Deep Learning: Part I

Introduction to Artificial Neural Networks

Nicolas Courty
ncourty@irisa.fr

M2 AIDN/University of Bretagne Sud, Vannes

Course Overview

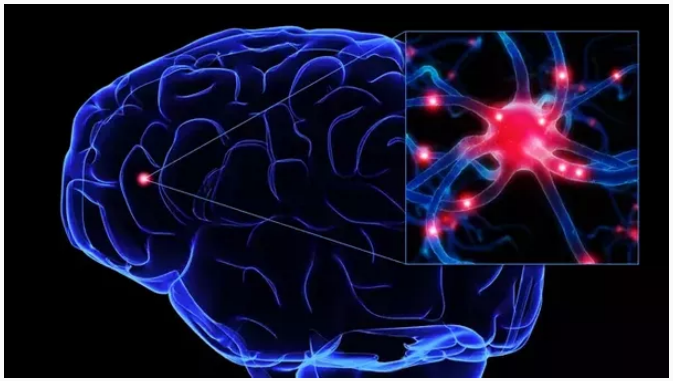
- Biological Inspiration and Historic Overview
- Introduction to Deep Neural Net
- Tips and Tricks in Training Deep Nets

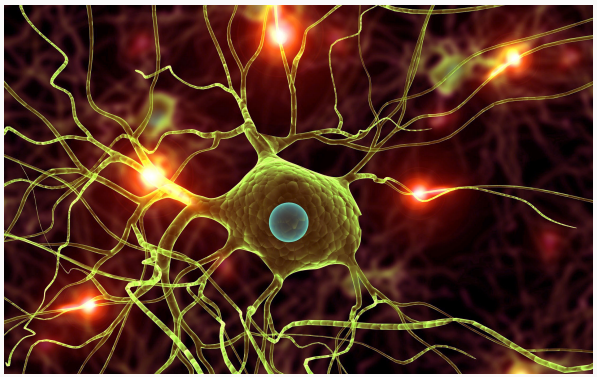
Table of content

Biological Inspiration and Historic Overview

Introduction

- What is Artificial Neural Network (ANN)?
- ANN is the information processing system that has certain characteristics in common with biological neurons.





- Approximately 10 billion neurons in the human cortex
- Each biological neuron is connected to several thousands of other neurons, in a highly complex manner

Biological Inspiration

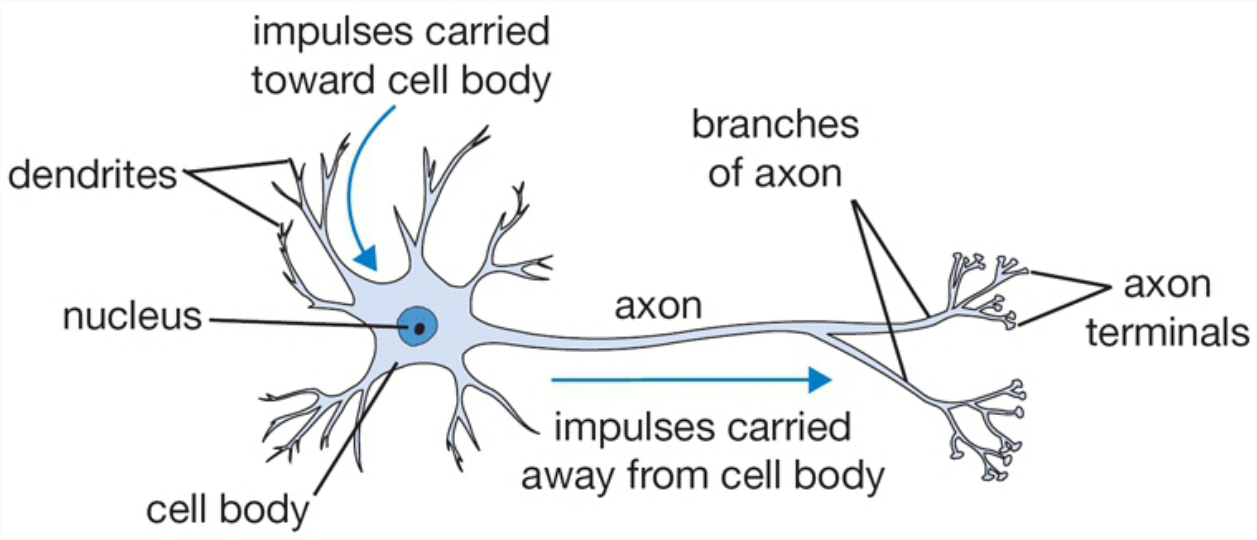


Figure 1: Components of single neuron

Components of single neurons

- Neuron $\rightarrow$ information switch with input and output information
- Neuron $\rightarrow$ Dendrites, Nucleus (Soma), Axon, Synapsis

Biological Inspiration

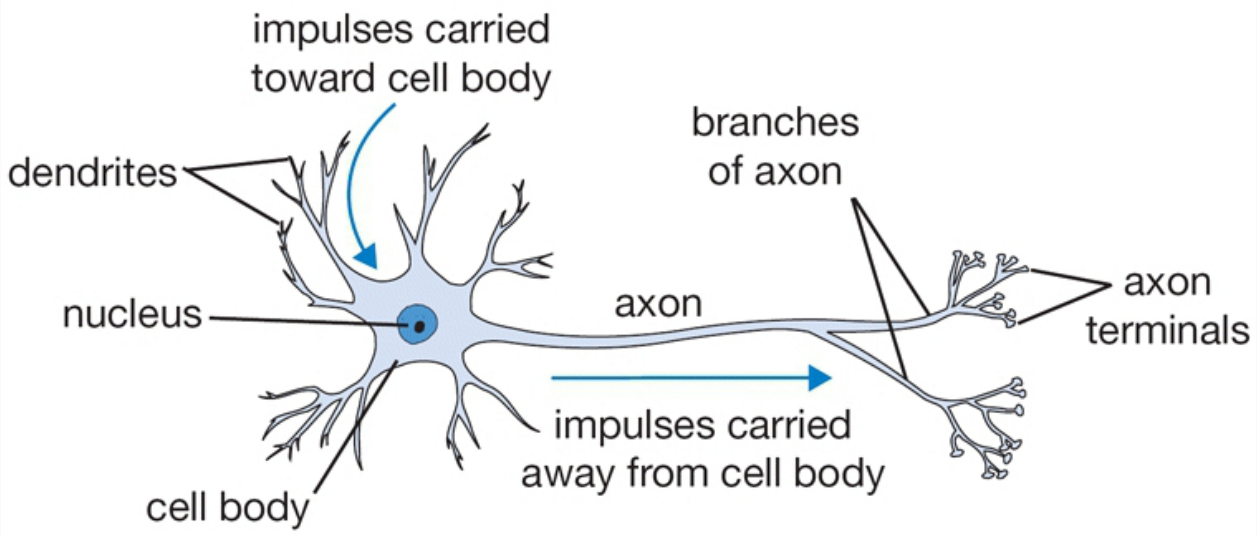


Figure 2: Components of single neuron

- Neuron $\rightarrow$ Dendrites, Nucleus (Soma), Axon, Synapsis
- Dendrites $\rightarrow$ The dendrites of a neuron receive the information from another neurons (by special connections, the synapses)

Biological Inspiration

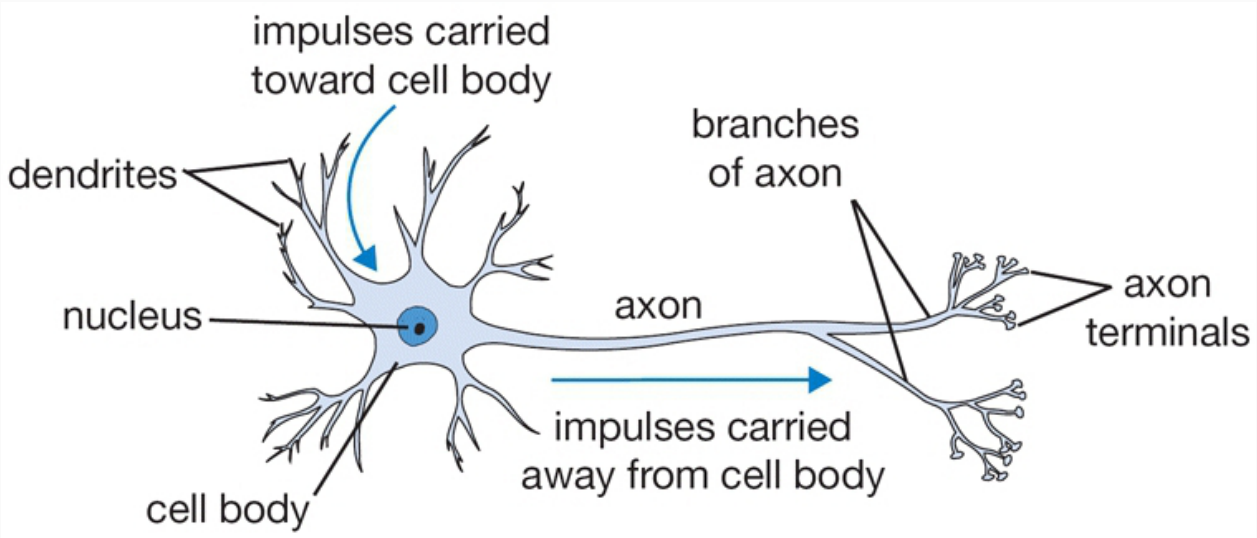


Figure 3: Components of single neuron

- Neuron $\rightarrow$ Dendrites, Nucleus (Soma), Axon, Synapsis
- Soma $\rightarrow$ the soma accumulates the signals received by the dendrites in the nucleus. If the accumulation exceeds certain threshold, the soma activates the signals
- Threshold: Step function threshold

Biological Inspiration

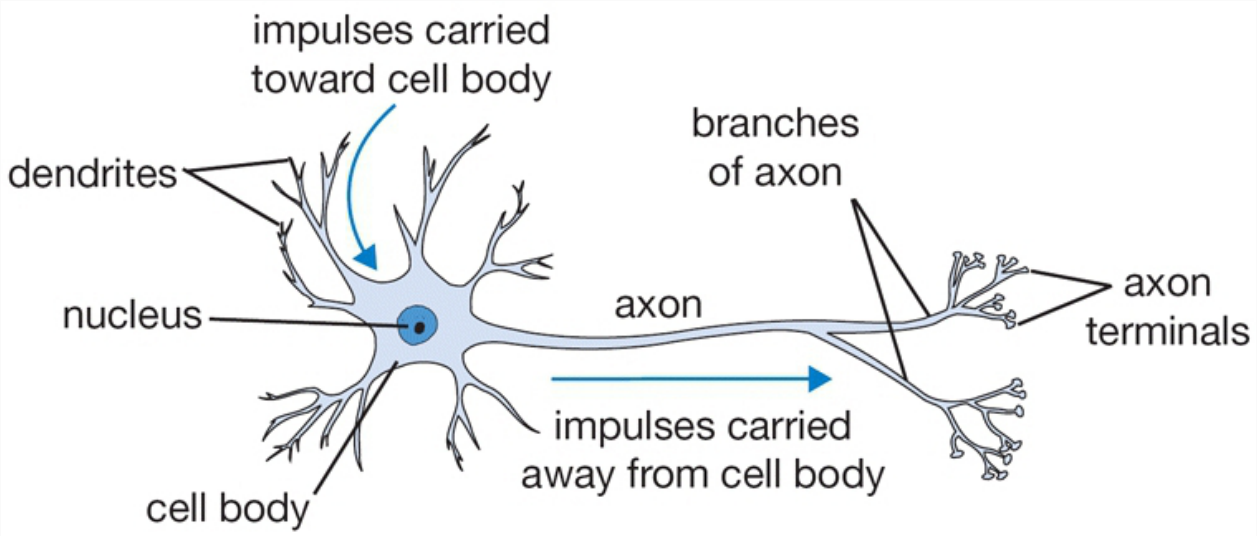


Figure 4: Components of single neuron

- Neuron $\rightarrow$ Dendrites, Nucleus (Soma), Axon, Synapsis
- Axon $\rightarrow$ The activated pluse from soma is transferred to other neurons by the means of axon. The axon terminals lead to dendrites

