TensorFlow (Part 1)

AI Workshop, 2-4 December 2019

Somnuk Phon-Amnuaisuk
School of Computing and Informatics
Centre for Innovative Engineering
Universiti Teknologi Brunei

meme

A meme (/ˈ mi: m/ meem), a neologism coined by Richard Dawkins, is "an idea, behavior, or style that spreads from person to person within a culture". A meme acts as a unit for carrying cultural ideas, symbols, or practices that can be transmitted from one mind to another through writing, speech, gestures, rituals, or other imitable phenomena with a mimicked theme.

Disclaimer

This lecture is compiled from my lectures as well as materials gathering from lectures found in public domains.

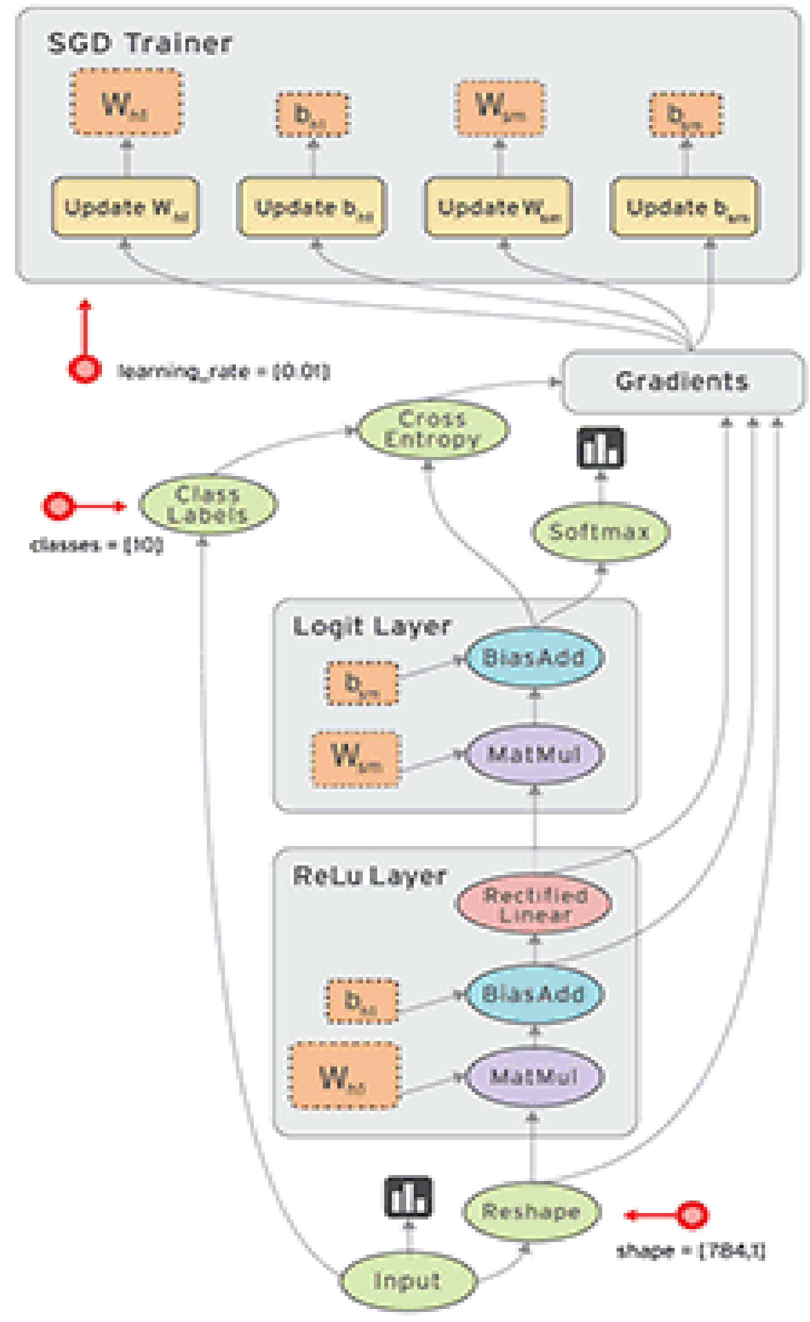
Outline

- Machine Learning Practical
  - TensorFlow
  - Perceptron
  - Loss function and Training

TensorFlow

TF 0.x →TF 1.x → TF 2.x

*** Credit Google site, slideshare site and Stanford University



Tensorflow

- Starting in 2011, Google Brain built DistBelief as a proprietary machine learning system based on deep learning neural networks.
- TensorFlow is Google Brain's second-generation system. Version 1.0.0 was released on February 11, 2017 (version 2.0 was released in October 2019).
- TensorFlow computations are expressed as stateful dataflow graphs. The name TensorFlow derives from the operations that such neural networks perform on multidimensional data arrays, which are referred to as tensors.
- TensorFlow provides stable Python (for version 3.7 across all platforms) and C APIs; and without API backwards compatibility guarantee: C++, Go, Java, JavaScript and Swift (early release).
- Third-party packages are available for C#, Haskell, Julia, R, Scala, Rust, OCaml, and Crystal.

A Quick Look at TensorFlow (TF 1.x)

Build the model

```
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data

mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
x = tf.placeholder("float", shape=[None, 784])
W = tf.Variable(tf.zeros([784,10]))
b = tf.Variable(tf.zeros([10]))
y = tf.nn.softmax(tf.matmul(x, W) + b)
```

A Quick Look at TensorFlow

Construct a training outline

```
y_ = tf.placeholder(tf.float32, [None, 10])
cross_entropy = -tf.reduce_sum(y_*tf.log(y))
opt = tf.train.GradientDescentOptimizer(0.01)
train_op = opt.minimize(cross_entropy)
```

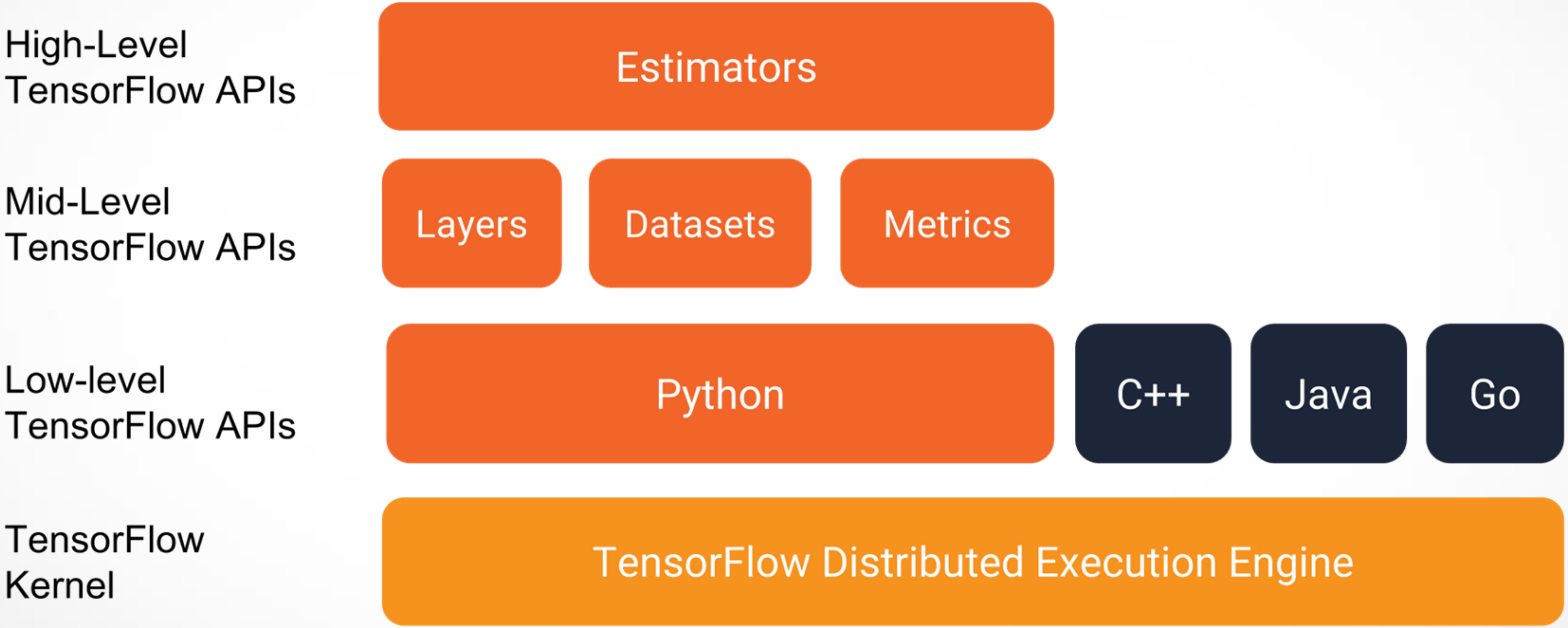
Cross entropy is defined as $H(p,q) = -\sum_{x\in\mathcal{X}} p(x)\,\log q(x)$

A Quick Look at TensorFlow

Execute the program

```
init = tf.initialize_all_variables()
sess = tf.Session()
sess.run(init)
for i in range(1000):
  batch_xs, batch_ys = mnist.train.next_batch(100)
  sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
```

TensorFlow


High-Level TensorFlow APIs
Estimators
Mid-Level TensorFlow APIs
Layers
Datasets
Metrics
Low-level TensorFlow APIs
Python
C++
Java
Go
TensorFlow Kernel
TensorFlow Distributed Execution Engine

TensorFlow Execution Modes

Graph Execution	Eager Execution

Graph Execution

- Operations construct a computational graph to be run later.
- Operations return tensors information
- Benefits:
  - Distributed training
  - Performance optimizations
  - More suitable for production deployment.

