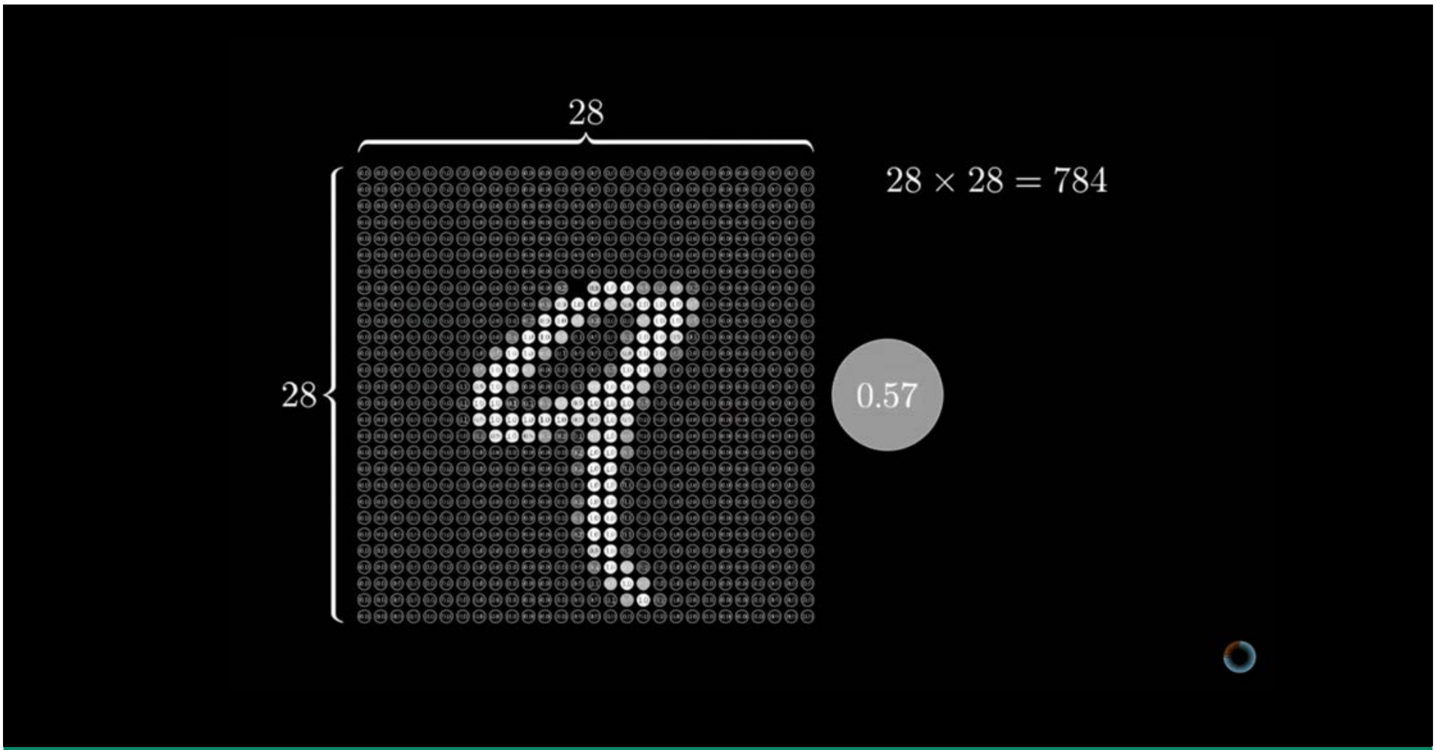
Recall: Digit Recognition



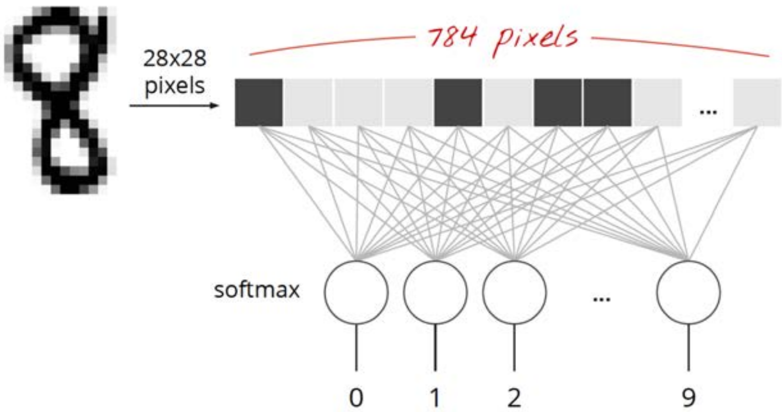
The MNIST Data Set

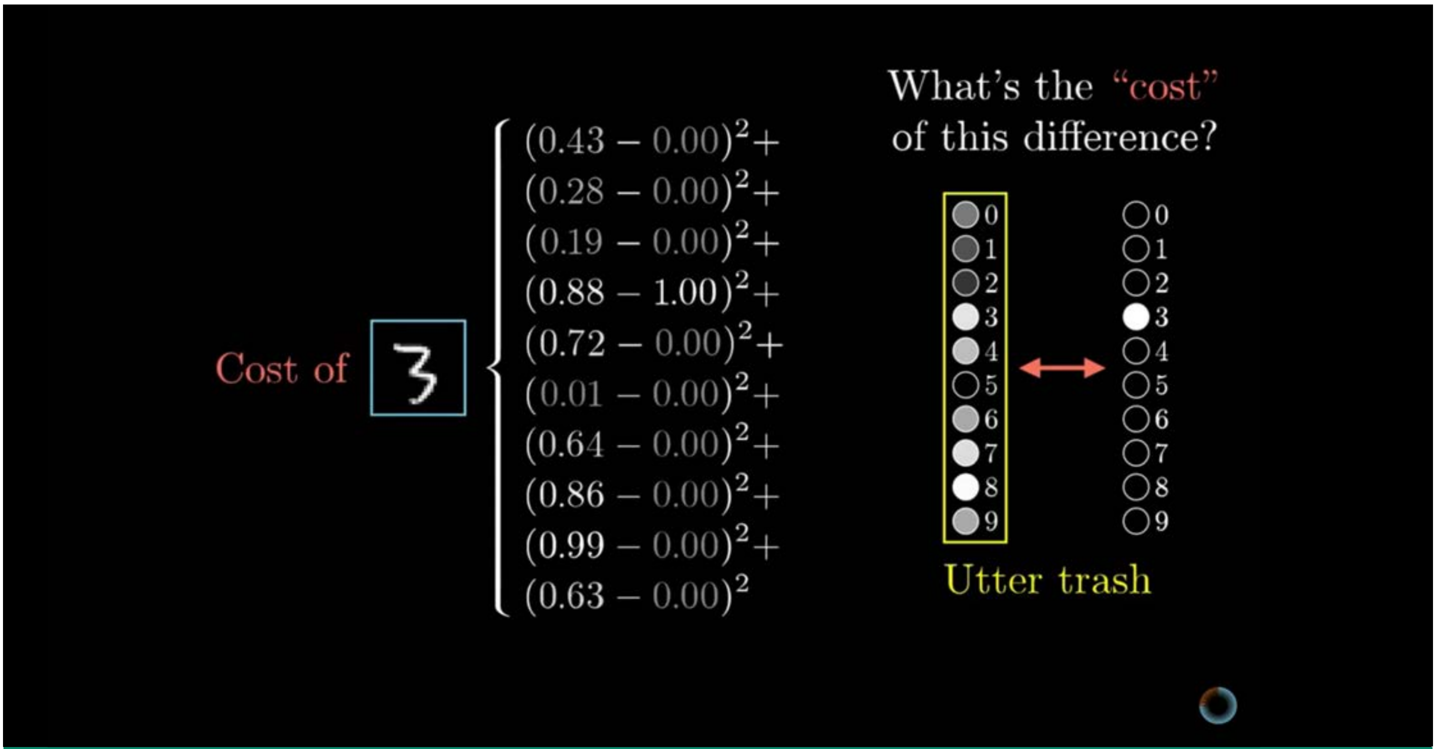
- MNIST (Mixed National Institute of Standards and Technology database) large database of handwritten digits.
- Used by almost everyone to demonstrate the power of deep learning

