DEPARTAMENTO DE SEÑALES, SISTEMAS Y RADIOCOMUNICACIONES

Acoustic Signal Processing

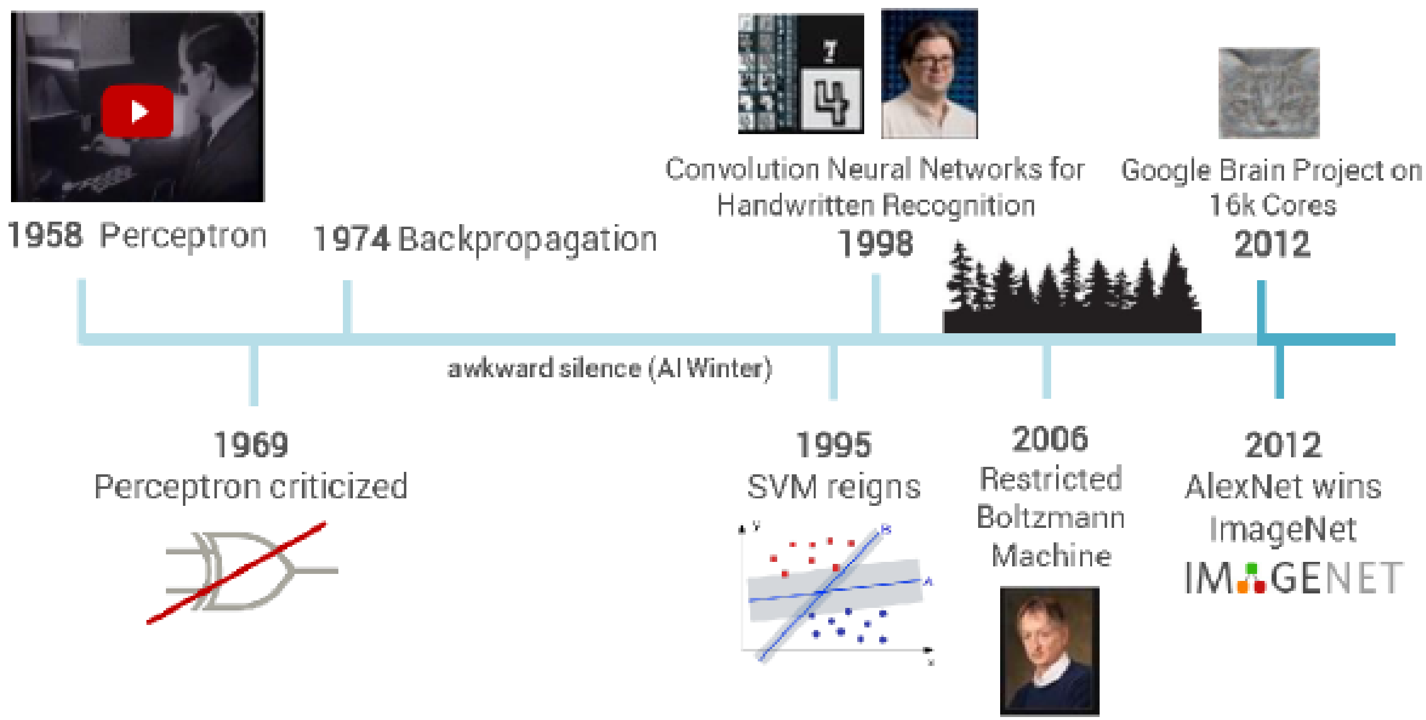
Máster Universitario en Ingeniería de Telecomunicación

Prof. Eduardo López Gonzalo
eduardo.lopez@upm.es
Prof. Luis A. Hernández Gómez
luisalfoso.hernandez@upm.es

Departamento de Señales, Sistemas y Radiocomunicaciones
E.T.S. Ingenieros de Telecomunicación
Universidad Politécnica de Madrid

Deep Learning: Hype or Reality?

A Brief History



Geoffrey Hinton:

Deep Learning: Hype or Reality?

DNNs better than humans at image recognition

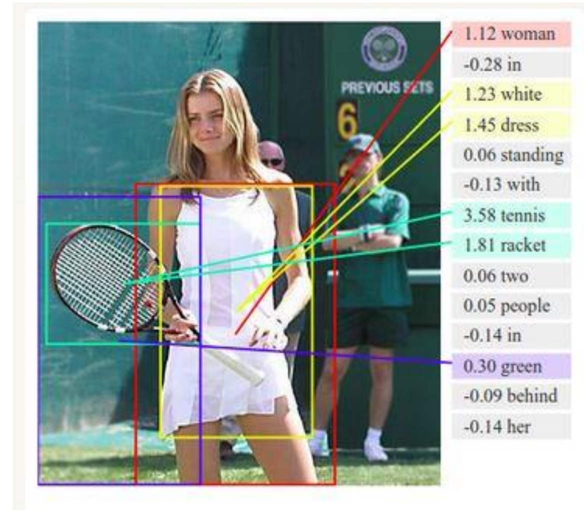
ImageNet

1000 categories

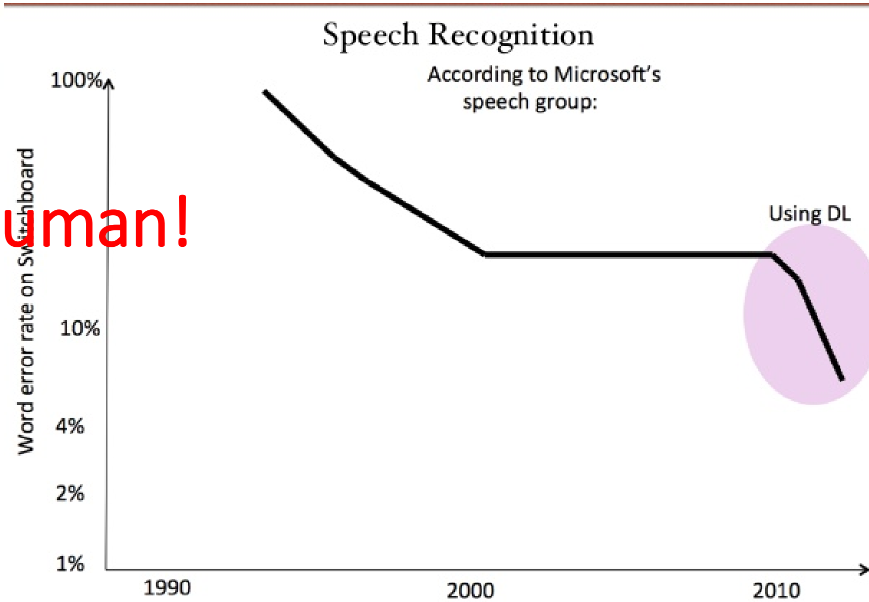
1.3M images

Human error: 5%

DNN: 3%



Super Human!



Deep Learning: Hype or Reality?

Deep Reinforcement Learning

AlphaGo

...going unsupervised! Deep Clustering

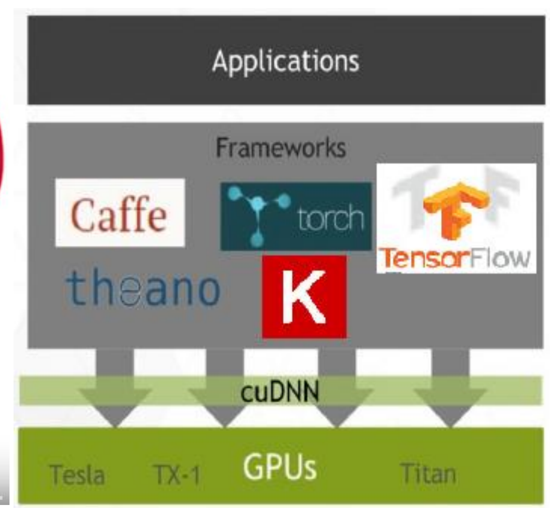
What Changed?
Old wine in new bottles

UDACITY
FREE COURSE
Deep Learning by Google

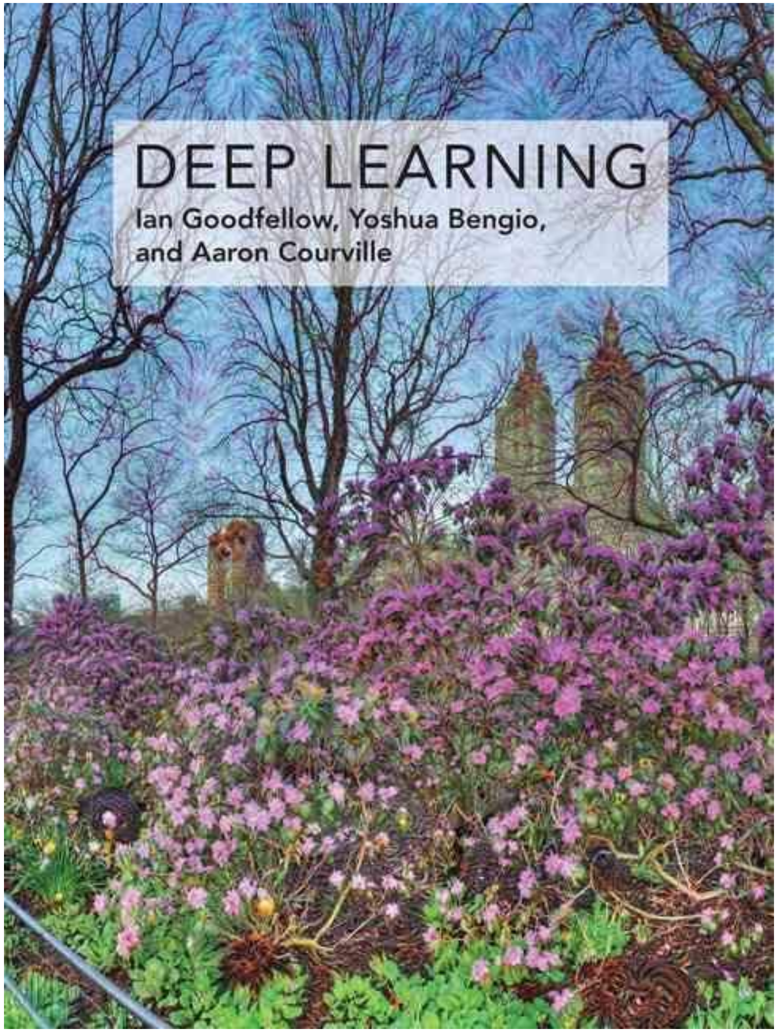
Big Data
(Digitalization)

Computation
(Moore's Law, GPUs)

Algorithmic Progress



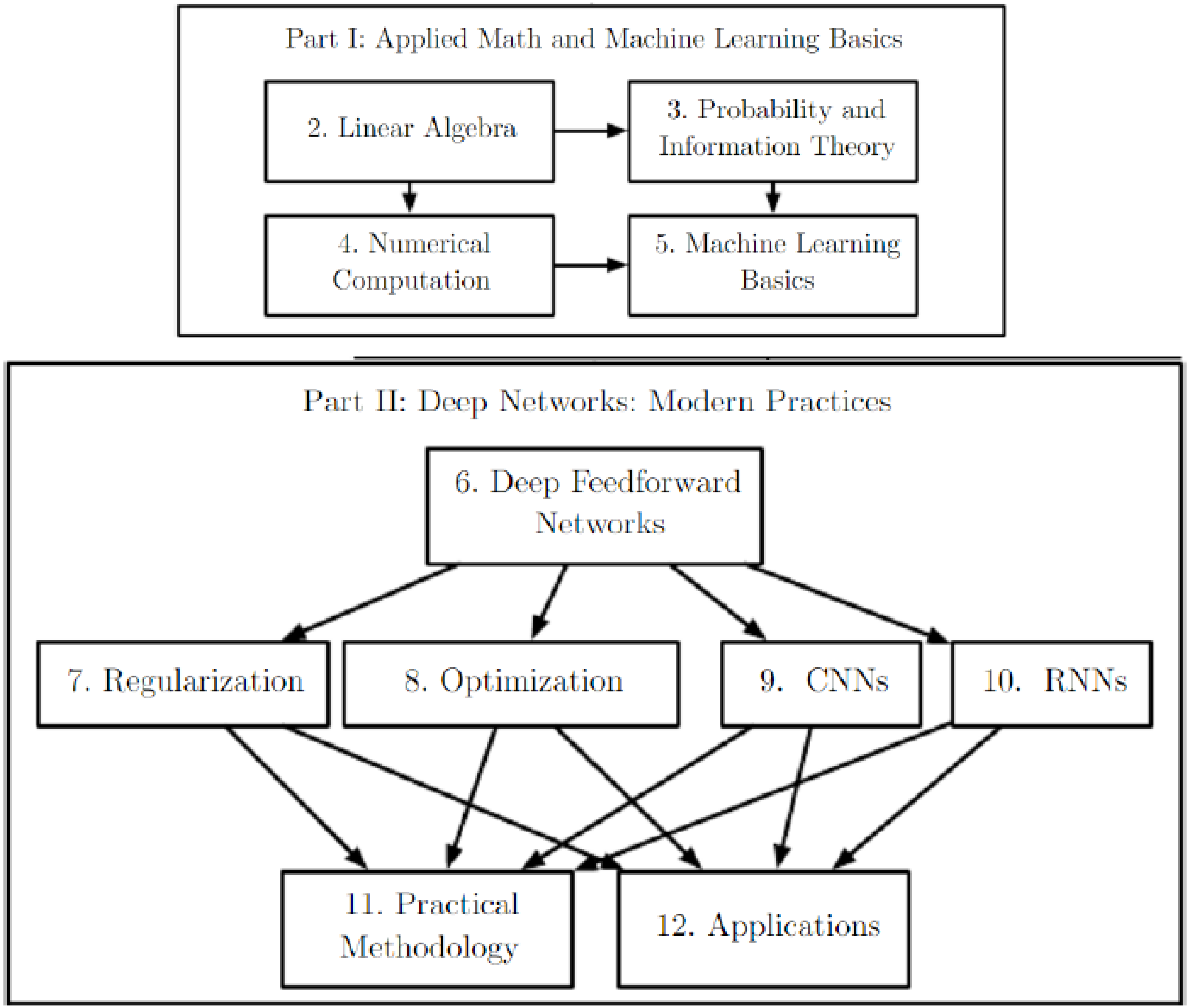
Learn the whole Machine Learning context



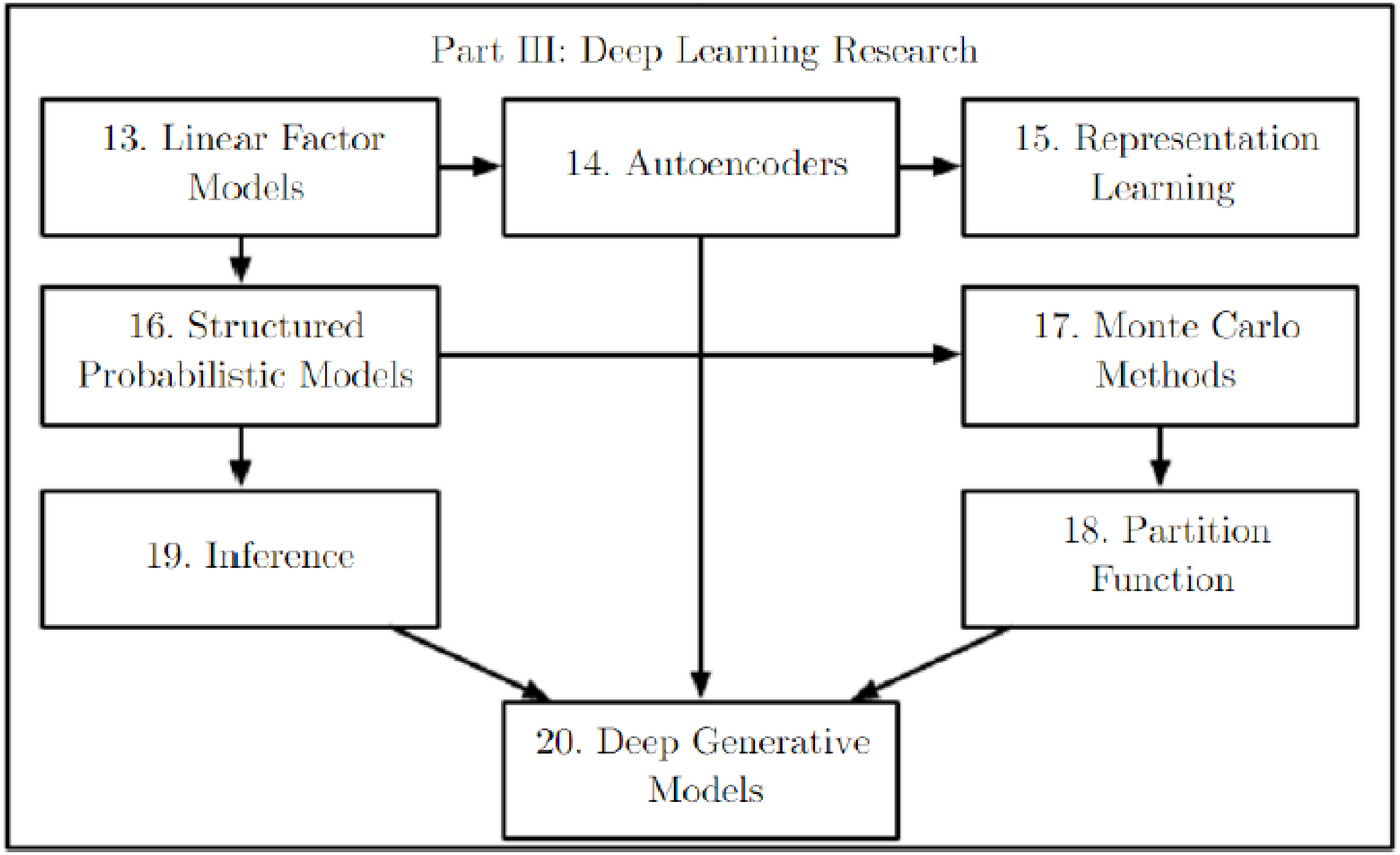
On line:
www.deeplearningbook.org

Deep Learning courses
Prof. Hung-yi Lee National Taiwan University (NTU) Taipei

Introduction to Deep Learning
...the Machine Learning background...