Biological Neuron to Artificial Neuron

- Transition to the artificial neurons

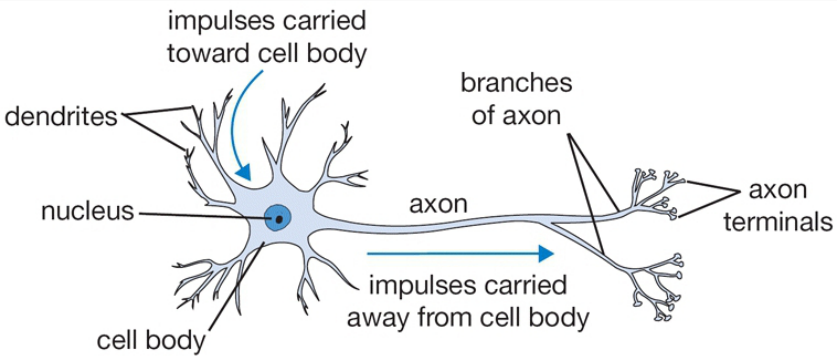


Figure 5: Components of single neuron

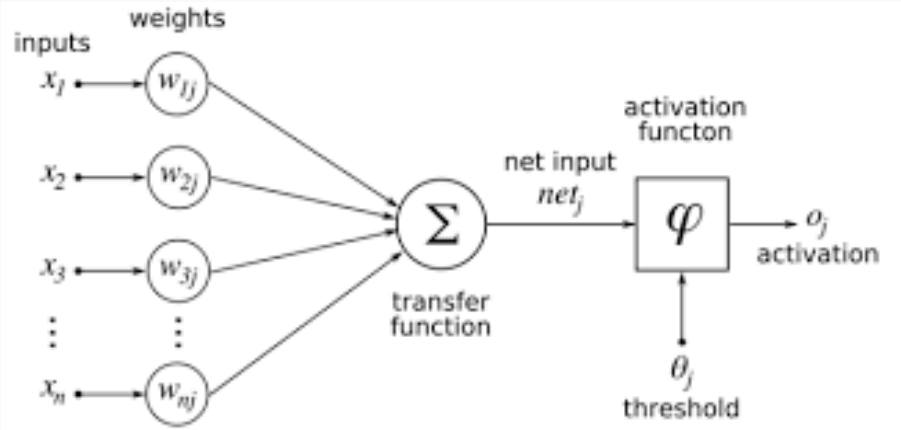


Figure 6: Single artificial neuron

Biological Neuron to Artificial Neuron

- Examples of Activation functions

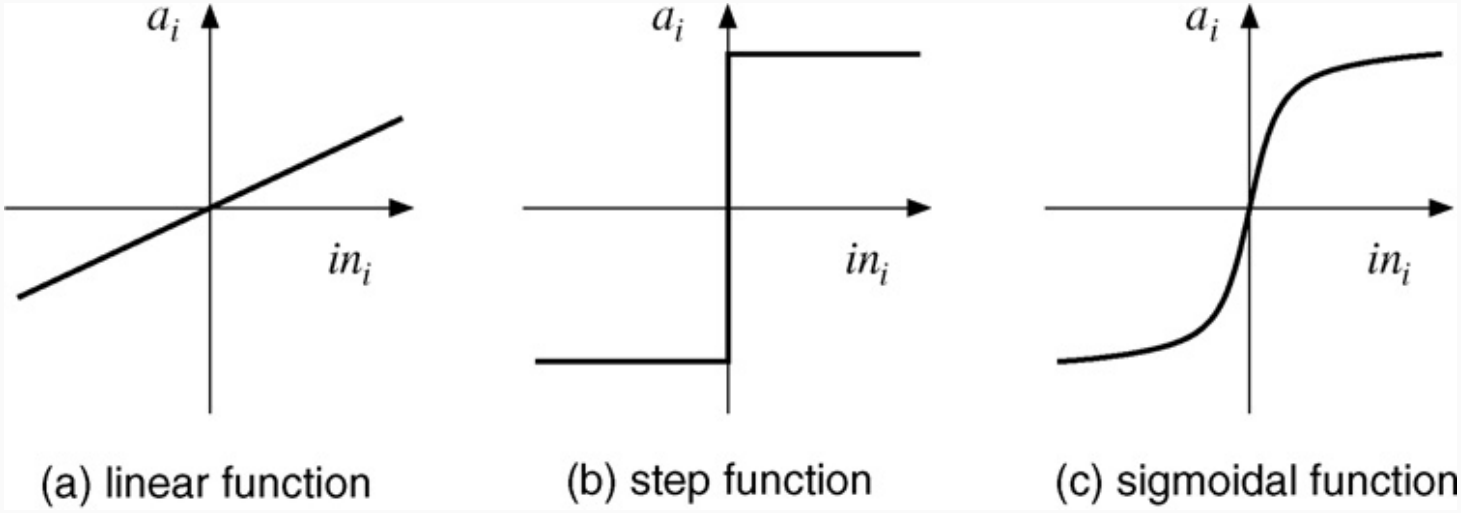


Figure 7: Components of single neuron

Benefits of ANN

1. Non-linearity
2. Input and output mapping
3. Adaptivity
4. Evidential Response
5. Fault Tolerance
6. Parallel Nature

Short History

- 1940's : McCulloch-Pitts Neurons
- 1949 : Hebb learning rule - Donald Hebb
- 1958 : Perceptron - Frank Rosenblatt
- 1982 : Hopfield Network
- 1985 : Back propagation

McCulloch-Pitts Neurons

- First attempt to model the biological neuron
- How the brain could produce highly complex patterns by using many basic cells that are connected together
- Bird with two receiver : 1) shape detector , 2) color detector
- design a logic gate for a bird to eat blueberry

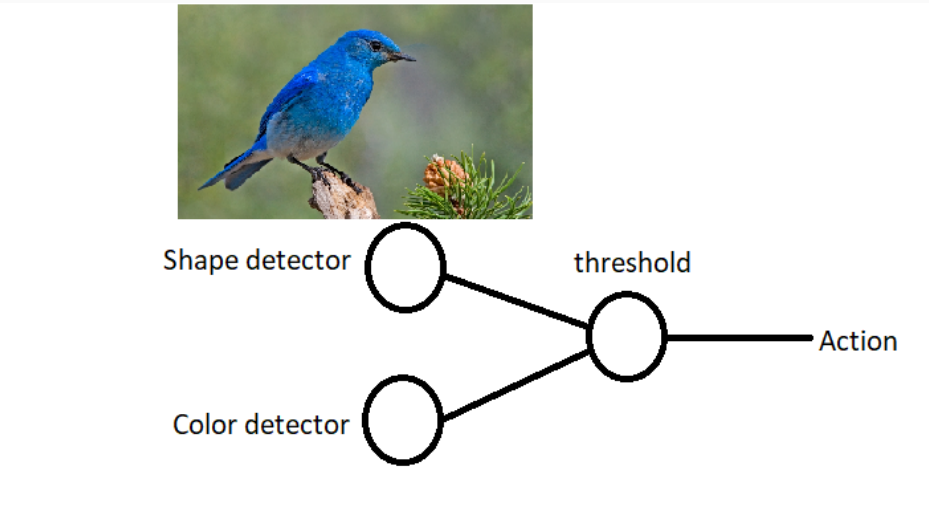
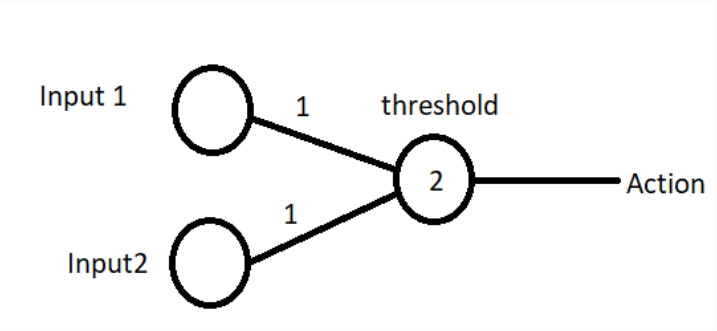


Figure 8: MCP Neurons

McCulloch-Pitts Neurons

- Suppose there are four objects : Blueberry, Golf ball, Violet, Hot Dog
- Design a MCP neuron for the bird to eat blueberry

Object	Purple	Round	Eat
Blueberry	1	1	1
Golf ball	0	1	0
Violet	1	0	0
Hot Dog	0	0	0



MCP Neurons: Multiple Inputs

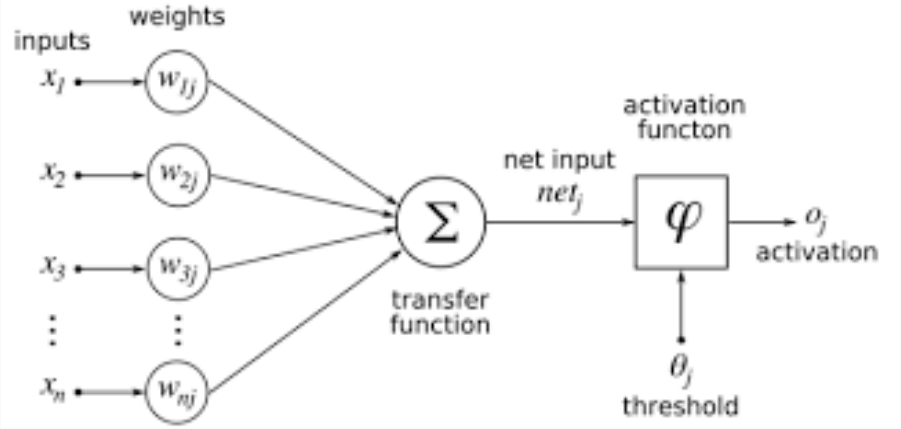


Figure 9: MCP Neuron

- the weights are fixed and threshold is pre-designed according to the output
- By varying the threshold, we can adapt MCP neurons for the desired output
- Logic gate design using MCP Neurons

MCP Neurons: Logic gates

- Basic operations: OR, AND, and NOT

AND

in_1	in_2	out
0	0	0
0	1	0
1	0	0
1	1	1

OR

in_1	in_2	out
0	0	0
0	1	1
1	0	1
1	1	1

NOT

in	out
0	1
1	0

- can you design the logic gates using MCP Neurons ?

MCP Neurons: Logic gates

- Basic operations: OR, AND, and NOT

AND

in_1	in_2	out
0	0	0
0	1	0
1	0	0
1	1	1

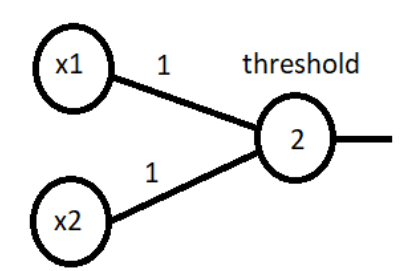
OR

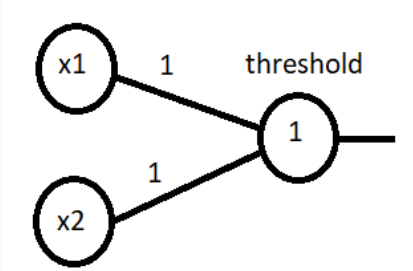
in_1	in_2	out
0	0	0
0	1	1
1	0	1
1	1	1

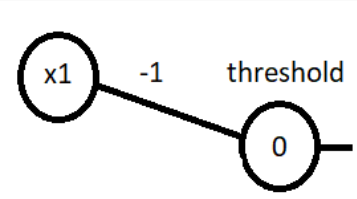
NOT

in	out
0	1
1	0

- can you design the logic gates using MCP Neurons ?
-







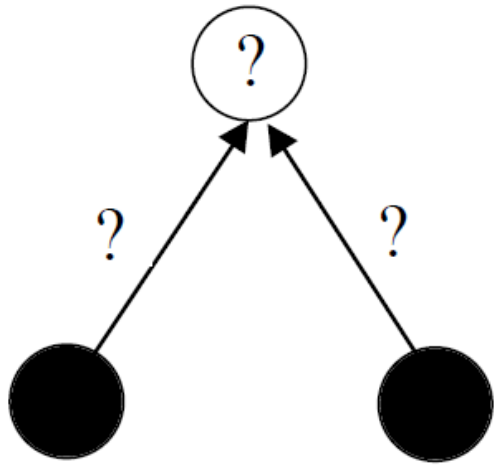
- Is it possible to change the weights and (or) threshold, and to end-up with the same results

MCP Neurons: XOR

- What about designing MCP neuron for XOR?

XOR

in_1	in_2	out
0	0	0
0	1	1
1	0	1
1	1	0



- what is the weight values and threshold value to obtain XOR?

MCP Neurons: Limitation of single Neuron

- single MCP neuron is limited in its capacity
- why does the single MCP neuron cannot perform this task?
- To see let's plot the OR and AND in the two dimensional space

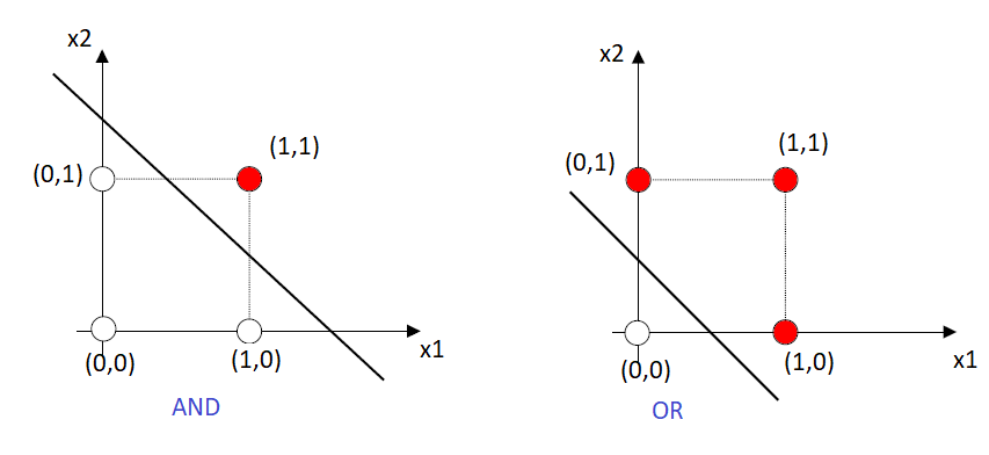


Figure 10: plot of AND, OR in two dimensional space

- Basically, MCP neurons finds a decision boundary to separate the binary outcomes
- Likewise can you plot XOR in $2 - D$ space

MCP Neurons: XOR

- XOR in two dimensional space

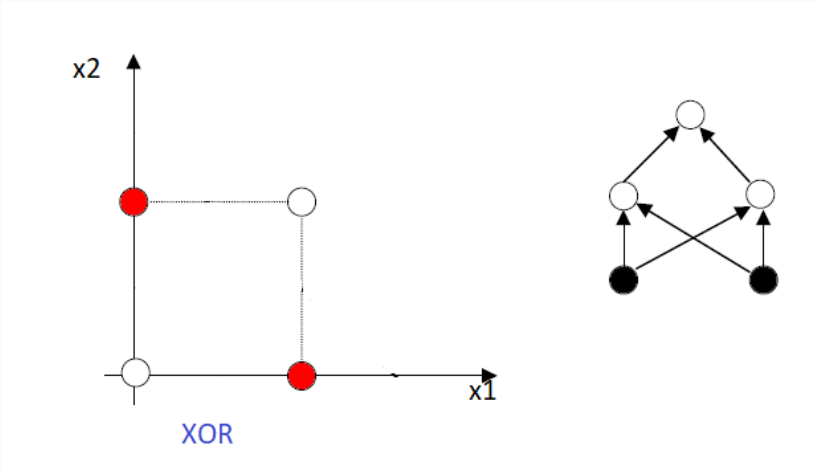


Figure 11: XOR in $2 - D$

- we need two lines to separate the outcomes
- So, single MCP neuron is limited with the complexity of the model grows. we need more number of MCP neurons to solve the complex tasks.
- XOR problem can be solved using two MCP neurons