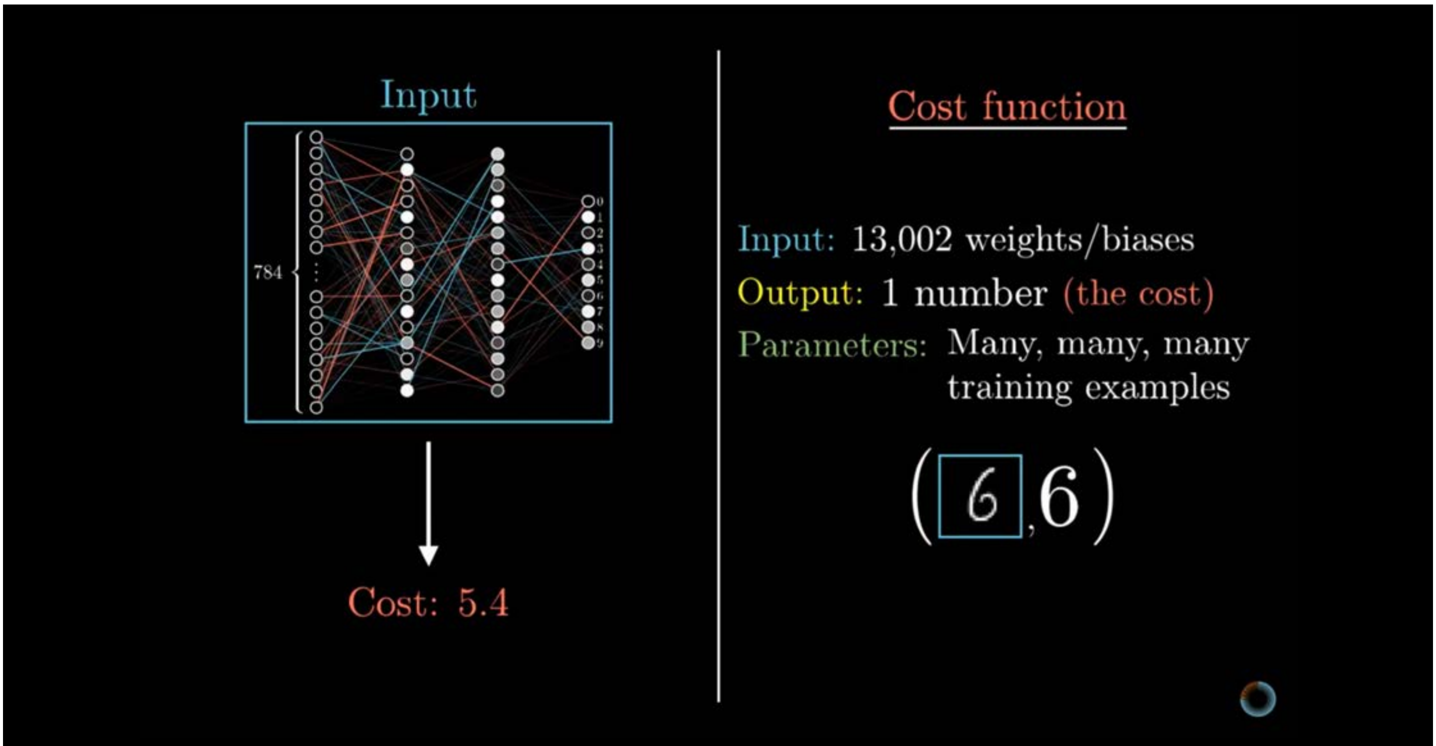


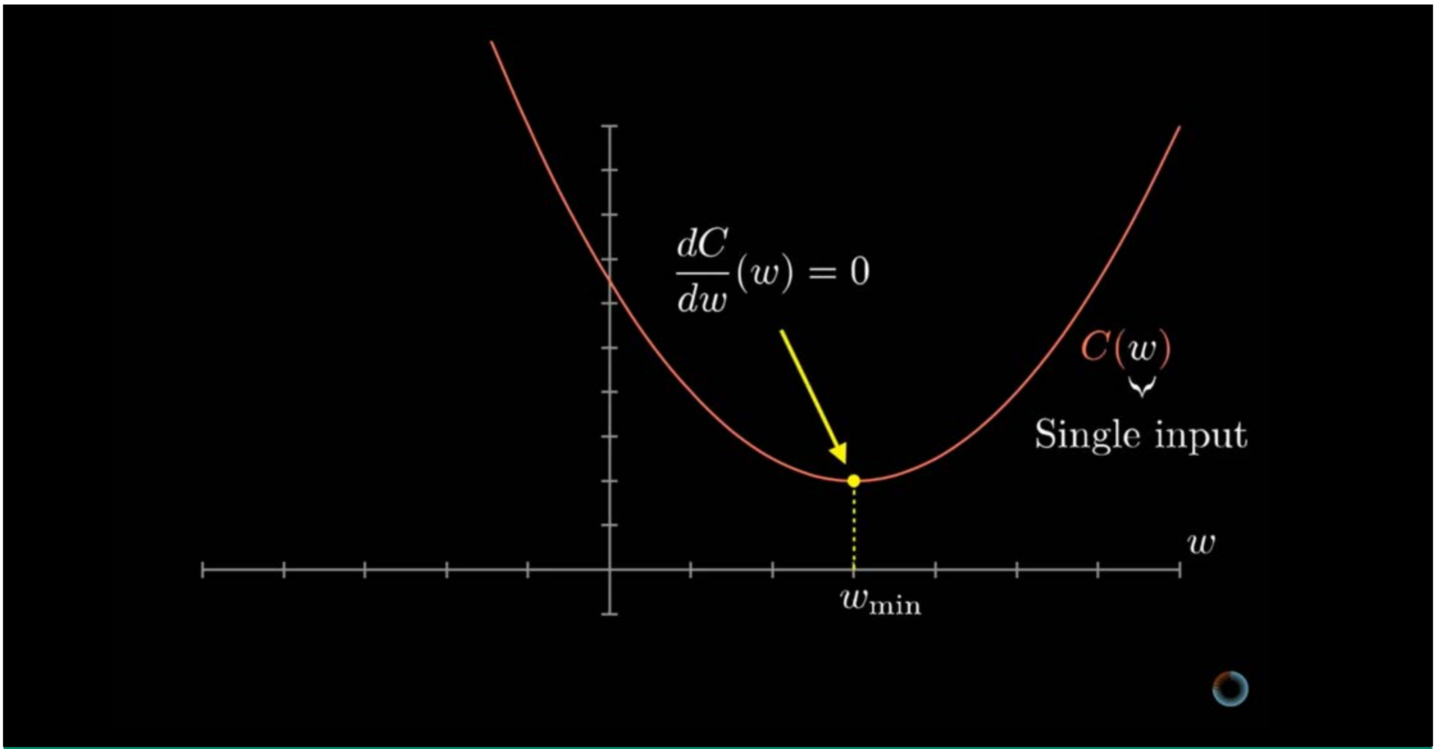


The MNIST Data Set

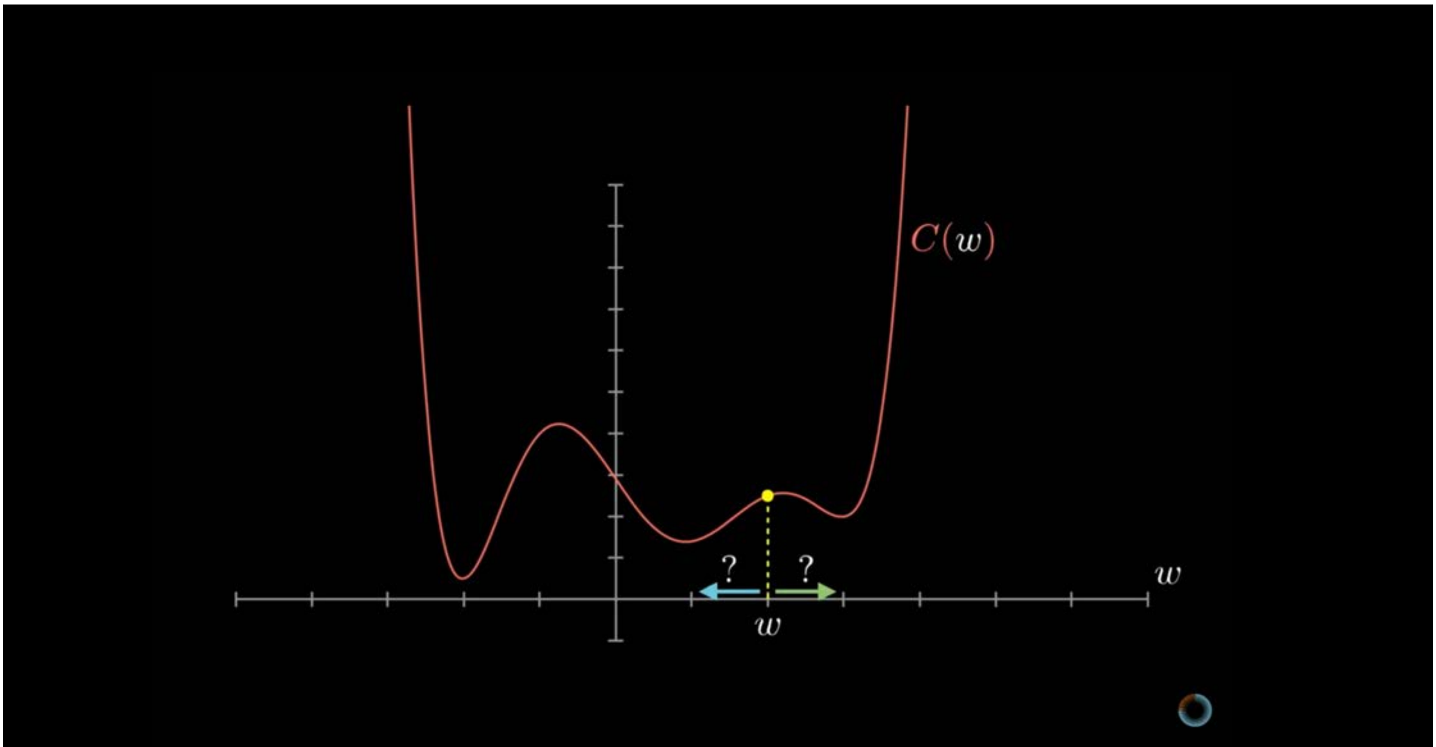



What's the "cost" of this difference?
Cost of
3
(0.43 – 0.00)²+
(0.28 – 0.00)²+
(0.19 – 0.00)²+
(0.88 – 1.00)²+
(0.72 – 0.00)²+
(0.01 – 0.00)²+
(0.64 – 0.00)²+
(0.86 – 0.00)²+
(0.99 – 0.00)²+
(0.63 – 0.00)²
0 1 2 3 4 5 6 7 8 9
0 1 2 3 4 5 6 7 8 9
Utter trash

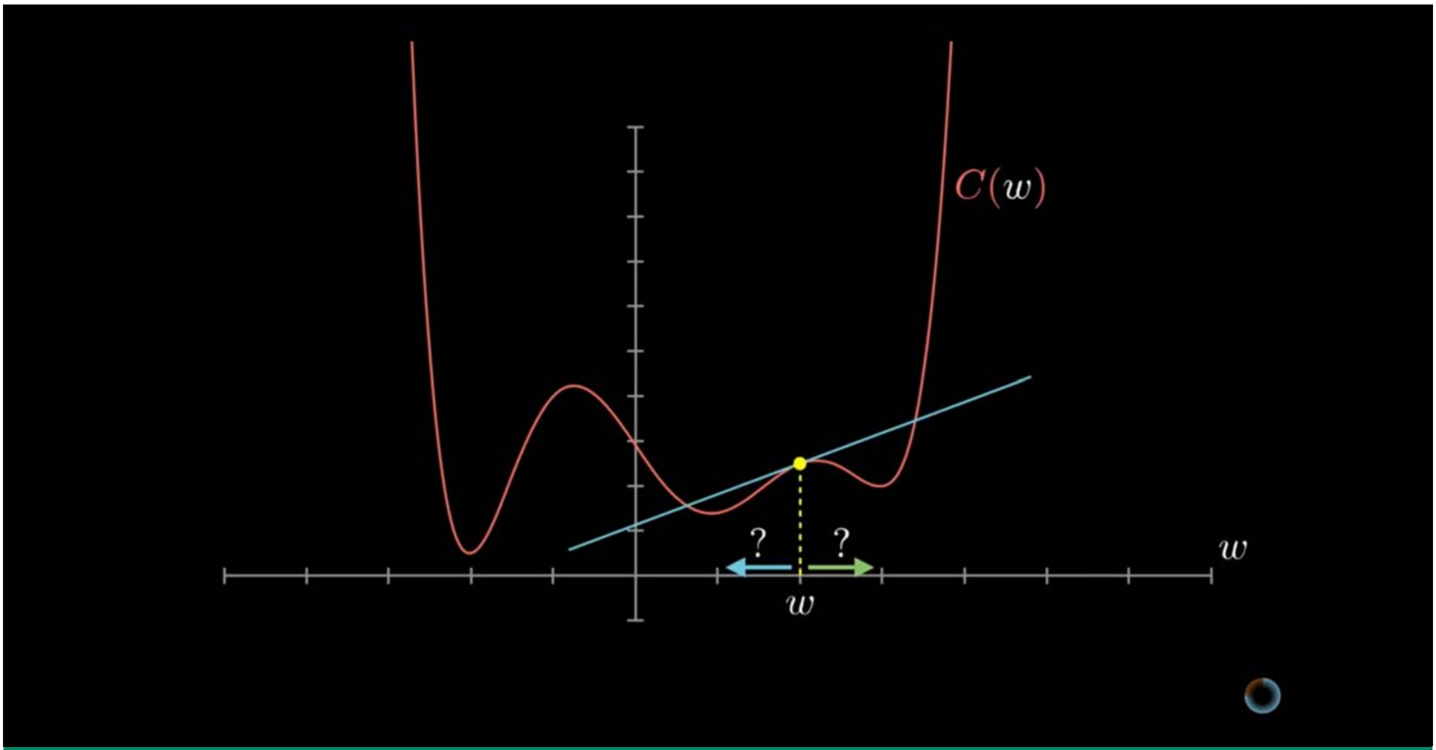

Input
784
0 1 2 3 4 5 6 7 8 9
Cost: 5.4
Cost function
Input: 13,002 weights/biases
Output: 1 number (the cost)
Parameters: Many, many, many training examples
(6, 6)