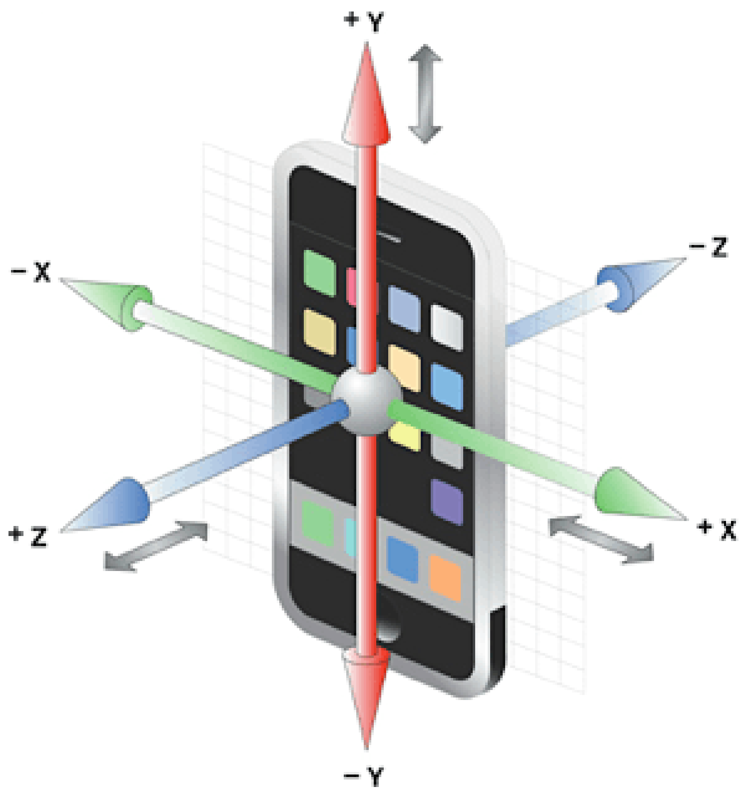


Introduction to Deep Learning
…the Machine Learning background…

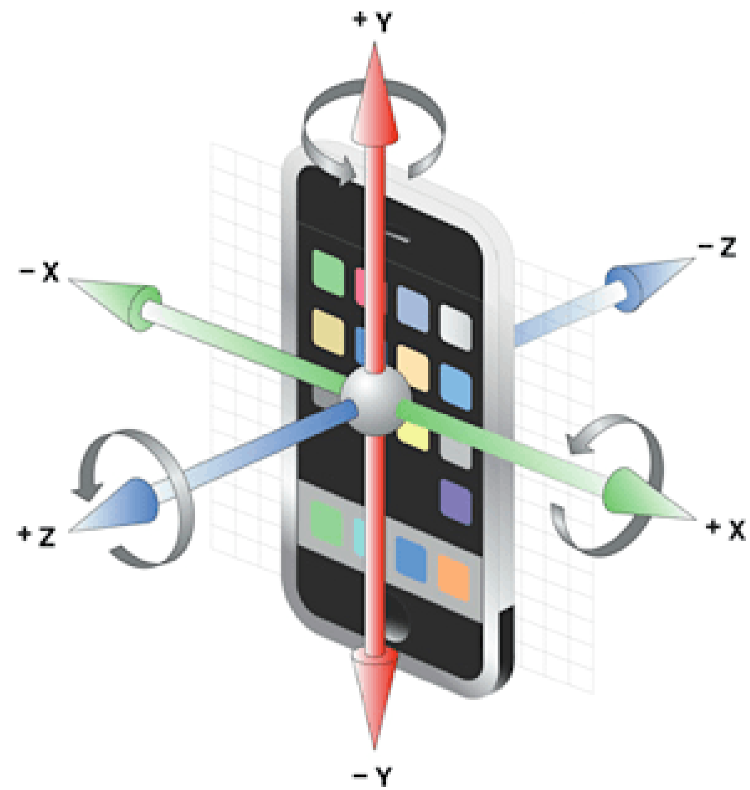


Drivies use case:

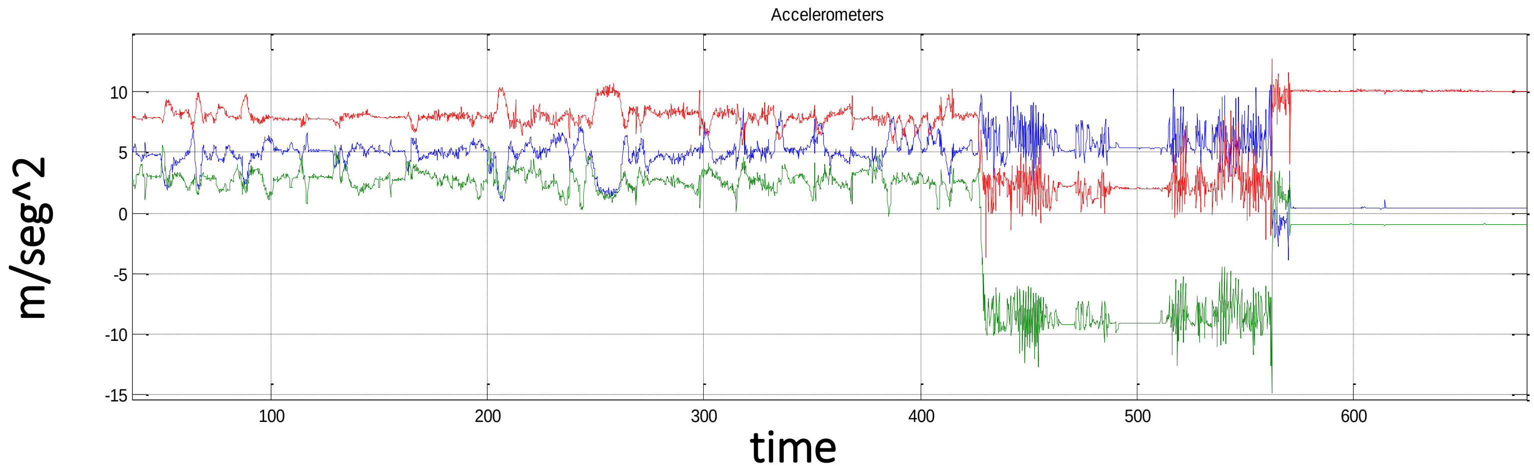
www.driviesapp.com

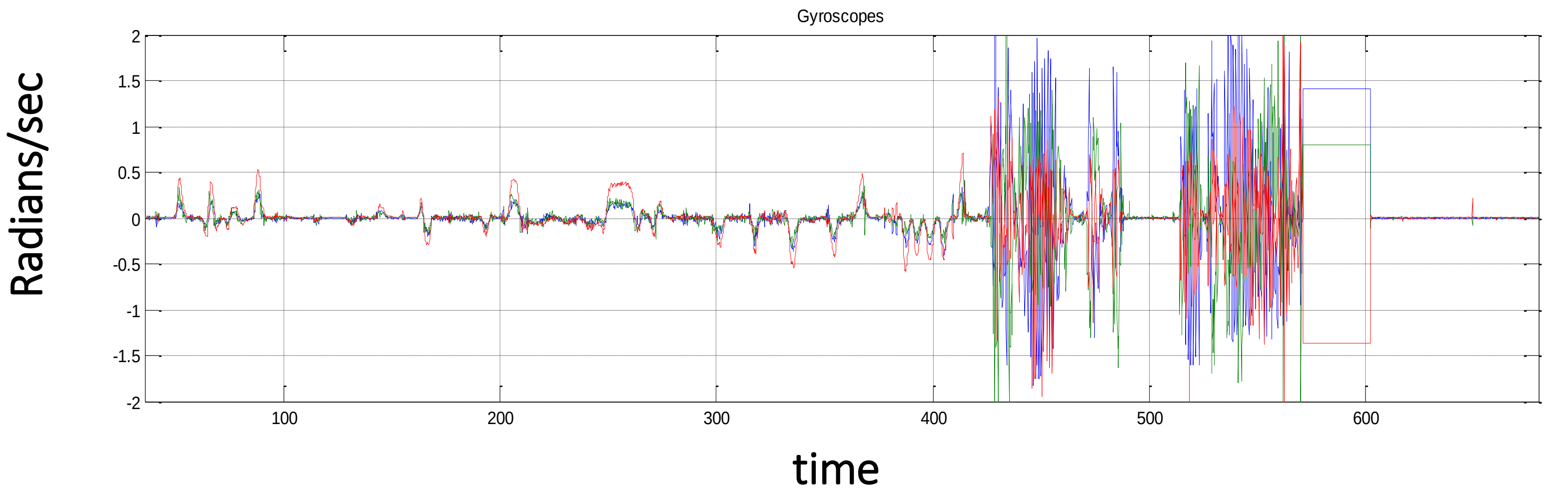


Accelerometer



Gyroscope

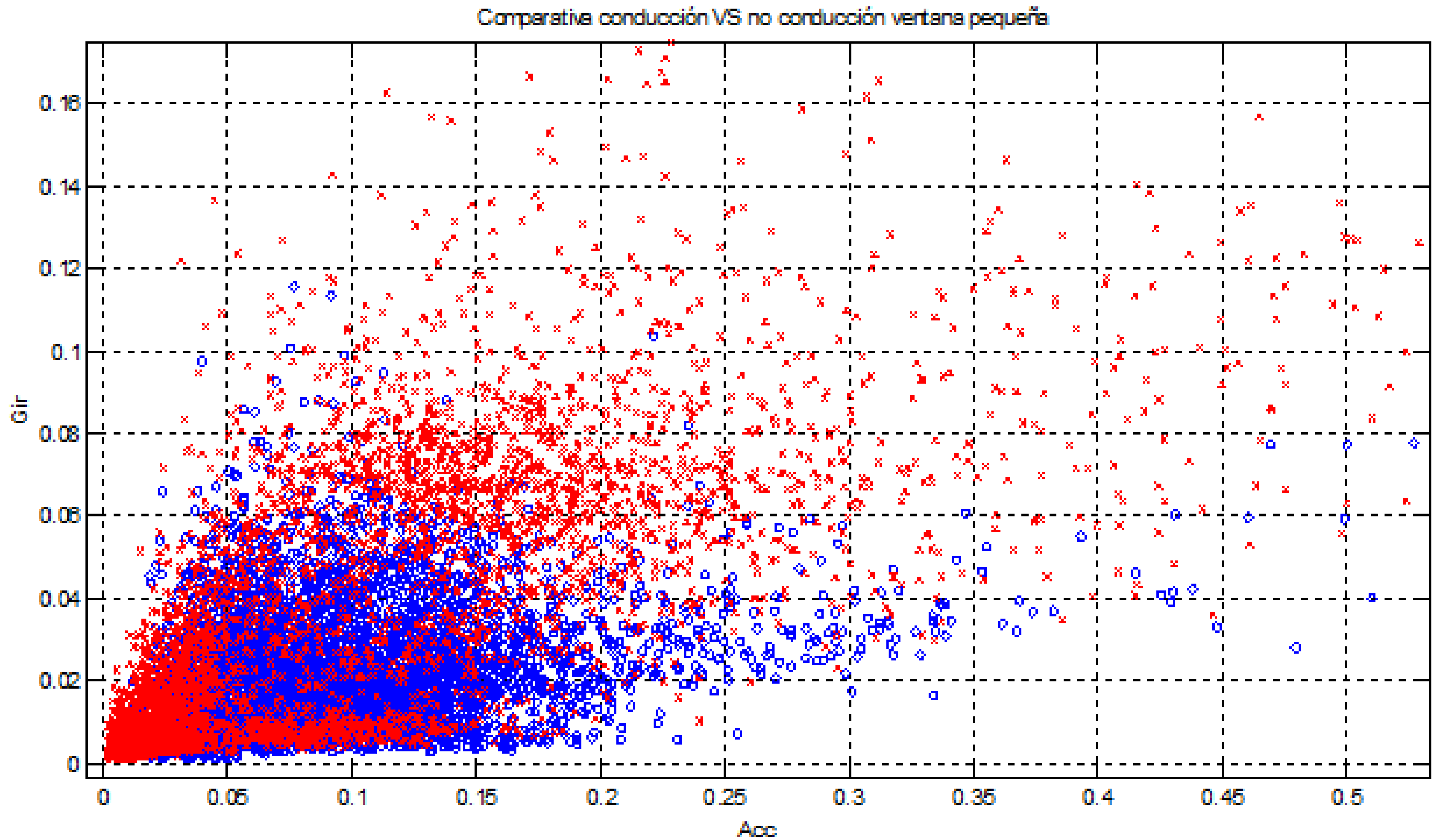

Accelerometers
m/seg^2
10
5
0
-5
-10
-15
100
200
300
400
500
600
time


Gyroscopes
Radians/sec
2
1.5
1
0.5
0
-0.5
-1
-1.5
-2
100
200
300
400
500
600
time

From manual - linear classifiers TO Neural Networks

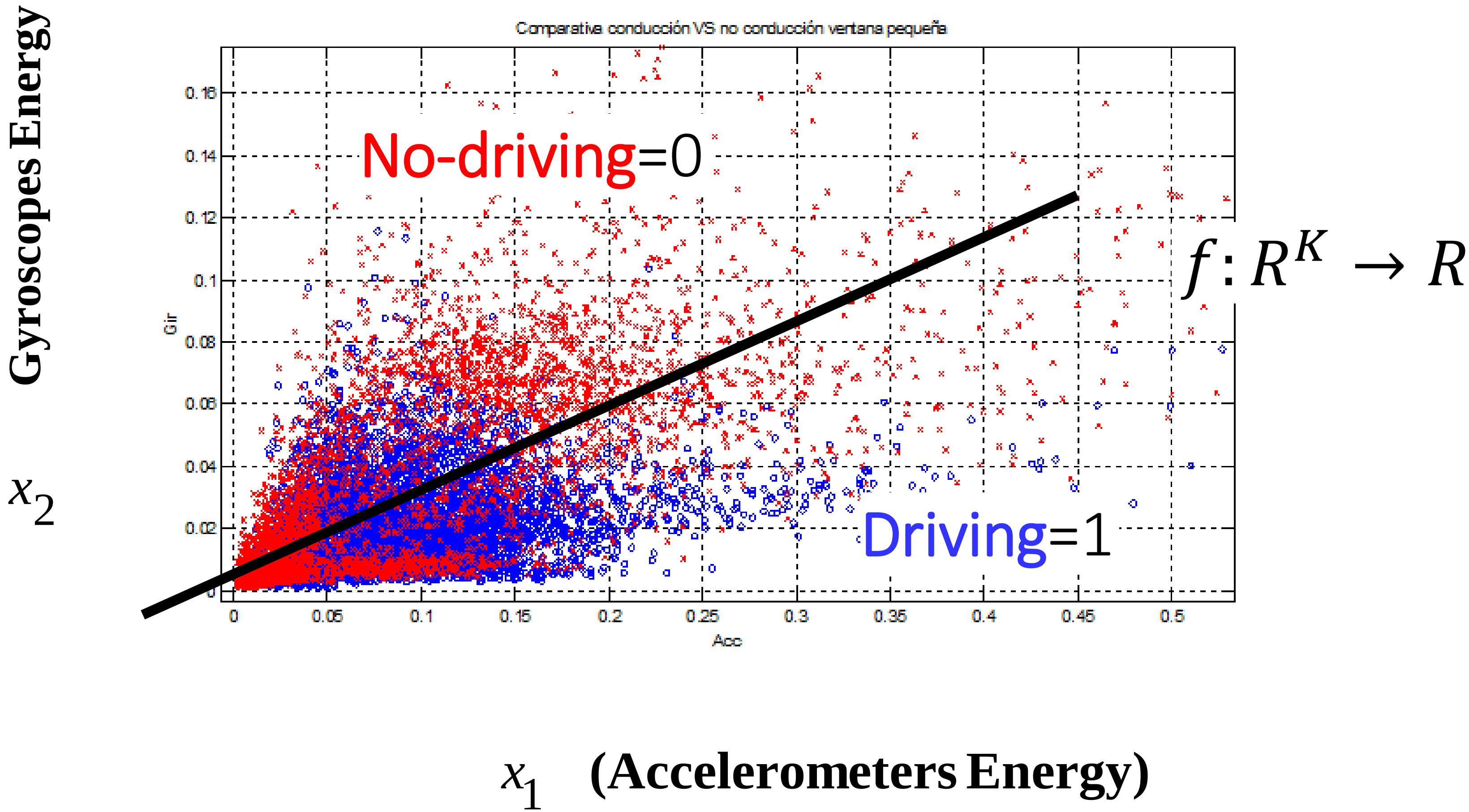
Gyroscopes Energy

x_2

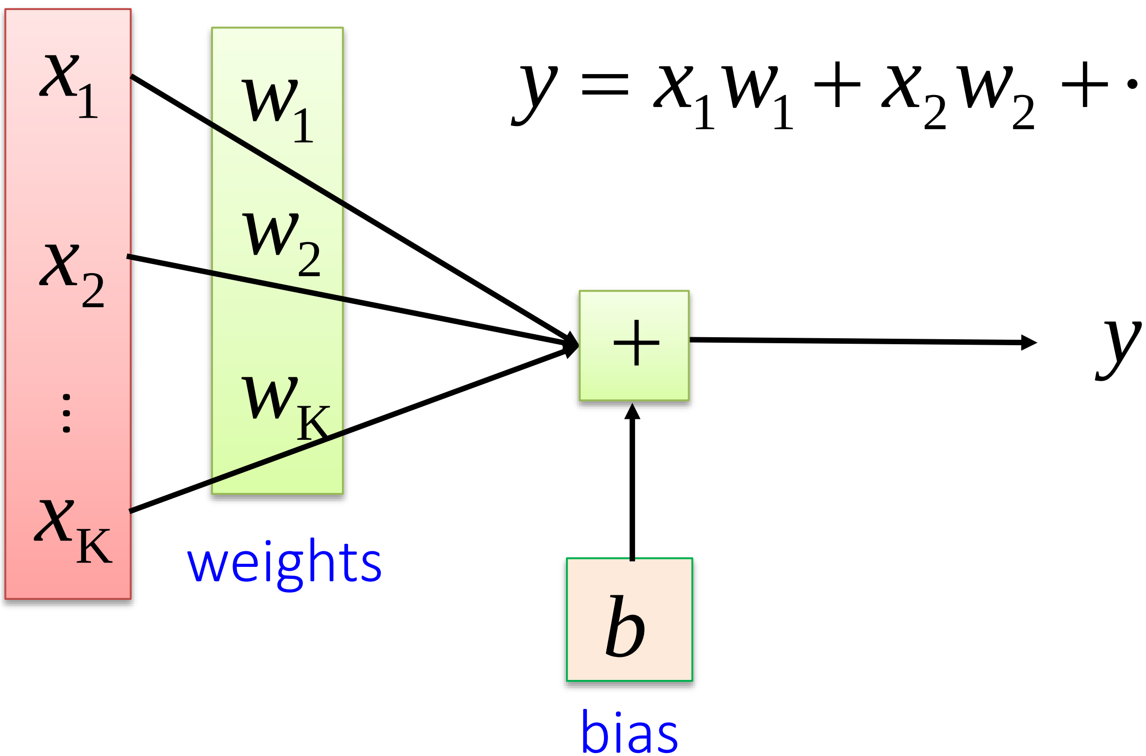


x_1 **: Accelerometers Energy**

Driving detection (yes/no)
= define a **decision function**



A Linear **decision function**



$$y = x_1 w_1 + x_2 w_2 + \cdots + x_K w_K + b$$

A Linear **decision function**

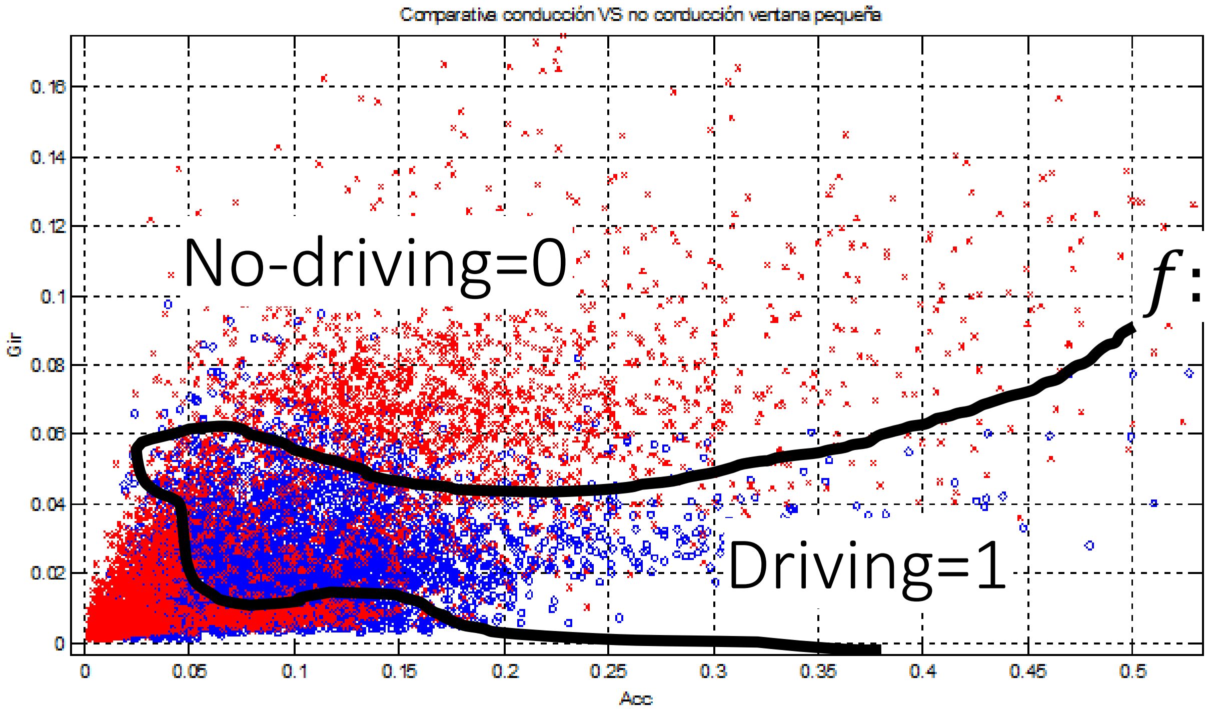
$$y = x_1 w_1 + x_2 w_2 + \cdots + x_K w_K + b$$