MCP Neuron: XOR

- We know that any logic function can be expressed using basic logical operators
- XOR can be decomposed as: $(x_1 + x_2).(\bar{x_1} + \bar{x_2})$, here $'.'$ denotes AND operation and $'+'$ denotes OR operation

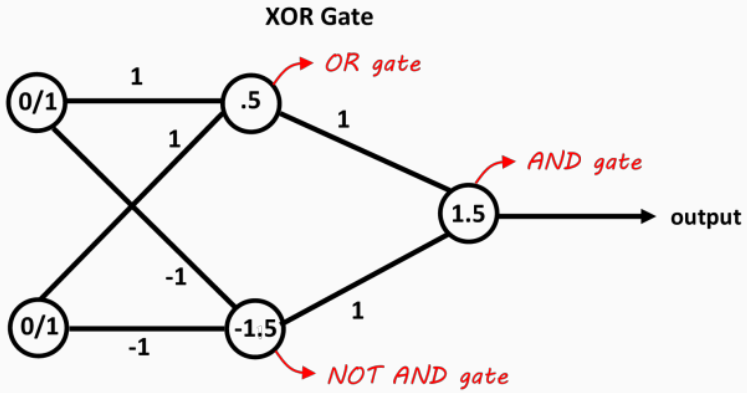


Figure 12: MCP Neuron for XOR logic function

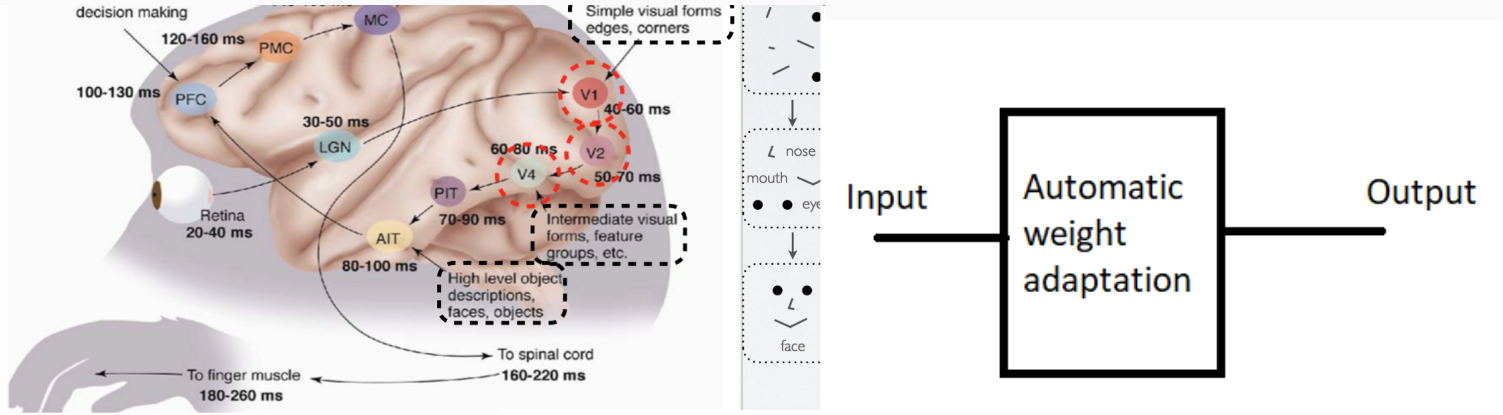
- How can we write XOR in the other form?
- $(x_1.\bar{x_2}) + (\bar{x_1}.x_2)$

Limitations of MCP Neurons

- The weights are pre-designed, and it is specific to the problem at hand
- There is no learning involved while designing the weights
- So it cannot be generalized to the variety of tasks
- It will be nice, if the algorithm can design weights automatically depending on the input and output pairs

Learning weights

- A learning system changes itself in order to adapt to e.g. environmental changes.



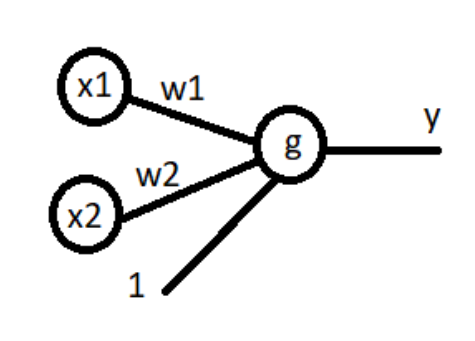
- Likewise, ANN should have the capability to change itself to adapt the changes in the input and output of the system, and improve the performance
- More weightage for the important inputs and less weights for non-desirable inputs

Hebb's learning rule: First learning rule

- In 1949, Donald Hebb proposed one of the key ideas in biological learning, commonly known as Hebbs Law.
- Hebbs Law states that learning occurs by modification of the synapse strengths (weights) in a manner such that if two interconnection neurons are both 'on' at the same time, then weight between these neurons should be increased.
- here we only consider a single layer net (one output unit)

Hebb's rule

- Let's consider the following neuron



- Adjust weights for :
  $w_i(new) = w_i(old) + x_i y, (i = 1 \rightarrow \ n)$
- Adjust the bias:
  $b(new) = b(old) + y$
- weight change: $\delta w_i = x_i y$

Hebb's rule: Example

(a) AND function in Binary form

Input			Output
x1	x2	1	y
1	1	1	1
1	0	1	0
0	1	1	0
0	0	1	0

(b) AND function in Bi-polar form

Input			Output
x1	x2	1	y
1	1	1	1
1	-1	1	-1
-1	1	1	-1
-1	-1	1	-1

- Weight change is the product of input and target.
  $\delta w_1 = x_1 y_1, \delta w_2 = x_2 y_1, \delta b = y$
- why we need bi-polar form ?

Hebb's rule: AND function

INPUT	TARGET	WEIGHT CHANGES	WEIGHTS
$(x_1 \quad x_2 \quad 1)$		$(\Delta w_1 \quad \Delta w_2 \quad \Delta b)$	$(w_1 \quad w_2 \quad b)$
			(0 0 0)
(1 1 1)	1	(1 1 1)	(1 1 1)

The separating line (see Section 2.1.3) becomes

$$x_2 = -x_1 - 1.$$

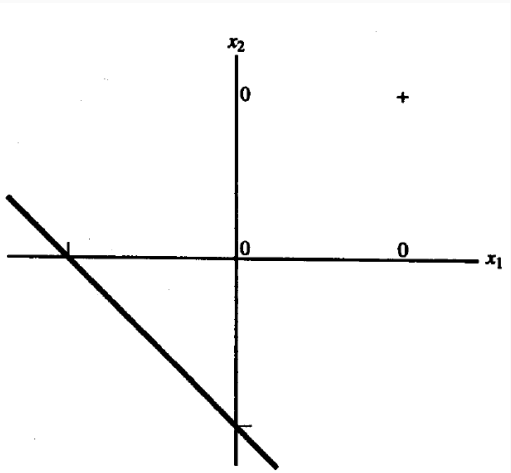


Figure 13: Decision boundary after first training pair

- Let's see what happen's after subsequent training pairs

Hebb's rule: AND function

INPUT			TARGET	WEIGHT CHANGES			WEIGHTS		
$(x_1$	x_2	$1)$		$(\Delta w_1$	Δw_2	$\Delta b)$	$(w_1$	w_2	$b)$
(1	0	1)	0	(0	0	0)	(1	1	1)
(0	1	1)	0	(0	0	0)	(1	1	1)
(0	0	1)	0	(0	0	0)	(1	1	1)

- Learning did not happen for the other pairs of samples, why ?
- so we need to convert the data representation in bi-polar form

Hebb's rule: AND function

INPUT			TARGET	WEIGHT CHANGES			WEIGHTS		
$(x_1$	x_2	$1)$		$(\Delta w_1$	Δw_2	$\Delta b)$	$(w_1$	w_2	$b)$
(1	0	1)	0	(0	0	0)	(1	1	1)
(0	1	1)	0	(0	0	0)	(1	1	1)
(0	0	1)	0	(0	0	0)	(1	1	1)

- Learning did not happen for the other pairs of samples, why ?
- what happens when you change the data representation of the target values in bi-polar form?
- Now we will convert the data representation of both input and output in bi-polar form, and will see whether we can draw the decision boundary correctly

Hebb's rule: AND function, bi-polar form

INPUT	TARGET	WEIGHT CHANGES	WEIGHTS
$(x_1 \quad x_2 \quad 1)$		$(\Delta w_1 \quad \Delta w_2 \quad \Delta b)$	$(w_1 \quad w_2 \quad b)$
			(0 0 0)
(1 1 1)	1	(1 1 1)	(1 1 1)

The separating line (see Section 2.1.3) becomes

$$x_2 = -x_1 - 1.$$

After the second pair

INPUT	TARGET	WEIGHT CHANGES	WEIGHTS
$(x_1 \quad x_2 \quad 1)$		$(\Delta w_1 \quad \Delta w_2 \quad \Delta b)$	$(w_1 \quad w_2 \quad b)$
			(1 1 1)
(1 −1 1)	−1	(−1 1 −1)	(0 2 0)

The separating line becomes

$$x_2 = 0.$$

Hebb's rule: AND function, bi-polar form

After the third pair

INPUT	TARGET	WEIGHT CHANGES	WEIGHTS
$(x_1 \quad x_2 \quad 1)$		$(\Delta w_1 \quad \Delta w_2 \quad \Delta b)$	$(w_1 \quad w_2 \quad b)$
			$(0 \quad 2 \quad 0)$
$(-1 \quad 1 \quad 1)$	-1	$(1 \quad -1 \quad -1)$	$(1 \quad 1 \quad -1)$

The separating line becomes

$$x_2 = -x_1 + 1.$$

After the last pair

INPUT	TARGET	WEIGHT CHANGES	WEIGHTS
$(x_1 \quad x_2 \quad 1)$		$(\Delta w_1 \quad \Delta w_2 \quad \Delta b)$	$(w_1 \quad w_2 \quad b)$
			$(1 \quad 1 \quad -1)$
$(-1 \quad -1 \quad 1)$	-1	$(1 \quad 1 \quad -1)$	$(2 \quad 2 \quad -2)$

Even though the weights have changed, the separating line is still

$$x_2 = -x_1 + 1,$$

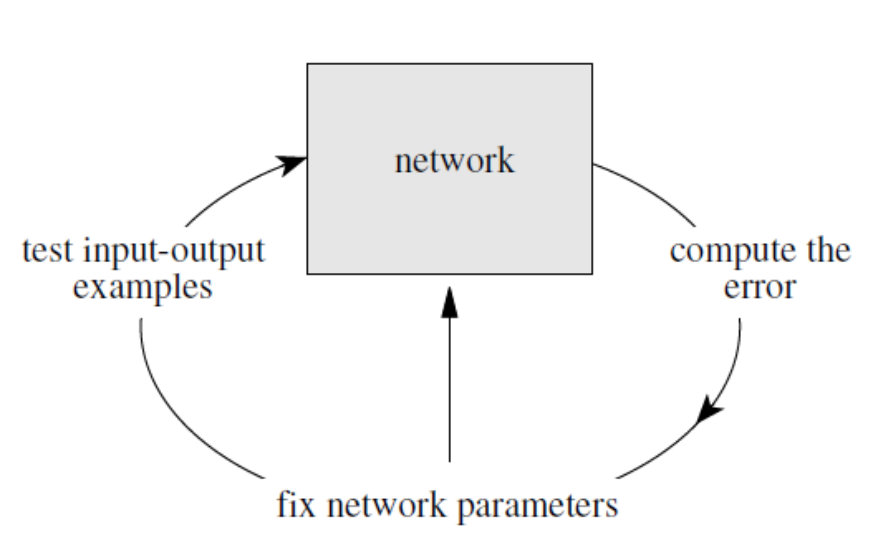
Hebb's rule: Failure cases

INPUT (x_1 x_2 x_3 1)	TARGET	WEIGHT CHANGE (Δw_1 Δw_2 Δw_3 Δb)	WEIGHT (w_1 w_2 w_3 b)
			(0 0 0 0)
(1 1 1 1)	1	(1 1 1 1)	(1 1 1 1)
(1 1 −1 1)	−1	(−1 −1 1 −1)	(0 0 2 0)
(1 −1 1 1)	−1	(−1 1 −1 −1)	(−1 1 1 −1)
(−1 1 1 1)	−1	(1 −1 −1 −1)	(0 0 0 −2)

- the weight did not give correct output for the first pattern
- Can guess what could be the correct decision boundary?
- why does the Hebb's rule fail?