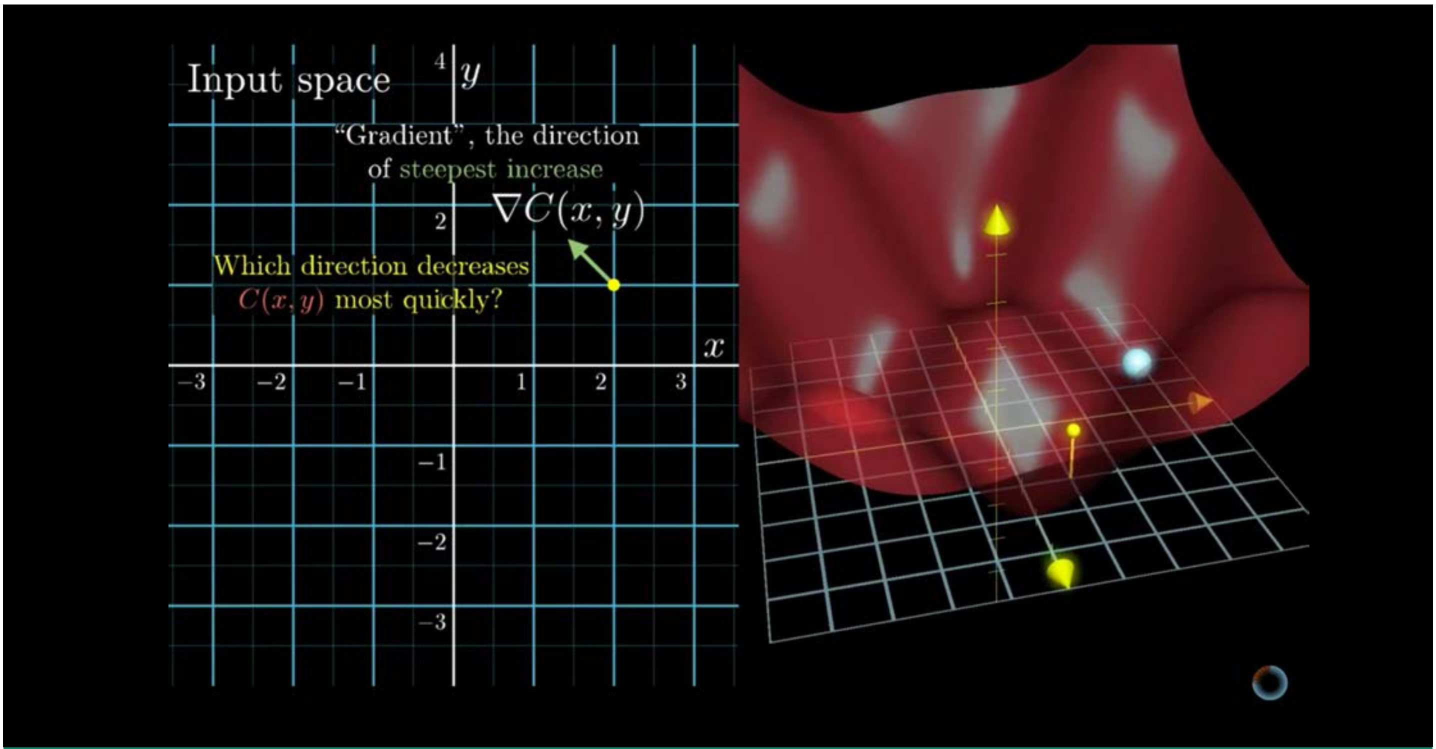


$\frac{dC}{dw}(w) = 0$
$C(w)$
Single input
w
w_{min}

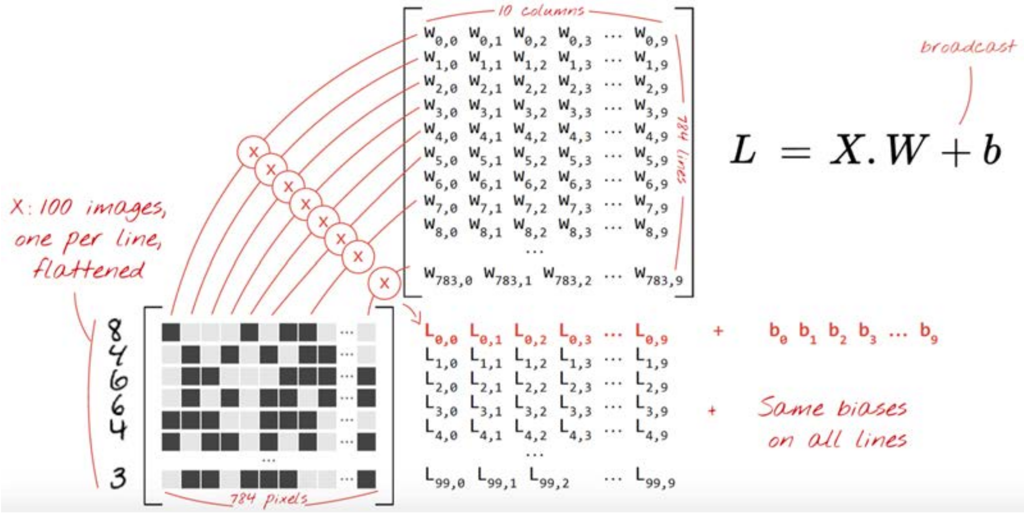


$C(w)$
?
?
w
w


C(w)
?
?
w
w


Input space
"Gradient", the direction
of steepest increase
∇C(x, y)
Which direction decreases
C(x, y) most quickly?
x
y

Matrix Notation

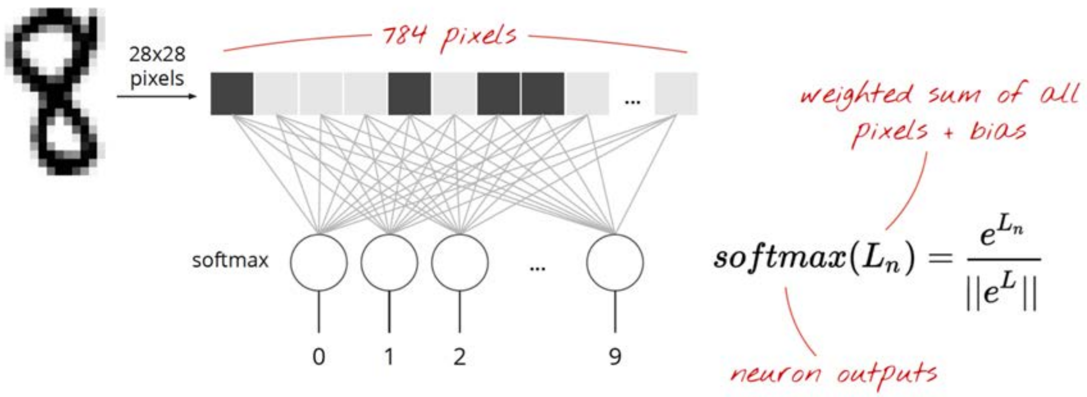


Softmax Function

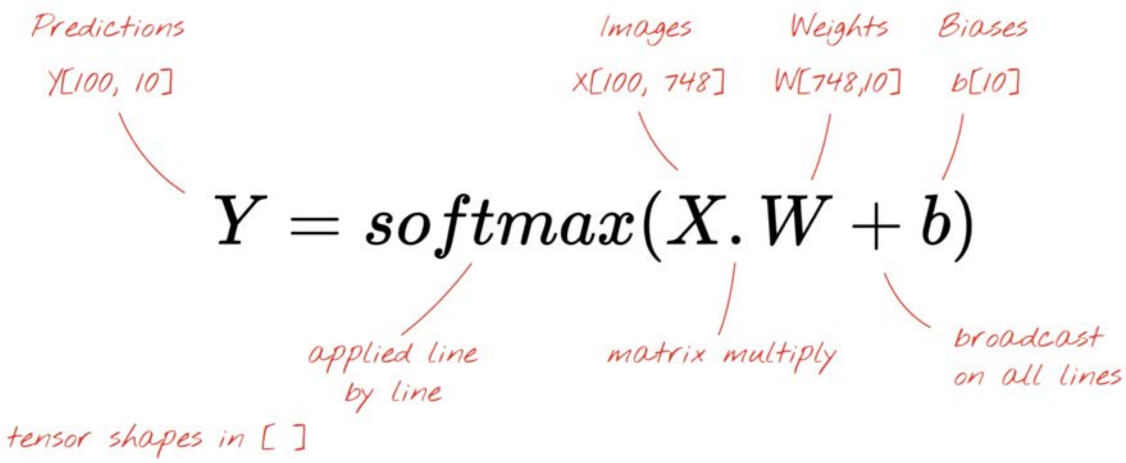
- The softmax function or the normalized exponential function is a generalization of the logistic function that "squashes" a K-dimensional vector *Z* of arbitrary real values to a K-dimensional vector of real values in the range [0, 1] that add up to 1.
- The function is given by

$$\sigma(\mathbf{z})_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \quad \text{for } j = 1, \ldots, K.$$