$$y = \mathbf{x}^{\mathrm{T}} \mathbf{w}$$

From linear classifiers TO Neural Networks

Nonlinear **decision function?**

x_2



$f: R^K \rightarrow R$

x_1

Non-linear decision functions

$$y = x_1 w_1 + x_1{}^2 w_2 + x_1{}^3 w_3 + x_1 x_2 w_4 + \cdots + b$$

A linear model of transformed inputs:

$$y = \phi(\mathbf{x})^{\mathrm{T}} \mathbf{w}$$

$\phi(\mathbf{x})$ where ϕ is a non linear transformation

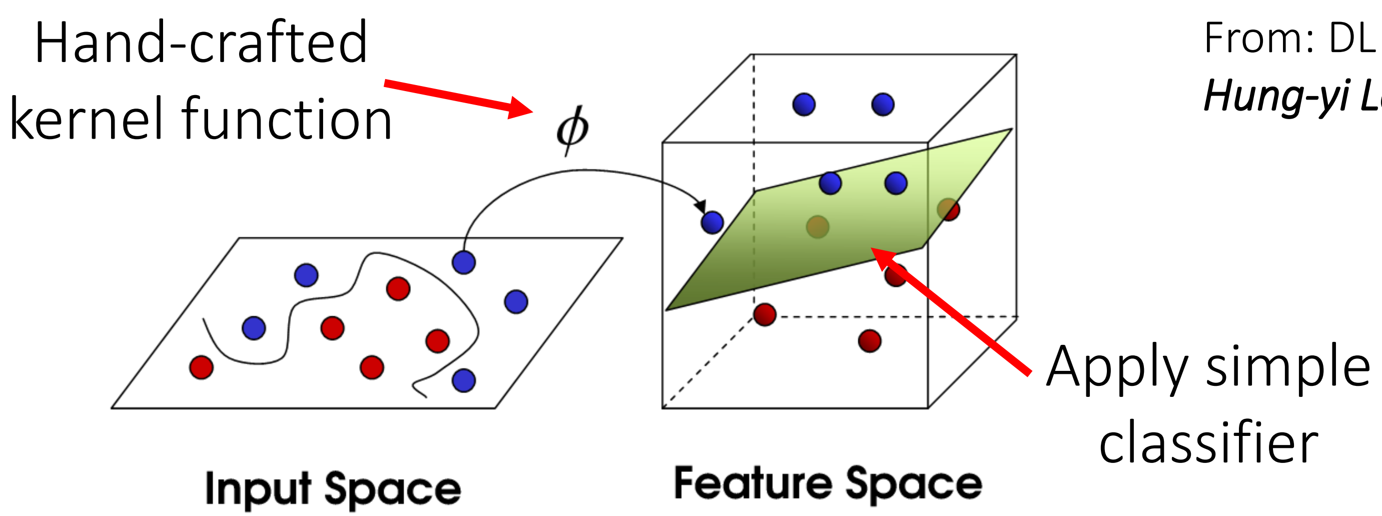
How choosing the mapping $\phi(.)$?

1. To feature engineer $\phi(.)$

2. Use a very generic $\phi(.)$ as kernel machines (e.g. SVM, RBF kernel)

3. The strategy of **deep learning** : to learn $\phi(.)$

SVM



From: DL Tutorial
Hung-yi Lee

Source of image: http://www.gipsa-lab.grenoble-inp.fr/transfert/seminaire/455_Kadri2013Gipsa-lab.pdf

The DL approach: learn $\phi(.)$

Now we have:

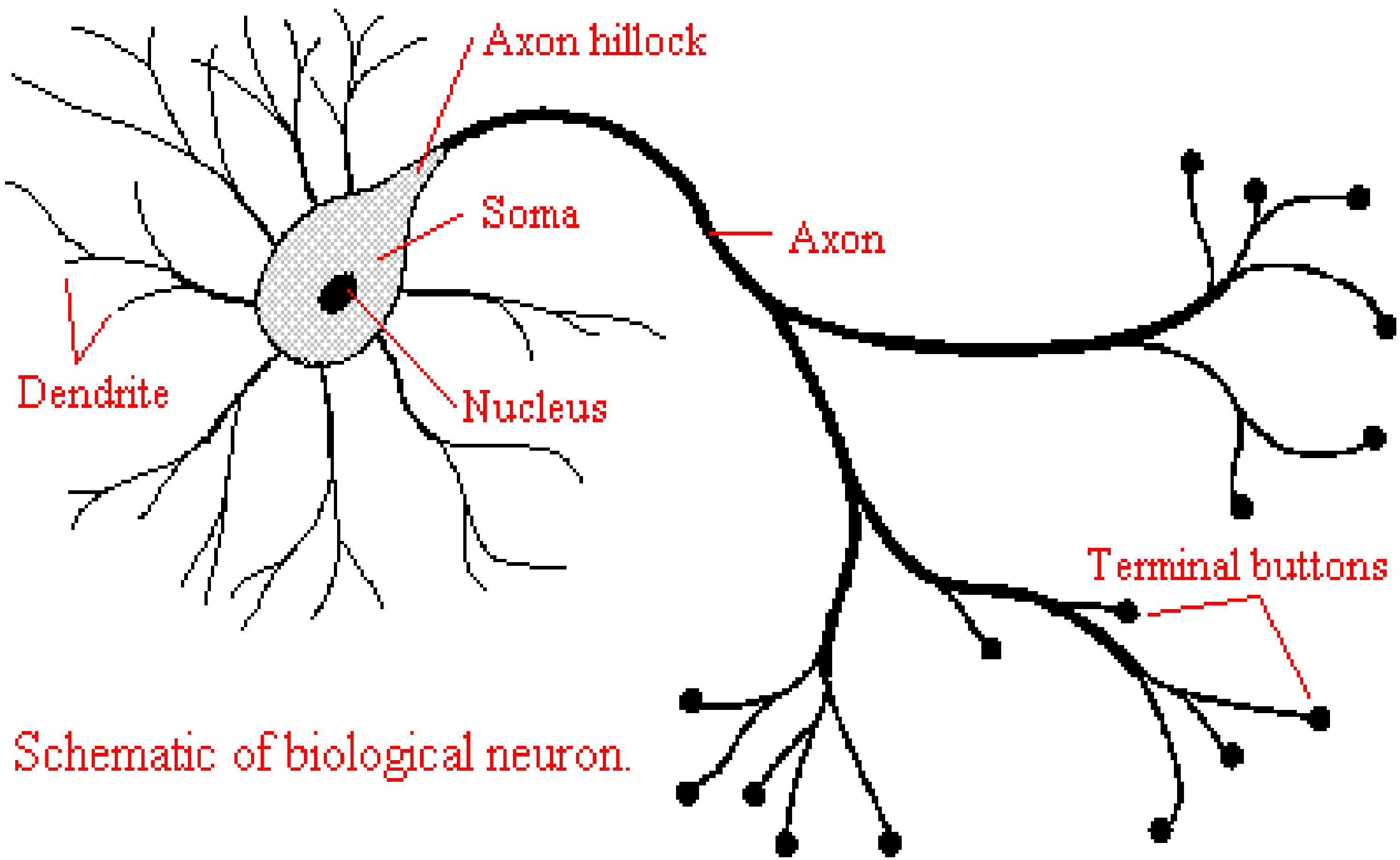
- Parameters $\boldsymbol{\theta}$ that we use to learn $\phi(.)$ from a broad class of functions

- Parameters $\mathbf{w}$ that map $\phi(\mathbf{x})$ to he desired output

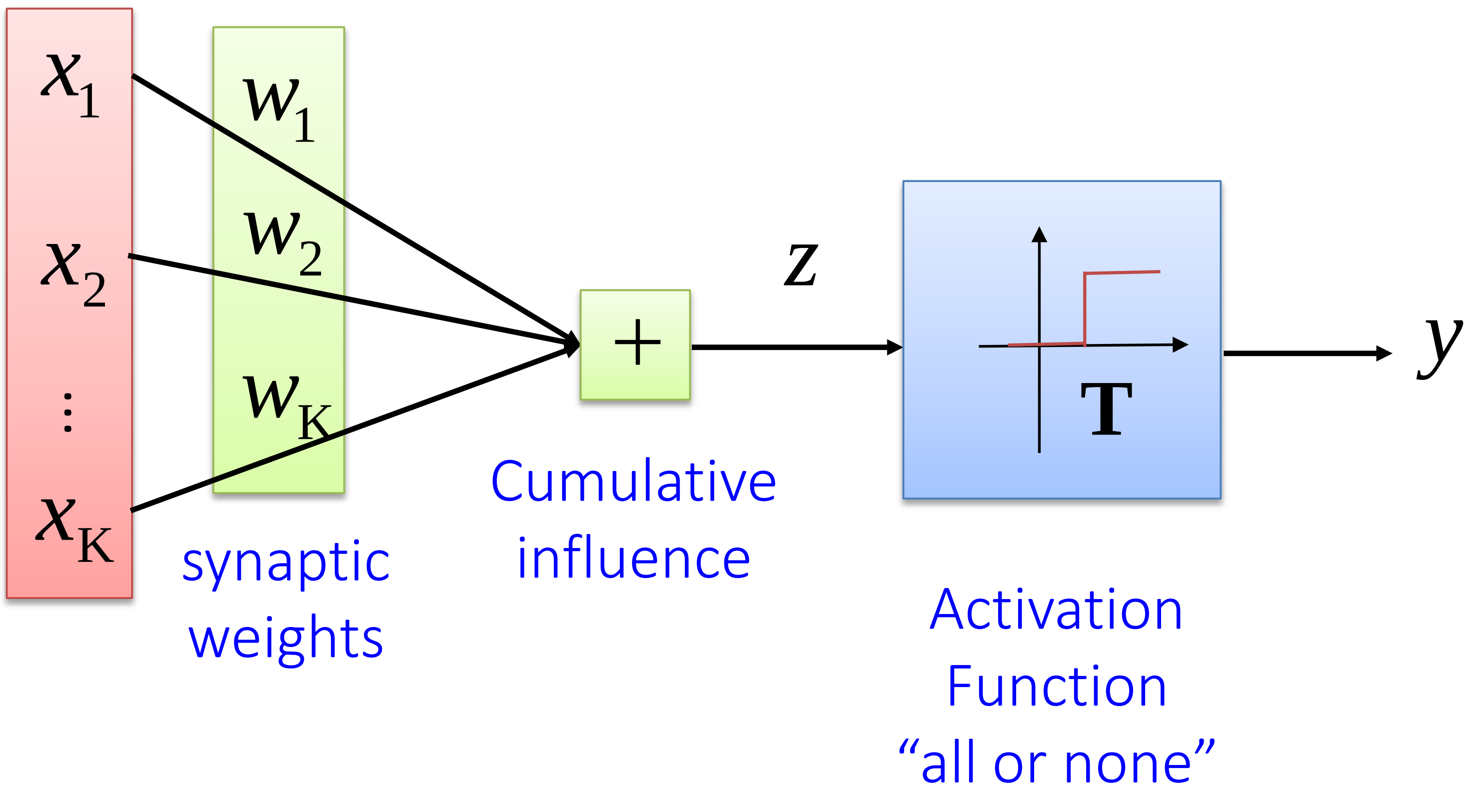
$$y = f(\mathbf{x}; \boldsymbol{\theta}, \mathbf{w}) = \phi(\mathbf{x})^{\mathrm{T}} \mathbf{w}$$

The DL approach: learn $\phi(\mathbf{x})$

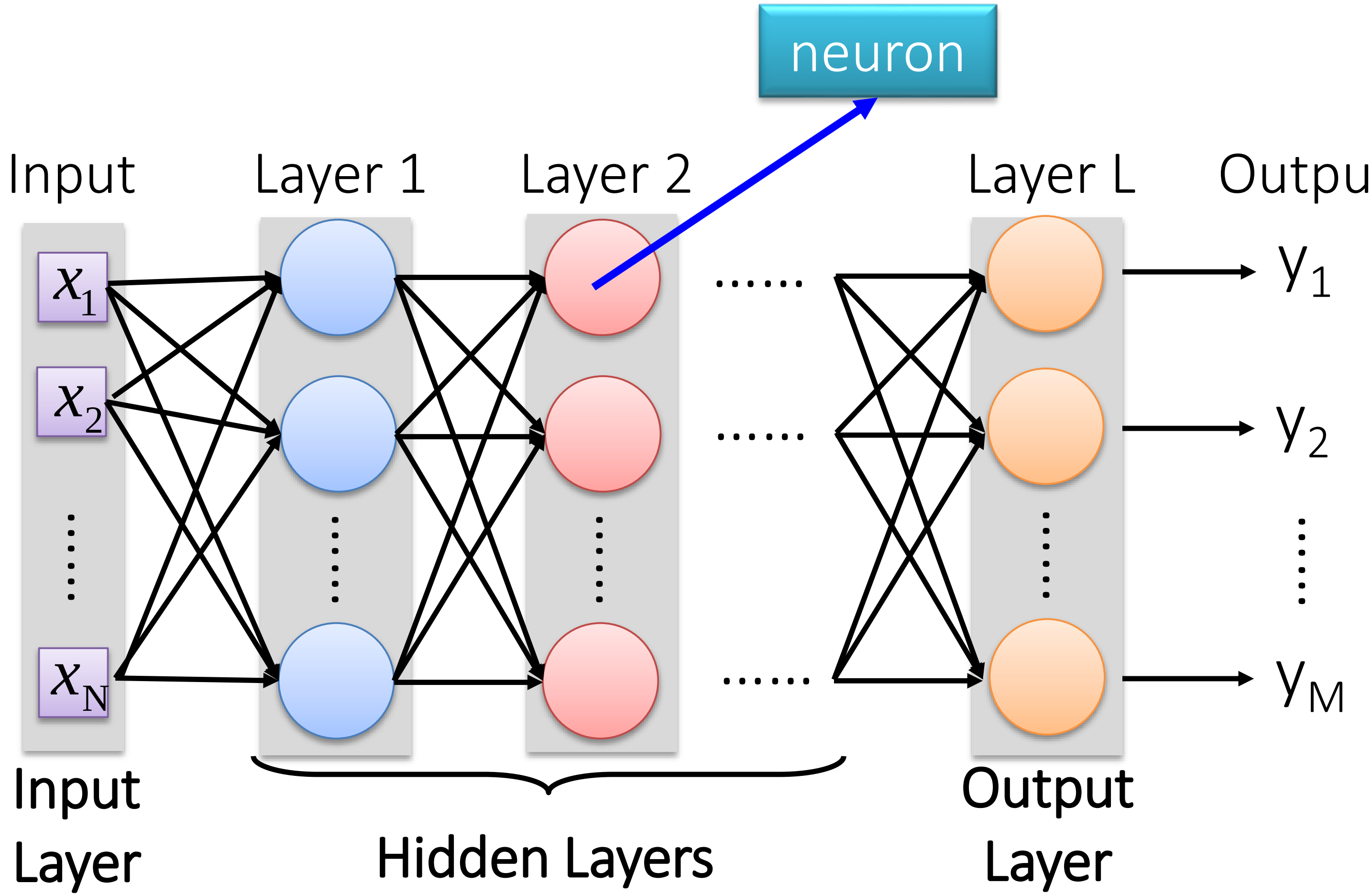
...from a broad class of functions



Neuron approach....



Neural Network (from Hung-yi Lee "Deep Learning Tutorial")



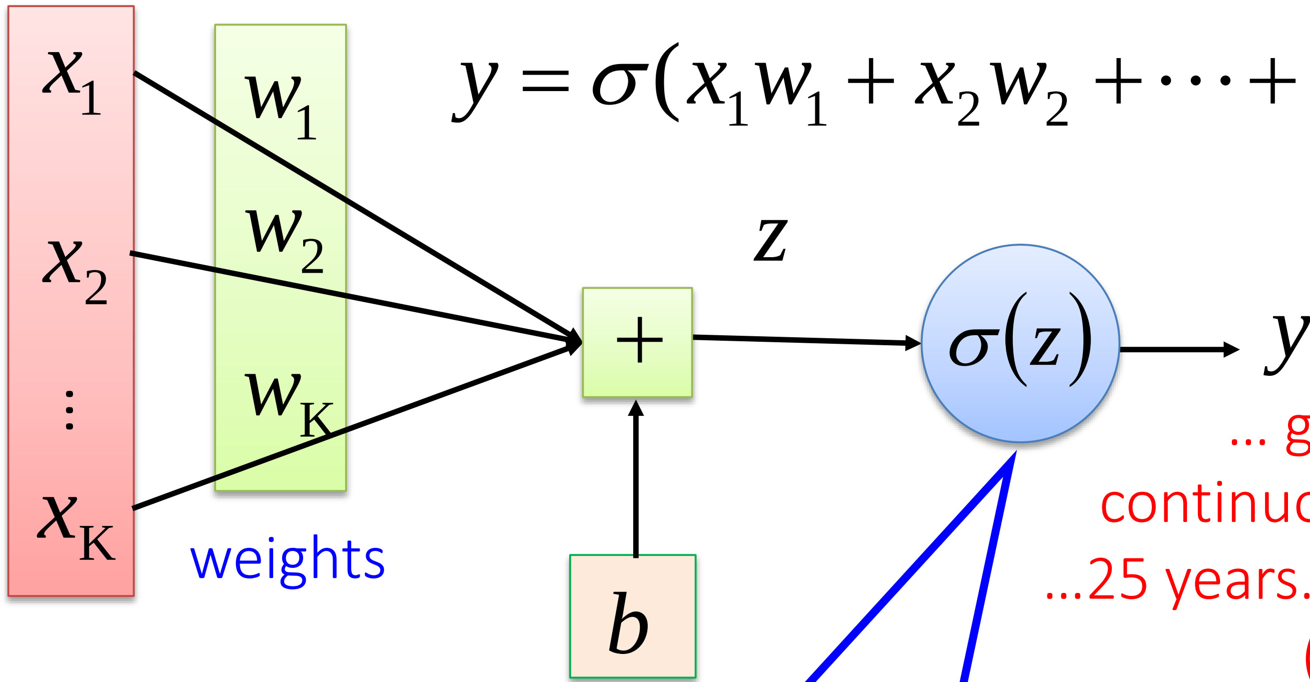
Deep means many hidden layers

From linear classifiers TO Neural Networks

- Parameters $\boldsymbol{\theta}$ that we use to learn $\phi(.)$ from a broad class of functions

- Weights and thresholds are estimated from training examples:
  - to minimize a **loss function** (i.e. similarity between NN outputs y and desired outputs $\hat{y}$)

$$y = \sigma(x_1 w_1 + x_2 w_2 + \cdots + x_K w_K + b)$$



... gradients!