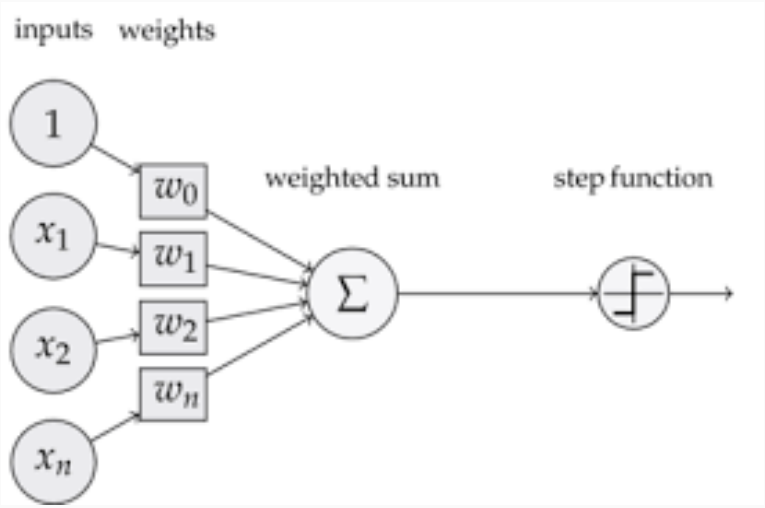
Perceptron

- Rosenblatt invented the idea of perceptron learning rule in 1962.
- Hebb's learning rule even failed in the linear separable cases



- If the data points are linearly separable, then perceptron algorithm finds the correct decision boundary

Perceptron

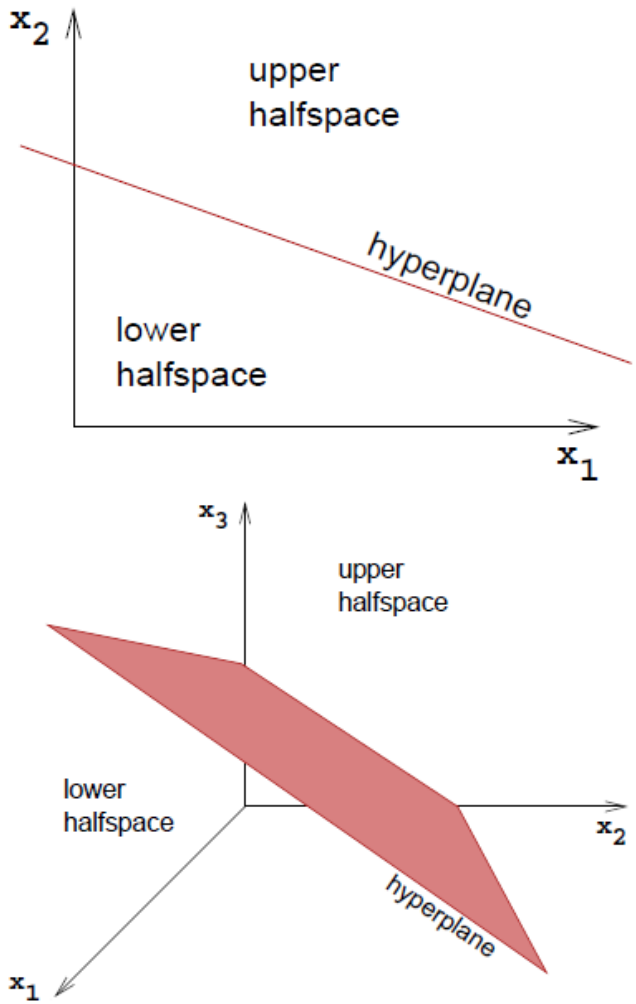


- output $f = g(\sum_{i=1}^{n} w_i x_i)$,
- if $\mathbf{x} = (x_1, x_2, \ldots, x_n, 1)$, $\mathbf{w} = (w_1, w_2, \ldots, w_n, w_0)$, then $f = g(\mathbf{w}^T \mathbf{x})$

$$f = \begin{cases} 1 & \text{if } x_1 w_1 + x_2 w_2 + \ldots + x_n w_n \geq 0, \\ 0 & \text{if } x_1 w_1 + x_2 w_2 + \ldots + x_n w_n < 0, \end{cases} \tag{1}$$

Perceptron

- $\mathbf{w}^T\mathbf{x}$ is the hyperplane
- Idea is to partition the hyperplane
- points with $\mathbf{w}^T\mathbf{x} = 0$ is on the hyperplane
- points with $\mathbf{w}^T\mathbf{x} > 0$ is above the hyperplane
- points with $\mathbf{w}^T\mathbf{x} < 0$ is below the hyperplane



Perceptron

- how to learn the proper weights
- one way is to try different possible weights, and choose the one which has minimum error

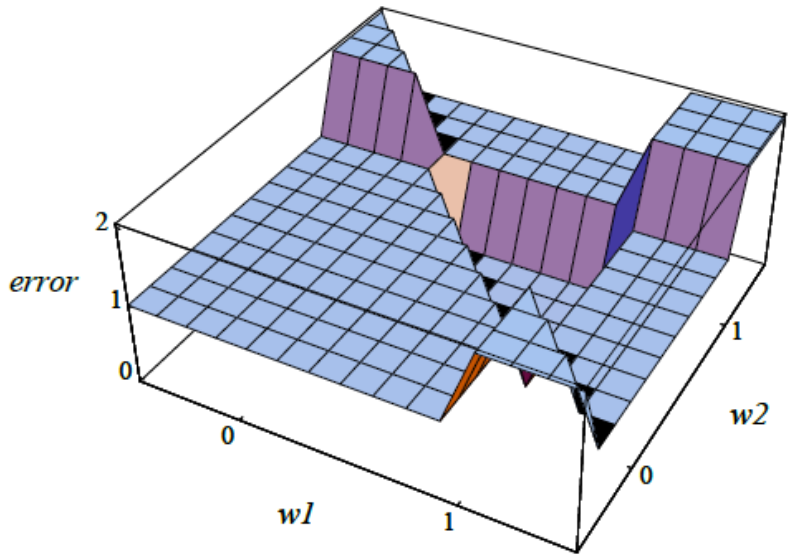
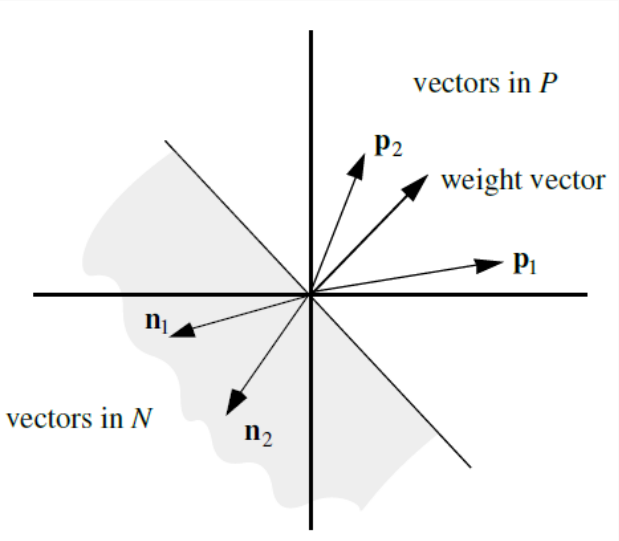


Fig. 4.5. Error function for the AND function

Perceptron: Learning algorithm

- change weight only when wrong classification occurs
- Let $\mathbf{p_1}, \mathbf{p_2} \in R^2$ belongs to the positive class and $\mathbf{n_1}, \mathbf{n_2} \in R^2$ belongs to the negative class



start: The weight vector $\mathbf{w}_0$ is generated randomly, set $t := 0$

test: A vector $\mathbf{x} \in P \cup N$ is selected randomly,

if $\mathbf{x} \in P$ and $\mathbf{w}_t \cdot \mathbf{x} > 0$ go to *test*,
if $\mathbf{x} \in P$ and $\mathbf{w}_t \cdot \mathbf{x} \leq 0$ go to *add*,
if $\mathbf{x} \in N$ and $\mathbf{w}_t \cdot \mathbf{x} < 0$ go to *test*,
if $\mathbf{x} \in N$ and $\mathbf{w}_t \cdot \mathbf{x} \geq 0$ go to *subtract*.

add: set $\mathbf{w}_{t+1} = \mathbf{w}_t + \mathbf{x}$ and $t := t + 1$, goto *test*

subtract: set $\mathbf{w}_{t+1} = \mathbf{w}_t - \mathbf{x}$ and $t := t + 1$, goto *test*

Perceptron: Learning

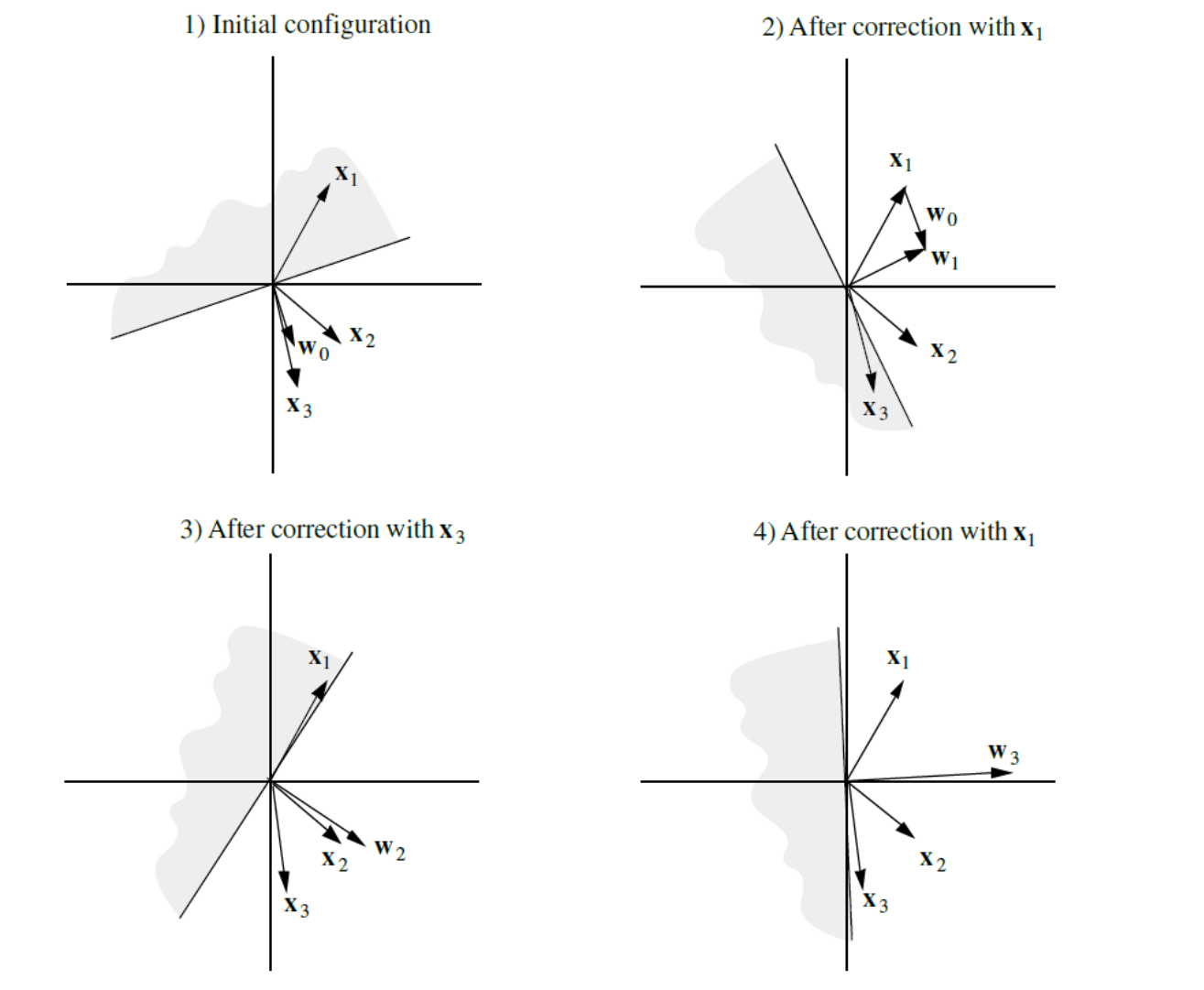


Figure 15: convergence of learning algorithm

Perceptron criteria function:Deriving the update rule

- Define the cost function: as sum of the misclassified samples
- $J(\mathbf{w}) = -\sum_{i \in M} y_i(\mathbf{w^T x_i})$
- when all the samples are correctly classified, $J(\mathbf{w})$ becomes zero
- To minimize, compute the gradients:

$$\frac{\partial J}{\partial \mathbf{w}} = -\sum_{i \in M} y_i x_i$$

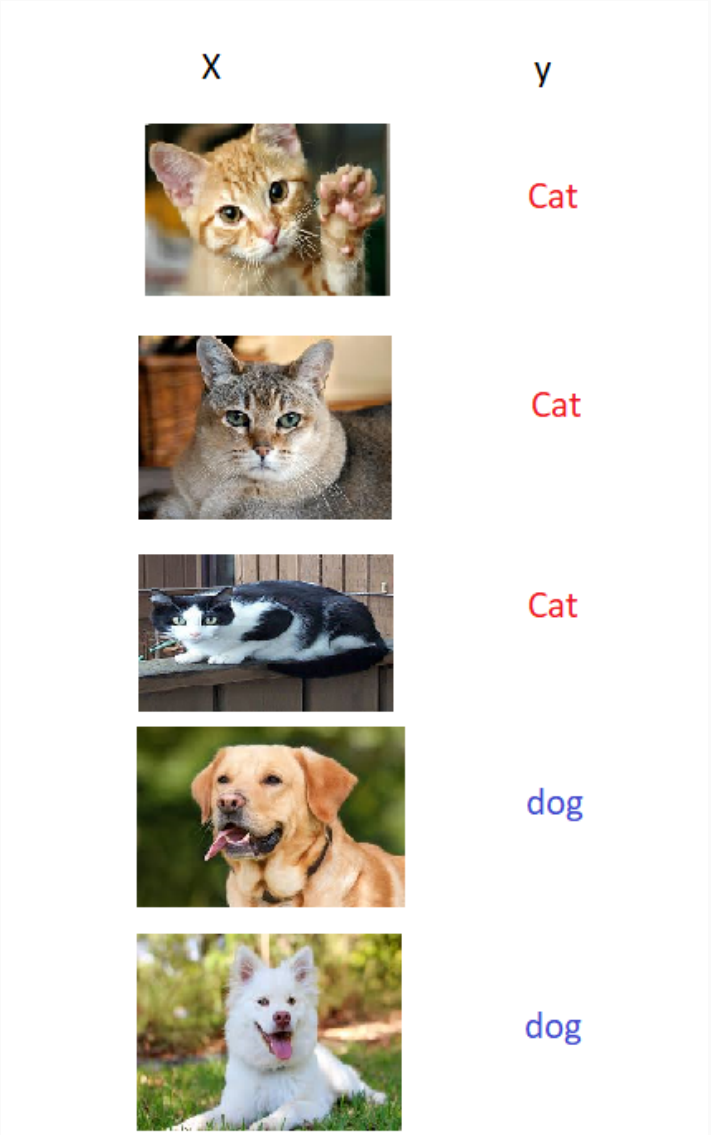
$$\mathbf{w_{t+1}} = \mathbf{w_t} + \alpha \frac{\partial J}{\partial \mathbf{w}}$$