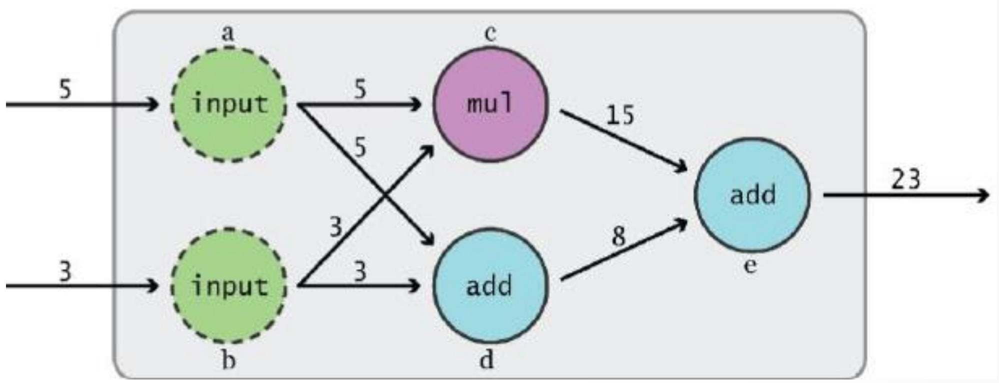
Eager Execution

- Imperative programming environment that evaluates operations immediately, without building graphs
- Operations return concrete values
- Benefits:
  - An intuitive interface
  - Easier debugging
  - Control flow in Python instead of graph control flow

Graphs and Computations

TensorFlow Graph Execution separates definition of computations from their execution

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



Benefits of Graphs

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

Tensors

- An n-dimensional array

  0-d tensor: scalar (number)

  1-d tensor: vector

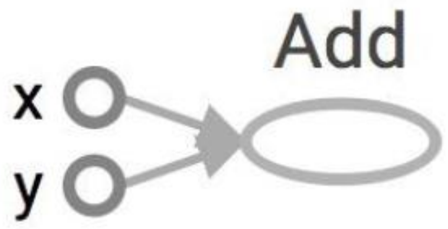
  2-d tensor: matrix

  and so on

- Common tensor types
  – Constant → tf.constant(…)
  – Variable → tf.Variable(…)
  – Placeholder → tf.placeholder(…)

Data Flow Graph

```
import tensorflow as tf

a = tf.add(3, 5)
```



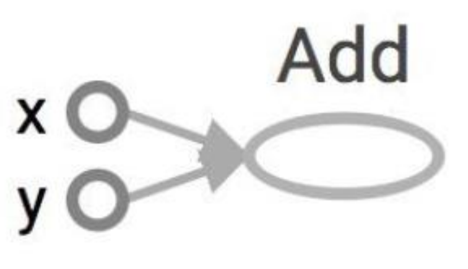
TF automatically names the nodes when you don't explicitly name them.

x = 3

y = 5

Data Flow Graph

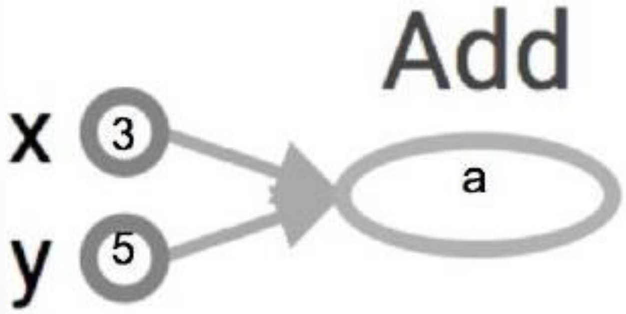
```
import tensorflow as tf
a = tf.add(3, 5)
print(a)

>> Tensor("Add:0", shape=(), dtype=int32)
(Not 8)
```



Getting the value of a Tensor

Create a **session**, assign tensor to a variable so we can refer to it

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))
sess.close()
>> 8
```

~~sess = tf.Session()~~

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

~~sess.close()~~

```
>> 8
```

Session (TF 1.x)

Session is class for running TensorFlow operations

```
# Build a graph.
a = tf.constant(5.0)
b = tf.constant(6.0)
c = a * b

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

# Evaluate the tensor `c`.
print(sess.run(c))
```

Session

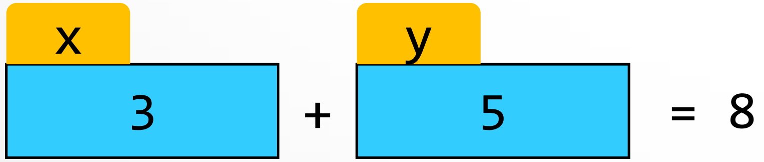
- A Session object encapsulates the environment in which Operation objects are executed, and Tensor objects are evaluated.
- Session will also allocate memory to store the current values of variables.
- A session may own resources, hence, it is important to release resources i.e., tf.Session.close(), or enclose session within ‘with tf.Session()’ block.
- Session is removed from TF 2.x

Graph = Symbolic Expression

Programming

- variables hold values and operations compute values

```
x = 3
y = 5
x + y
8
```

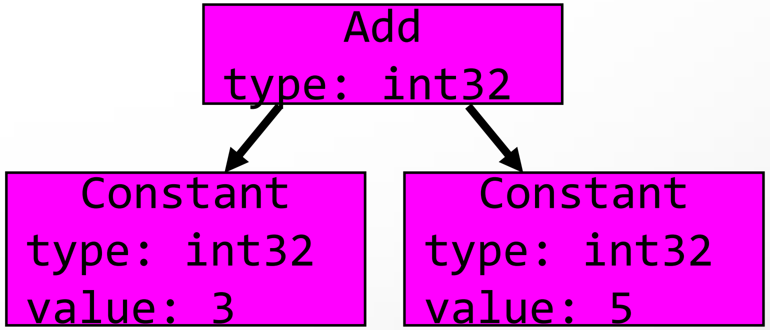


Symbolic Computing

- Variables represent themselves, with a name and a type
- Operations build expressions

```
x = tf.constant(3)
y = tf.constant(5)
x + y
Tensor("Add:0", shape=(), dtype=int32)
```

Overloaded + operator



- Tensors have no value (except constants), but can be evaluated to produce a value
- Evaluation requires a Session (TF 1.x), that contains the memory for the values associated to variables.
- Values are supplied through a dictionary

```
a = tf.placeholder(tf.int8)
b = tf.placeholder(tf.int8)
sess.run(a+b, feed_dict={a: 10, b: 32})
```

Summary of Common Tensors

- Constant

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

- Variable

  w = tf.Variable(<initial-value>, name=<optional-name>)

- Variables need to be initialized before being used.
- Placeholder can be seen as a variable that we assign data in at execution time.

```
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')
```

Summary of Common Tensors

```
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.]
```

Graphs = Symbolic Expression

- TensorFlow supports automatic differentiation
- TensorFlow automatically builds the backpropagation graph
- TensorFlow runtime automatically partitions the graph and distributes the execution on multiple devices.
- So the gradient computation in TensorFlow will also be distributed to run on multiple devices

Symbolic Computations

- Expressions can be transformed, before being evaluated
- In particular symbolic differentiation can be computed
- TensorFlow applies differentiation rules for known functions or composition thereof by applying the chain rule

Gradients

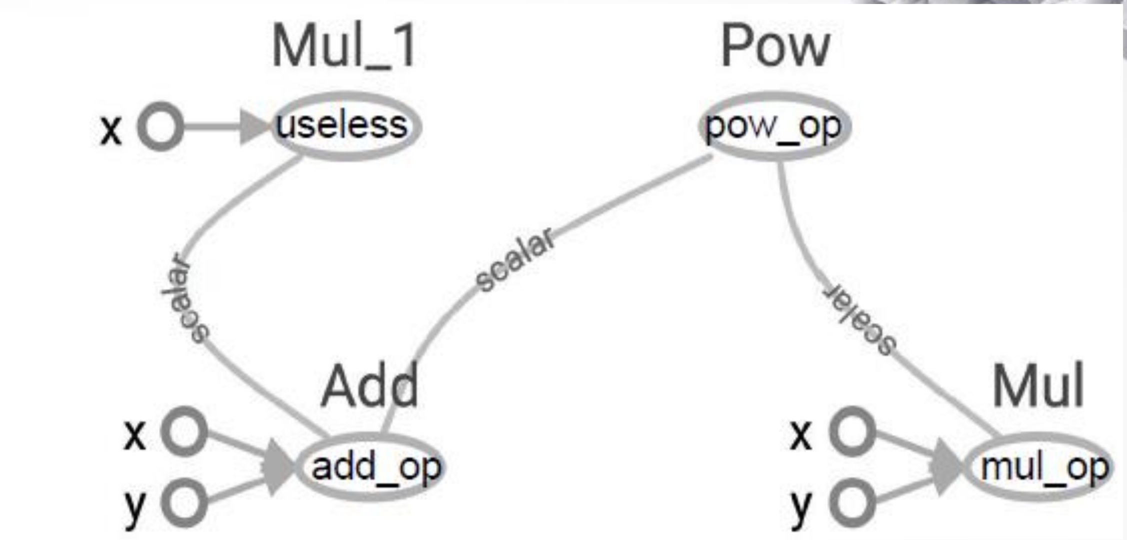
The `gradients_function` call takes a Python function as an argument and returns a Python callable that computes the partial derivatives with respect to its inputs.

Here is the derivative of `square()`:

```
x = tf.placeholder(tf.float32)
square = tf.multiply(x, x)
grad = tf.gradients(square,x)

with tf.Session() as sess:
  print(sess.run([square,grad], feed_dict={x:3.0}))
```

Beneficial Features (Save computation)