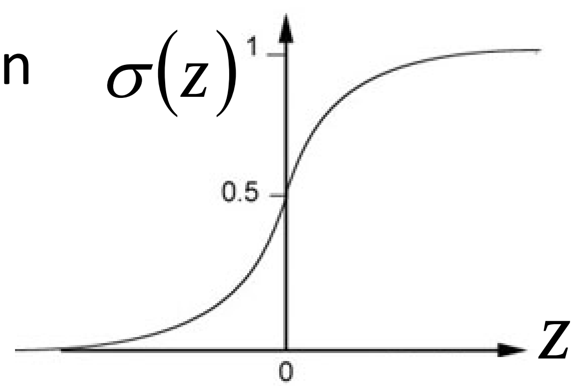
continuous activation!

...25 years... Paul J. Werbos (1974)

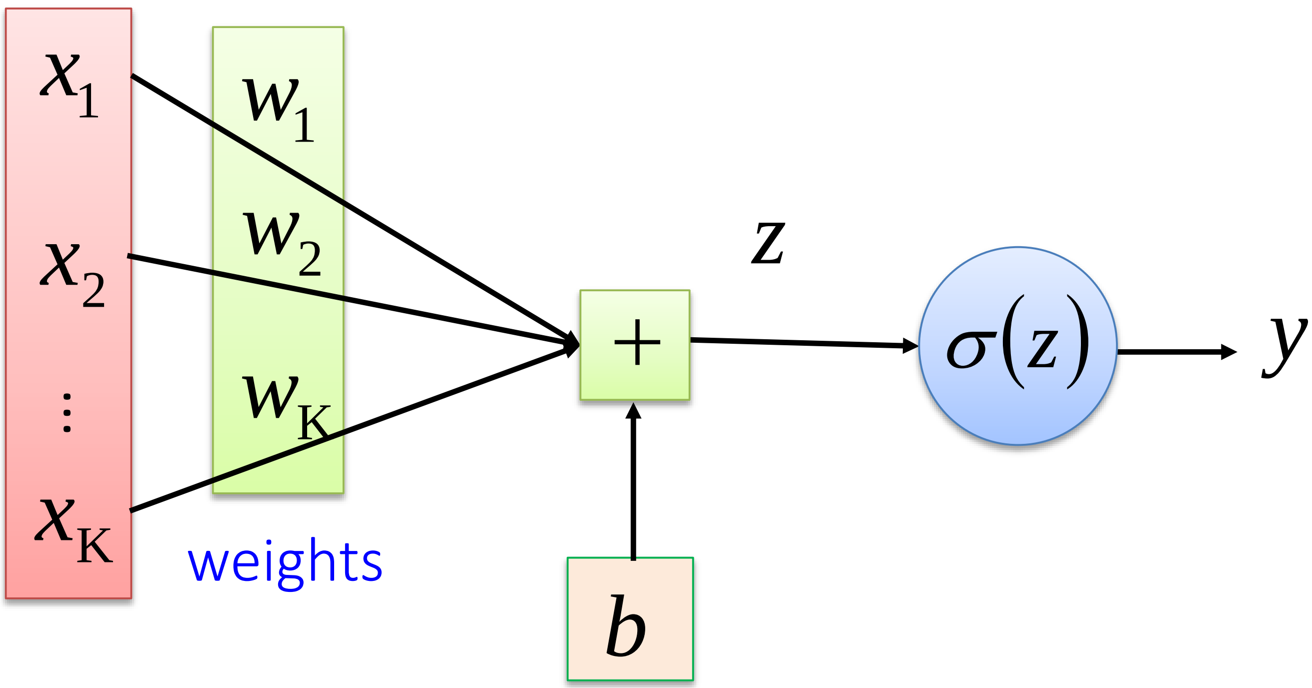
...as Sigmoid function

Sigmoid Function

$$\sigma(z) = \frac{1}{1+e^{-z}}$$



- But recall that this is also logistic regression!



From linear classifiers TO Neural Networks

- So let’s stop here and start playing with

Google TensorFlow

- Library for writing "machine intelligence" algorithms

- Very popular for deep learning and neural networks

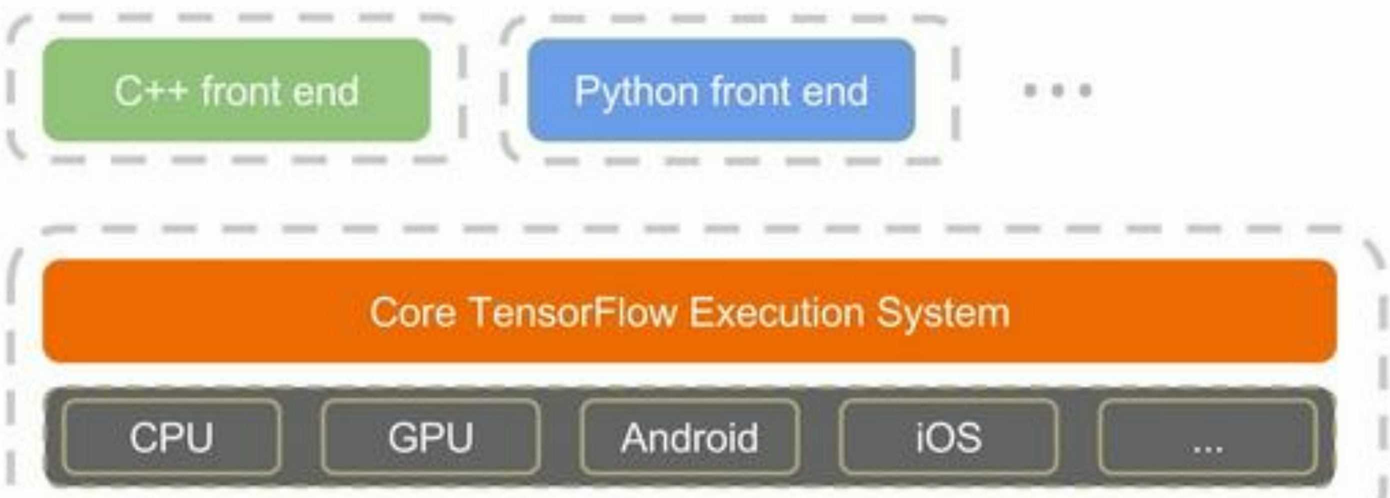
- Can also be used for general purpose numerical computations

- Interface in C++ and Python

TensorFlow: Expressing High-Level ML Computations

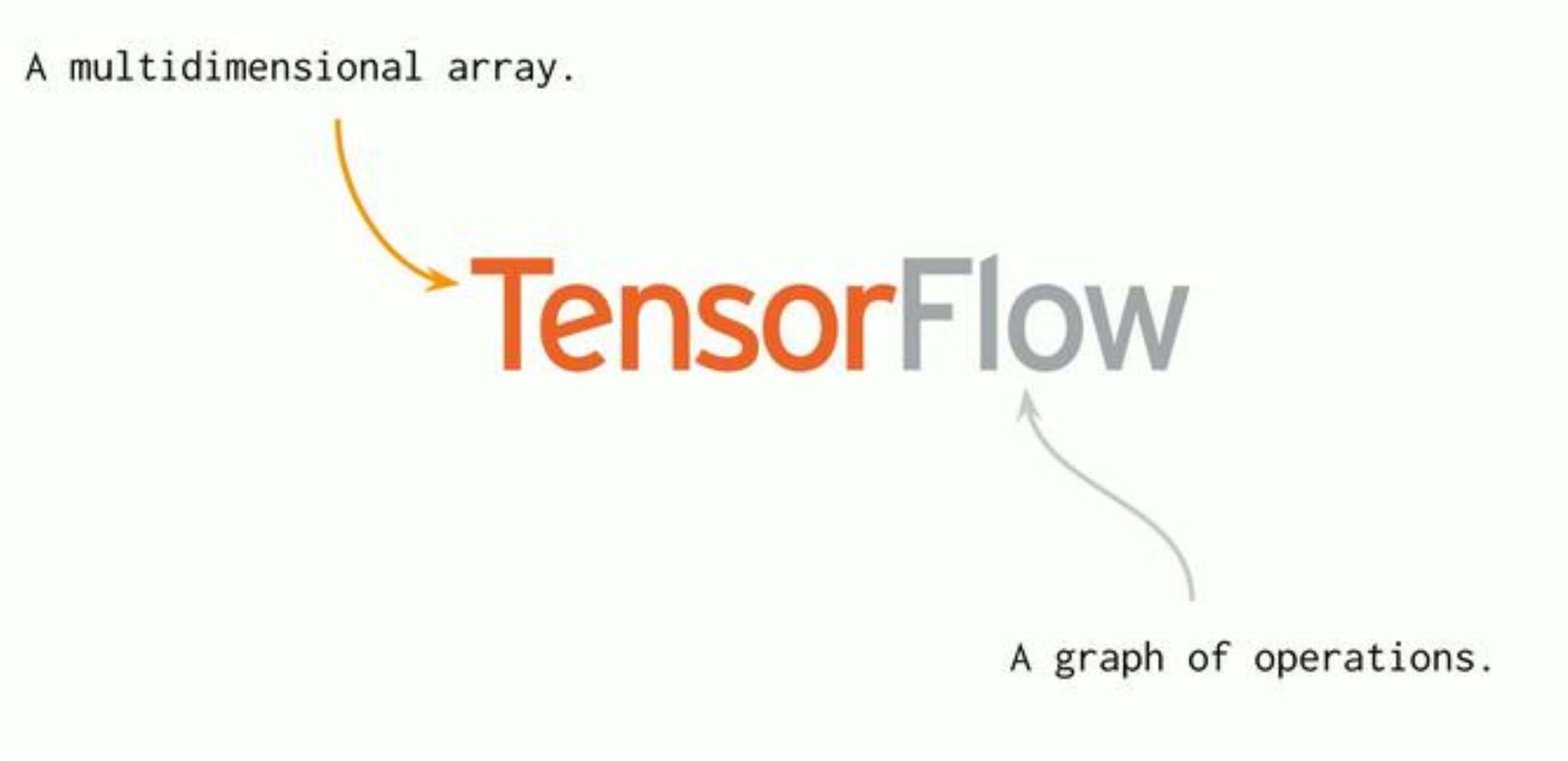
- Core in C++
- Different front ends for specifying/driving the computation

Keras



A word of caution: the APIs in languages other than Python are not yet covered by the API stability promises.

- Python
- C++
- Java
- Go



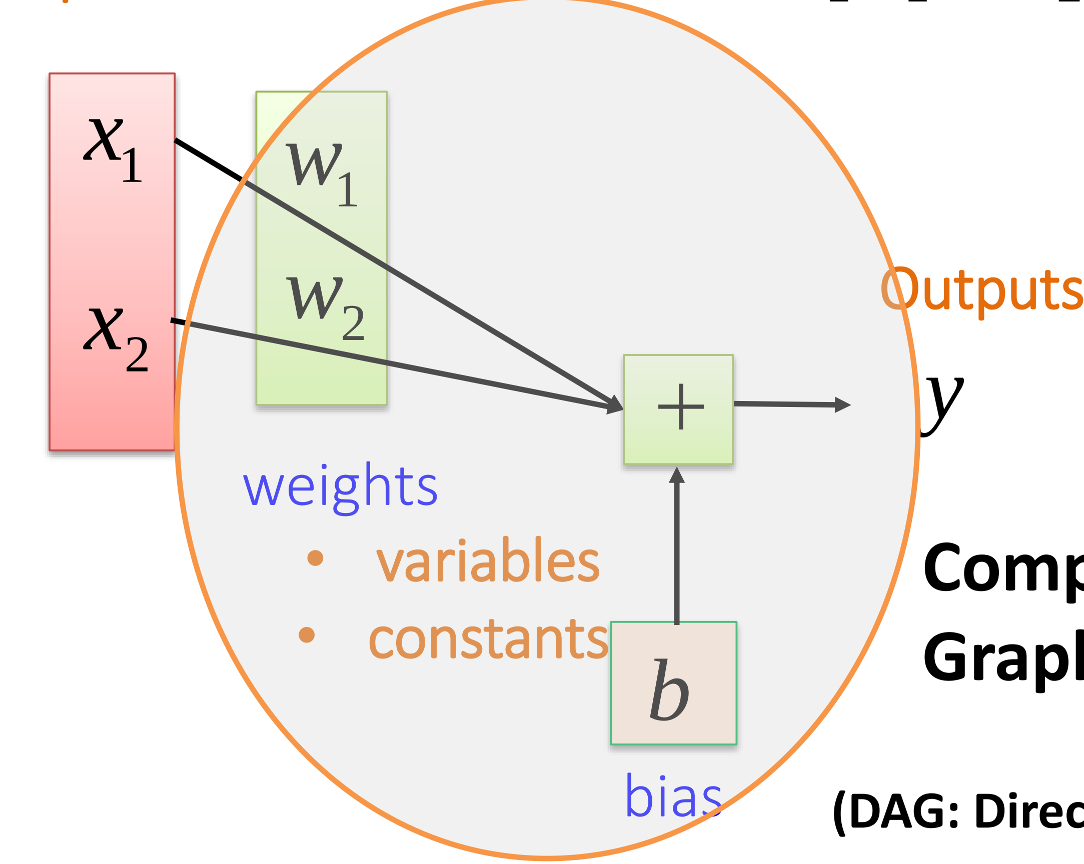
From Mihaela Rosca Talk (Deep Mind)

Generates a computational graph like Theano
Everything about TensorFlow is here:

https://www.tensorflow.org

$$y = x_1 w_1 + x_2 w_2 + b$$

Inputs

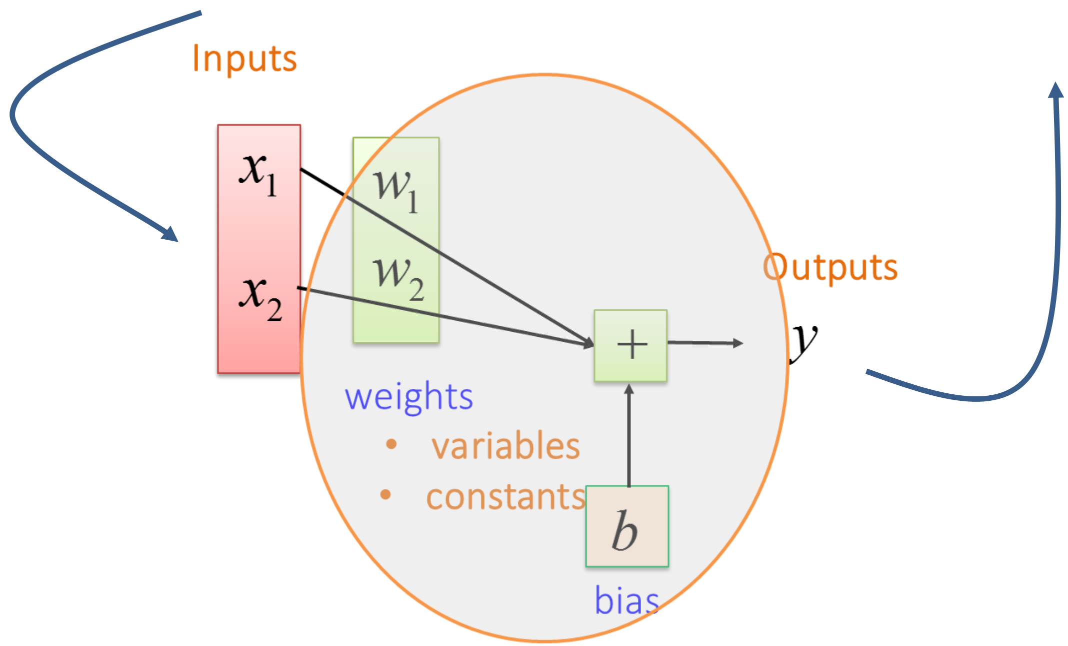


Computational Graph

(DAG: Directed Acyclic Graph)

TensorFlow Recipe:

- Define a series of expressions
- Initialize variables
- Start a session (launch a graph)
- Run the graph, feed some data, fetch some values



TensorFlow Essentials:

Four types of objects make TensorFlow unique from other frameworks

- Session
- Computational graph
- Variables
- Placeholder

Let’s start playing with TensorFlow !!

How?

we have prepared some Interactive Python notebooks (Jupyter) http://jupyter.org/about.html

TSA_IntroTF_1.ipynb

https://github.com/MUIT-TSA/Python/DeepLearning_TF_Keras/

TSA_IntroTF_1.ipynb

We recommend you try:
Colaboratory

It's a Jupyter notebook environment that requires no setup to use and runs entirely in the cloud.

Colaboratory is free to use.

Welcome to Colaboratory!

- Colaboratory is a Google research project created to help disseminate machine learning education and research.

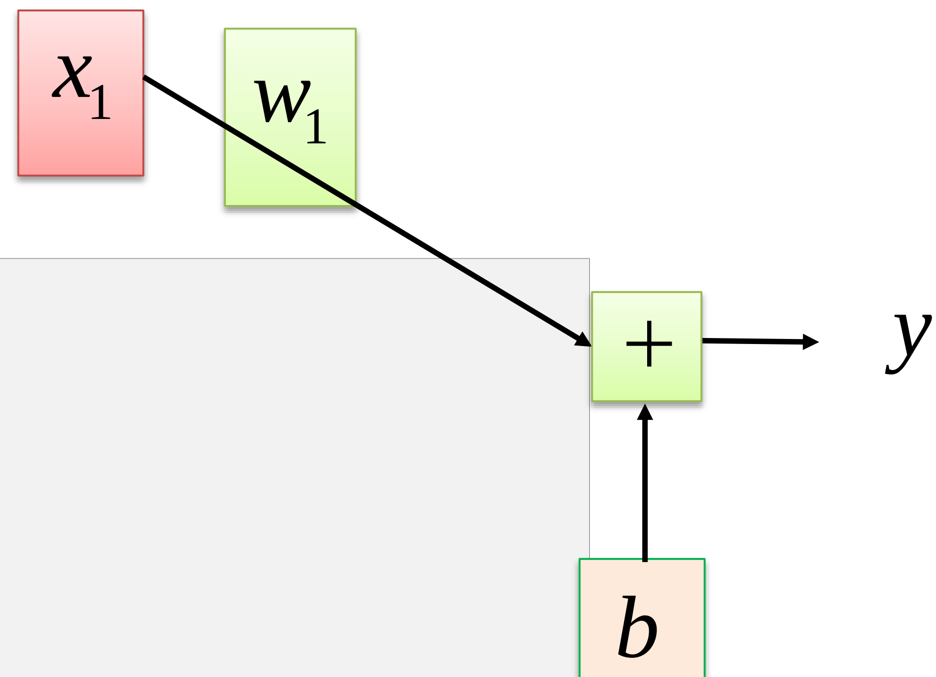
- Colaboratory notebooks are stored in Google Drive and can

GPU Support (NEW!)
Colab now supports running TensorFlow computations on a GPU.