Supervised Learning

Supervised Learning

Let $\{(\mathbf{x}_1, y_2), \ldots, (\mathbf{x}_n, y_n)\}$, $\mathbf{x}_i \in \mathrm{R}^n$,$y_i \in \mathrm{R}$ be the N pair of training samples and its associated class labels, then objective is to seek a function $f : \mathbf{X} \rightarrow \mathbf{y}$, where $\mathbf{X}$ is the input space and $\mathbf{y}$ is the output space. If there are c classes, then $y \in \{1, \ldots, c\}$

$$X = \begin{bmatrix} 0.1 & 0.2 & 0.3 \\ 0.25 & 0.32 & 0.39 \\ \vdots & \vdots & \vdots \\ 0.3 & 0.2 & 0.4 \end{bmatrix} \quad y = \begin{bmatrix} 0 \\ 0 \\ 1 \\ \vdots \\ 2 \\ 2 \end{bmatrix}$$



ARTIFICIAL NEURON

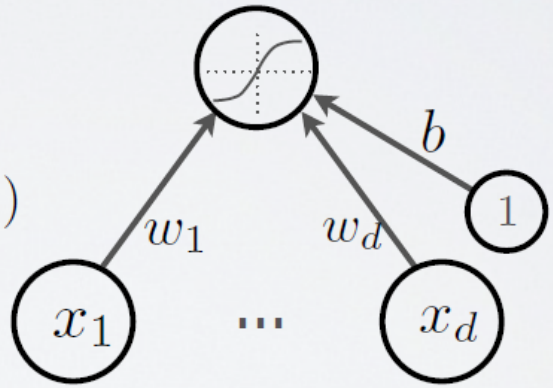
Topics: connection weights, bias, activation function

- Neuron pre-activation (or input activation):

$$a(\mathbf{x}) = b + \sum_i w_i x_i = b + \mathbf{w}^\top \mathbf{x}$$

- Neuron (output) activation

$$h(\mathbf{x}) = g(a(\mathbf{x})) = g(b + \sum_i w_i x_i)$$



- $\mathbf{w}$ are the connection weights
- b is the neuron bias
- $g(\cdot)$ is called the activation function

Activation functions

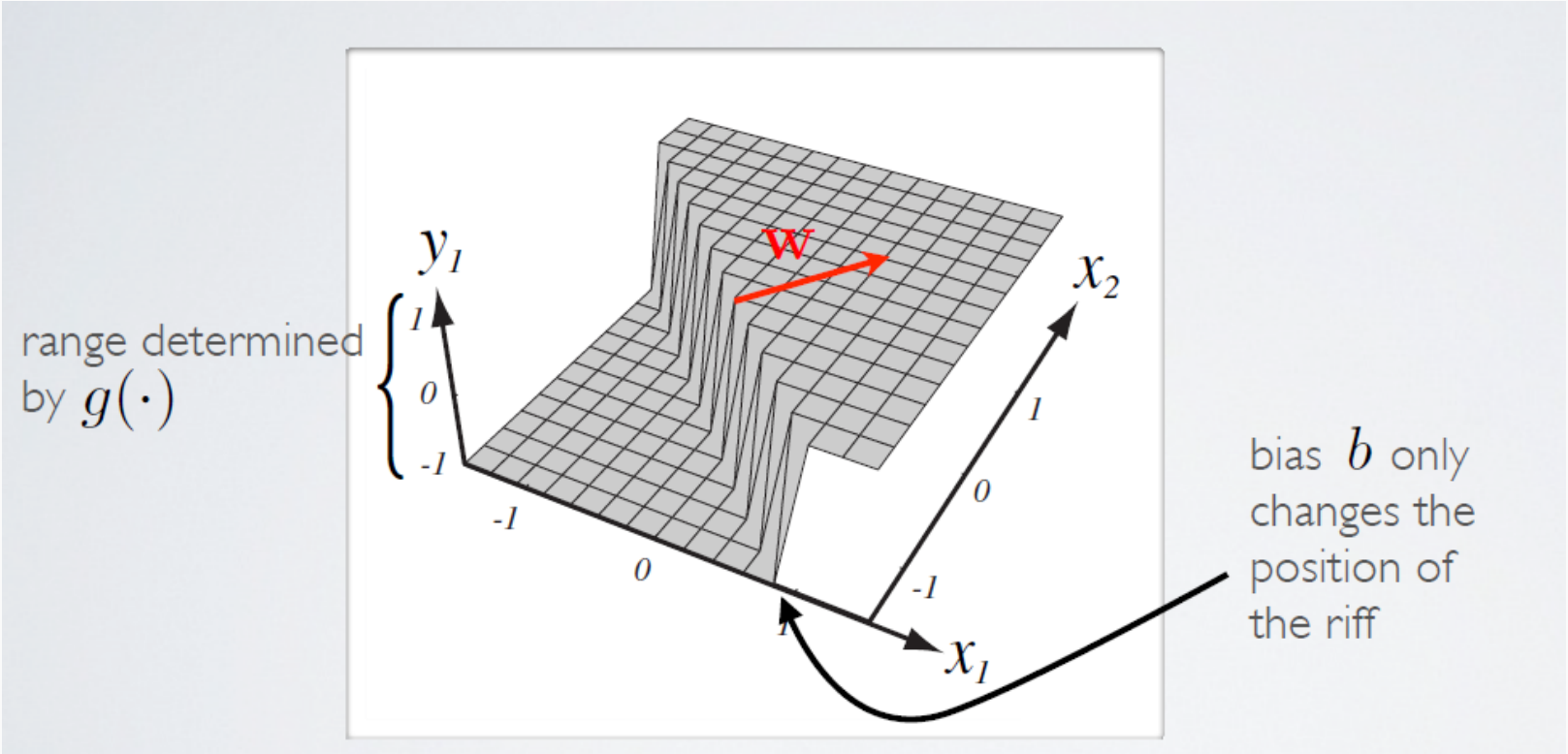
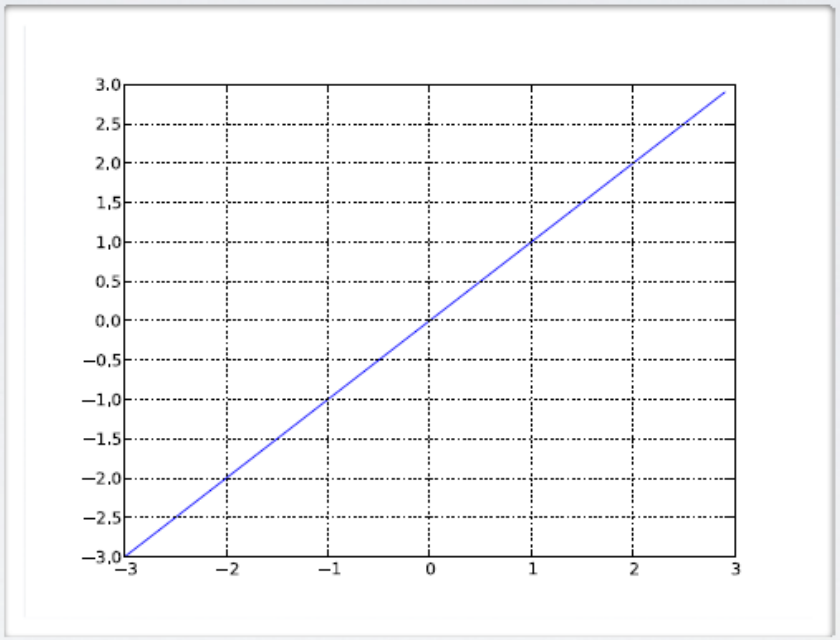


Figure 1: Activation function

Activation functions

- Performs no input squashing
- Not very interesting...



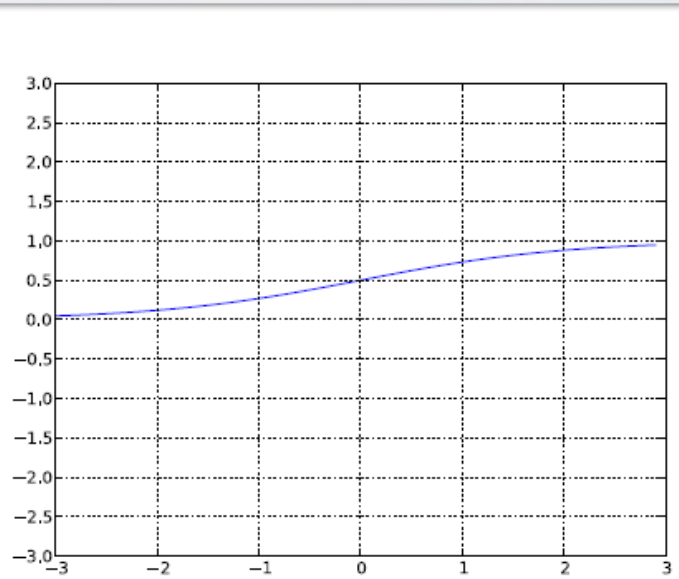
$$g(a) = a$$

Figure 2: Linear activation function

- doesn't introduce any non-linearity in the output

Activation functions

- Squashes the neuron's pre-activation between 0 and 1
- Always positive
- Bounded
- Strictly increasing



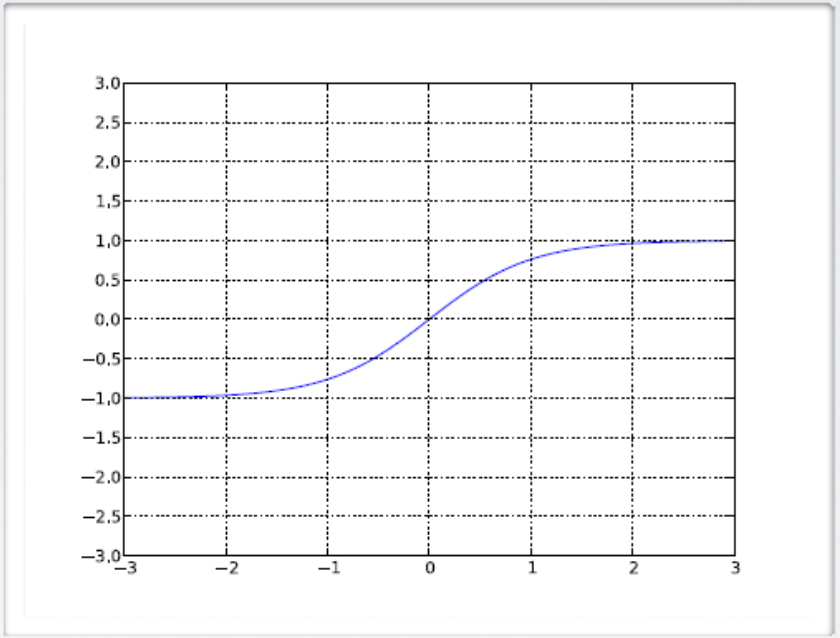
$$g(a) = \text{sigm}(a) = \frac{1}{1+\exp(-a)}$$

Figure 3: sigmoid activation function

- bigger the activation saturates close to 1
- smaller the activation saturates close to 0

Activation functions

- Squashes the neuron's pre-activation between -1 and 1
- Can be positive or negative
- Bounded
- Strictly increasing

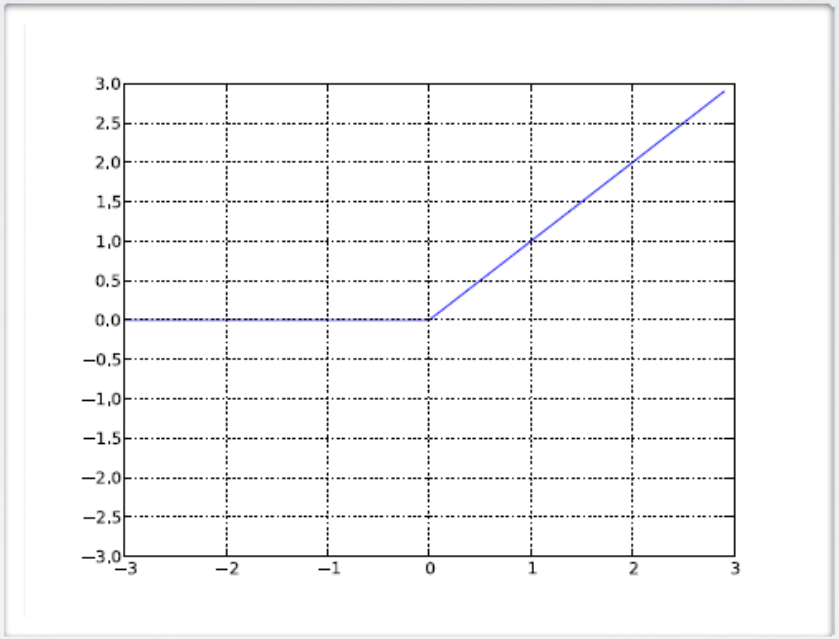


$$g(a) = \tanh(a) = \frac{\exp(a) - \exp(-a)}{\exp(a) + \exp(-a)} = \frac{\exp(2a) - 1}{\exp(2a) + 1}$$