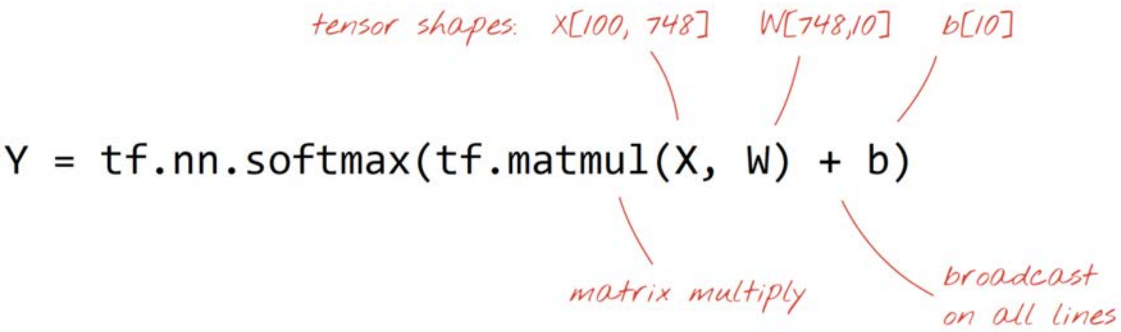
Softmax Simple Model



Softmax Simple Model



In TensorFlow



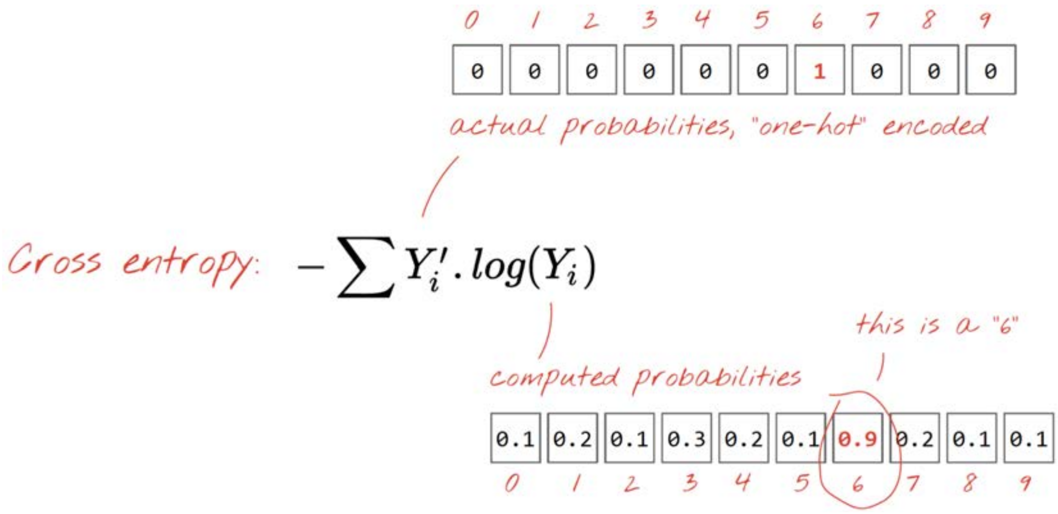
Check for Success

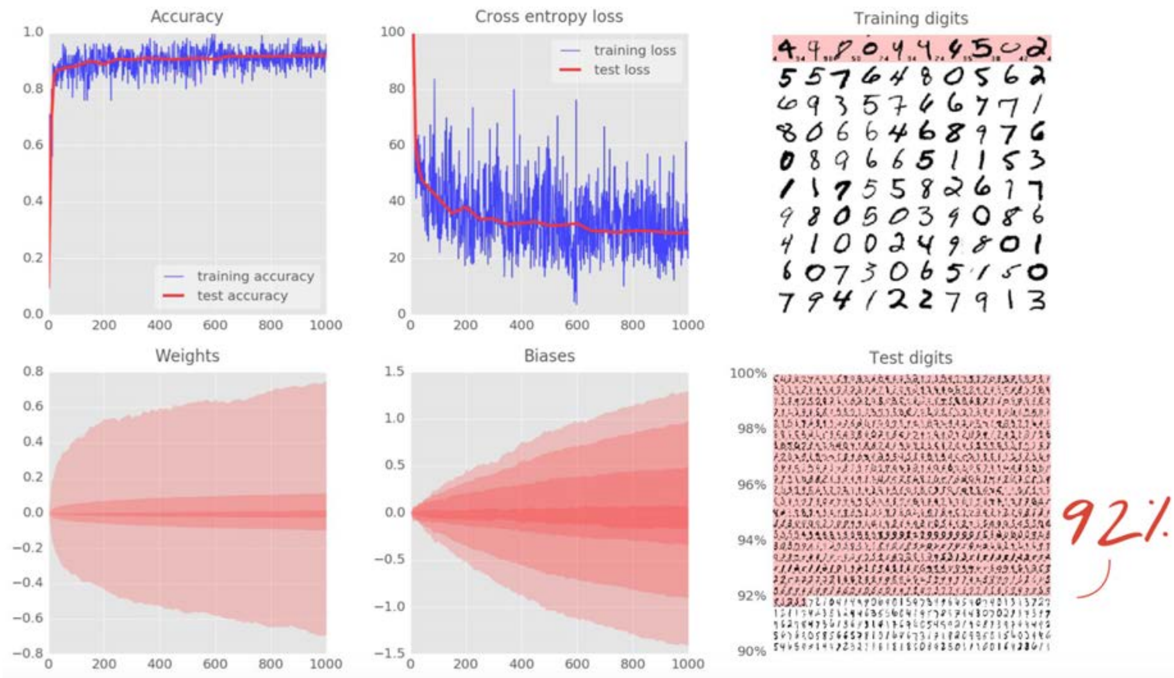
- Need to include a cost or loss function for the optimization/backpropagation to work on
- Use the cross entropy cost function, represented by:

$$J = -\frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{n} y_j^{(i)} log(y_{j\_}^{(i)}) + (1-y_j^{(i)}) log(1-y_{j\_}^{(i)})$$

Where $y_j^{(i)}$ is the ith training label for output node *j*, $y_{j\_}^{(i)}$ is the ith predicted label for output node *j*, *m* is the number of training / batch samples and *n* is the number . There are two operations occurring in the above equation. The first is the summation of the logarithmic products and additions *across all the output nodes*. The second is taking a mean of this summation *across all the training samples*

Check for Success





Initialization

```
import tensorflow as tf

X = tf.placeholder(tf.float32, [None, 28, 28, 1])
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))

init = tf.initialize_all_variables()
```

this will become the batch size, 100

28 x 28 grayscale images

Training = computing variables W and b

Compute and Check for Success

flattening images

```
# model
Y = tf.nn.softmax(tf.matmul(tf.reshape(X, [-1, 784]), W) + b)
# placeholder for correct answers
Y_ = tf.placeholder(tf.float32, [None, 10])

# loss function
cross_entropy = -tf.reduce_sum(Y_ * tf.log(Y))

# % of correct answers found in batch
is_correct = tf.equal(tf.argmax(Y,1), tf.argmax(Y_,1))
accuracy = tf.reduce_mean(tf.cast(is_correct, tf.float32))
```

"one-hot" encoded

"one-hot" decoding

TensorFlow: Training

learning rate

```
optimizer = tf.train.GradientDescentOptimizer(0.003)
train_step = optimizer.minimize(cross_entropy)
```

loss function

TensorFlow: Run

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

for i in range(1000):
    # Load batch of images and correct answers
    batch_X, batch_Y = mnist.train.next_batch(100)
    train_data={X: batch_X, Y_: batch_Y}

    # train
    sess.run(train_step, feed_dict=train_data)

    # success ?
    a,c = sess.run([accuracy, cross_entropy], feed_dict=train_data)

    # success on test data ?
    test_data={X: mnist.test.images, Y_: mnist.test.labels}
    a,c = sess.run([accuracy, cross_entropy, It], feed=test_data)
```

running a Tensorflow computation, feeding placeholders

Tip: do this every 100 iterations

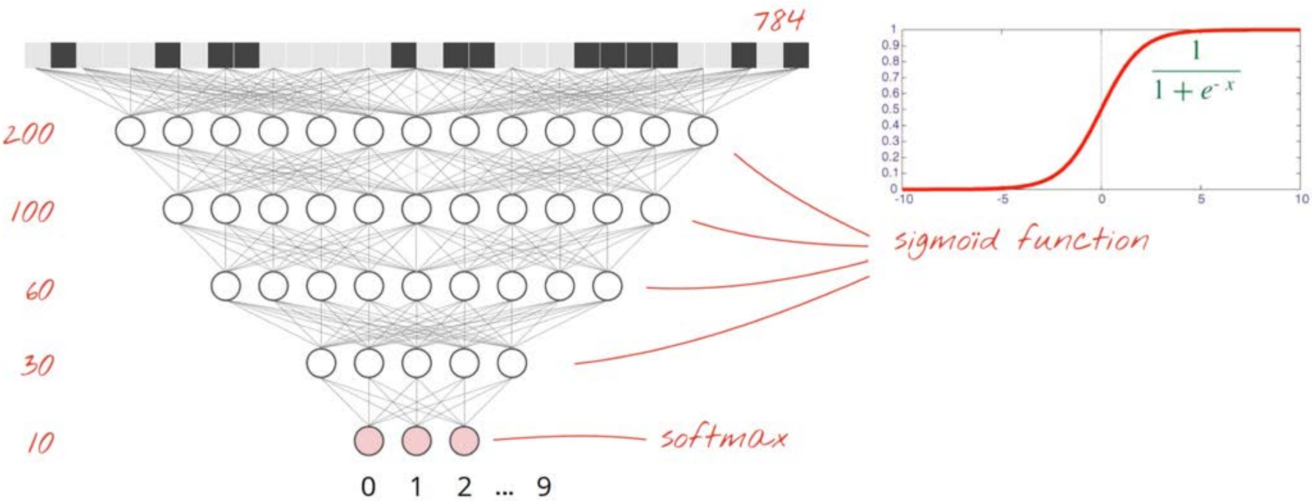
TensorFlow: Full Code

initialisation · model · success metrics · training step · Run

```
import tensorflow as tf

X = tf.placeholder(tf.float32, [None, 28, 28, 1])
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
init = tf.initialize_all_variables()

# model
Y=tf.nn.softmax(tf.matmul(tf.reshape(X,[-1, 784]), W) + b)

# placeholder for correct answers
Y_ = tf.placeholder(tf.float32, [None, 10])

# loss function
cross_entropy = -tf.reduce_sum(Y_ * tf.log(Y))

# % of correct answers found in batch
is_correct = tf.equal(tf.argmax(Y,1), tf.argmax(Y_,1))
accuracy = tf.reduce_mean(tf.cast(is_correct,tf.float32))
```

```
optimizer = tf.train.GradientDescentOptimizer(0.003)
train_step = optimizer.minimize(cross_entropy)

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

for i in range(10000):
    # load batch of images and correct answers
    batch_X, batch_Y = mnist.train.next_batch(100)
    train_data={X: batch_X, Y_: batch_Y}

    # train
    sess.run(train_step, feed_dict=train_data)

    # success ? add code to print it
    a,c = sess.run([accuracy, cross_entropy], feed=train_data)

    # success on test data ?
    test_data={X:mnist.test.images, Y_:mnist.test.labels}
    a,c = sess.run([accuracy, cross_entropy], feed=test_data)
```


Go Deep: Redo with 5 Layers



TensorFlow: Initialisation