TensorFlow Docs:

"TensorFlow programs are usually structured into a construction phase, that assembles a graph…"

$$y = x_1 w_1 + b$$




```
import tensorflow as tf

x=tf.constant(1.0)
W=tf.constant(6.0)
b=tf.constant(1.5)

y=x*W+b

print(y)
Tensor("add:0", shape=(), dtype=float32)
```

TensorFlow Docs:

- "A <u>Session object</u> encapsulates the environment in which Tensor objects are evaluated…"
- "..and an <u>execution phase</u> that uses a session to execute ops in the graph"

```
import tensorflow as tf

x=tf.constant(1.0)
W=tf.constant(6.0)
b=tf.constant(1.5)

y=x*W+b

with tf.Session() as sess:
        print(sess.run(y))

7.5
```

HOWEVER *Tensorflow* is open new venues!

See: Eager Execution for TensorFlow ¶

Announcing TensorFlow 1.5 Friday, January 26, 2018

With Eager Execution for TensorFlow enabled, you can execute TensorFlow operations immediately as they are called from Python.

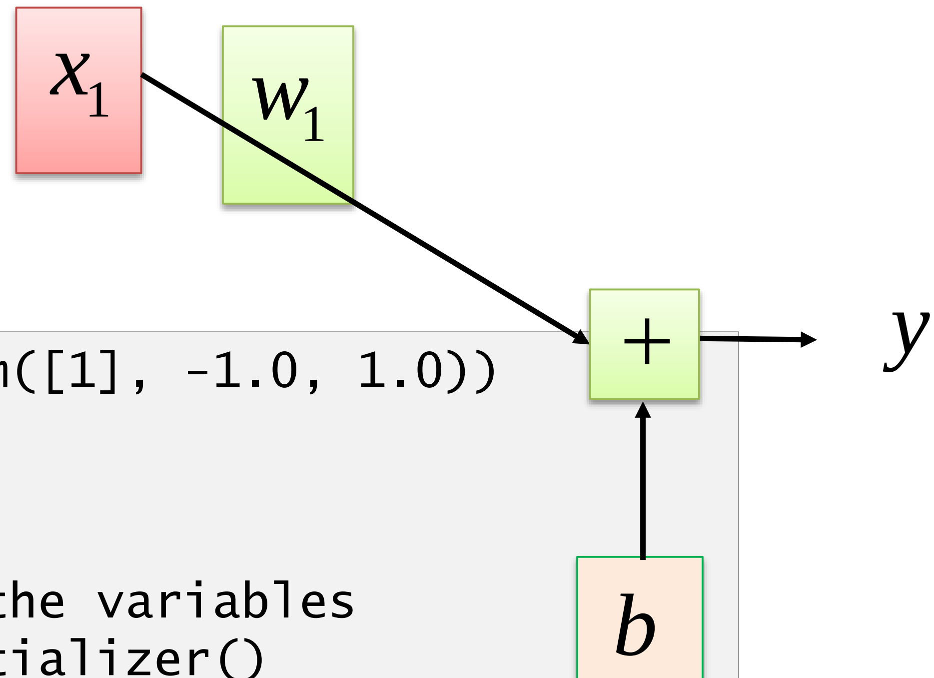
This makes it easier to get started with TensorFlow, and can make research and development more intuitive.

TensorFlow Variables

- "… hold and update **parameters**"

$$y = x_1 w_1 + b$$

- ..and graph (session) is executed several times

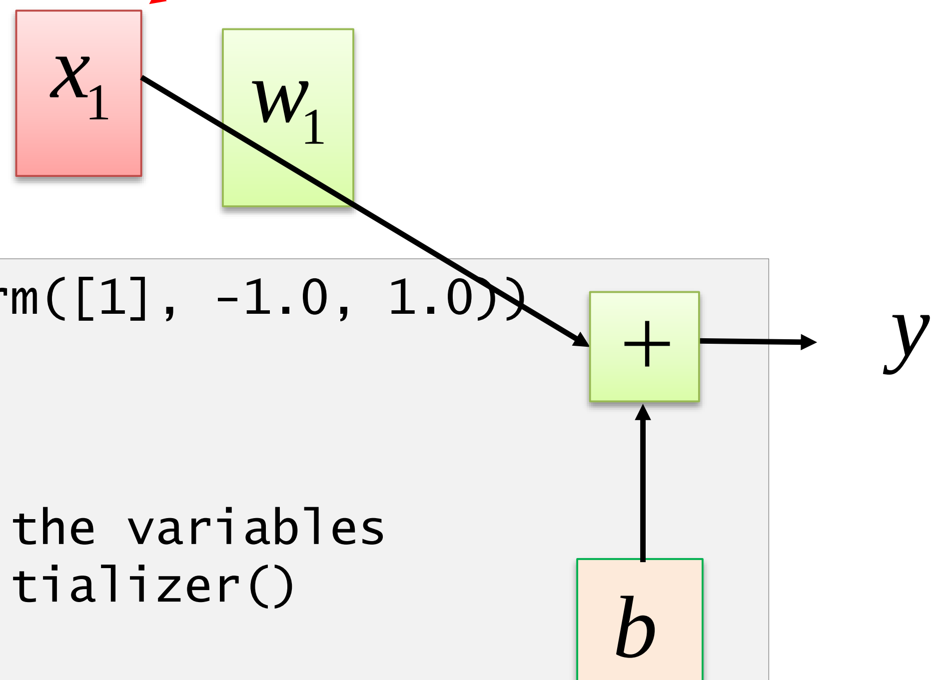


```
W=tf.Variable(tf.random_uniform([1], -1.0, 1.0))
b=tf.Variable(tf.zeros([1]))
x=tf.constant(1.0)

# Before starting, initialize the variables
init = tf.global_variables_initializer()

y=x*W+b

with tf.Session() as sess:
        sess.run(init)
        for step in range(4):
                print(sess.run(y))
```

TensorFlow Placeholders

- "… dummy nodes that provide **entry points** to the computational graph"

$$y = x_1 w_1 + b$$




```
W=tf.Variable(tf.random_uniform([1], -1.0, 1.0))
b=tf.Variable(tf.zeros([1]))
x=tf.placeholder(tf.float32)

# Before starting, initialize the variables
init = tf.global_variables_initializer()

y=x*W+b

with tf.Session() as sess:
      sess.run(init)
      for step in range(4):
            print(sess.run(y, feed_dict=(x:1)))
```

Why TensorFlow?

- Python + Numpy
- Graph based, easy to model
- Faster compile times than Theano
- **Tensorboard** for Visualization
- **http://playground.tensorflow.org**
- Open Sourced
- Data and Model Paralllelism
- Distributed supported

But what's a Tensor?

Tensor Ranks, Shapes, and Types

Briefly: ***A tensor is an array of n-dimension*** *containing the same type of data*

- **Tensor rank** (sometimes referred to as order or degree or n-dimension) is the number of dimensions of the tensor.

Rank	Math entity	Python example
0	Scalar (magnitude only)	`s = 483`
1	Vector (magnitude and direction)	`v = [1.1, 2.2, 3.3]`
2	Matrix (table of numbers)	`m = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]`
3	3-Tensor (cube of numbers)	`t = [[[2], [4], [6]], [[8], [10], [12]], [[14], [16], [18]]]`
n	n-Tensor (you get the idea)	`....`

But what's a Tensor?

Tensor Ranks, Shapes, and Types

- **Tensor shape**: The TensorFlow documentation uses three notational conventions to describe tensor dimensionality: rank, shape, and dimension number.

Rank	Shape	Dimension number	Example
0	[]	0-D	A 0-D tensor. A scalar.
1	[D0]	1-D	A 1-D tensor with shape [5].
2	[D0, D1]	2-D	A 2-D tensor with shape [3, 4].
3	[D0, D1, D2]	3-D	A 3-D tensor with shape [1, 4, 3].
n	[D0, D1, ... Dn-1]	n-D	A tensor with shape [D0, D1, ... Dn-1].

Shapes can be represented via Python lists / tuples of ints, or with the `tf.TensorShape`.

But what's a Tensor?

Tensor Ranks, Shapes, and **Types**

Data type	Python type	Description
DT_FLOAT	tf.float32	32 bits floating point.
DT_DOUBLE	tf.float64	64 bits floating point.
DT_INT8	tf.int8	8 bits signed integer.
DT_INT16	tf.int16	16 bits signed integer.
DT_INT32	tf.int32	32 bits signed integer.
DT_INT64	tf.int64	64 bits signed integer.
DT_UINT8	tf.uint8	8 bits unsigned integer.
DT_UINT16	tf.uint16	16 bits unsigned integer.
DT_STRING	tf.string	Variable length byte arrays. Each element of a Tensor is a byte array.
DT_BOOL	tf.bool	Boolean.

....

....

But what's a Tensor?

A Tensor in TensorFlow has 2 shapes! The **static shape** AND the **dynamic shape**

- The static shape can be read using the tf.Tensor.get_shape() method: this shape is inferred from the operations that were used to create the tensor, and may be partially complete.

```
x = tf.placeholder(tf.int32, shape=[4])
print x.get_shape()
# ==> '(4,)‘
```

- If the static shape is not fully defined, the dynamic shape of a Tensor t can be determined by evaluating tf.shape(t).

```
y, _ = tf.unique(x)
print y.get_shape()
# ==> '(?,)'
```

But what’s a Tensor?

A Tensor in TensorFlow has 2 shapes! The **static shape** AND the **dynamic shape**

- getting dynamic shape of a Tensor t can be determined by evaluating tf.shape(t).

```
y, _ = tf.unique(x)

sess = tf.Session()
print sess.run(y, feed_dict={x: [0, 1, 2, 3]}).shape
# ==> '(4,)‘

print sess.run(y, feed_dict={x: [0, 0, 0, 0]}).shape
# ==> '(1,)'
```

But what's a Tensor?

mathematical operations to manipulate the *tensors*

Operation	Description
tf.add	sum
tf.sub	substraction
tf.mul	multiplication
tf.div	division
tf.mod	module
tf.abs	return the absolute value
tf.neg	return negative value
tf.sign	return the sign
tf.inv	returns the inverse
tf.square	calculates the square
tf.round	returns the nearest integer
tf.sqrt	calculates the square root
tf.pow	calculates the power
tf.exp	calculates the exponential
tf.log	calculates the logarithm

....

But what's a Tensor?

Dealing with shapes is one of the major issues when working with TensorFlow!

Placeholders: Feeding data…

tf.placeholder()

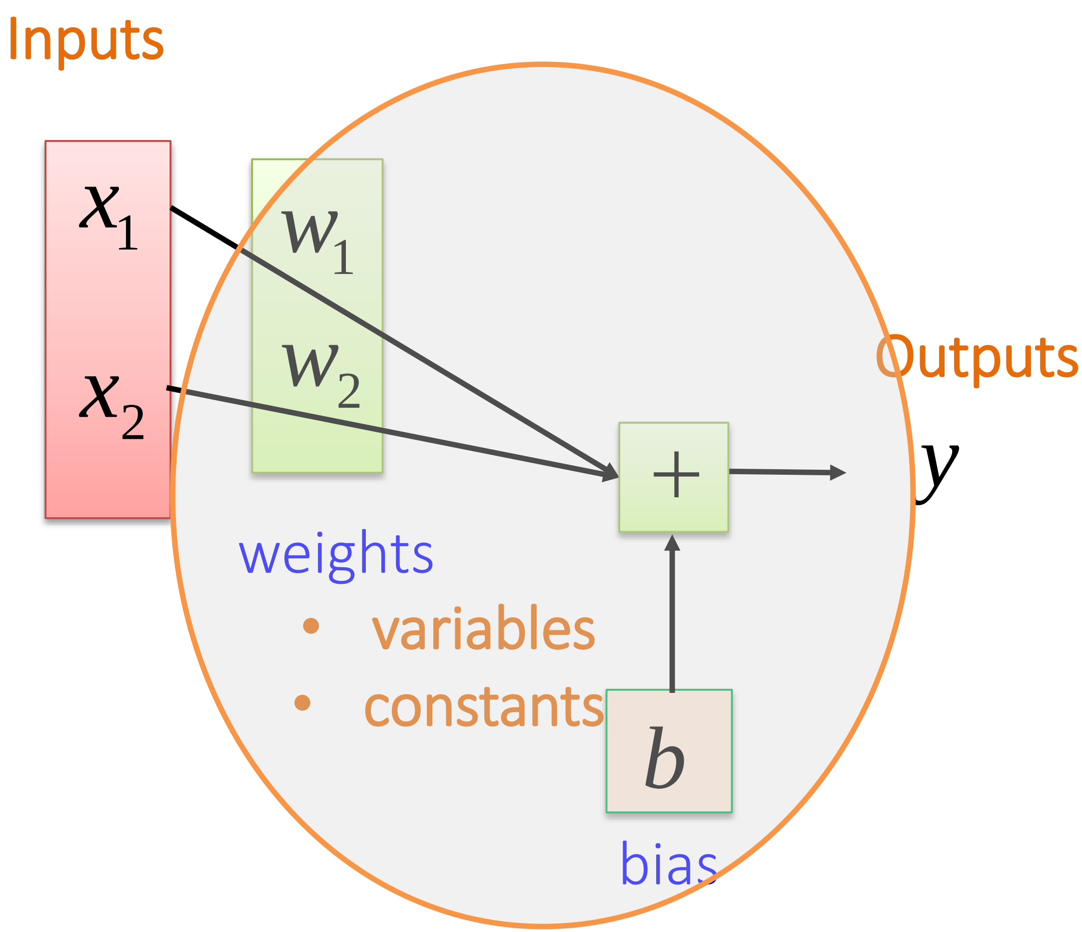
Placeholders allow you to pass in numpy arrays of data

At run time, we feed using a **feed_dic**

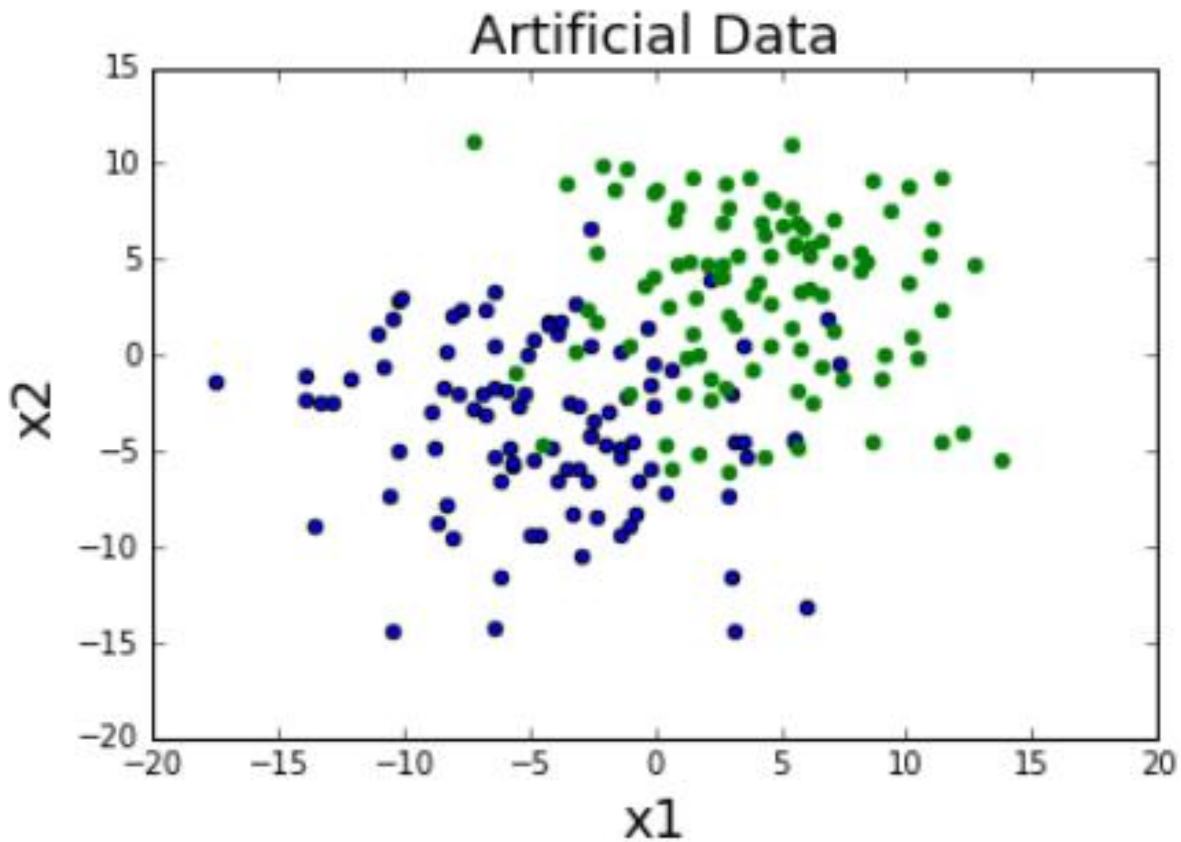
Arguments: data type (mandatory), shape (optional), name (optional)

Let's try a simple linear classifier

$$y = x_1 w_1 + x_2 w_2 + b$$

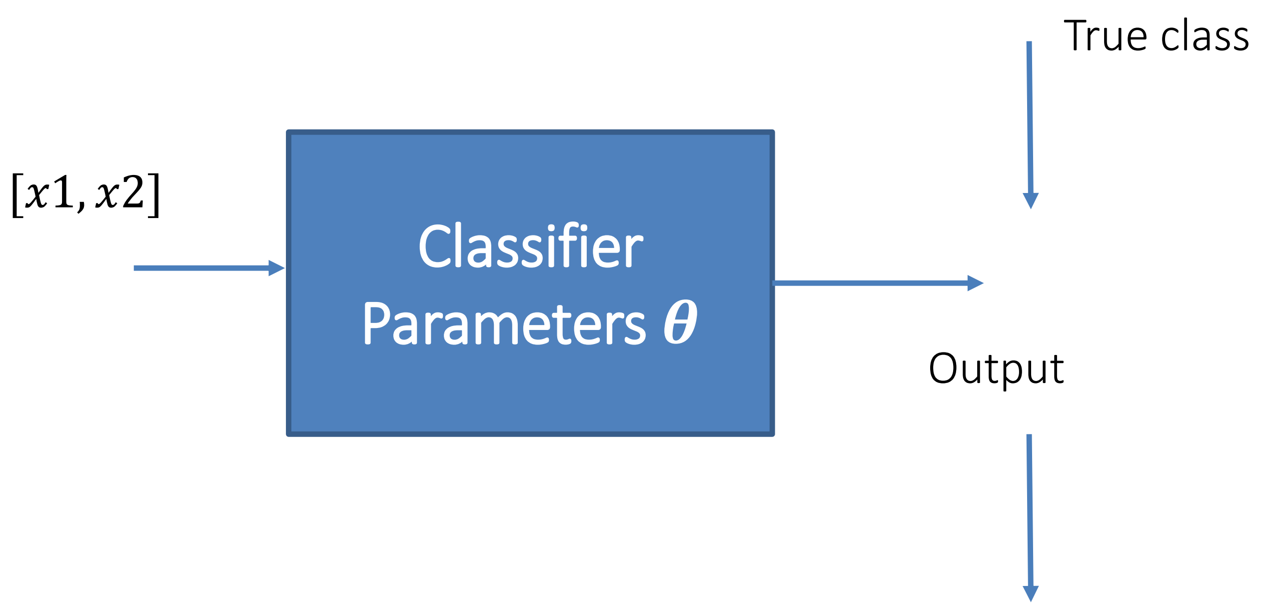


Binary (two-class) classifier
from synthetic data



Binary (two-class) classifier
from synthetic data

Training: How to find the parameters $\boldsymbol{\theta}$: **W**, b ?