Figure 4: tanh activation function

Activation functions

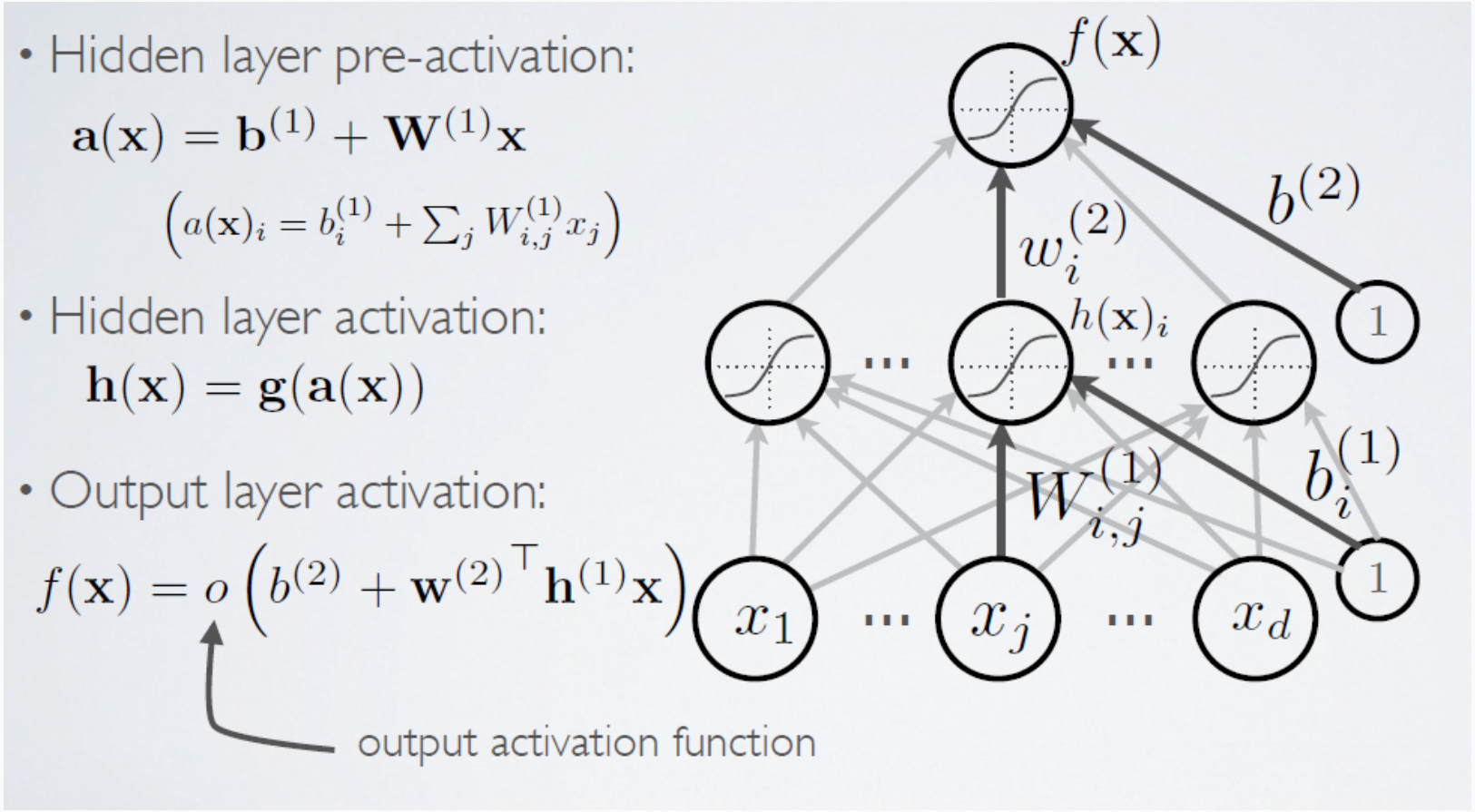
- Bounded below by 0 (always non-negative)
- Not upper bounded
- Strictly increasing
- Tends to give neurons with sparse activities



$$g(a) = \text{reclin}(a) = \max(0, a)$$

Figure 5: Relu activation function

Single layer neuron



- number of output layers depends on number of the classes to predict

Single layer neuron: Softmax Activation

- For multi-class classification:
  - we need multiple outputs (1 output per class)
  - we would like to estimate the conditional probability $p(y = c|\mathbf{x})$

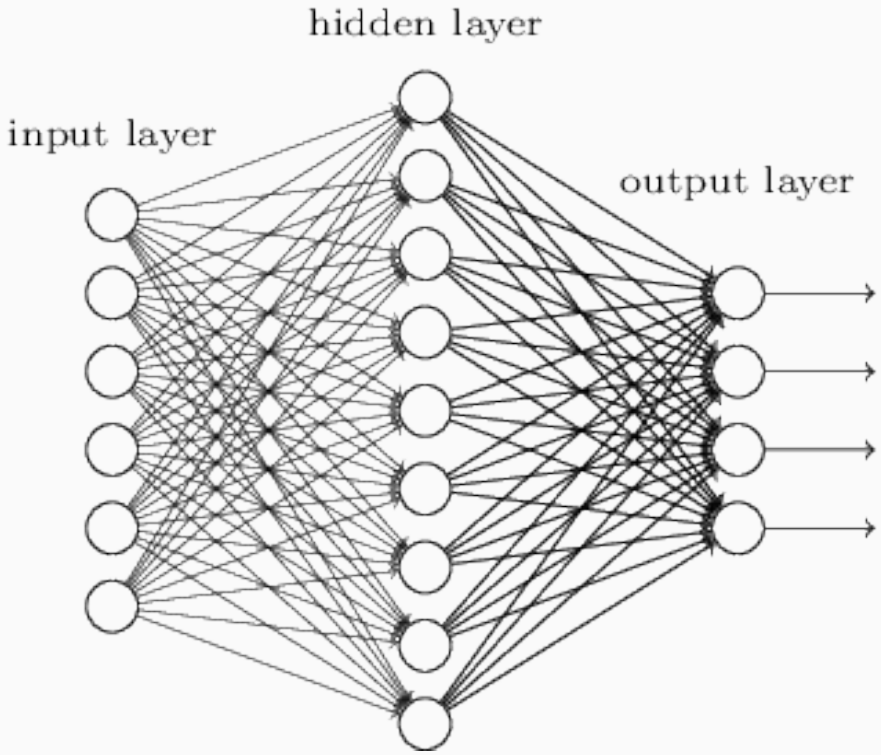
- We use the softmax activation function at the output:

$$\mathbf{o}(\mathbf{a}) = \text{softmax}(\mathbf{a}) = \left[\frac{\exp(a_1)}{\sum_c \exp(a_c)} \cdots \frac{\exp(a_C)}{\sum_c \exp(a_c)} \right]^\top$$

  - strictly positive
  - sums to one

- Predicted class is the one with highest estimated probability

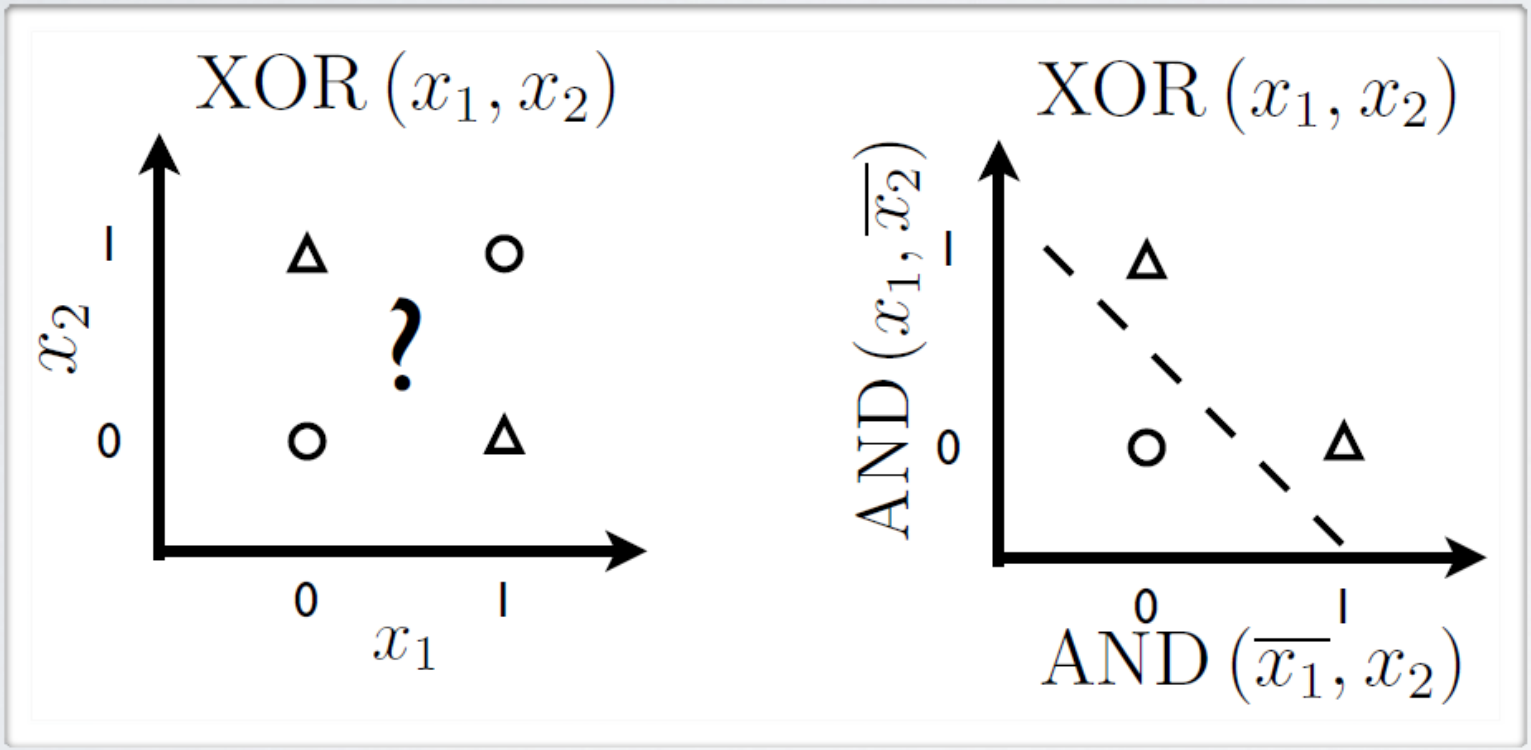
Single layer neuron



- hidden layer can be viewed as feature transformation layer
- number of output layer depends on number of classes in the data

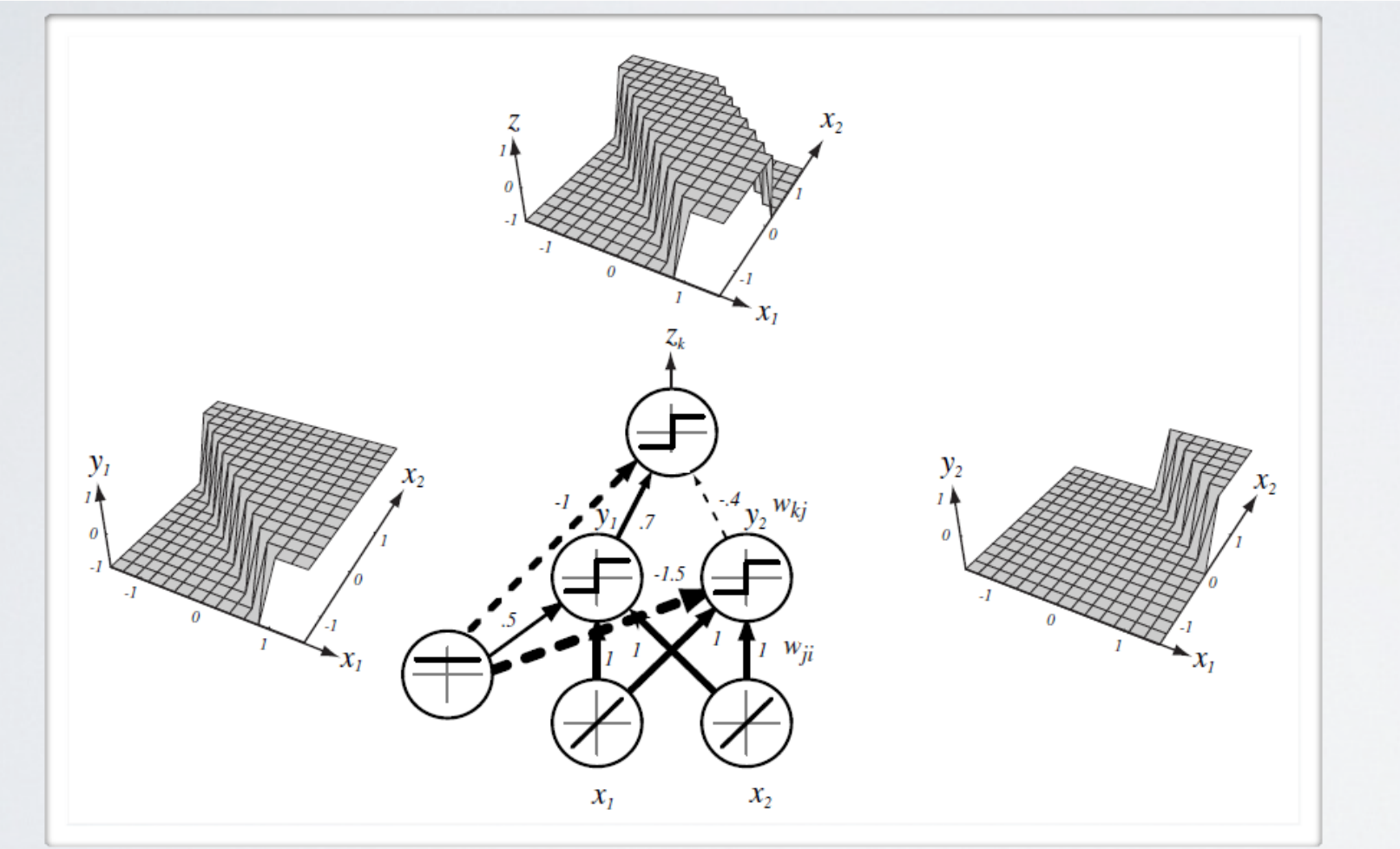
Capacity of Neural Network

- Can't solve non linearly separable problems...