```
K = 200
L = 100
M = 60
N = 30

W1 = tf.Variable(tf.truncated_normal([28*28, K] ,stddev=0.1))
B1 = tf.Variable(tf.zeros([K])
W2 = tf.Variable(tf.truncated_normal([K, L], stddev=0.1))
B2 = tf.Variable(tf.zeros([L])
W3 = tf.Variable(tf.truncated_normal([L, M], stddev=0.1))
B3 = tf.Variable(tf.zeros([M])
W4 = tf.Variable(tf.truncated_normal([M, N], stddev=0.1))
B4 = tf.Variable(tf.zeros([N])
W5 = tf.Variable(tf.truncated_normal([N, 10], stddev=0.1))
B5 = tf.Variable(tf.zeros([10]))
```

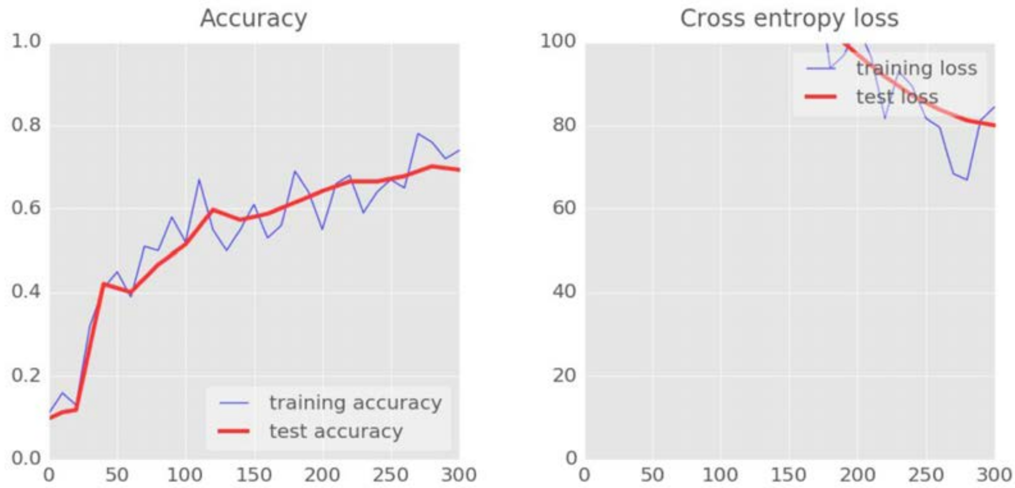
weights initialised with random values

TensorFlow: The model

weights and biases

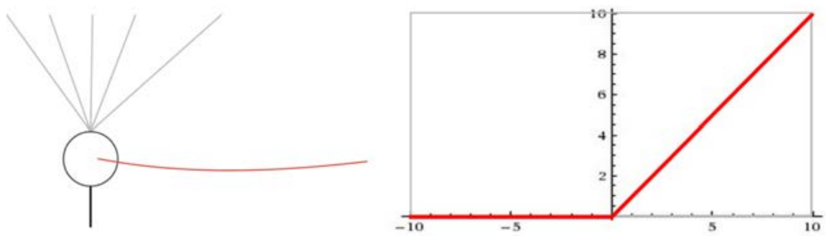
```
X = tf.reshape(X, [-1, 28*28])

Y1 = tf.nn.sigmoid(tf.matmul(X, W1) + B1)
Y2 = tf.nn.sigmoid(tf.matmul(Y1, W2) + B2)
Y3 = tf.nn.sigmoid(tf.matmul(Y2, W3) + B3)
Y4 = tf.nn.sigmoid(tf.matmul(Y3, W4) + B4)
Y = tf.nn.softmax(tf.matmul(Y4, W5) + B5)
```

Slow Start ?



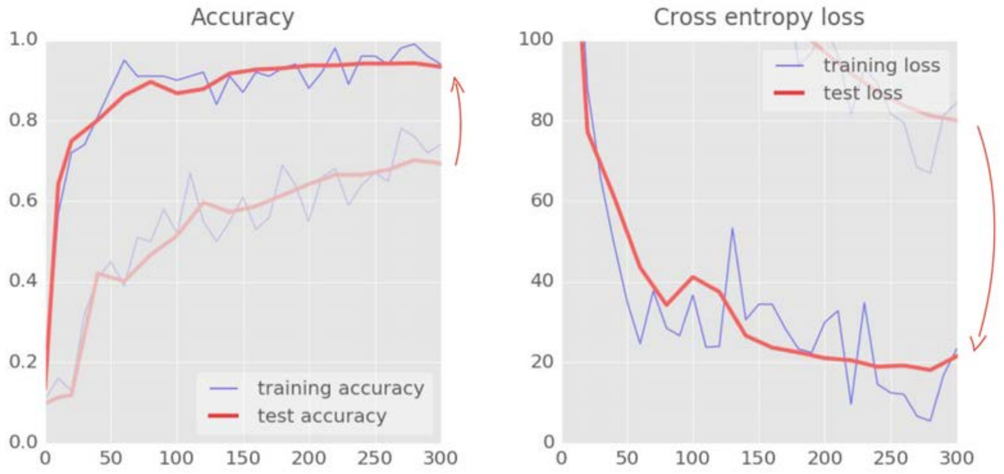
RELU

RELU = Rectified Linear Unit

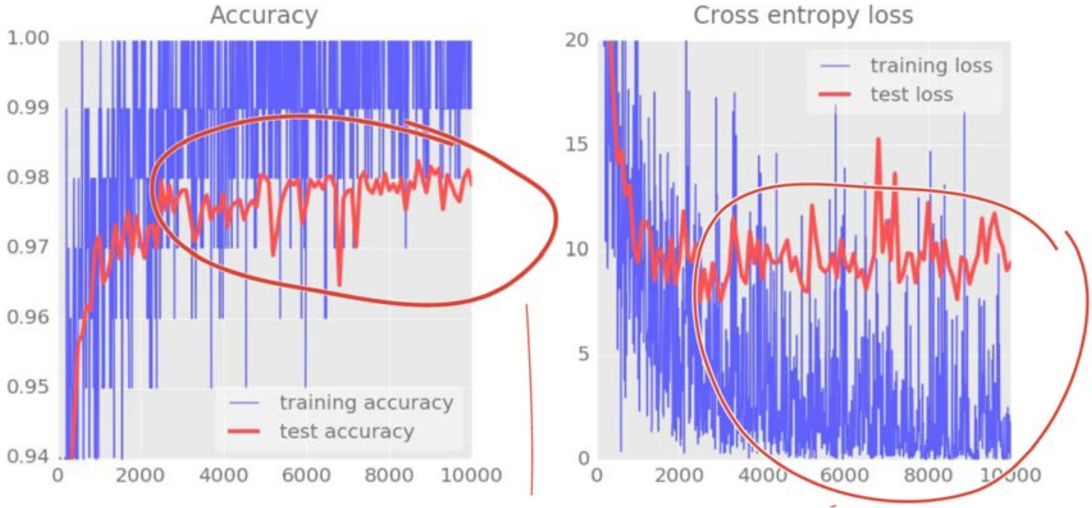