```
x = 2
y = 3
add_op = tf.add(x, y)
mul_op = tf.multiply(x, y)
useless = tf.multiply(x, add_op)
pow_op = tf.pow(add_op, mul_op)
with tf.Session() as sess:
    z = sess.run(pow_op)
```



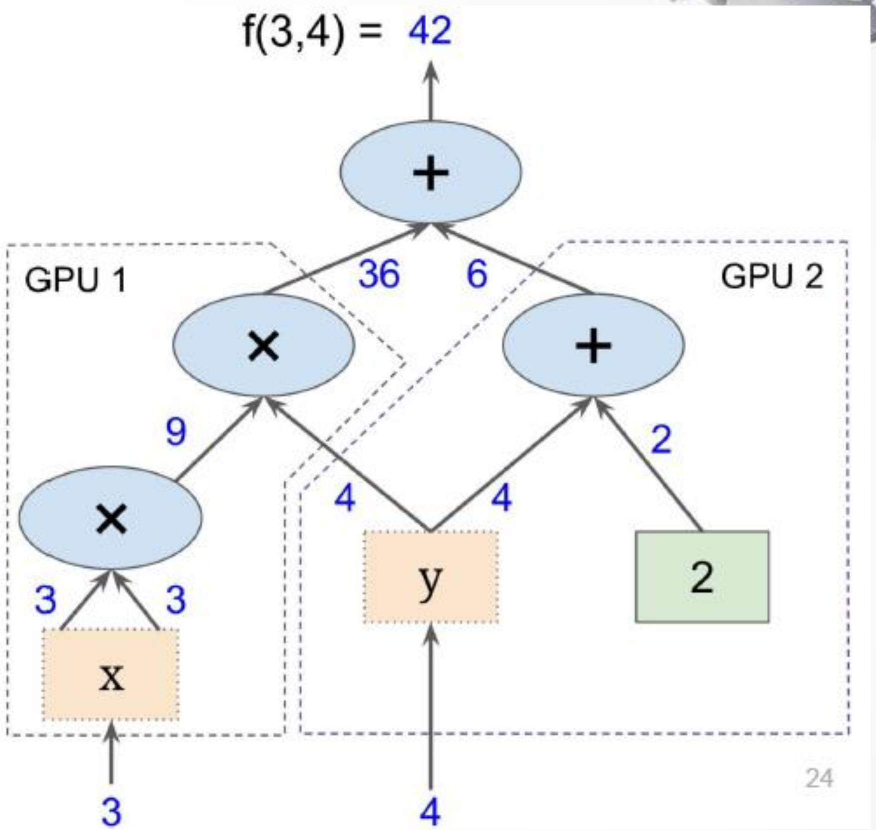
Because we only want the value of pow_op and pow_op doesn’t depend on useless, session won’t compute value of useless

→ save computation

Beneficial Features (Subgraphs)

Possible to break graphs into several chunks and run them parallelly across multiple CPUs, GPUs, TPUs, or other devices

Example: AlexNet



Graph from *Hands-On Machine Learning with Scikit-Learn and TensorFlow*

Beneficial Features (Distributed Computation)

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

```
# Create 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.multiply(a, b)
# Create a session with log_device_placement set to True.
sess = tf.Session(config=tf.ConfigProto(log_device_placement=True))
# Run the op.
print(sess.run(c))
```

Practical Session
Constants, Sequences, Variables, Ops

AI workshop. Pre - Coding Conquest 2019 event

Tensor constructors

```
tf.zeros([2, 3], tf.int32) ==> [[0, 0, 0], [0, 0, 0]]

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

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

Constants

```
import tensorflow as tf
a = tf.constant([2, 2], name='a')
b = tf.constant([[0, 1], [2, 3]], name='b')
x = tf.multiply(a, b, name='mul')
with tf.Session() as sess:
  print(sess.run(x))
>> [[0 2]
    [4 6]]
```

Constants in Graphs

- Constants are stored in graph
- 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

Sequences

```
tf.linspace(start, stop, num, name=None)
tf.linspace(10.0, 13.0, 4) ==> [10. 11. 12. 13.]

tf.range(start, limit=None, delta=1, dtype=None, name='range')
tf.range(3, 18, 3) ==> [3 6 9 12 15]
tf.range(5) ==> [0 1 2 3 4]
```

Random Sequences

```
with tf.Session() as sess:
    r1 = random.normal([1,2])
    r2 = random.uniform([2])
    r3 = random.truncated_normal([2,3])
    print(sess.run([r1,r2,r3]))
```

Initialize seed at the beginning of a program to ensure replicability of experiments: `tf.set_random_seed(seed)`

Variables Initialization

Initialize all variables at once:

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

Initialize only a subset of variables:

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

Initialize a single variable:

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

Evaluating an expression

```
# 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)
```

Assignment

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

W.assign(100) creates an assign op. That op needs to be executed in a session to take effect.

```
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
```

Placeholders (TF 1.x)

A TF program often has 2 phases:

1. Assemble a graph
2. Use a session to execute operations in the graph.

⇒ Assemble the graph first without knowing the values needed for computation

Analogy:

Define the function f(x, y) = 2 * x + y without knowing value of x or y.

x, y are placeholders for the actual values.

Placeholders

```
tf.placeholder(dtype, shape=None, name=None)
# create a placeholder for a vector of 3 elements, type tf.float32
a = tf.placeholder(tf.float32, shape=[3])
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)) # >> ???
```

InvalidArgumentError: a doesn’t have an actual value

Supply values to placeholders

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

```
# create a placeholder for a vector of 3 elements, type tf.float32
a = tf.placeholder(tf.float32, shape=[3])
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:
    # the tensor a is the key, not the string 'a'
    print(sess.run(c, feed_dict={a: [1, 2, 3]}))
# >> [6, 7, 8]
```

Placeholders

shape=None means that tensor of any shape will be accepted as value for placeholder.

```
tf.placeholder(dtype, shape=None, name=None)
# create a placeholder for a vector of 3 elements, type tf.float32
a = tf.placeholder(tf.float32, shape=[3])
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, feed_dict={a: [1, 2, 3]}))
# >> [6, 7, 8]
```

shape=None also breaks all following shape inference, which makes many ops not work because they expect certain rank.

Feeding Data to Placeholders

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

Optimizers

Session looks at all **trainable** variables that loss depends on and updates them according to an optimizer

```
Loss = …loss function based on X and Y
optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.01).minimize(loss)
_, l = sess.run([optimizer, loss], feed_dict={X: x, Y:y})
```

Trainable Variable

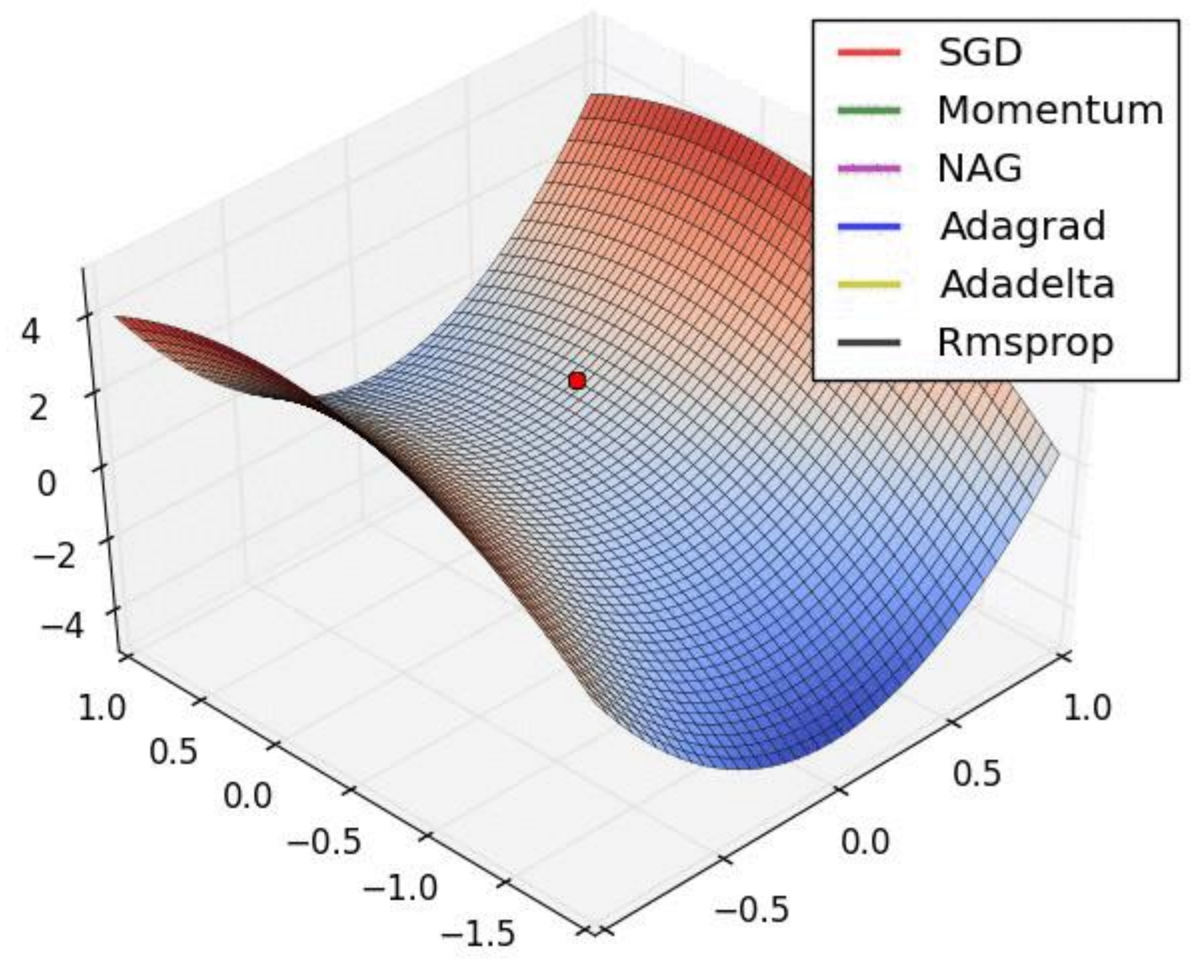
```
tf.Variable(initial_value=None, trainable=True,...)
```