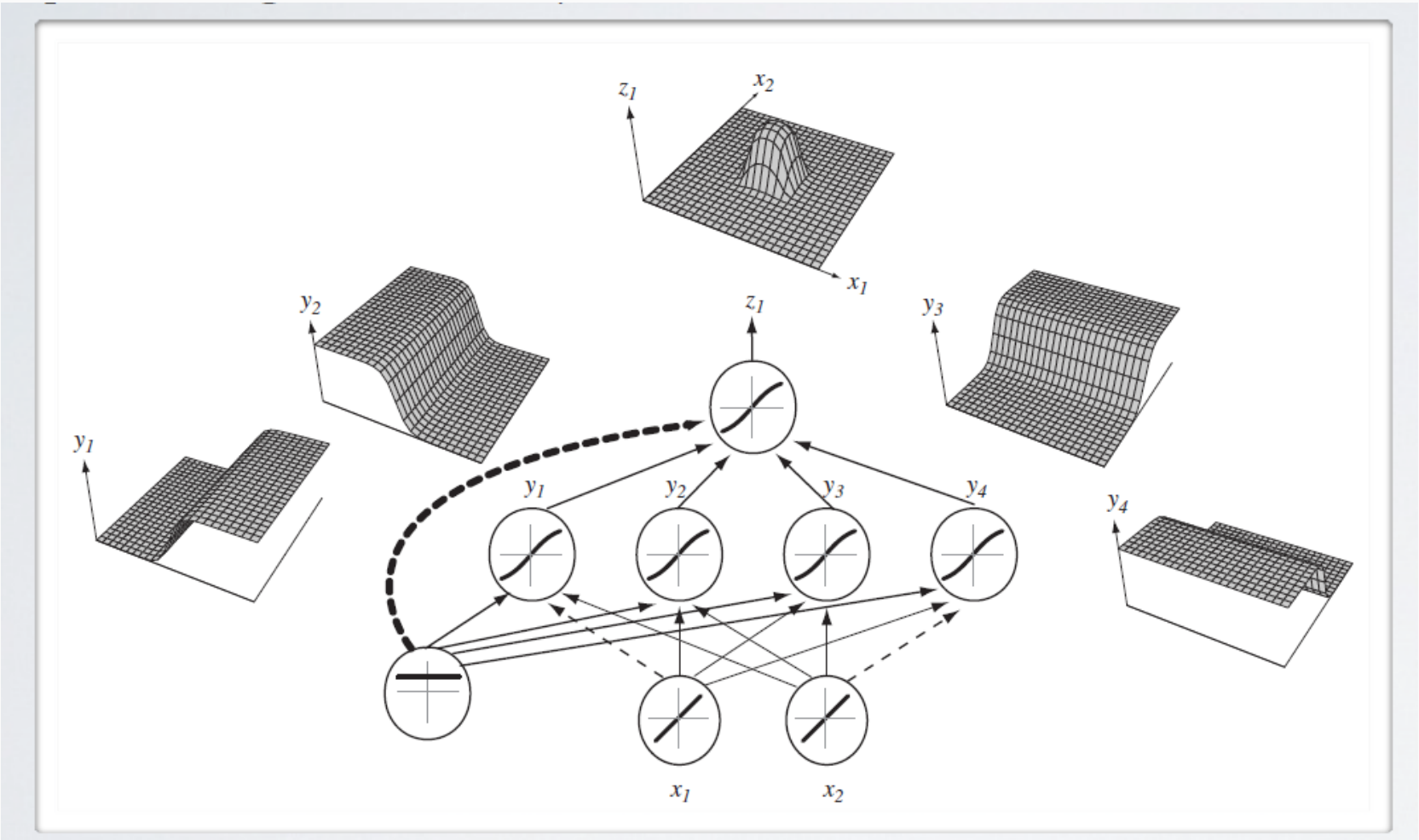


- ... unless the input is transformed in a better representation

Capacity of Neural Network



Capacity of Neural Network



- non-linear decision boundary can be obtained by combination of linear decision boundaries

Capacity of Neural Network

- Universal approximation theorem (Hornik, 1991):
  - "a single hidden layer neural network with a linear output unit can approximate any continuous function arbitrarily well, given enough hidden units"
- The result applies for sigmoid, tanh and many other hidden layer activation functions
- This is a good result, but it doesn't mean there is a learning algorithm that can find the necessary parameter values!

Multi-layer Neural Network

- Could have L hidden layers:
  - layer pre-activation for $k>0$ $(\mathbf{h}^{(0)}(\mathbf{x}) = \mathbf{x})$

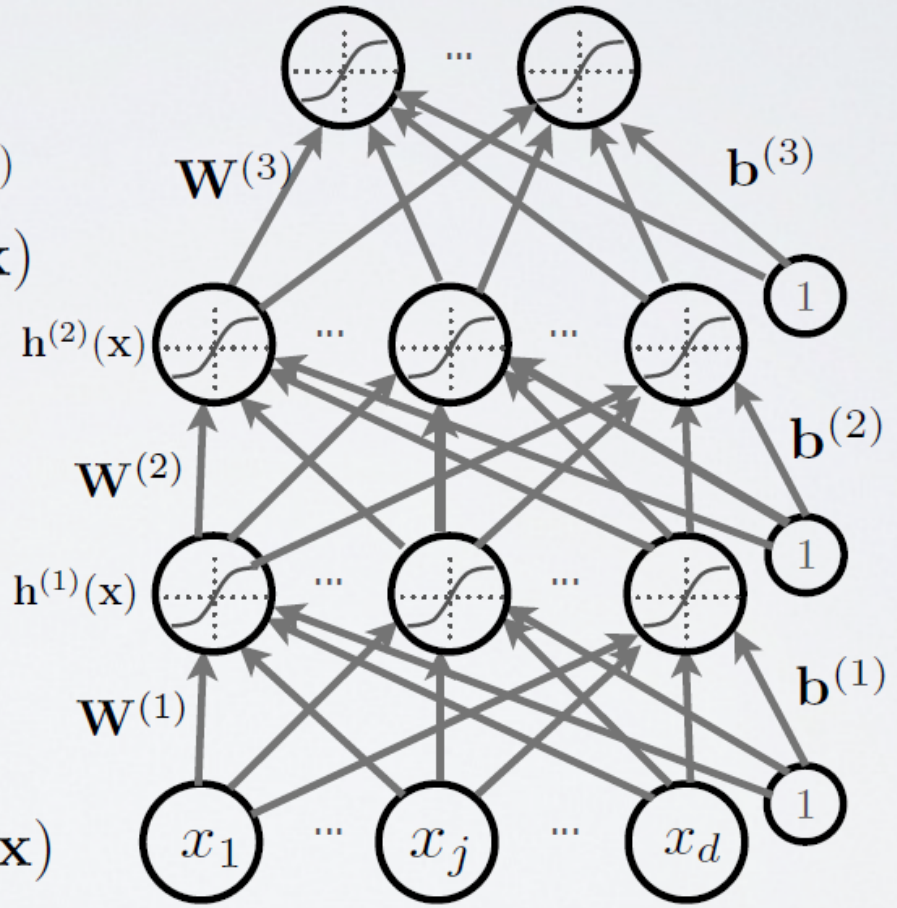
$$\mathbf{a}^{(k)}(\mathbf{x}) = \mathbf{b}^{(k)} + \mathbf{W}^{(k)}\mathbf{h}^{(k-1)}(\mathbf{x})$$

  - hidden layer activation (k from 1 to L):

$$\mathbf{h}^{(k)}(\mathbf{x}) = \mathbf{g}(\mathbf{a}^{(k)}(\mathbf{x}))$$

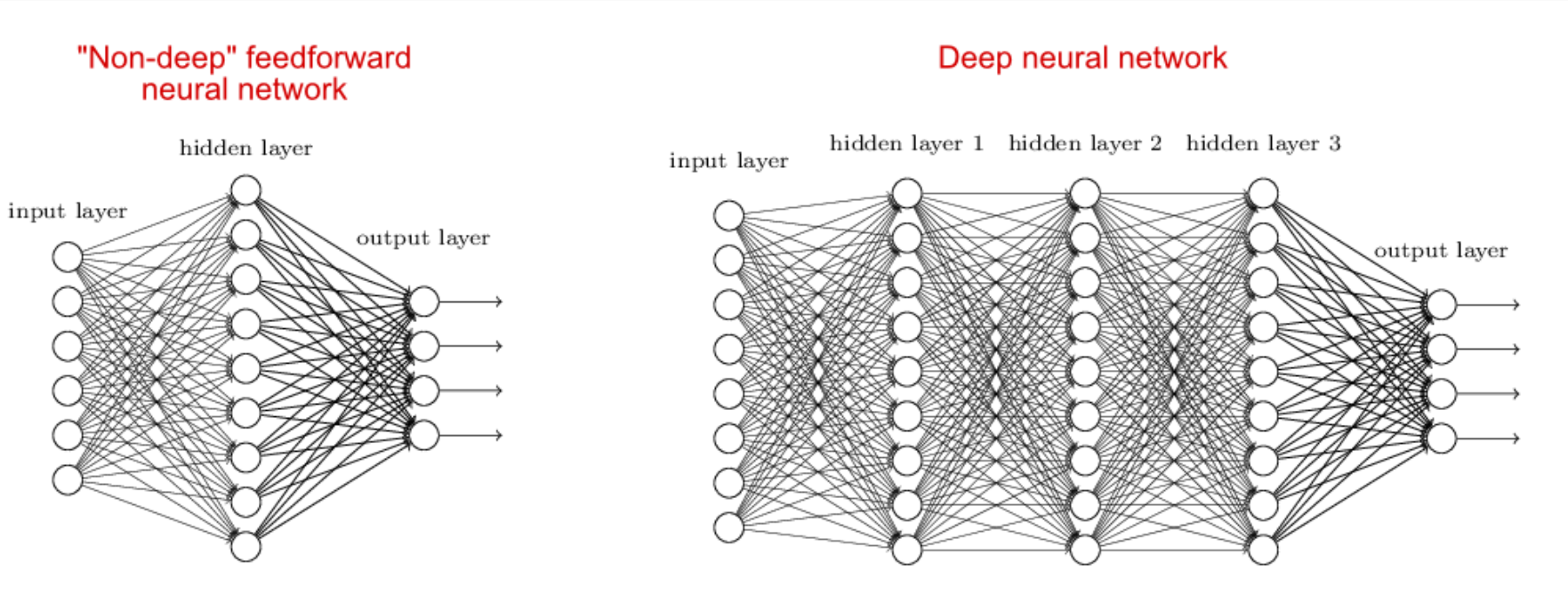
  - output layer activation ($k=L+1$):

$$\mathbf{h}^{(L+1)}(\mathbf{x}) = \mathbf{o}(\mathbf{a}^{(L+1)}(\mathbf{x})) = \mathbf{f}(\mathbf{x})$$



- feed forward network

Multi-layer Neural Network: Deep Networks



- depth of state of the art deep networks
  - VGG -16 and 19 layers
  - ResNet 50 layers
- we will see the complexity of model in final part of the course

Empricial Risk Minimization: Machine Learning

- Empirical risk minimization
  - framework to design learning algorithms

$$\arg\min_{\boldsymbol{\theta}} \frac{1}{T} \sum_{t} l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$$

  - $l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$ is a loss function

- Learning is cast as optimization
  - ideally, we'd optimize classification error, but it's not smooth
  - loss function is a surrogate for what we truly should optimize (e.g. upper bound)

- convert a training NN into a optimization problem
- maximum likelihood estimate

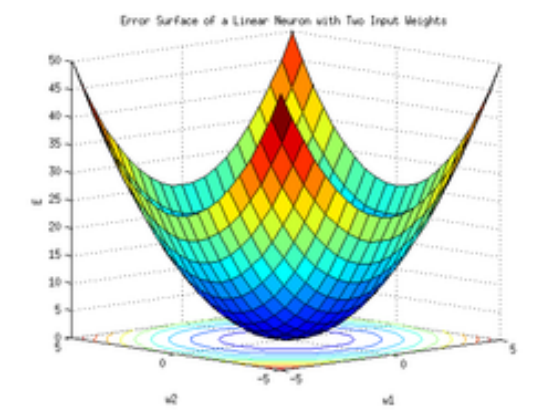
Loss functions: Mean square error

- mean square loss function - Regression problem
- when predicting real value quantities

predicted value

$$l(\mathbf{f}(\mathbf{x}), y) = (\mathbf{f}(\mathbf{x}) - y)^2$$

expected/actual value



Loss functions : Cross entropy

- cross entropy loss function
- one hot encoding

Loss functions : Cross entropy

- cross entropy loss function
- one hot encoding

Neural network estimates $f(\mathbf{x})_c = p(y = c|\mathbf{x})$

- we could maximize the probabilities of $y^{(t)}$ given $\mathbf{x}^{(t)}$ in the training set

To frame as minimization, we minimize the negative log-likelihood

natural log (ln)

$$l(\mathbf{f}(\mathbf{x}), y) = -\sum_c 1_{(y=c)} \log f(\mathbf{x})_c = -\log f(\mathbf{x})_y$$

- we take the log to simplify for numerical stability and math simplicity
- sometimes referred to as cross-entropy

Empricial Risk Minimization

- Empirical risk minimization
  - framework to design learning algorithms

$$\arg\min_{\boldsymbol{\theta}} \frac{1}{T} \sum_{t} l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$$

  - $l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$ is a loss function

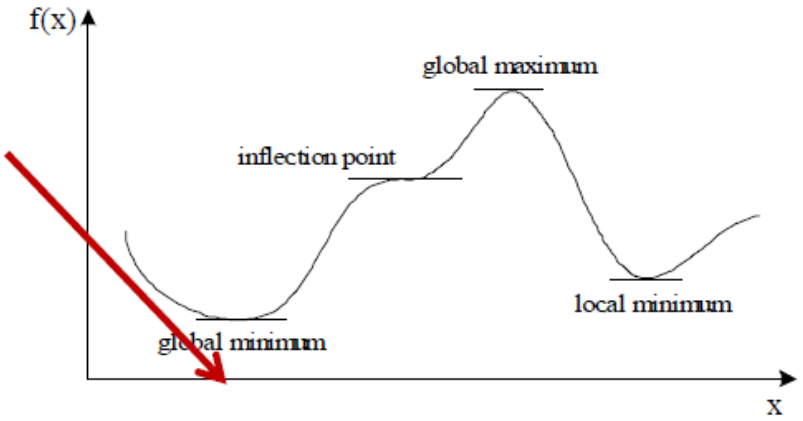
- Learning is cast as optimization
  - ideally, we'd optimize classification error, but it's not smooth
  - loss function is a surrogate for what we truly should optimize (e.g. upper bound)