OPTIMIZATION =>

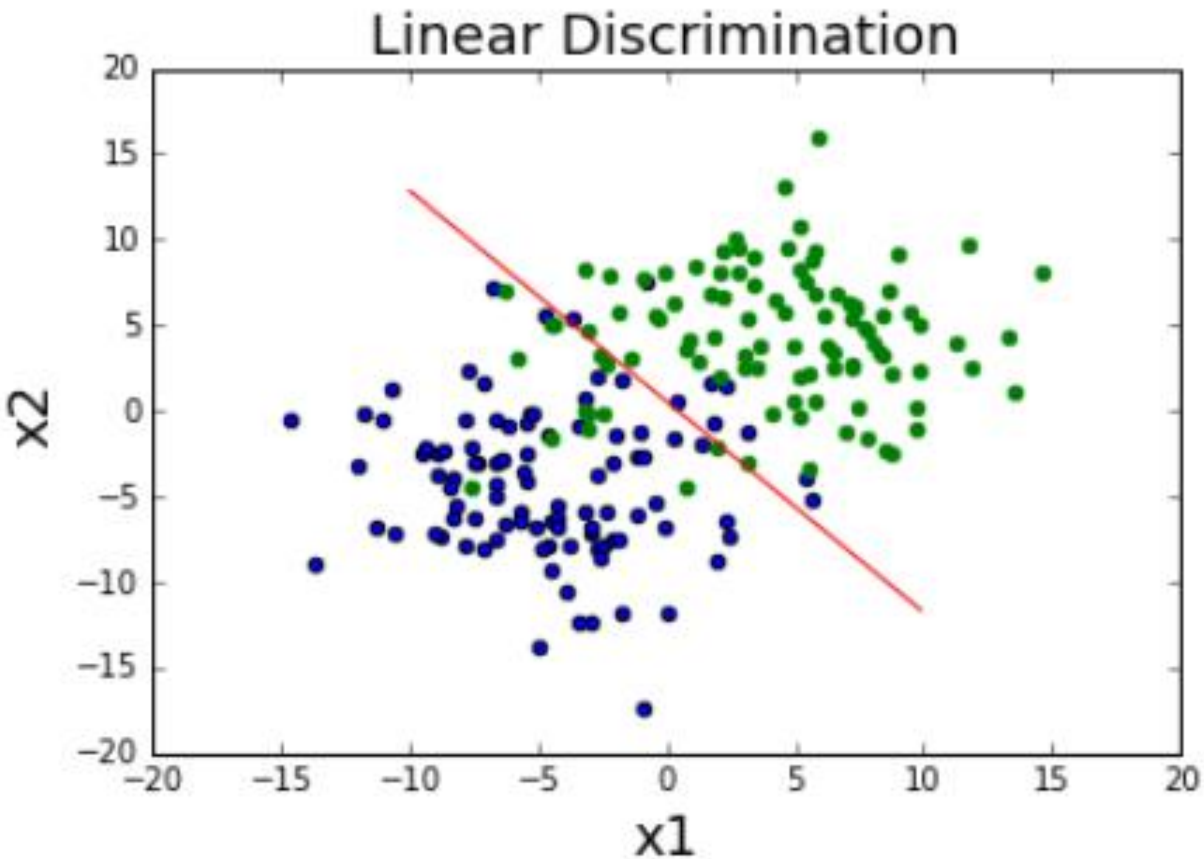
Classification Error
Cost a function of the parameters = C(θ)

TSA_IntroTF_2.ipynb

For particular values of W and b: $$x_1 w_1 + x_2 w_2 + b = 0$$

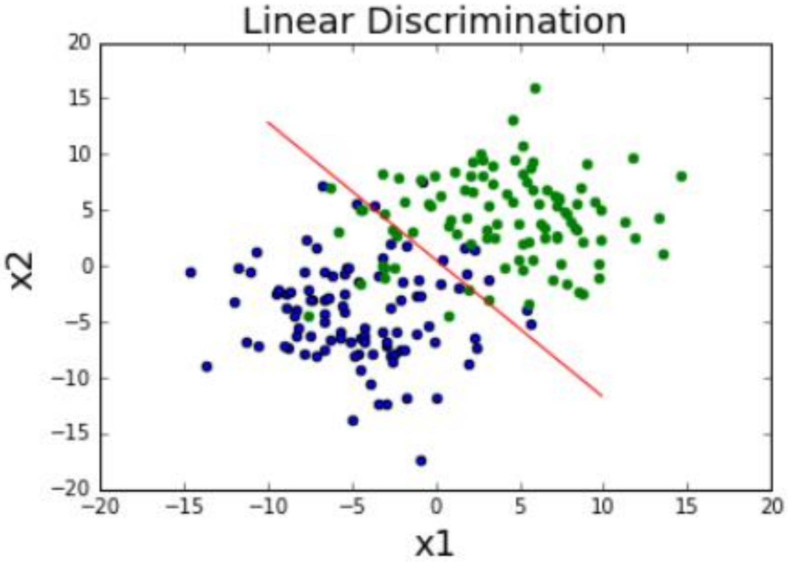
Red line is the set of points defined by:
... arbitrary ....

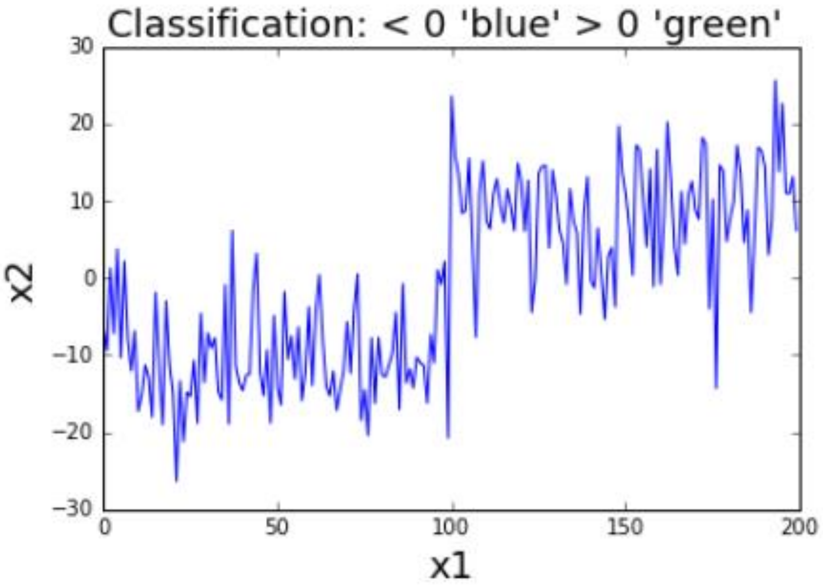
$$x_2 + 1.23 * x_1 - 0.55 = 0$$



Green points will give: $$x_2 + 1.23 * x_1 - 0.55 > 0$$

Blue points will give: $$x_2 + 1.23 * x_1 - 0.55 < 0$$





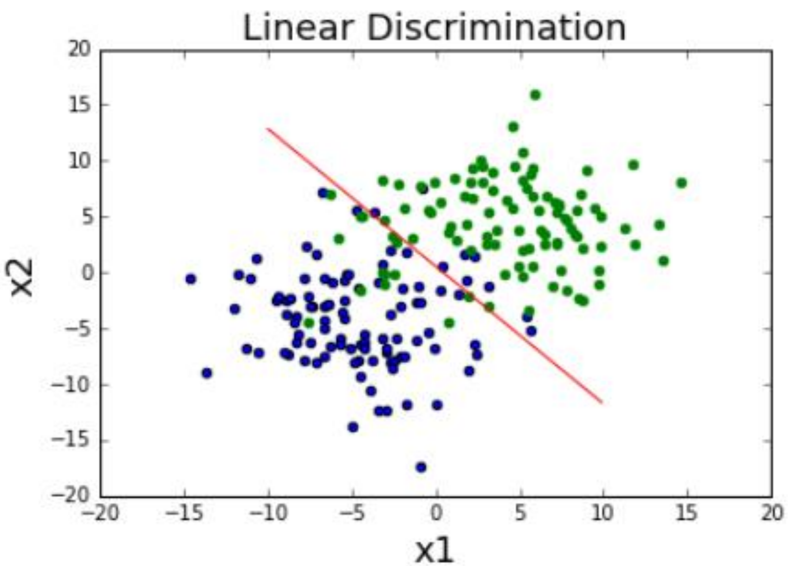
What Cost Function ?

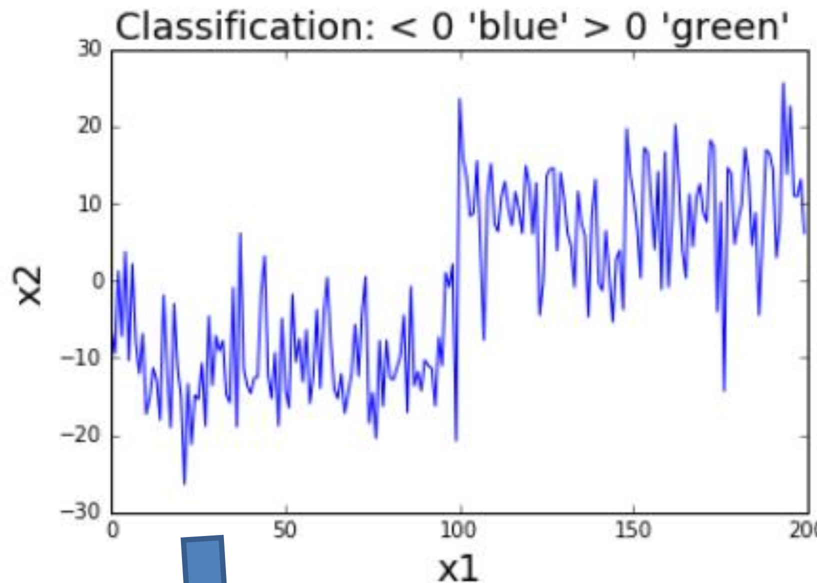
- Classification error (number of errors)? How derivate? gradient?

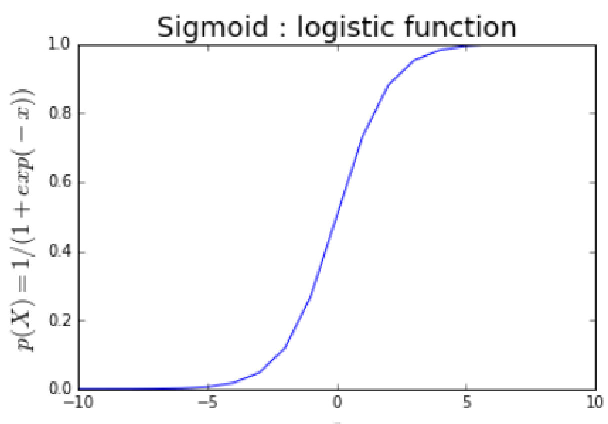
- Better think on terms of probability & look for nice ways to derivate…

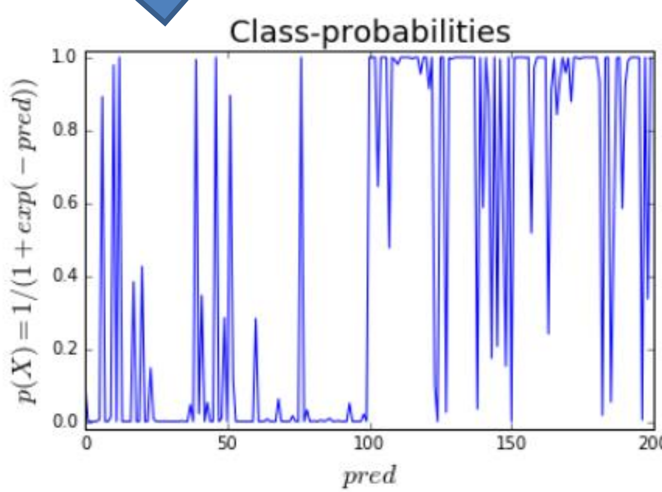
From scores to probabilities:

- Binary classification is easy: use sigmoid or logistic function









Cost function:
How to measure accuracy?

Cross-entropy
and
Maximum Likelihood Estimation

Intuitive approach for a two-class (binary) classifier

Take a data set:
$\{x_n, l_n\}$ of N vectors x_n belonging to two classes (labels l_n =0,1)

Consider our classifier as an estimator of the conditional probability:

- $p(l_n = 1|x_n) = p_{1n}$
- ...and so... $p(l_n = 0|x_n)$ = 1 - p_{1n}

A PERFECT classifier

- If l_n=1 then $p(l_n = 1|x_n) = p_{1n}$=1
- If l_n=0 then $p(l_n = 1|x_n) = p_{1n}$=0

In a single expression:

$$p_{1n}^{l_n}(1 - p_{1n})^{(1-l_n)}$$

In a perfect classifier this must be:
a “probability” always 1 for every data *n*!!!

A PERFECT classifier

All the probabilities/likelihood for each sample should be 1

Maximum Likelihood estimation of θ

$$\theta_{ML} = \prod_{n=1}^{N} p_{1n}^{l_n}(1 - p_{1n})^{(1-l_n)}$$

From that a Cost Function can be obtained taking: **the negative logarithm = Cross-entropy**

$$\text{Cross entropy} = -\sum_{n=1}^{N}(l_n \log(p_{1n}) + (1 - l_n)\log(1 - p_{1n}))$$

Cross entropy loss function

From Maximum Likelihood to Minimum Cross-entropy

$$\text{Cross entropy} = -\sum_{n=1}^{N}(l_n \log(p_{1n}) + (1 - l_n)\log(1 - p_{1n}))$$

Shannon entropy $\boldsymbol{H}(.)$: of uncertainty in a probability distribution P

$$H(x) = -E_{x\sim P}\left[\log\left(P(x)\right)\right]$$

Cross-entropy (two distributions : $Pdata$ and $Pmodel$)

$$-E_{x\sim P_{data}}\left[\log\left(P_{model}(x)\right)\right]$$

Cross entropy loss function

ANOTHER "PRACTICAL" POINT OF VIEW:

$$\text{Cross entropy} = -\sum_{n=1}^{N}\left(l_n \log(p_{1n}) + (1 - l_n)\log(1 - p_{1n})\right)$$

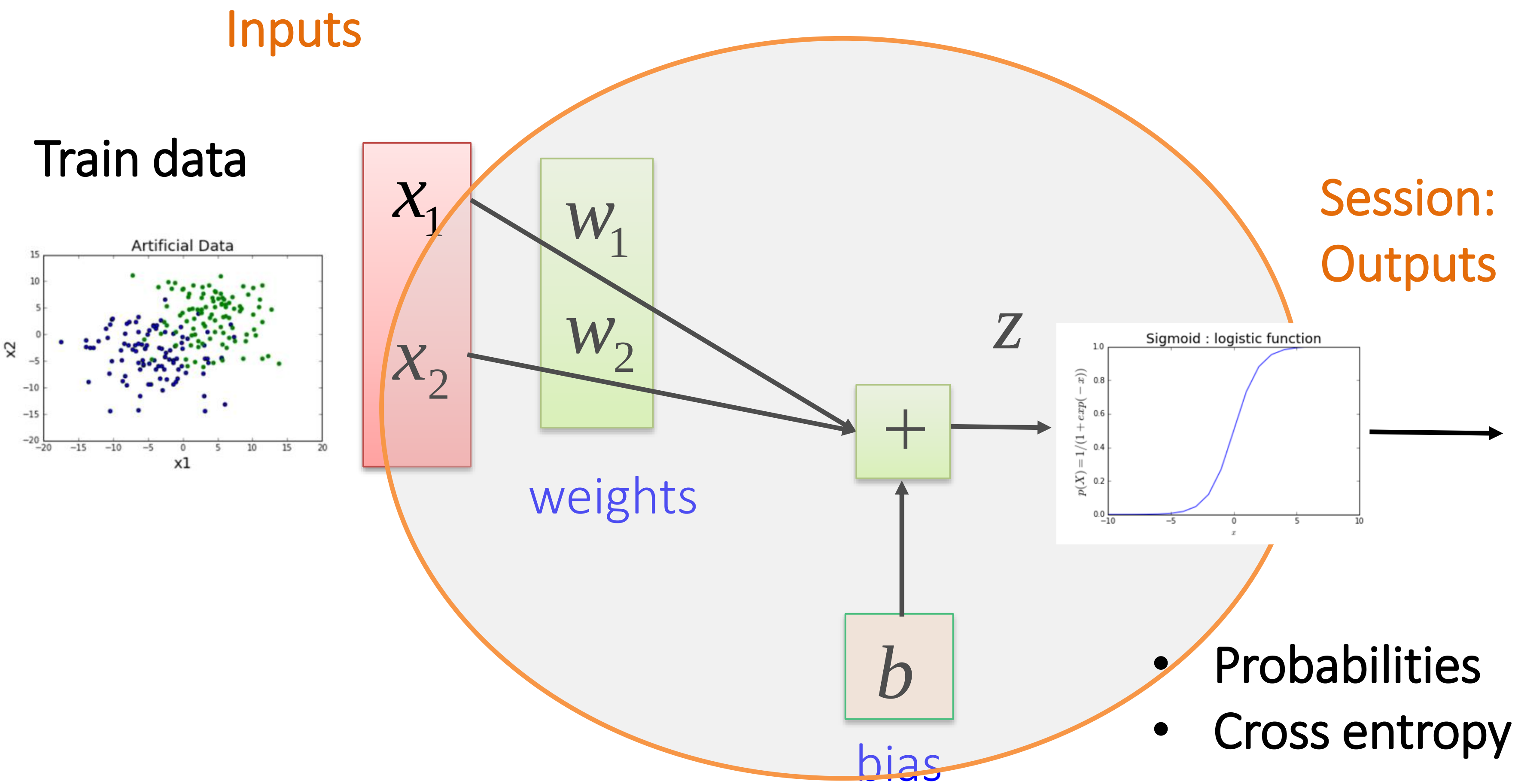
HEAVILY PENALIZES gross errors:

- l_n = 1 and p_{1n} close to 0 log(0)!!

- l_n = 0 and p_{1n} close to 1 !! as $(1 - p_{1n})$ close to 0 log(0)!!