Specify if a variable should be trained or not

By default, all variables are trainable

Available Optimizers

```
tf.train.GradientDescentOptimizer

tf.train.AdagradOptimizer

tf.train.MomentumOptimizer

tf.train.AdamOptimizer

tf.train.FtrlOptimizer

tf.train.RMSPropOptimizer

...
```



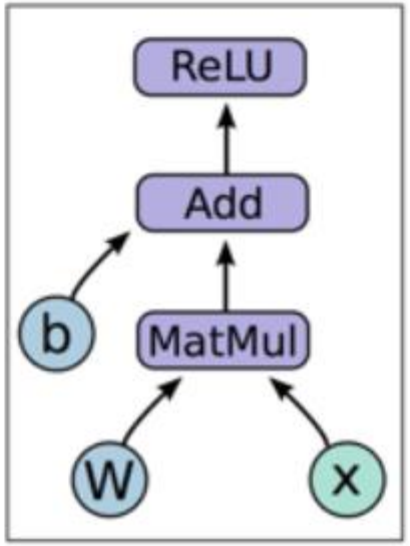
Optimization algorithms visualized over time in 3D space. (Source: Stanford class CS231n, MIT License)

Training Pipeline

Constructing a Computational Graph

```
# import the tensorflow library
import tensorflow as tf
import numpy as np

# create the input placeholder
X = tf.placeholder(tf.float32, shape=[None, 784], name="X")
weight_initer = tf.truncated_normal_initializer(mean=0.0, stddev=0.01)

# create network parameters
W = tf.get_variable(name="Weight", dtype=tf.float32, shape=[784, 200], initializer
=weight_initer)
bias_initer =tf.constant(0., shape=[200], dtype=tf.float32)
b = tf.get_variable(name="Bias", dtype=tf.float32, initializer=bias_initer)

# create MatMul node
x_w = tf.matmul(X, W, name="MatMul")
# create Add node
x_w_b = tf.add(x_w, b, name="Add")
# create ReLU node
h = tf.nn.relu(x_w_b, name="ReLU")

# Add an Op to initialize variables
init_op = tf.global_variables_initializer()
```

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

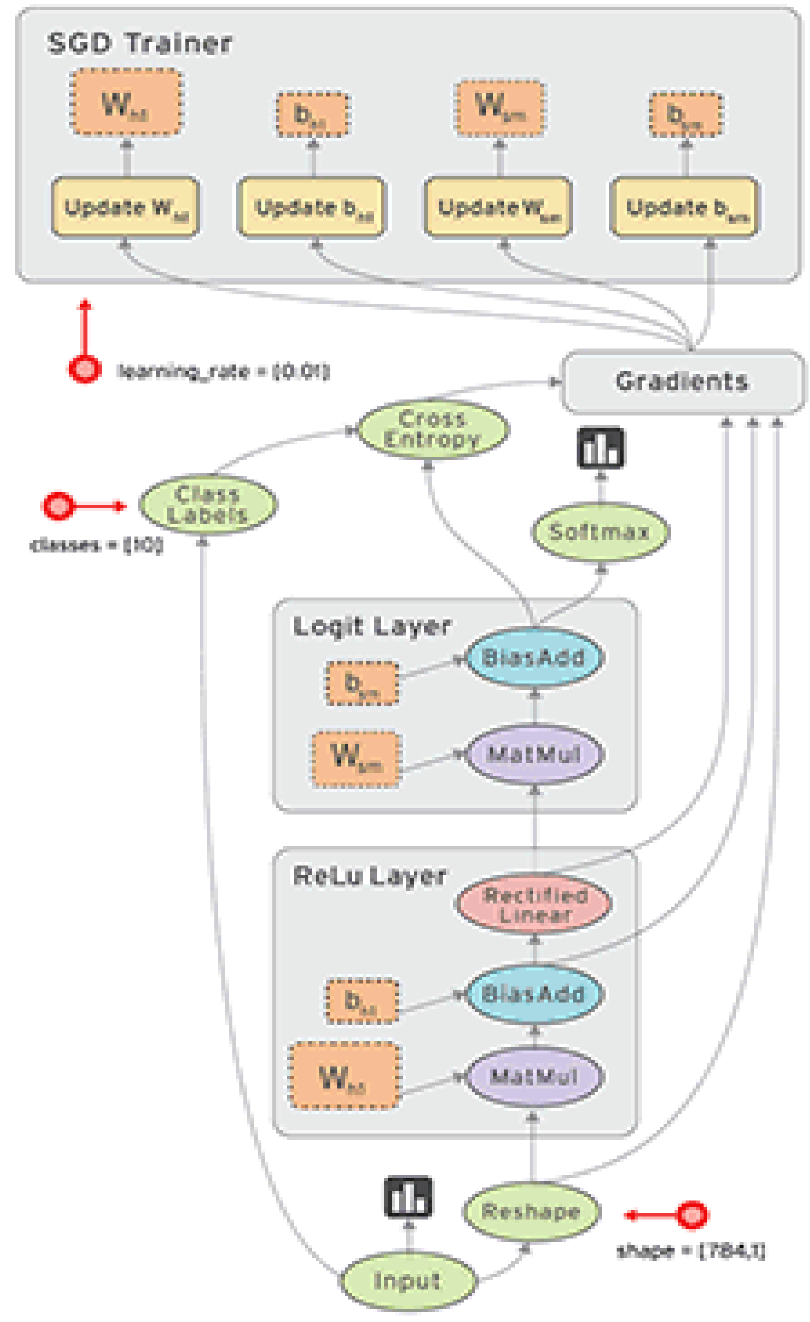


http://easy-tensorflow.com/tf-tutorials/basics/tensor-types

1. Create variables and placeholders (e.g. X, Y, W, b)
2. Assemble the graph
   - Define the output, e.g.:
     `Y_predicted = w * X + b`
   - Specify the loss function, e.g.:
     `loss = tf.square(Y - Y_predicted, name='loss’)`
3. Create an optimizer, e.g.:
   ```
   opt = tf.train.GradientDescentOptimizer(learning_rate=0.001)
   optimizer = opt.minimize(loss)
   ```
4. Train the model
   - Initialize variables
   - Run optimizer, feeding data into variables and placeholders

Practical: TensorFlow

*** Credit Google site, slideshare site and Stanford University



Practical 1: Quadratic Equation

Creating a Simple TensorFlow Program

Let's write a tensorflow graph to compute roots of a given quadratic equation

$$y = ax^2 + bx + c$$

$$e.g.,y = 2x^2 - 3x - 1$$

```
a = tf.constant(2)
b = tf.constant(-3)
c = tf.constant(-1)
root1 = tf.add(-b,tf.sqrt(bb - 4ac))/(2a)
root2 = tf.add(-b, - tf.sqrt(bb - 4ac))/(2a)
```

```
[16] # Create a graph.
     g2 = tf.Graph()

     # Establish the graph as the "default" graph.
     with g2.as_default():
       # Assemble a graph consisting of the following operations:
       a = tf.placeholder(tf.float32)
       b = tf.placeholder(tf.float32)
       c = tf.placeholder(tf.float32)
       print(a,b,c)
       root1 = tf.add(-b,tf.sqrt(b*b - 4*a*c))/(2*a)
       root2 = tf.add(-b, - tf.sqrt(b*b - 4*a*c))/(2*a)
       print(root1,root2)
       with tf.Session() as sess:
         #
         result1 = sess.run(root1, feed_dict={a:2, b:-3, c:-1})
         result2 = sess.run(root2, feed_dict={a:2, b:-3, c:-1})
         print(result1, result2)
```