```
Y = tf.nn.relu(tf.matmul(X, W) + b)
```

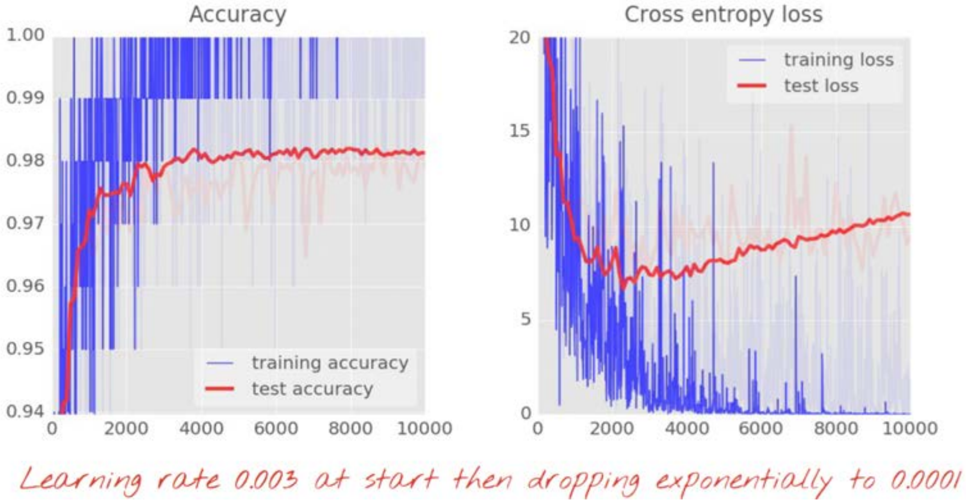
RELU



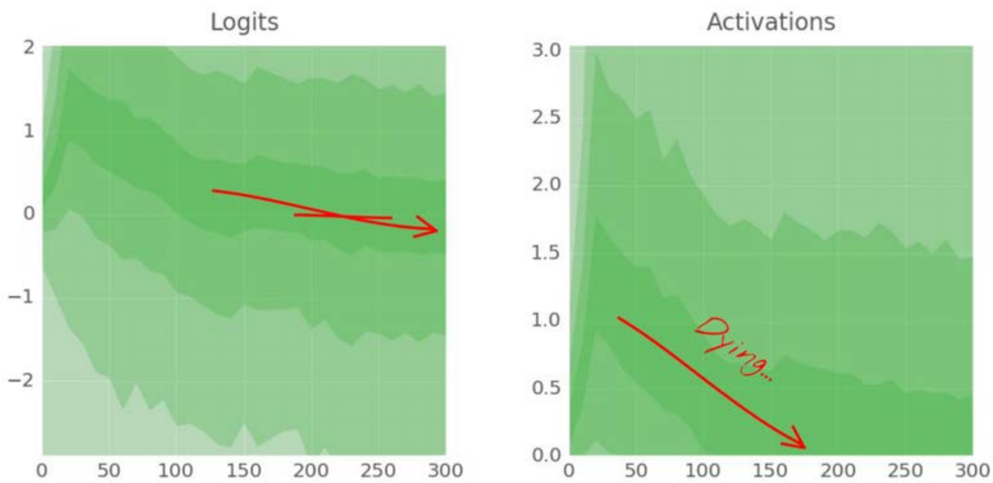
Noisy Accuracy Curve ?



Learning Rate Decay

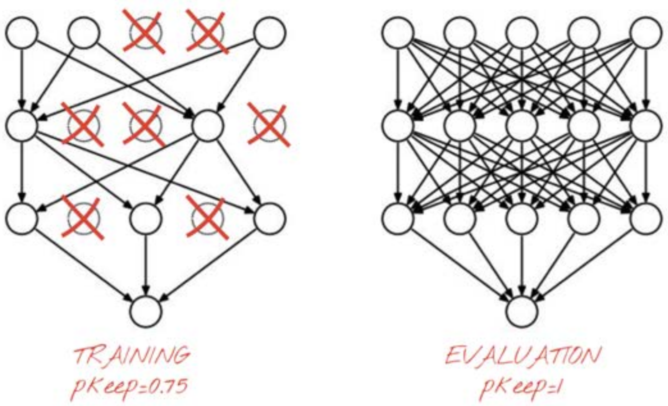


Dying Neurons

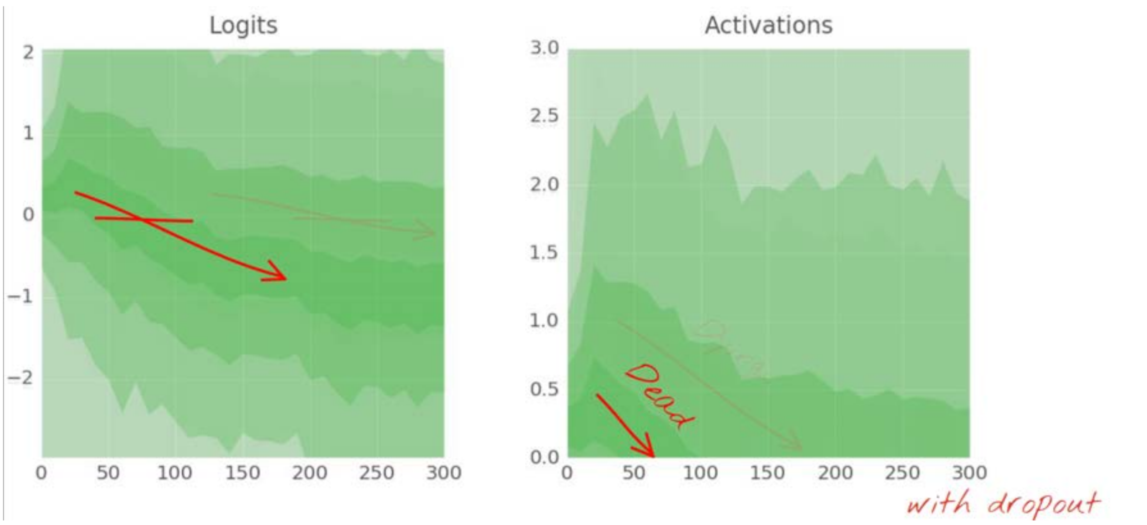


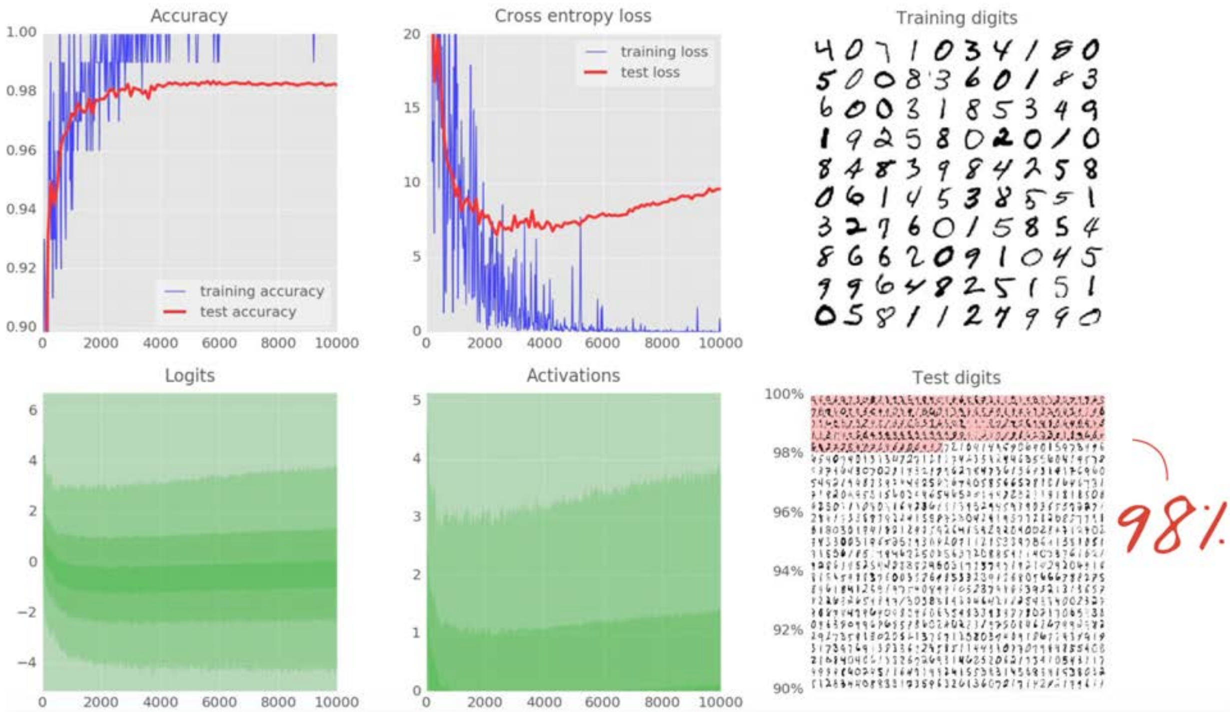
Dropout

Dropout

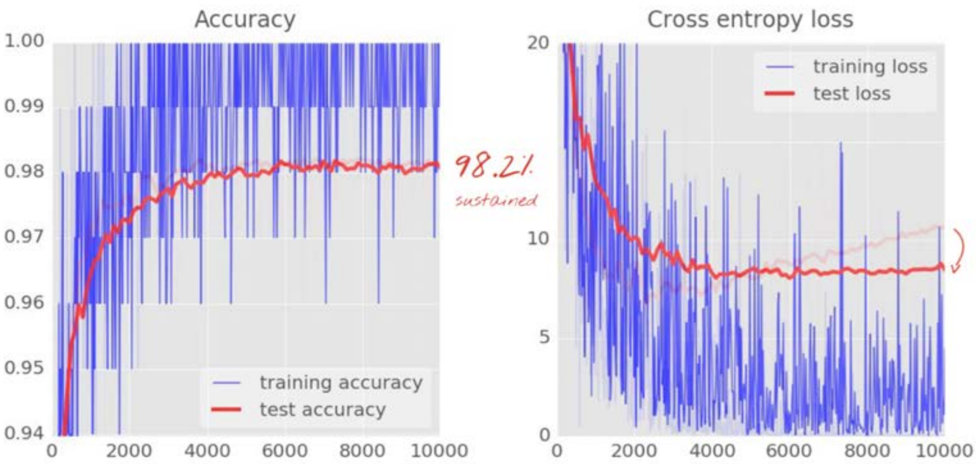



```
pkeep =
tf.placeholder(tf.float32)

Yf = tf.nn.relu(tf.matmul(X, W) + B)
Y  = tf.nn.dropout(Yf, pkeep)
```


Logits
Activations
Dead
with dropout

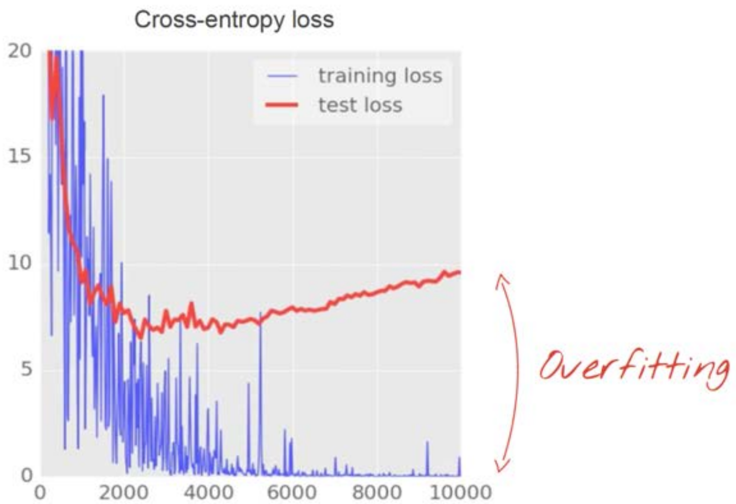

Accuracy
training accuracy
test accuracy
Cross entropy loss
training loss
test loss
Training digits
Logits
Activations
Test digits
98%

All the Party Tricks

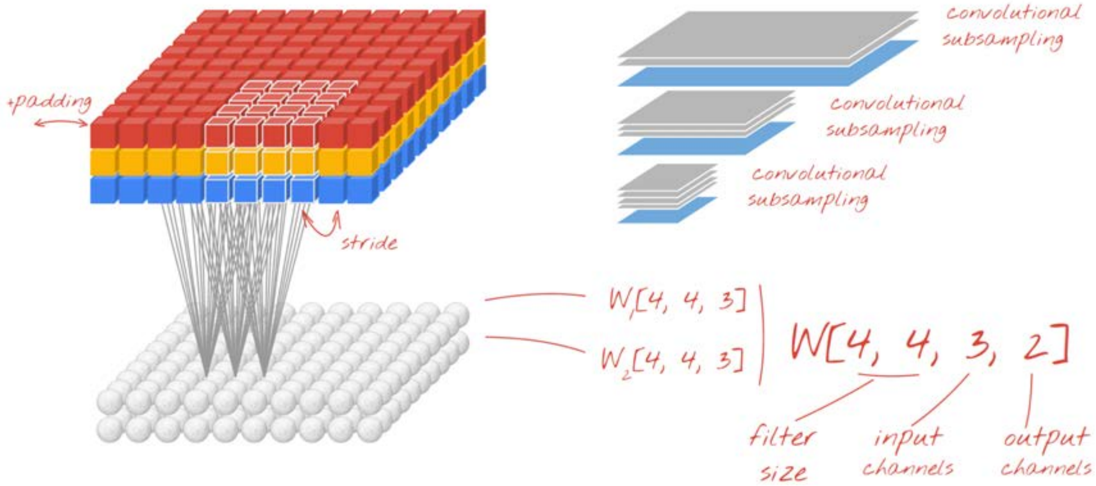


RELU, decaying learning rate 0.003 -> 0.0001 and dropout 0.75

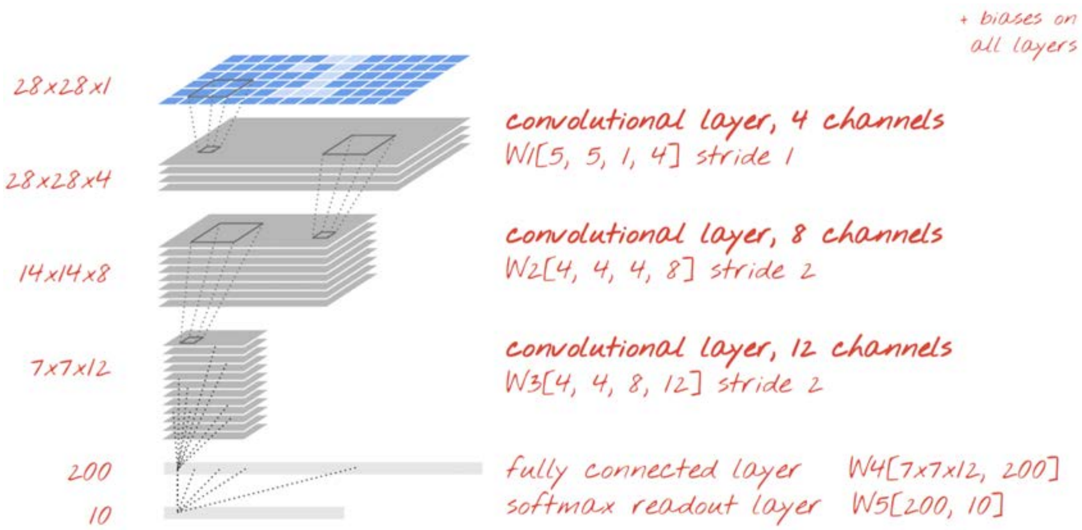
Overfitting



Convolutional Layer



Convolutional Neural Network



Tensorflow : Initialisation

filter size — input channels — output channels