Now let's feed our Training data to a simple linear classifier

$$x_2 + 1.23 * x_1 - 0.55 = z$$



- Probabilities
- Cross entropy

```
W=tf.constant([[1.23], [1.0]],name="weights")
b=tf.constant(-0.55,name="bias")
```

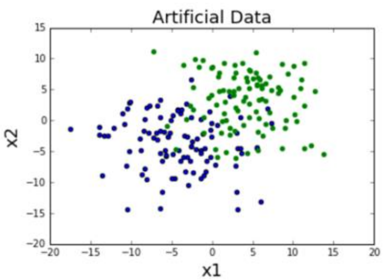
Inputs

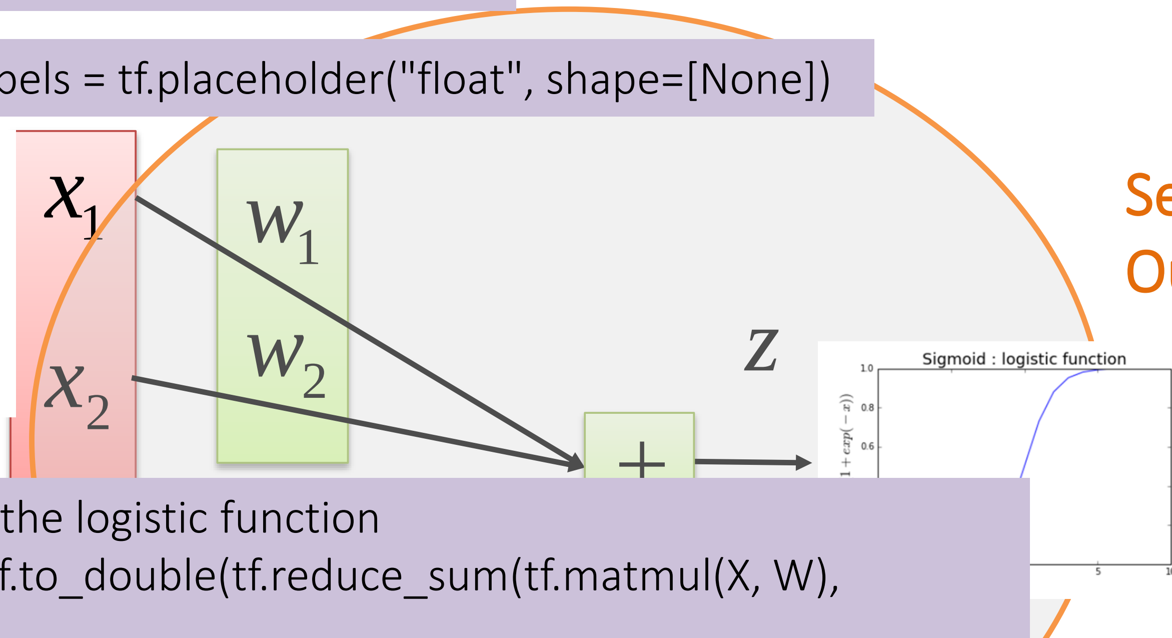
$$x_2 + 1.23 * x_1 - 0.55 = z$$

```
X = tf.placeholder("float", shape=[None, 2]
```

Train data

```
labels = tf.placeholder("float", shape=[None])
```





Session: Outputs

```
# Predictor is now the logistic function
pred = tf.sigmoid(tf.to_double(tf.reduce_sum(tf.matmul(X, W),
axis=[1]) + b))

# Cost function is cross-entropy
cost = -tf.reduce_sum(tf.to_double(labels) * tf.log(pred) + (1-
tf.to_double(labels)) * tf.log(1-pred))
```

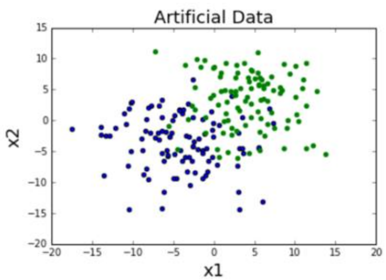
Probabilities
Cross entropy

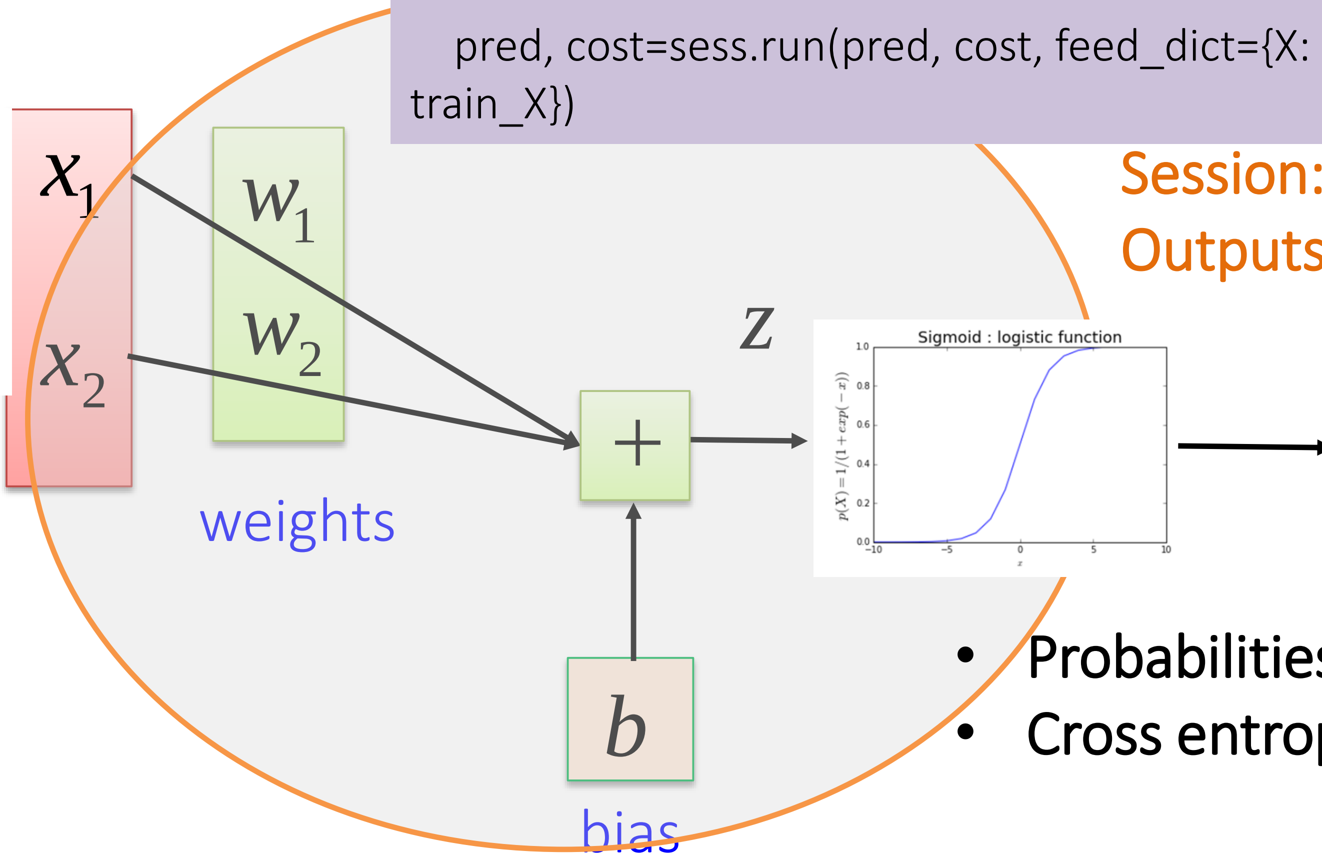
TensorFlow

Inputs

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

    sess.run(init)