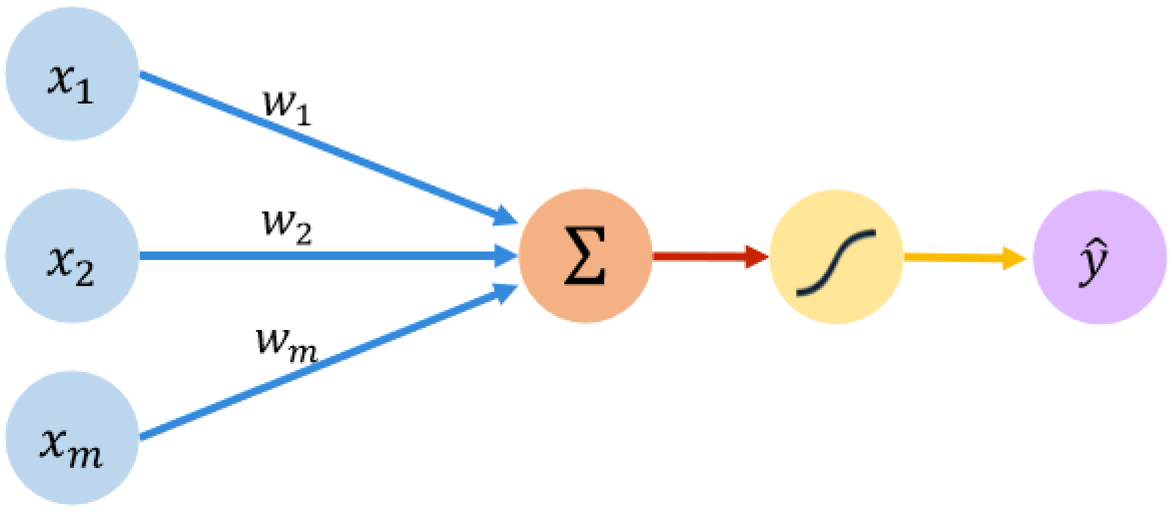
Practical 2: Perceptron

Output

Linear combination of inputs

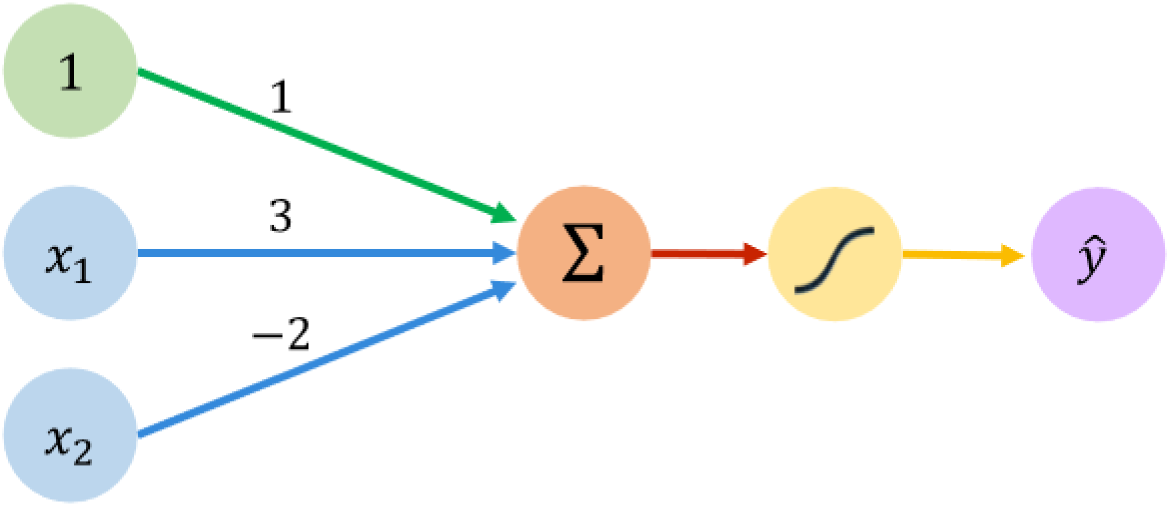
$$\hat{y} = g\left(\sum_{i=1}^{m} x_i \, w_i\right)$$

Non-linear activation function



We have: $w_0 = 1$ and $\boldsymbol{W} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

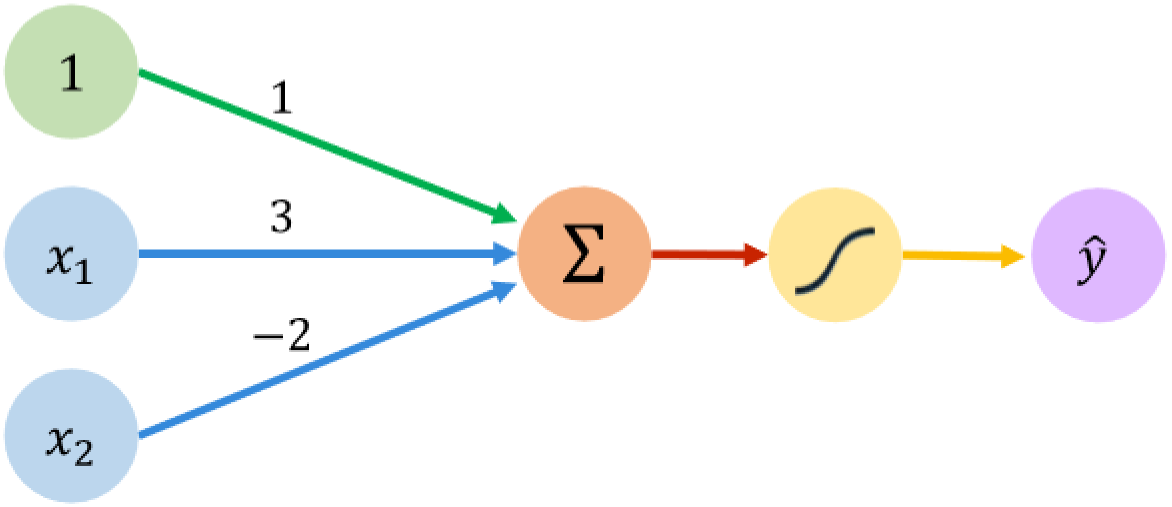
$$\begin{aligned} \hat{y} &= g(w_0 + \boldsymbol{X}^T \boldsymbol{W}) \\ &= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right) \\ \hat{y} &= g(1 + 3x_1 - 2x_2) \end{aligned}$$



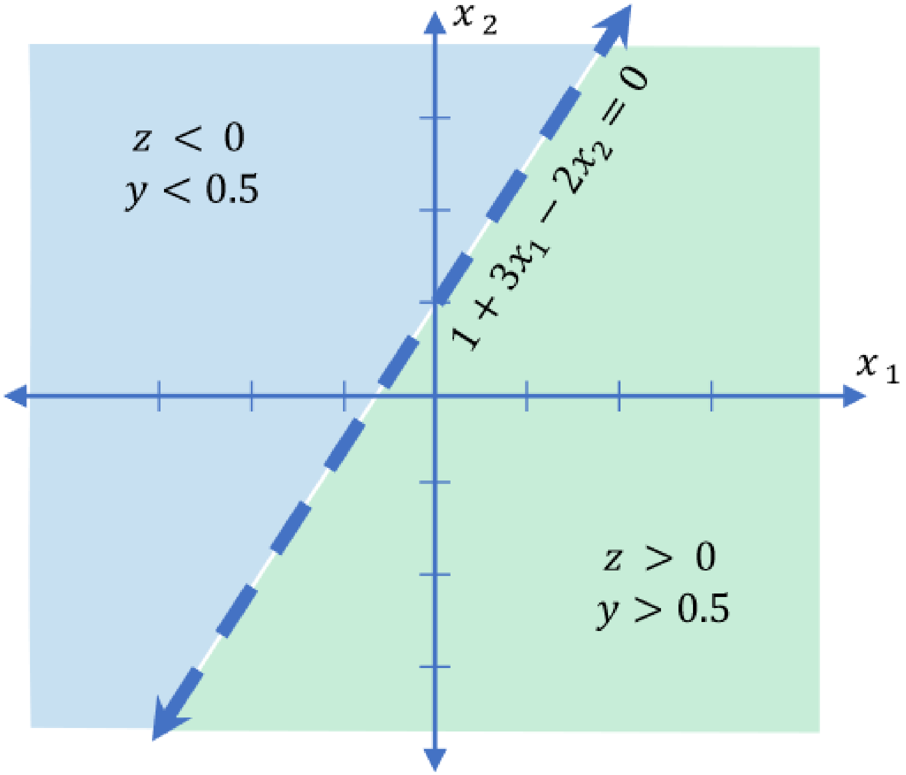
AI workshop. Pre - Coding Conquest 2019 event

We have: $w_0 = 1$ and $\boldsymbol{W} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

$$\hat{y} = g(w_0 + \boldsymbol{X}^T \boldsymbol{W})$$
$$= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right)$$
$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



$$\hat{y} = g(1 + 3x_1 - 2x_2)$$



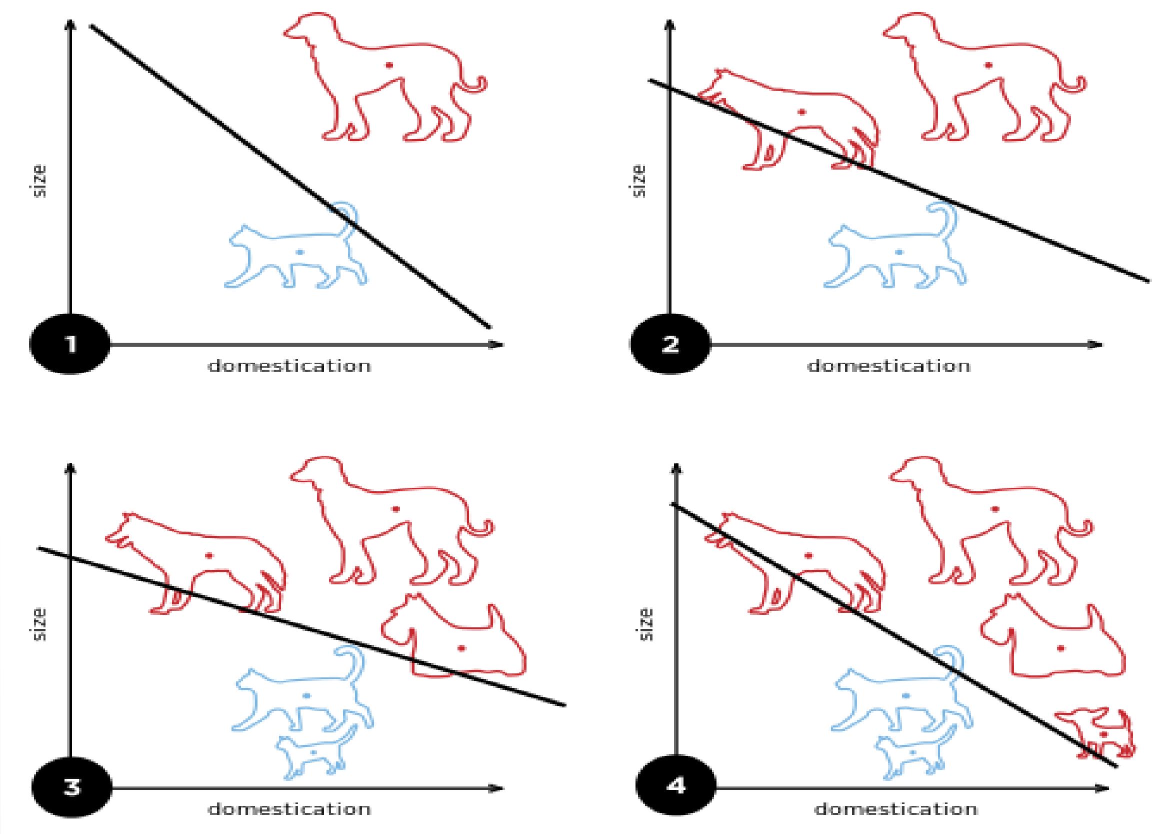
Prepare Data

```
[ ]  # prepare data set
     X = tf.placeholder( tf.float32, shape=(None,2),name=None )
     y = tf.placeholder( tf.float32, shape=(None,1),name=None )
     data = np.array( [[1,1],[1,0],[0,1],[0,0]]*50 )
     label = np.array([[1],[0],[0],[0]]*50)
     datID = np.random.permutation(range(200))
     datTrain = data[datID[:100],:]
     labTrain = label[datID[:100]]
     datTest = data[datID[100:],:]
     labTest = label[datID[100:]]
     #print(datTest, labTest)
```