```
K=4
L=8
M=12

W1 = tf.Variable(tf.truncated_normal([5, 5, 1, K] ,stddev=0.1))
B1 = tf.Variable(tf.ones([K])/10)
W2 = tf.Variable(tf.truncated_normal([5, 5, K, L] ,stddev=0.1))
B2 = tf.Variable(tf.ones([L])/10)
W3 = tf.Variable(tf.truncated_normal([4, 4, L, M] ,stddev=0.1))
B3 = tf.Variable(tf.ones([M])/10)

N=200

W4 = tf.Variable(tf.truncated_normal([7*7*M, N] ,stddev=0.1))
B4 = tf.Variable(tf.ones([N])/10)
W5 = tf.Variable(tf.truncated_normal([N, 10] ,stddev=0.1))
B5 = tf.Variable(tf.zeros([10])/10)
```

weights initialised with random values

Tensorflow: The model

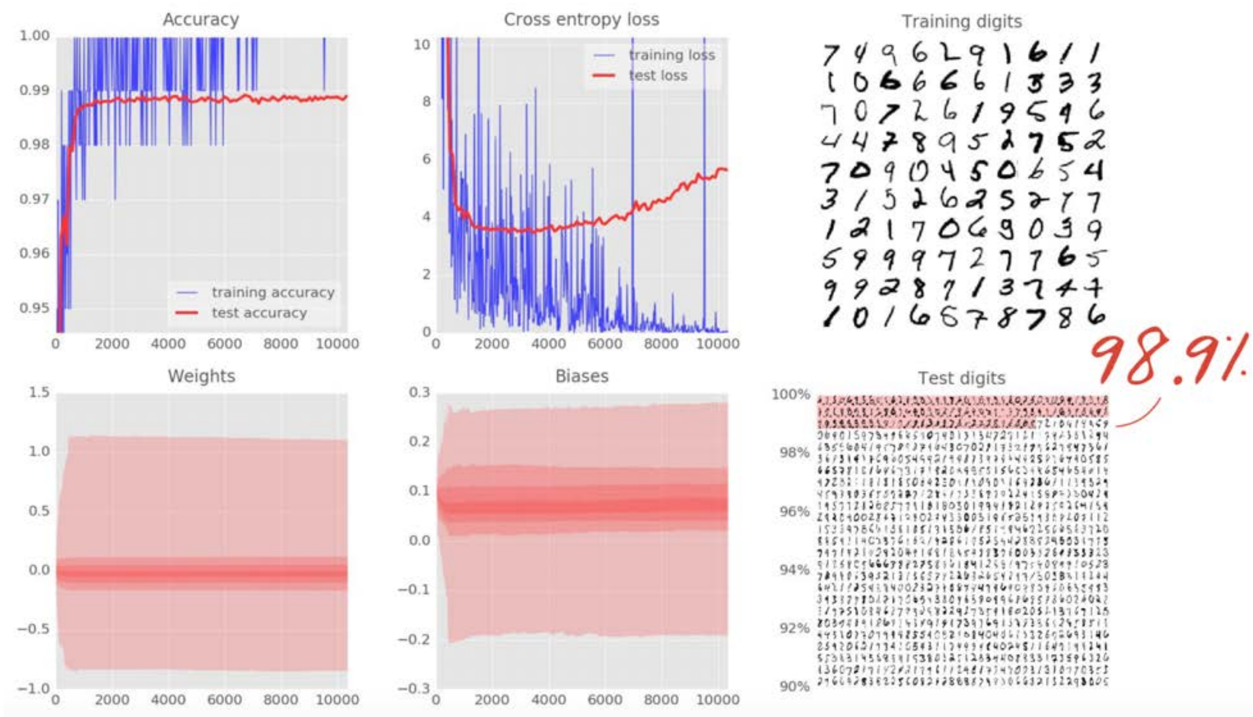
input image batch X[100, 28, 28, 1] — weights — stride — biases

```
Y1 = tf.nn.relu(tf.nn.conv2d(X, W1, strides=[1, 1, 1, 1], padding='SAME') + B1)
Y2 = tf.nn.relu(tf.nn.conv2d(Y1, W2, strides=[1, 2, 2, 1], padding='SAME') + B2)
Y3 = tf.nn.relu(tf.nn.conv2d(Y2, W3, strides=[1, 2, 2, 1], padding='SAME') + B3)

YY = tf.reshape(Y3, shape=[-1, 7 * 7 * M])