    pred, cost=sess.run(pred, cost, feed_dict={X: train_X})
```

Train data





Session: Outputs

- Probabilities
- Cross entropy

Check that our results are the same as before

How to get the best W and b values?

(that is: those who give you the lowest cost)

GradientDescentOptimizer(learning_rate).minimize(cost)

Next Slides are from:
Deep Learning Tutorial

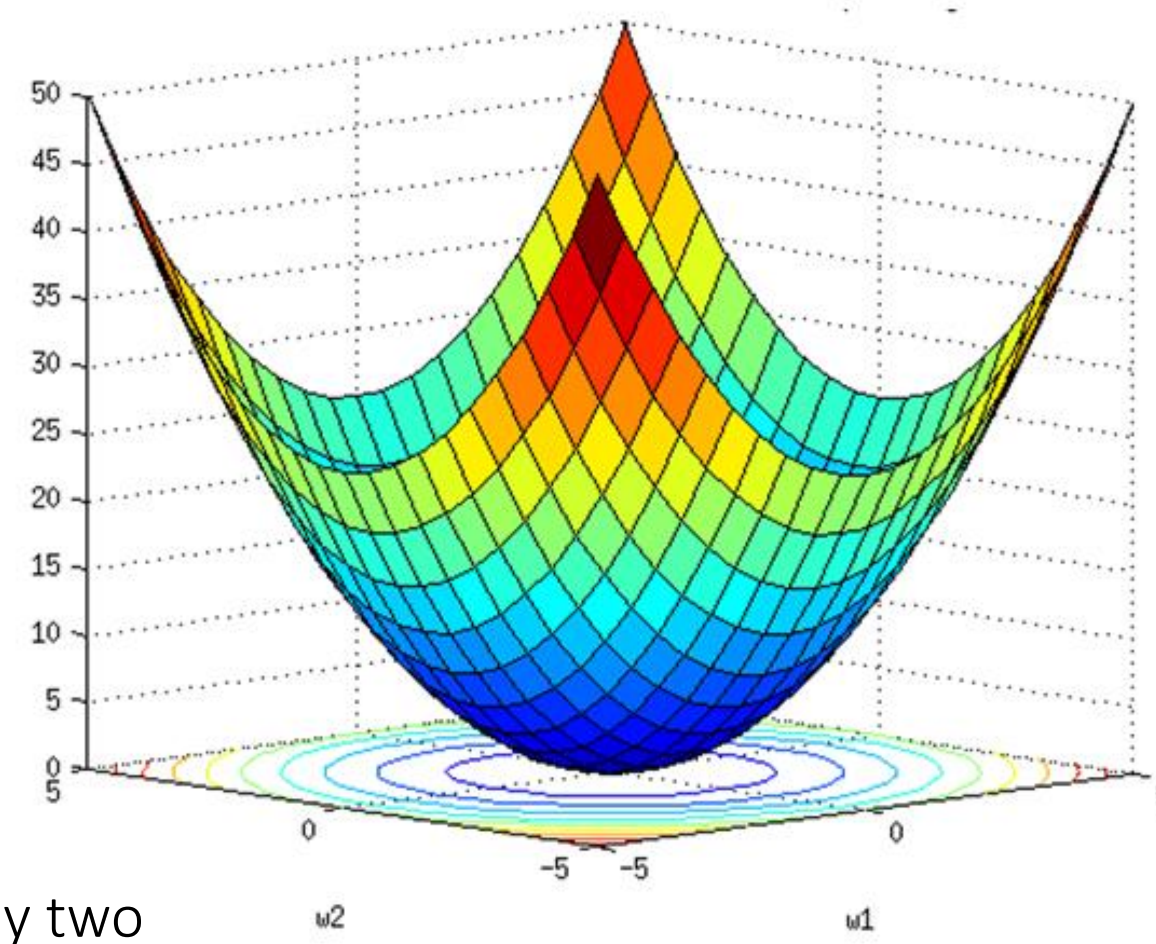
李宏毅

Hung-yi Lee

Gradient Descent

Cost
Or
Loss function

$C(\theta)$



Assume there are only two parameters w_1 and w_2 in a network.

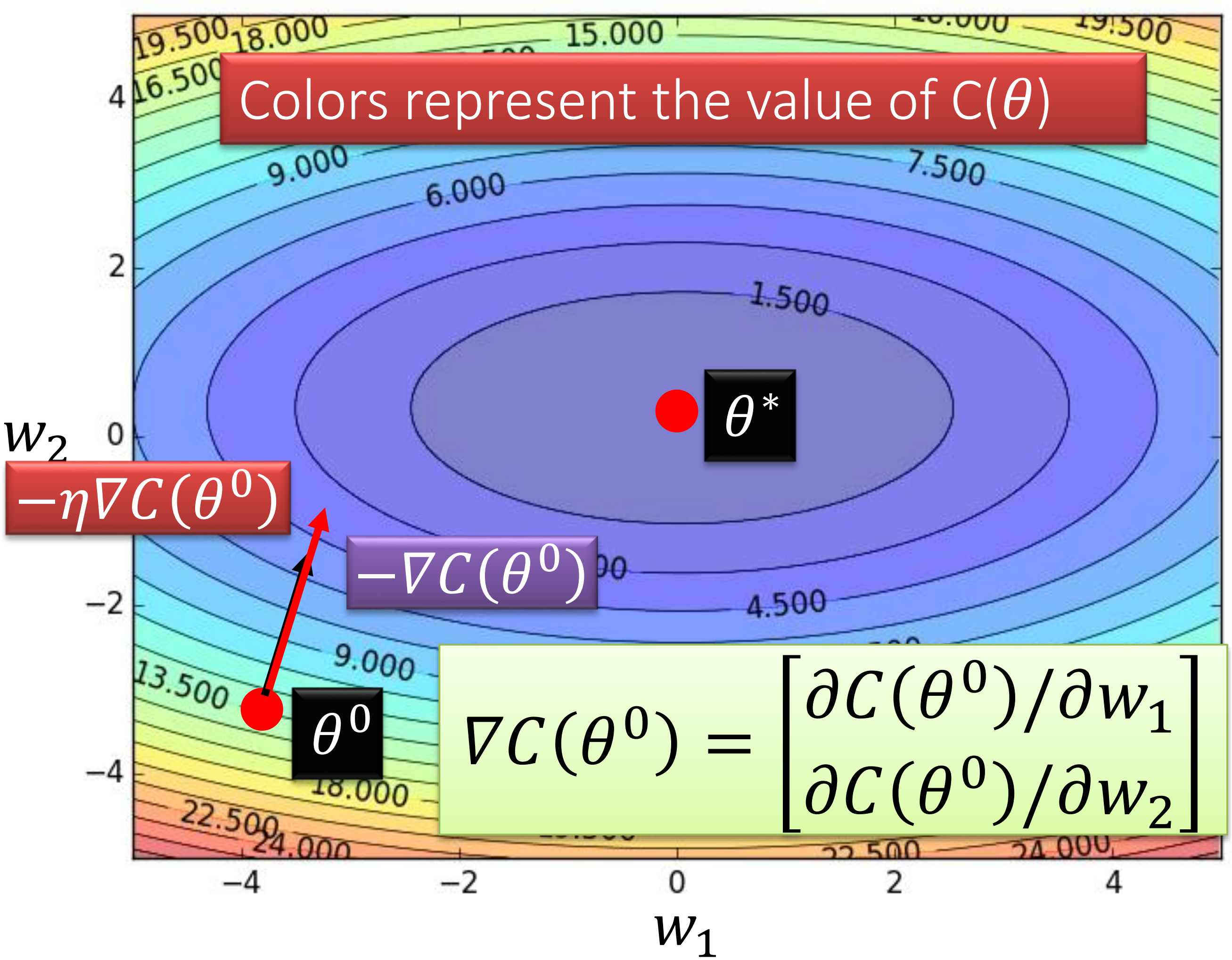
$$\theta = \{w_1, w_2\}$$

Gradient Descent

Assume there are only two parameters w_1 and w_2 in a network.

$$\theta = \{w_1, w_2\}$$

Error Surface



$$\nabla C(\theta^0) = \begin{bmatrix} \partial C(\theta^0)/\partial w_1 \\ \partial C(\theta^0)/\partial w_2 \end{bmatrix}$$

Randomly pick a starting point θ^0

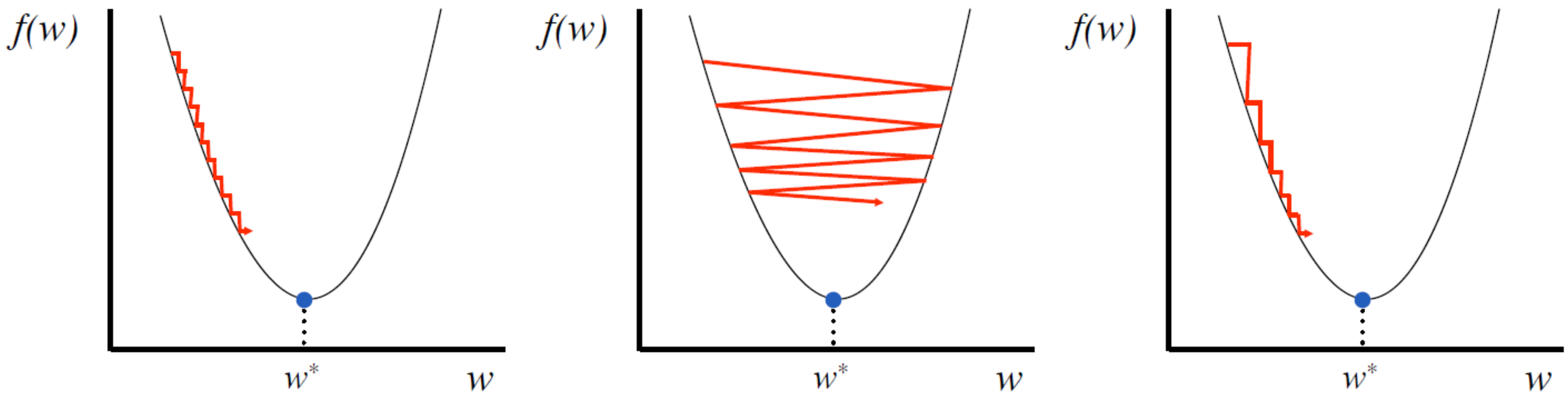
Compute the negative gradient at θ^0

$\Rightarrow -\nabla C(\theta^0)$

Times the learning rate η

$\Rightarrow -\eta\nabla C(\theta^0)$

Choosing Step Size



Too small: converge very slowly | Too big: overshoot and even diverge | Reduce size over time

Theoretical convergence results for various step sizes

A common step size is $\alpha_i = \dfrac{\alpha}{n\sqrt{i}}$

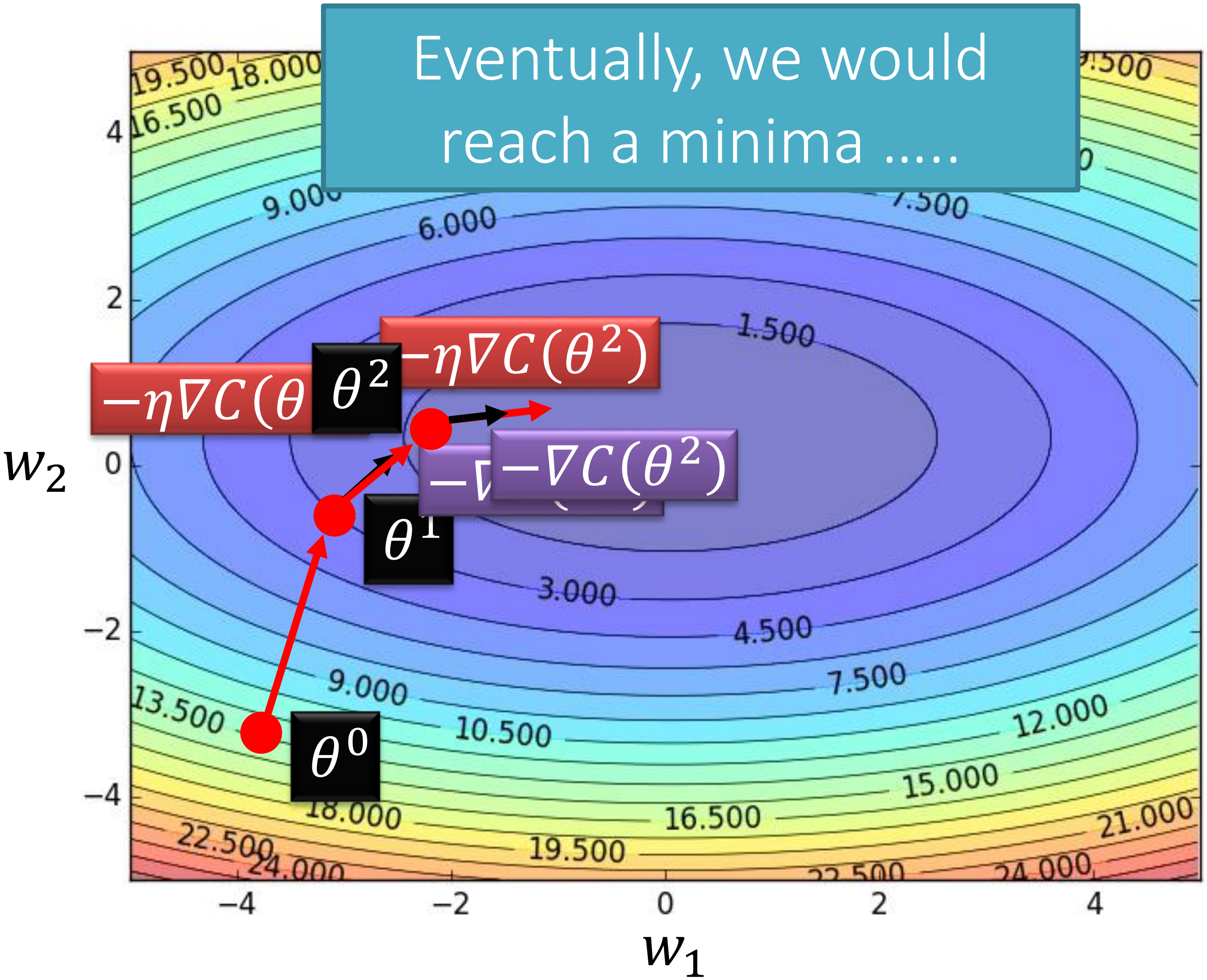
Constant

Iteration #

Training Points

Source: BerkeleyX: CS190.1x Scalable Machine Learning

Gradient Descent



Randomly pick a starting point θ^0

Compute the negative gradient at θ^0

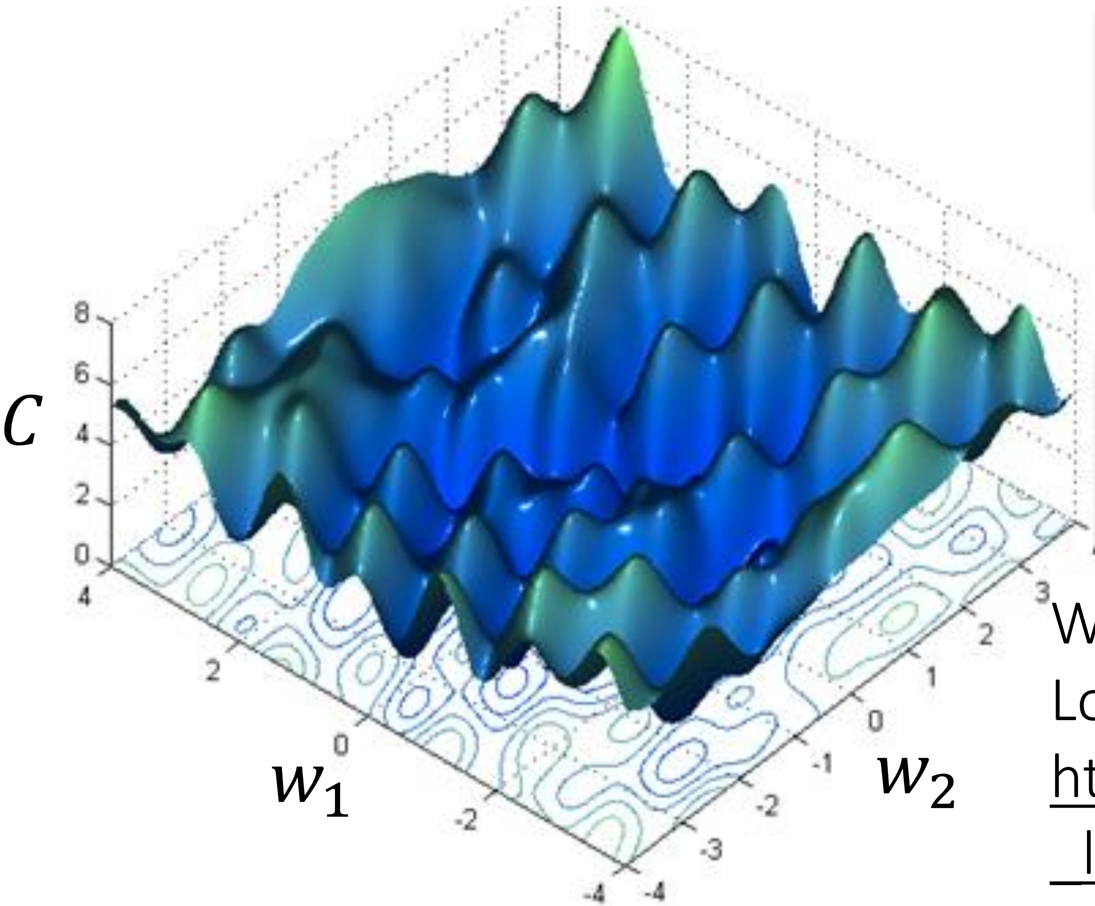
$\Rightarrow -\nabla C(\theta^0)$

Times the learning rate η

$\Rightarrow -\eta\nabla C(\theta^0)$

Local Minima

- Gradient descent never guarantee global minima

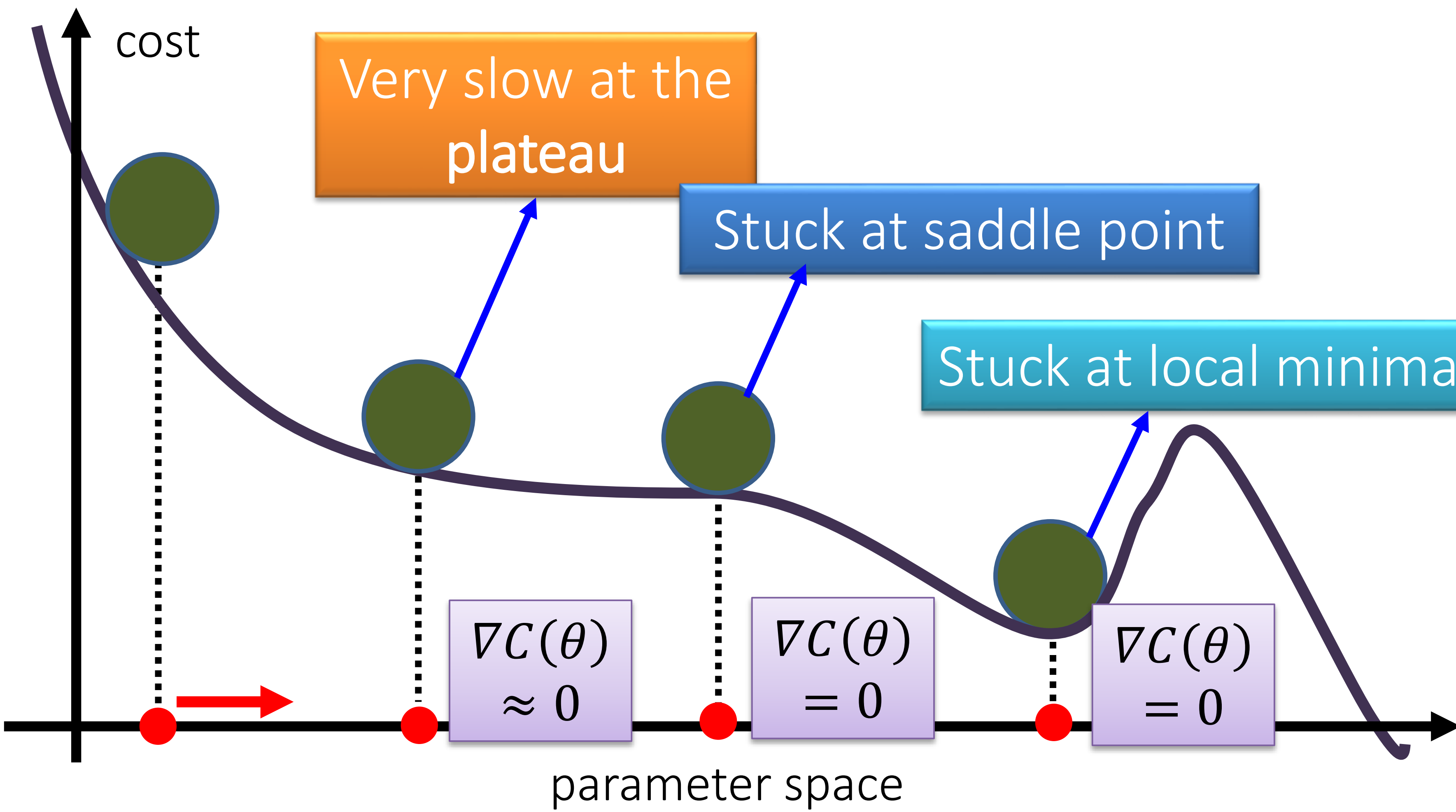


Different initial point θ^0

Reach different minima, so different results

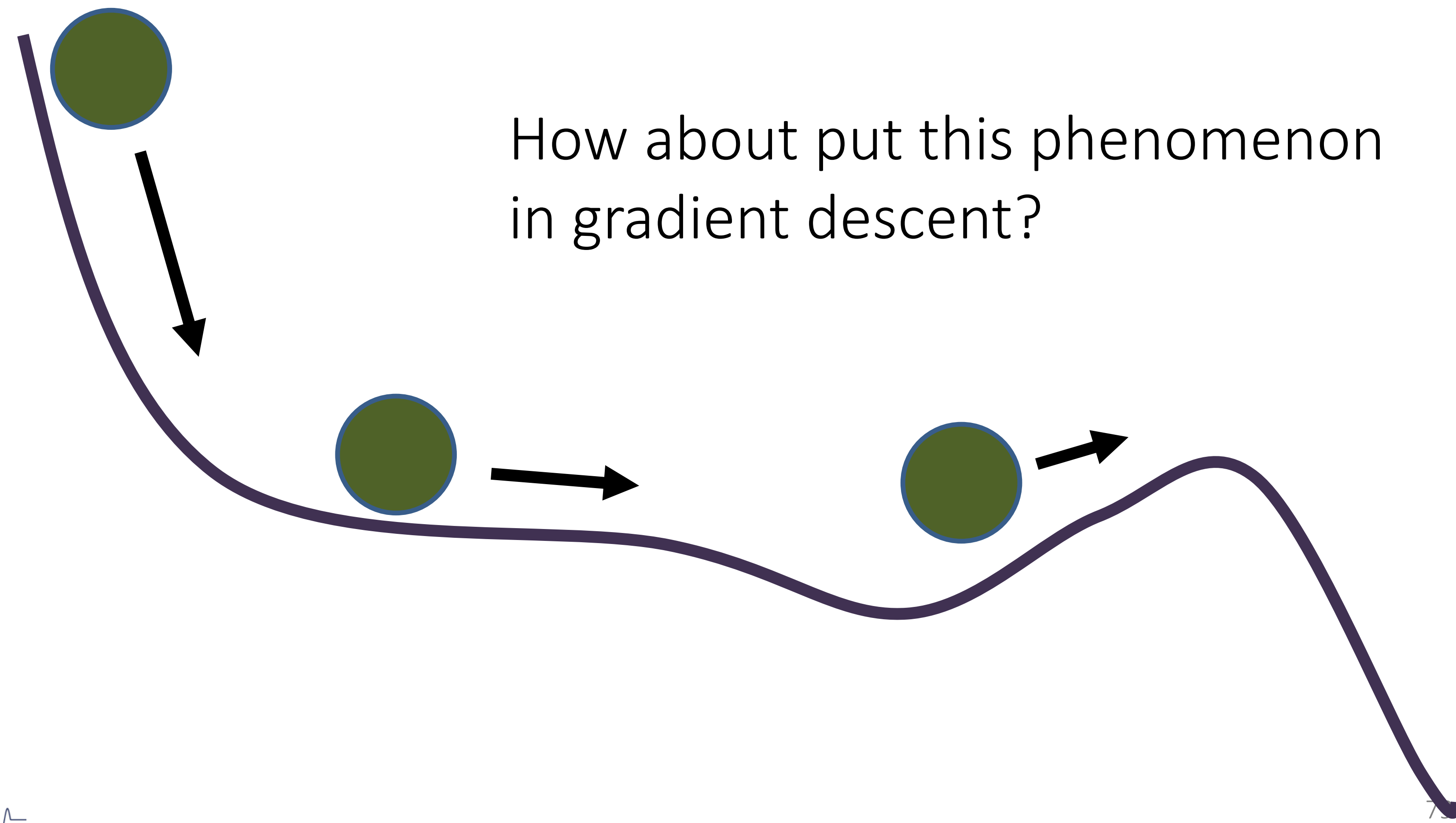
Who is Afraid of Non-Convex Loss Functions?
http://videolectures.net/eml07_lecun_wia/

Besides local minima ……



In physical world ……

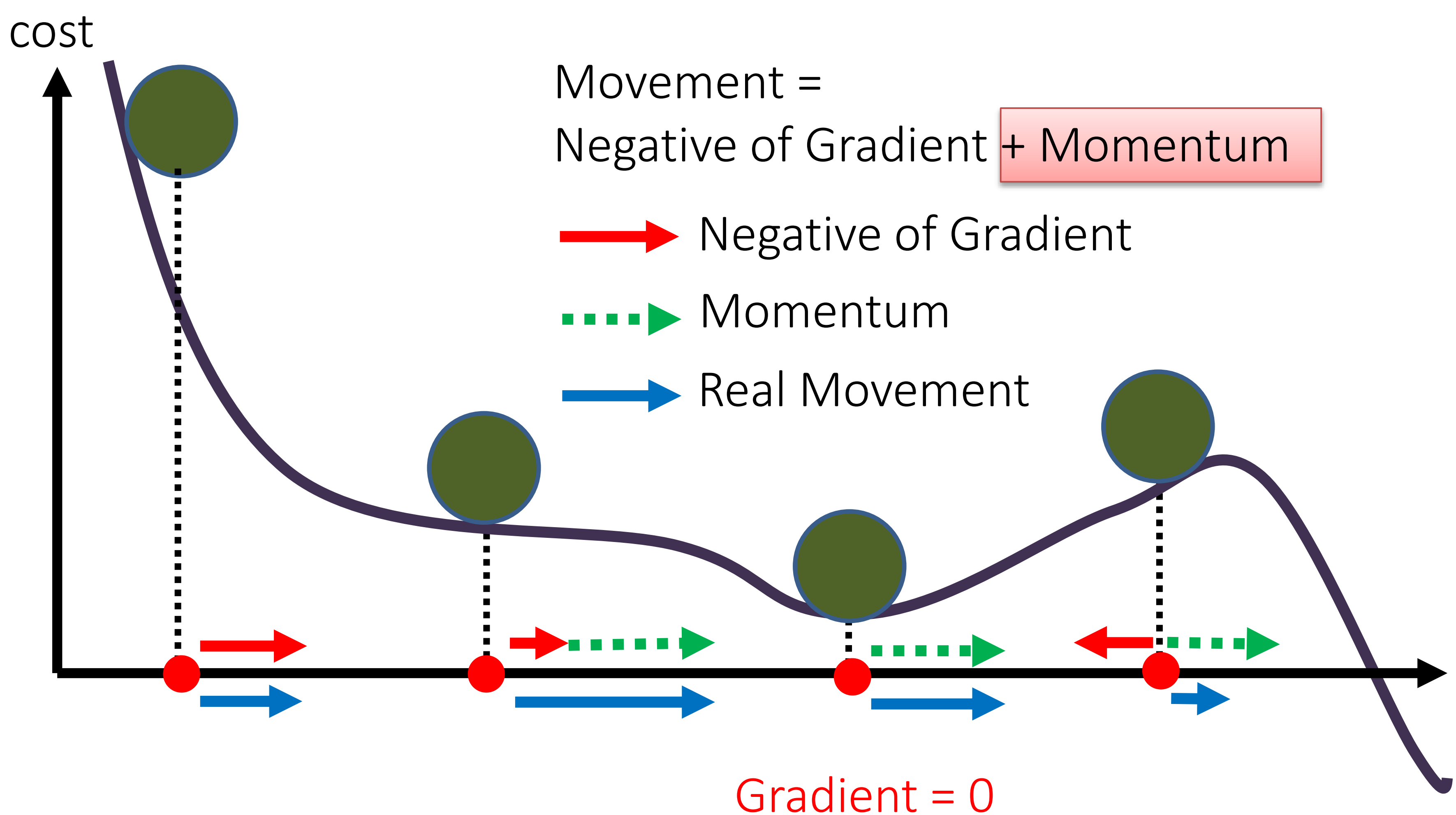
- Momentum



How about put this phenomenon in gradient descent?

Momentum

Still not guarantee reaching global minima, but give some hope ……



So now let's train!:

- **W** and **b** will be variables (tensors)

```
W = tf.Variable(tf.zeros([2, 1], "float"), name="weight")
b = tf.Variable(tf.zeros([1], "float"), name="bias")
```

- **and we use cross-entropy and gradient descend**

```
cost = -tf.reduce_sum(tf.to_double(labels) * tf.log(pred) + (1-tf.to_double(labels))
* tf.log(1-pred))

# Gradient descent
learning_rate = 0.001
optimizer = tf.train.GradientDescentOptimizer(learning_rate).minimize(cost)
```

- We have used all the training data several runs (Epochs)

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

    # We can Run the optimization algorithm several times
    for i in range(100):
        cost_out,W_out,b_out,_=sess.run([cost, W,b, optimizer], feed_dict={X:
train_X, labels: train_labels})
        print("Epoch : %d Cost= %s "%(i,cost_out))
        print(W_out)
        print(b_out)
```

When large amounts of data divide data into mini-batches

- Most optimization algorithms converge much faster (in terms of total computation, not in terms of number of updates) if they are allowed to rapidly compute approximate estimates of the gradient rather than slowly computing the exact gradient.

- Another consideration motivating statistical estimation of the gradient from a small number of samples is redundancy in the training set.

- Optimization algorithms that use the entire training set are called batch or deterministic gradient

- Stochastic Gradient Descend (SGD): Optimization algorithms that use only a single example at a time

- Most algorithms used for deep learning fall somewhere in between: minibatch or minibatch stochastic methods

Confusing terminology:

- The word “batch” is also often used to describe the minibatch used by minibatch stochastic gradient descent.

- It is very common to use the term “batch size” to describe the size of a minibatch

See more details on how choosing minibatch size on Deep Learning Book (Chap 8 : Optimization)

…. See details in Notebook
… and practice with it….

- Random initialization of variables
- Stepsize
- Optimizers
- Interactive Session
- Tf Debugging?

Deep Learning Seminar (materials)

Deep Learning using TensorFlow and TensorFlow-Slim

Dipendra Jha Northwestern University

dipendra009@gmail.com https://www.linkedin.com/in/dipendra009

Deep Learning courses

Prof. Hung-yi Lee National Taiwan University (NTU) Taipei

Introduction to Deep Learning
Yingyu Liang Princeton University

http://jrmeyer.github.io/tutorial/2016/02/01/TensorFlow-Tutorial.html

http://www.psi.toronto.edu/~jimmy/ece521/Tut1.pdf

…and of course look videos/course by Geoffrey Hinton: The Godfather of Deep Learning