Learning: Perceptron

- Rosenblatt: Perceptron

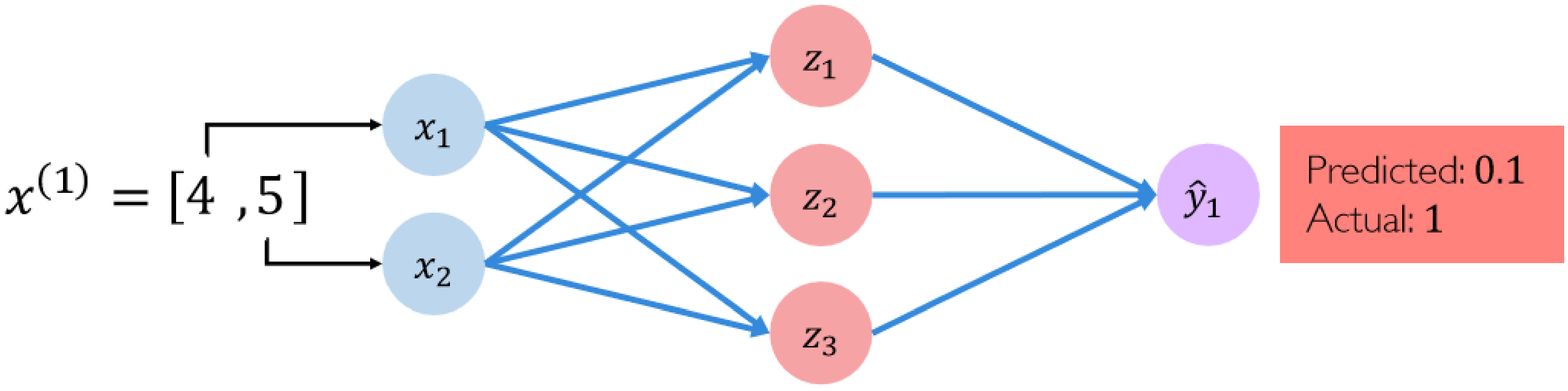
$$w_i(t+1) = w_i(t) + \alpha(d_j - y_j(t))x_{j,i}$$



```
import matplotlib.pyplot as plt
#
epoch = 300
lr = 0.1
#
ind = np.random.permutation(range(100))
traindat = np.array([[1,1],[1,0],[0,1],[0,0]]*25)
target = np.array([[1],[0],[0],[0]]*25)
traindat = traindat[ind,:]
target = target[ind,:]
init = tf.global_variables_initializer()
#
with tf.Session() as sess:
  sess.run(init)
  for n in range(epoch):
    yfw,w,bias = sess.run([y,W,b], feed_dict={X:traindat})
    deltaw = np.sum((target-yfw)*traindat,axis=0)/100
    deltaw = np.reshape(deltaw,[2,1])
    deltab = np.sum(target-yfw)/100
    w = w + lr*deltaw
    bias = bias + lr*deltab
    W.load(w)
    b.load(bias)
```

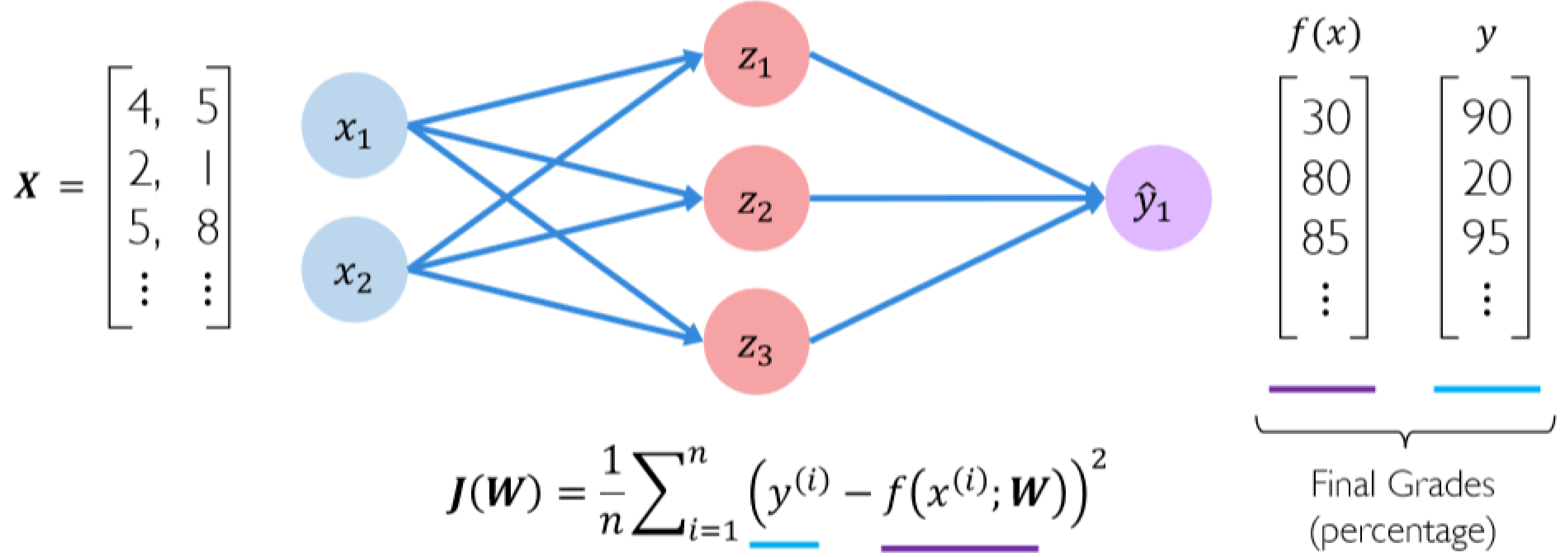
```python
# Parameters
learning_rate = 0.1
training_epochs = 500
display_step = 10

# Prepare preceptron model
# Set model weights
W = tf.Variable(tf.random_normal([2,1], stddev=0.35),name="weights")
b = tf.Variable(tf.random_normal([1], stddev=0.35),name="bias")
# Construct model
activation = tf.sigmoid(tf.matmul(X, W) + b)

# Minimize square error
squarederror = (y-activation)**2
cost = tf.reduce_mean(tf.reduce_sum(squarederror,reduction_indices = 1))
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
```

Loss Function



$$\mathcal{L}\left(\underbrace{f\left(x^{(i)}; \boldsymbol{W}\right)}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}}\right)$$

Mean-Square Error Loss

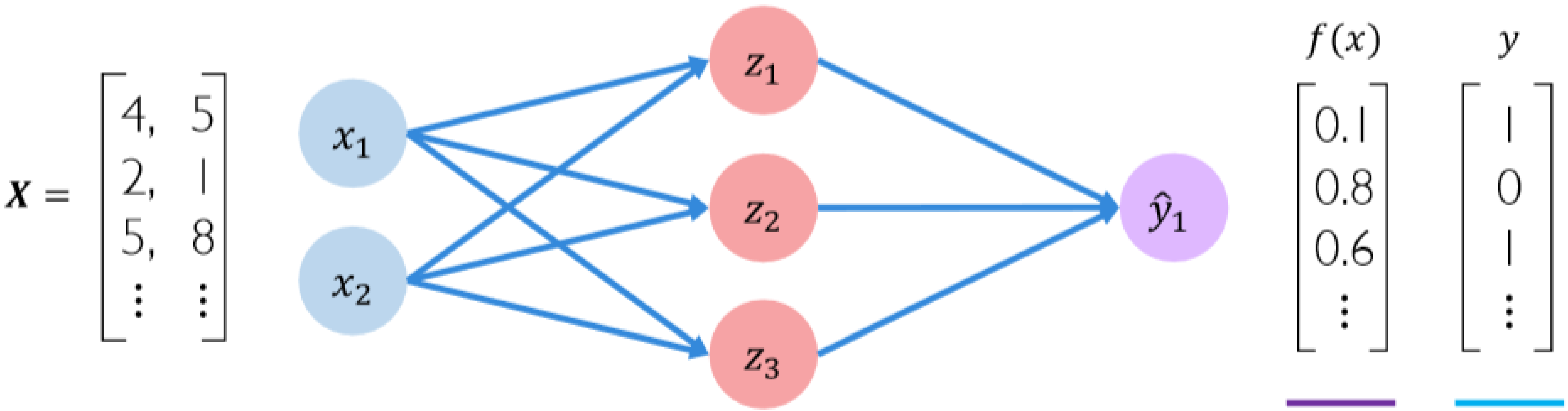


$$J(W) = \frac{1}{n}\sum_{i=1}^{n}\Big(\underbrace{y^{(i)}}_{\text{Actual}} - \underbrace{f(x^{(i)}; W)}_{\text{Predicted}}\Big)^2$$

```
loss = tf.reduce_mean( tf.square(tf.subtract(model.y, model.pred) )
```