Y4 = tf.nn.relu(tf.matmul(YY, W4) + B4)
Y  = tf.nn.softmax(tf.matmul(Y4, W5) + B5)
```

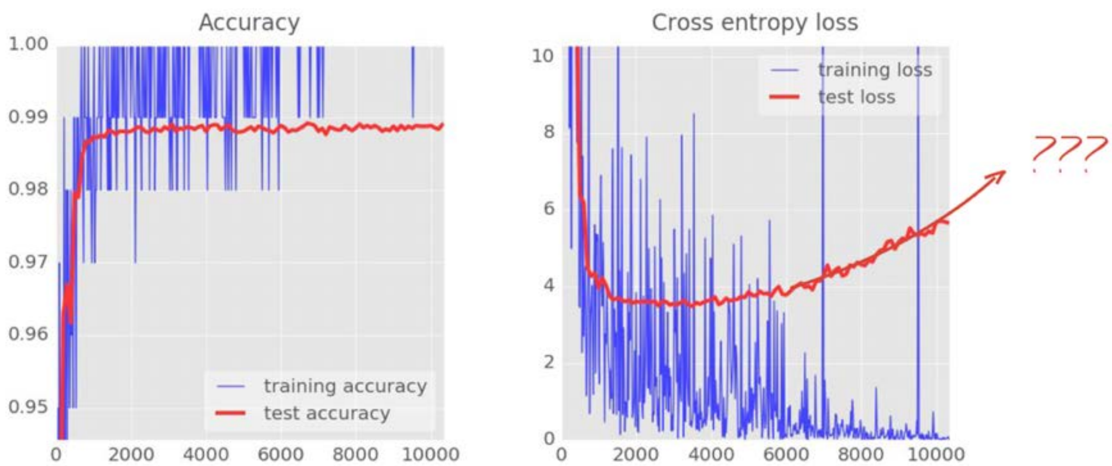
flatten all values for fully connected layer

Y3 [100, 7, 7, 12]

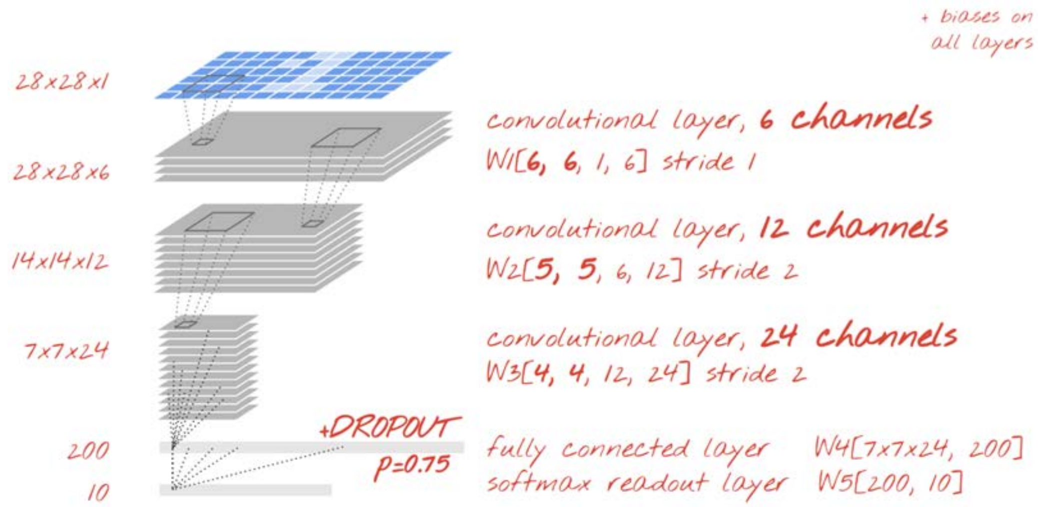
YY [100, 7x7x12]

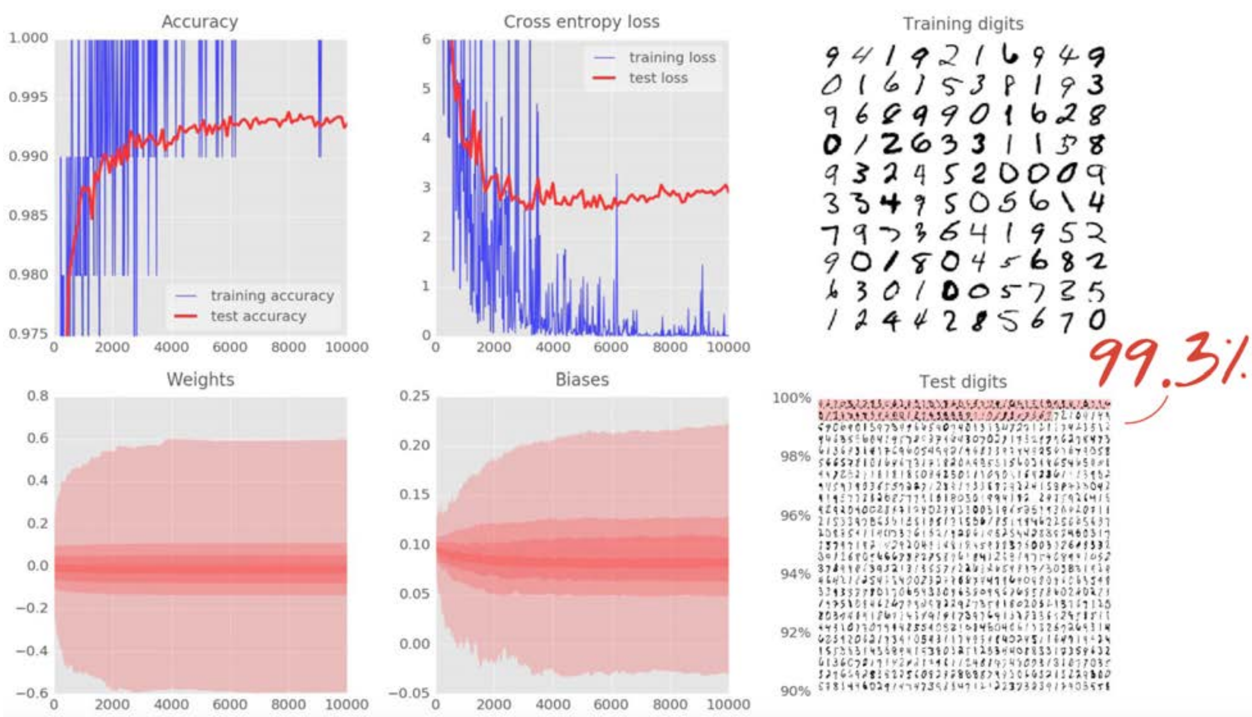


Can We do Better?

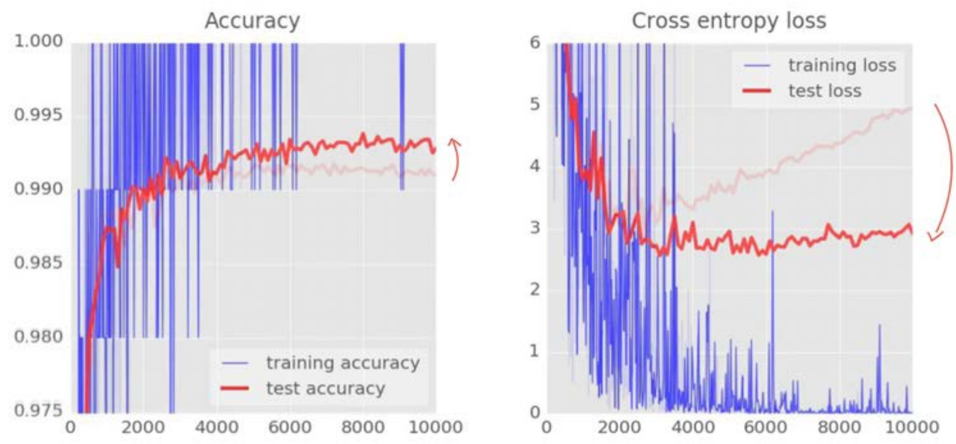


Bigger Convolutional Network + Dropout





Better!



with dropout

References

- Notes by:
  - Martin Gorner [The Examples we just did]
  - Tzar C. Umang
  - CS 20SI: TensorFlow for Deep Learning Research

  - *Code: github.com/martin-gorner/tensorflow-mnist-tutorial*

Tensorflow Resources

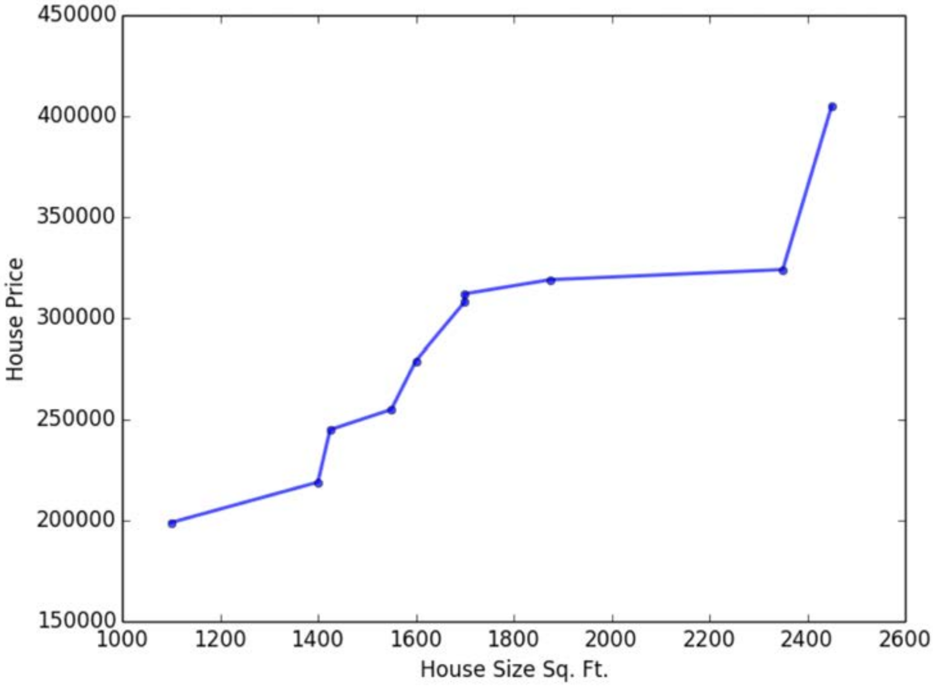
- Main Site https://www.tensorflow.org/
- Tutorials
  - https://github.com/nlintz/TensorFlow-Tutorials/

Appendix

Houses Prices

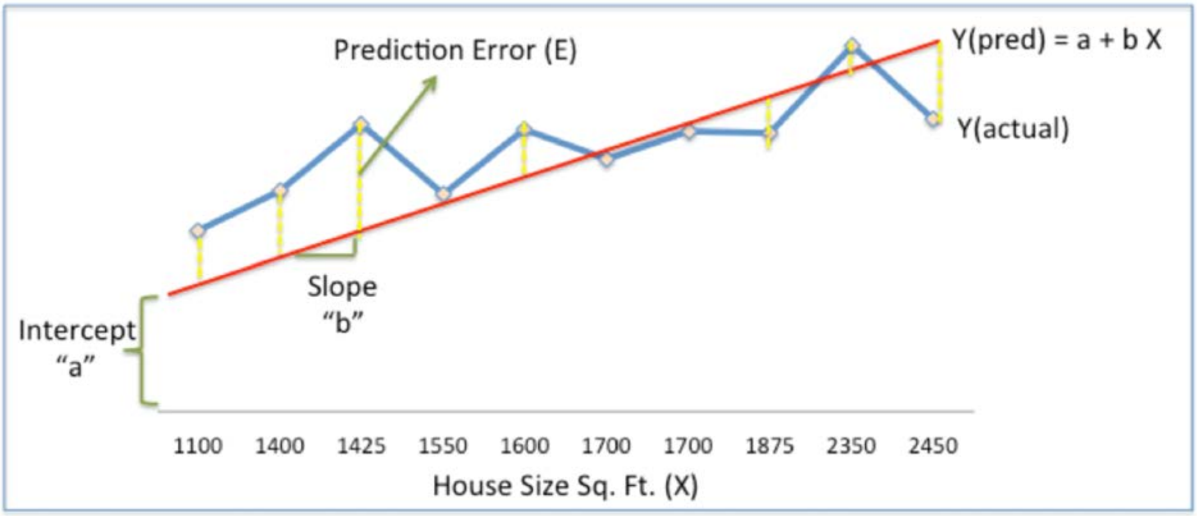
- Predict the price of a house given its area

House Size (ft^2)	1400	1600	1700	1875	1100	1550	2350	2450	1425	1700
House Price \$ (Y)	245,000	312,000	279,000	308,000	199,000	219,000	405,000	324,000	319,000	255,000



Predict Housing Prices

- Use a simple linear model, where we fit a line on the historical data, to predict the price of a new house (Ypred) given its size (X)
- $Y_{pred} = a+bX$



- The blue line gives the actual house prices from historical data (Y_{actual})
- The difference between Y_{actual} and Y_{pred} (given by the yellow dashed lines) is the prediction error (E)

Predict Housing Prices

- Need to find a line with optimal a and b weights that best fits the historical data by reducing the prediction error and improving prediction accuracy
- So, our objective is to find optimal **a, b** weights that minimize the error between actual and predicted values of house price
  - Sum of Squared Errors (SSE) = ½ Sum (Actual House Price – Predicted House Price)^2 = ½ Sum$(Y - Y_{pred})^2$
  - (1/2 is for mathematical convenience since it helps in calculating gradients in calculus)

Gradient Descent Algorithm

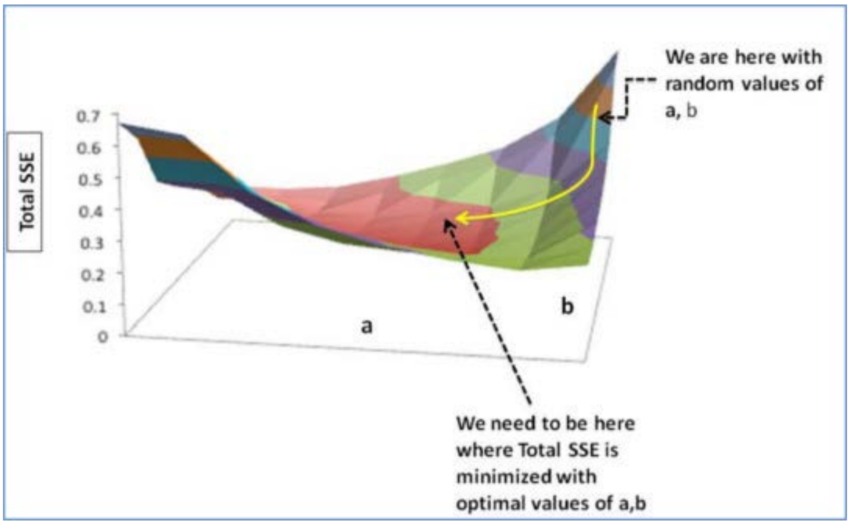
1) **Step 1:** Initialize the weights (a and b) with random values and calculate Error (SSE)
2) **Step 2:** Calculate the gradient i.e. change in SSE when the weights (a and b) are changed by a very small value from their original randomly initialized value. This helps us move the values of a and b in the direction in which SSE is minimized.
3) **Step 3:** Adjust the weights with the gradients to reach the optimal values where SSE is minimized
4) **Step 4:** Use the new weights for prediction and to calculate the new SSE
5) **Step 5:** Repeat steps 2 and 3 till further adjustments to weights doesn't significantly reduce the Error

Step 2: Calculate the error gradient w.r.t the weights

- $Y_p = a+b*X$
- $\partial_{SSE}/\partial_a = -(Y-Y_P)$ and $\partial_{SSE}/\partial_b = -(Y-Y_P)X$
- Here, $SSE=½ (Y-Y_P)^2 = ½(Y - (a+bX))^2$

The gradient vector, $[\partial_{SSE}/\partial_a\ \partial_{SSE}/\partial_b]^T$, gives the direction of the movement of a and b with respect to SSE

Step 3: Adjust the weights with the gradients to reach the optimal values where SSE is minimized



Update a and b

- Update rules:
  - a – ∂SSE/∂a
  - b – ∂SSE/∂b
- So, update rules:
  - New a = a – r * $\partial_{SSE}/\partial_a$ = 0.45 - 0.01*3.300 = 0.42
  - New b = b – r * $\partial_{SSE}/\partial_b$= 0.75 - 0.01*1.545 = 0.73
- Here, r is the learning rate = 0.01, which is the pace of adjustment to the weights.

Step 5: Repeat step 3 and 4

- Repeat step 3 and 4 till the time further adjustments to a, b doesn't significantly reduces the error. At that time, we have arrived at the optimal a,b with the highest prediction accuracy.