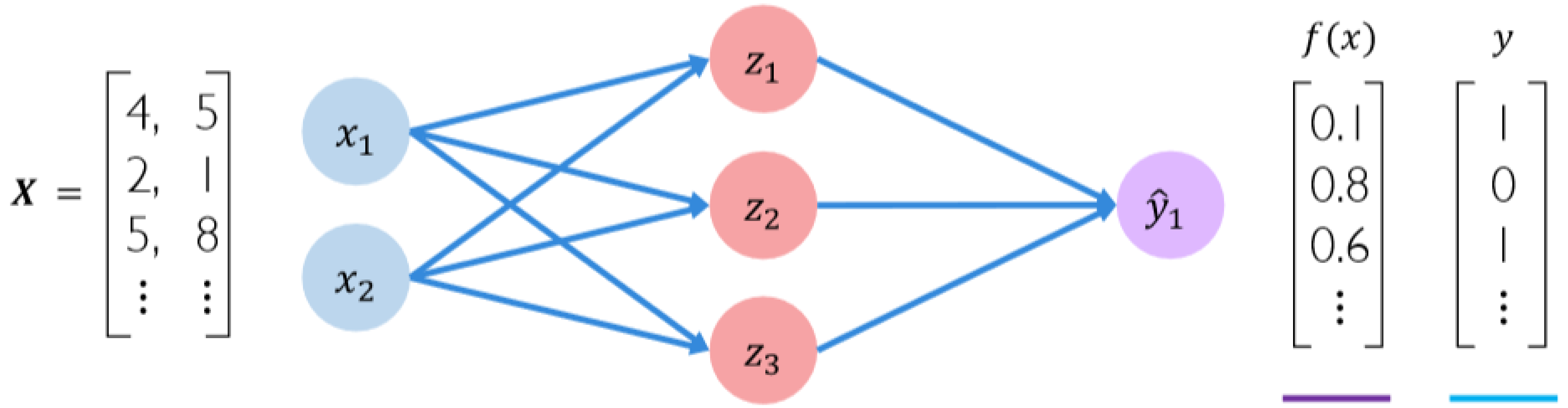
AI workshop. Pre - Coding Conquest 2019 event

Empirical Loss



$$\boldsymbol{J}(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}\Big(\underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}, \underbrace{y^{(i)}}_{\text{Actual}}\Big)$$

Binary Cross Entropy Loss



$$J(\boldsymbol{W}) = \frac{1}{n}\sum_{i=1}^{n} \underbrace{y^{(i)}}_{\text{Actual}} \log\Big(\underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big) + \big(1-\underbrace{y^{(i)}}_{\text{Actual}}\big)\log\Big(1-\underbrace{f\big(x^{(i)};\boldsymbol{W}\big)}_{\text{Predicted}}\Big)$$

```
loss = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(model.y, model.pred) )
```

Gradient Descent

Algorithm

1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$

```
weights = tf.random_normal(shape, stddev=sigma)
```

2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

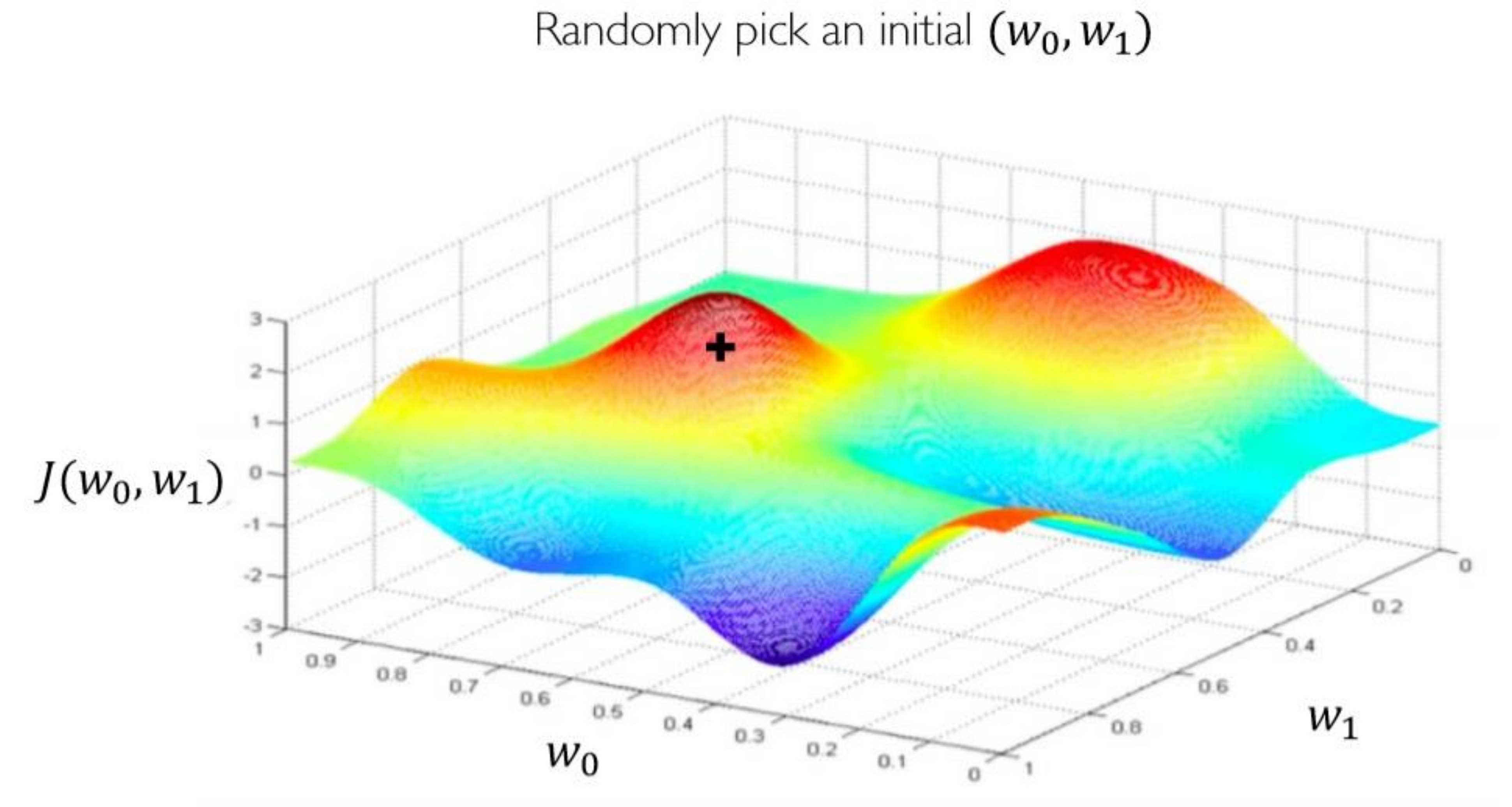
```
grads = tf.gradients(ys=loss, xs=weights)
```

4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$

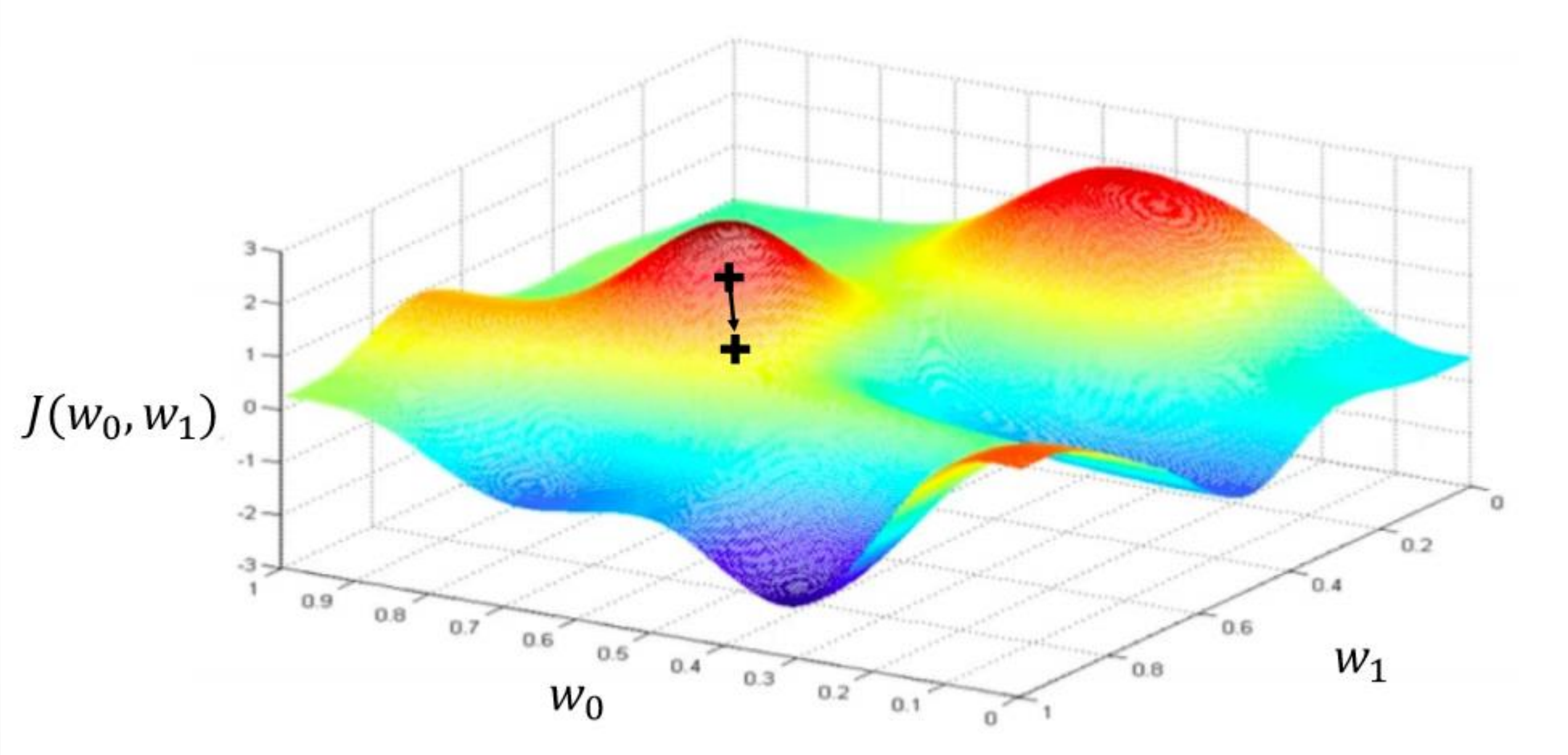
```
weights_new = weights.assign(weights - lr * grads)
```

5. Return weights

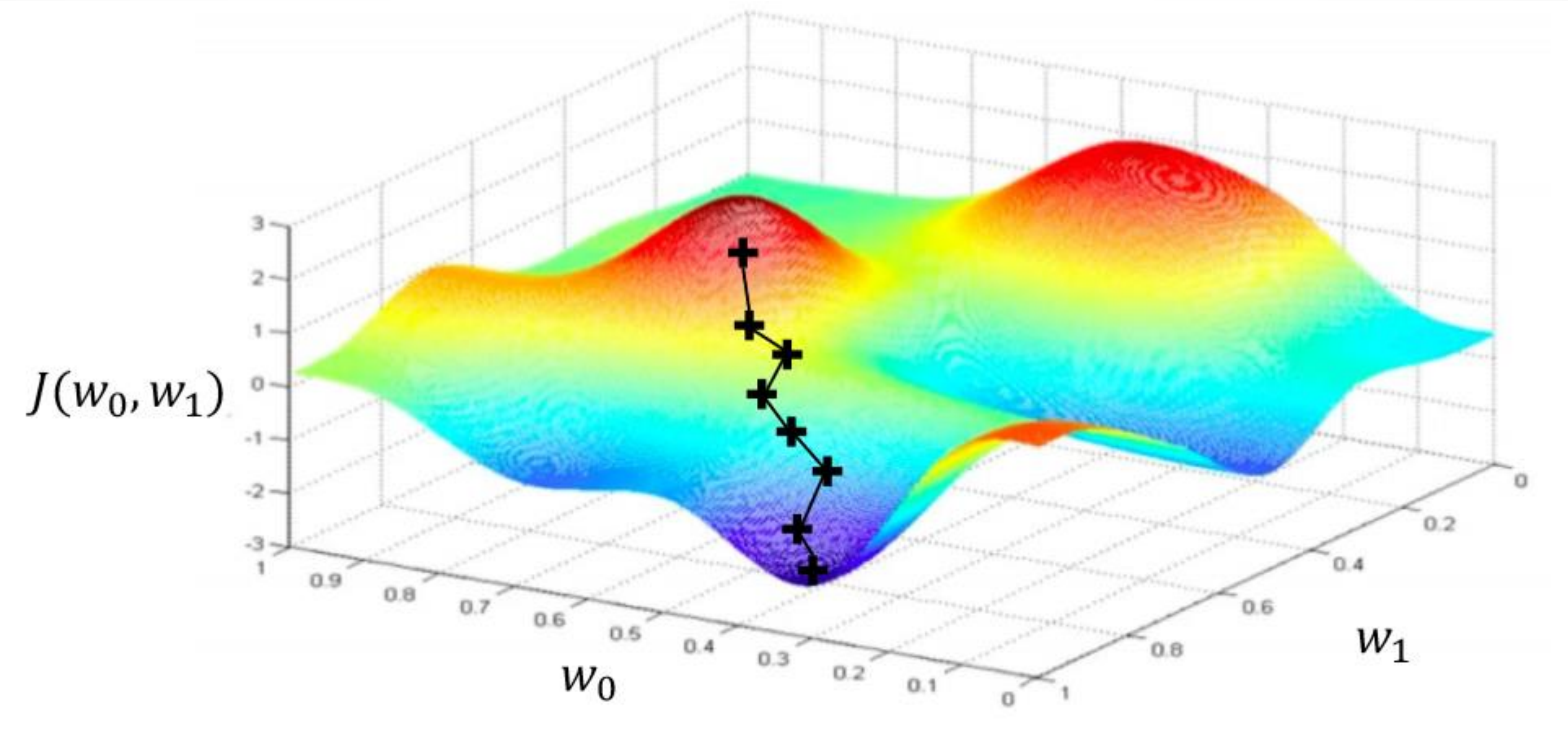
Gradient Descent



Gradient Descent



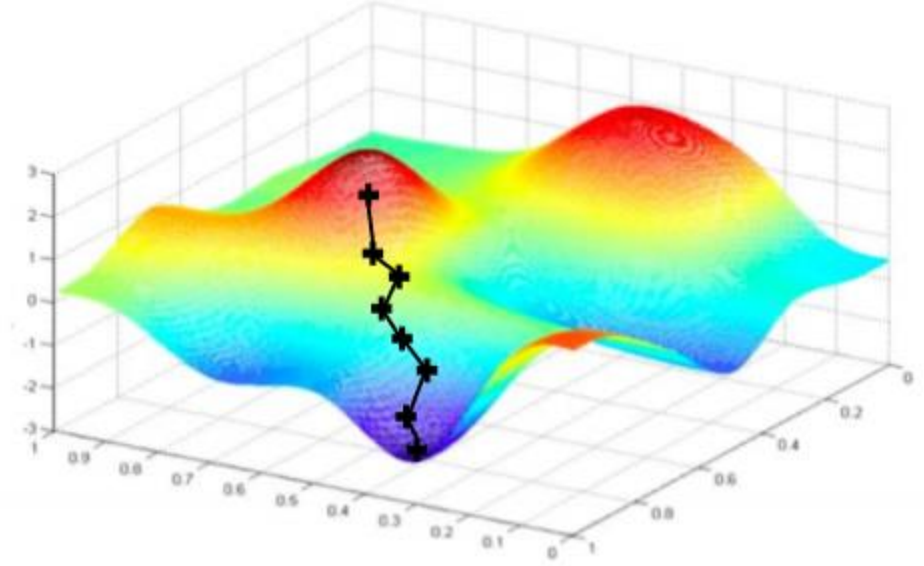
Gradient Descent



Gradient Descent

Algorithm

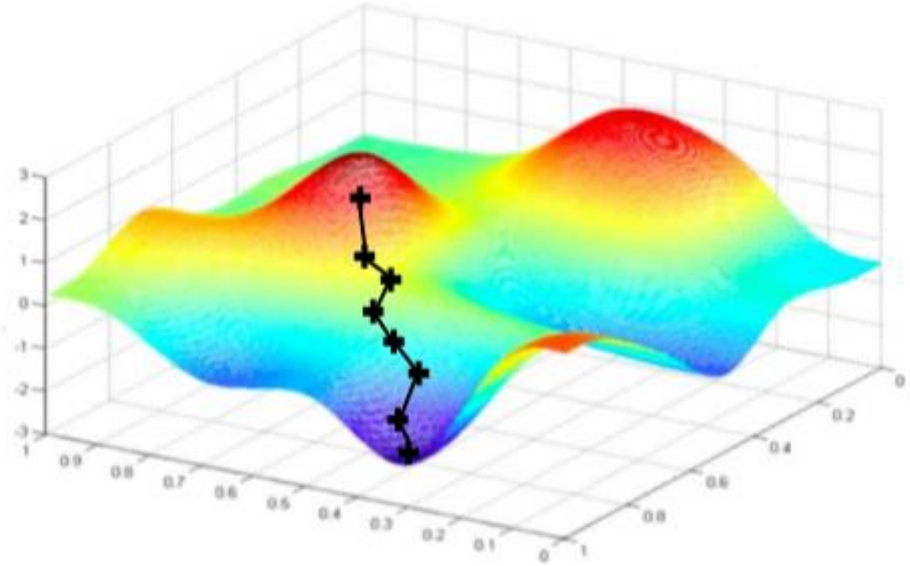
1. Initialize weights randomly $\sim\mathcal{N}(0,\sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Return weights



Stochastic Gradient Descent

Algorithm

1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick single data point $\boldsymbol{i}$
4. Compute gradient, $\boxed{\frac{\partial J_i(\boldsymbol{W})}{\partial \boldsymbol{W}}}$
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
6. Return weights

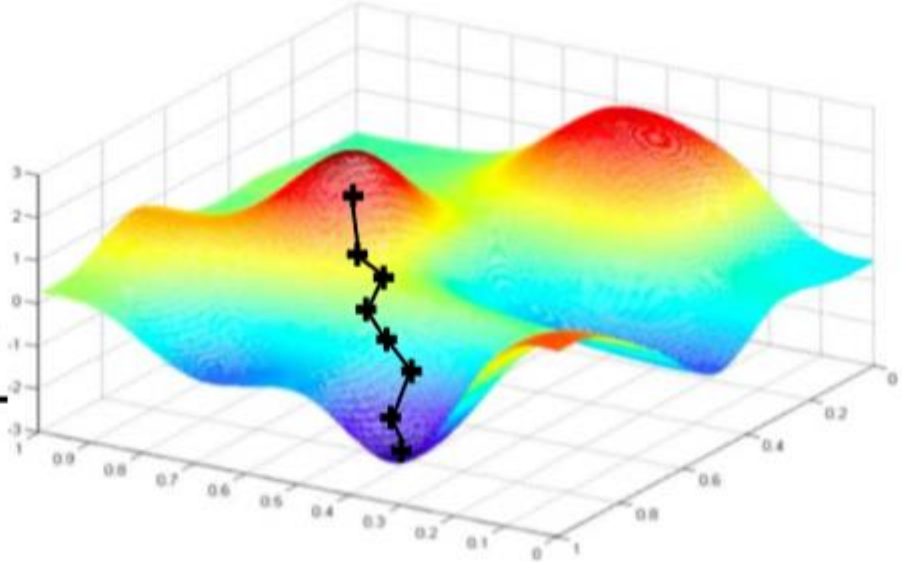


Easy to compute but
very noisy
(stochastic)!

Stochastic Gradient Descent

Algorithm

1. Initialize weights randomly $\sim\mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient, $\frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}} = \frac{1}{B}\sum_{k=1}^{B} \frac{\partial J_k(\boldsymbol{W})}{\partial \boldsymbol{W}}$
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
6. Return weights



Adaptive Learning Algorithms

• Momentum	tf.train.MomentumOptimizer
• Adagrad	tf.train.AdagradOptimizer
• Adadelta	tf.train.AdadeltaOptimizer
• Adam	tf.train.AdamOptimizer
• RMSProp	tf.train.RMSPropOptimizer

Q & A