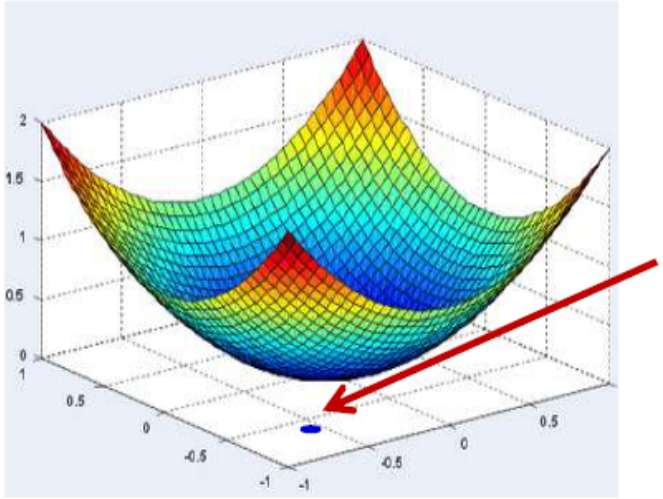
- maximum likelihood estimate

Caveat about following slides

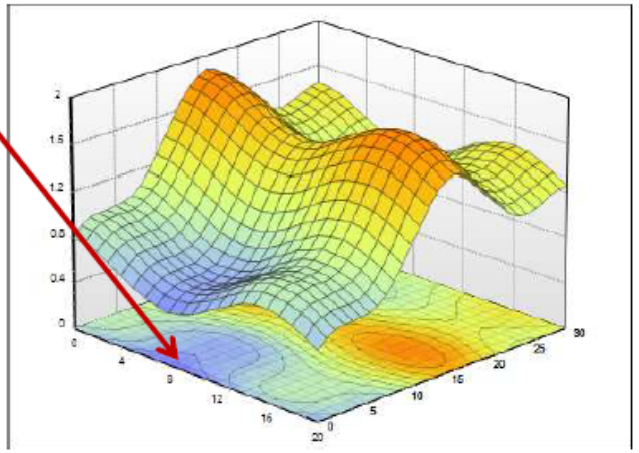
- The following slides speak of optimizing a function w.r.t a variable “x”
- This is only mathematical notation. In our actual network optimization problem we would be optimizing w.r.t. network weights “w”
- To reiterate – “x” in the slides represents the variable that we’re optimizing a function over and not the input to a neural network
- **Do not get confused!**

The problem of optimization

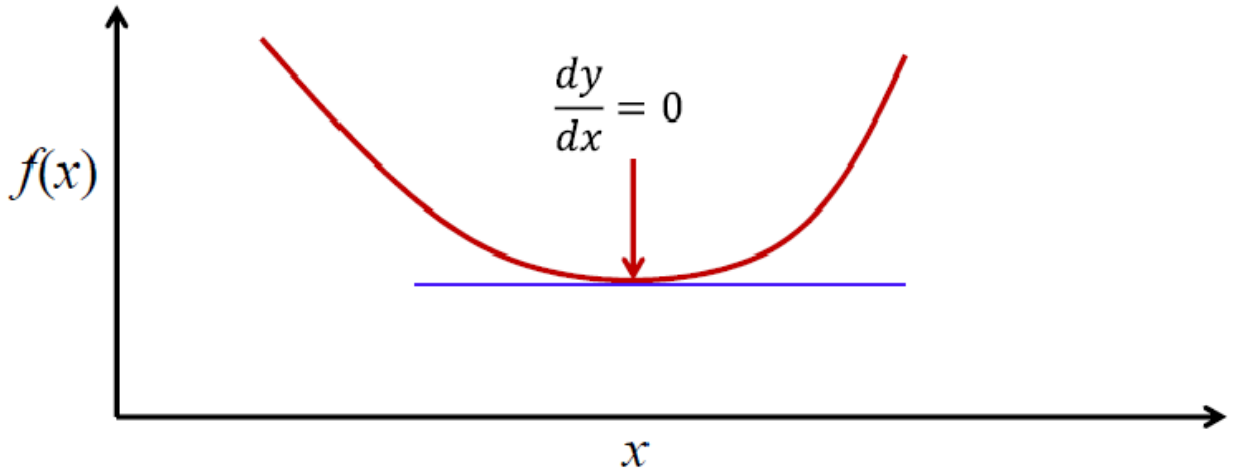




- General problem of optimization: find the value of x where $\mathbf{f}(x)$ is minimum



Finding the minimum of a function

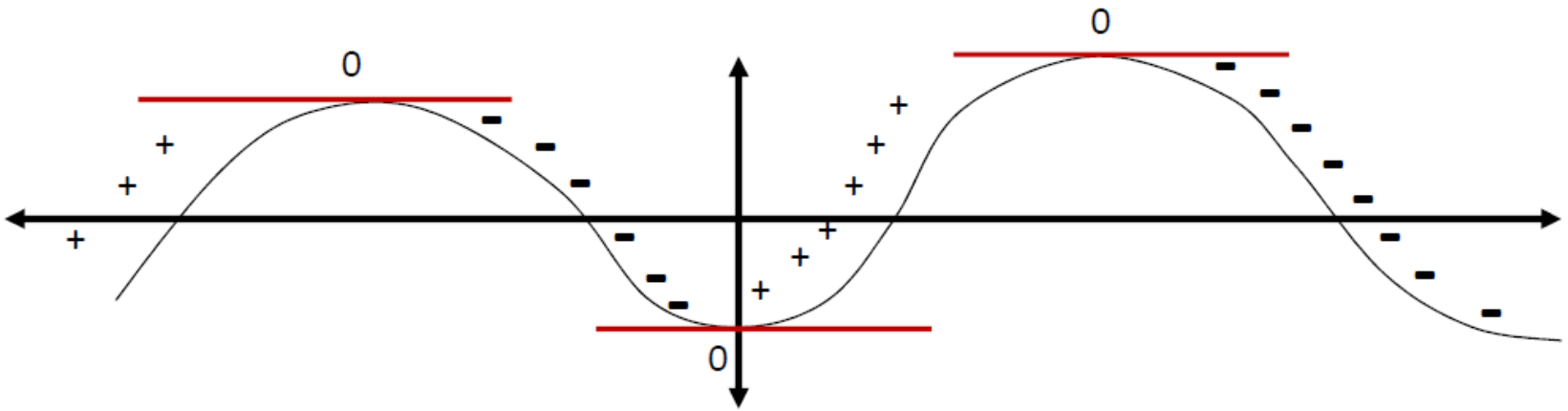


- Find the value x at which $f'(x) = 0$
  - Solve

$$\frac{df(x)}{dx} = 0$$

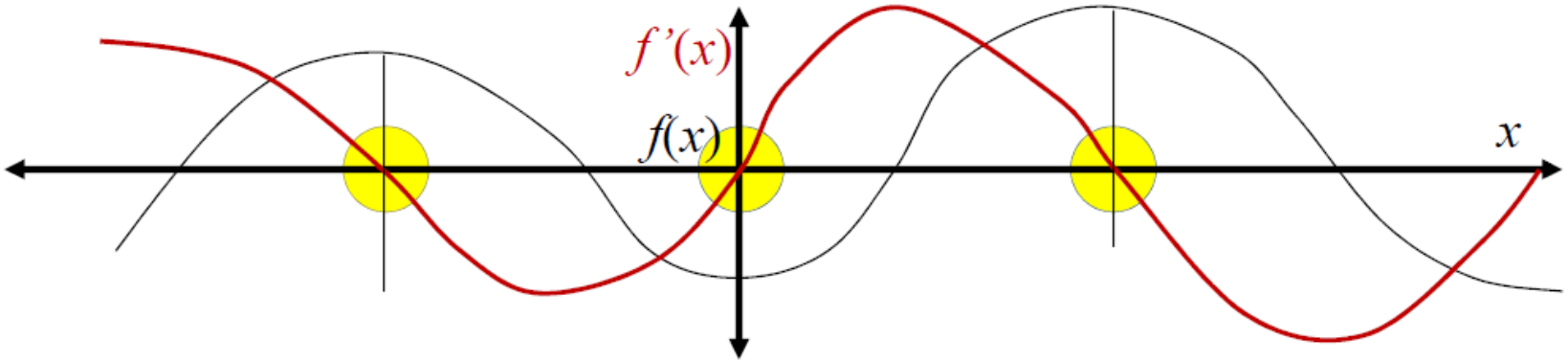
- The solution is a "turning point"
  - Derivatives go from positive to negative or vice versa at this point
- But is it a minimum?

Turning Points



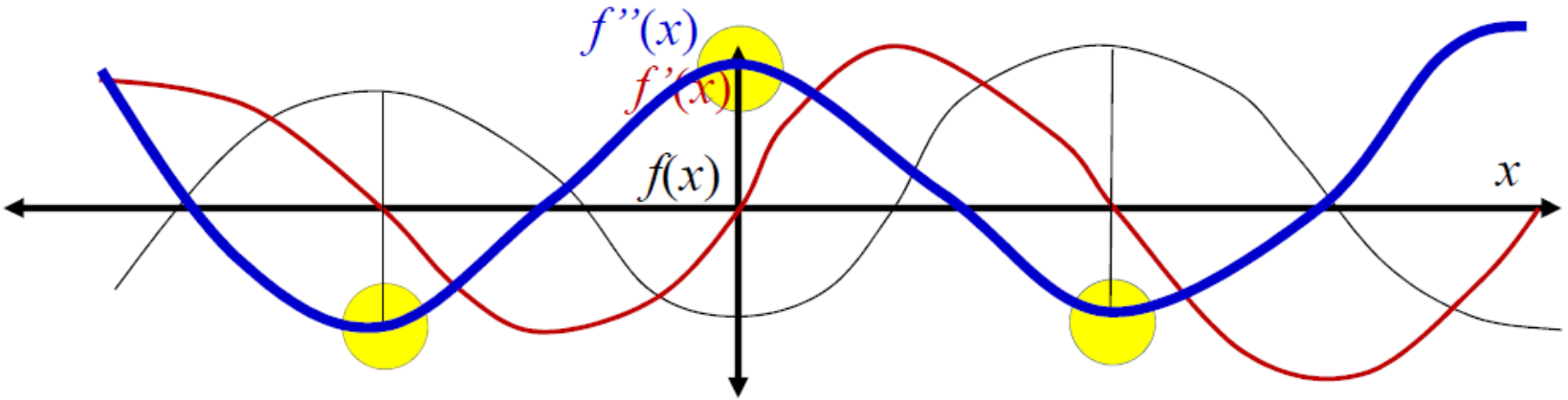
- Both *maxima* and *minima* have zero derivative
- Both are turning points

Derivatives of a curve



- Both *maxima* and *minima* are turning points
- Both *maxima* and *minima* have zero derivative

Derivative of the derivative of the curve

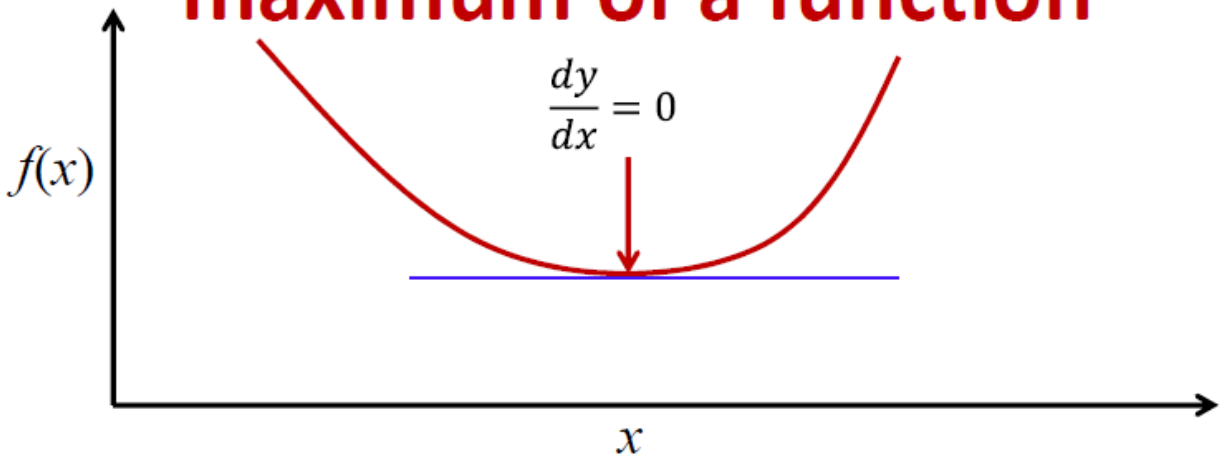


- Both *maxima* and *minima* are turning points
- Both *maxima* and *minima* have zero derivative

- The *second derivative* $f''(x)$ is –ve at maxima and +ve at minima!

Optimization: Recap

Soln: Finding the minimum or maximum of a function



- Find the value x at which $f'(x) = 0$: Solve

$$\frac{df(x)}{dx} = 0$$

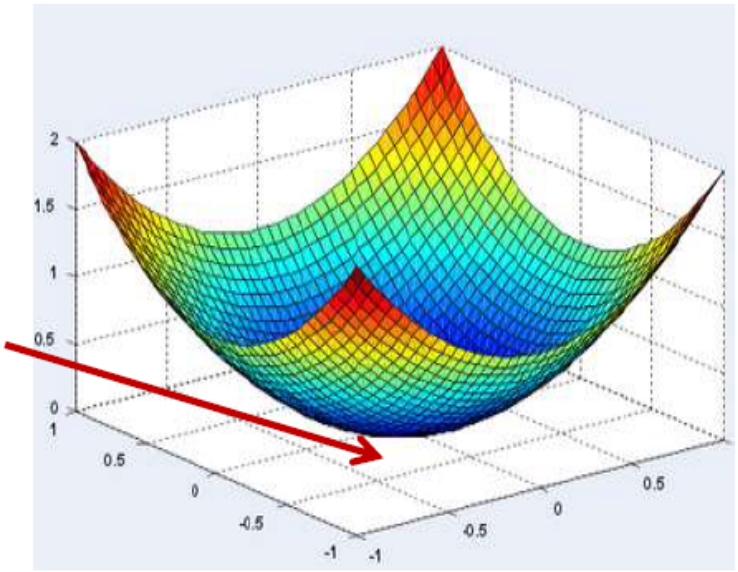
- The solution x_{soln} is a turning point
- Check the double derivative at x_{soln} : compute

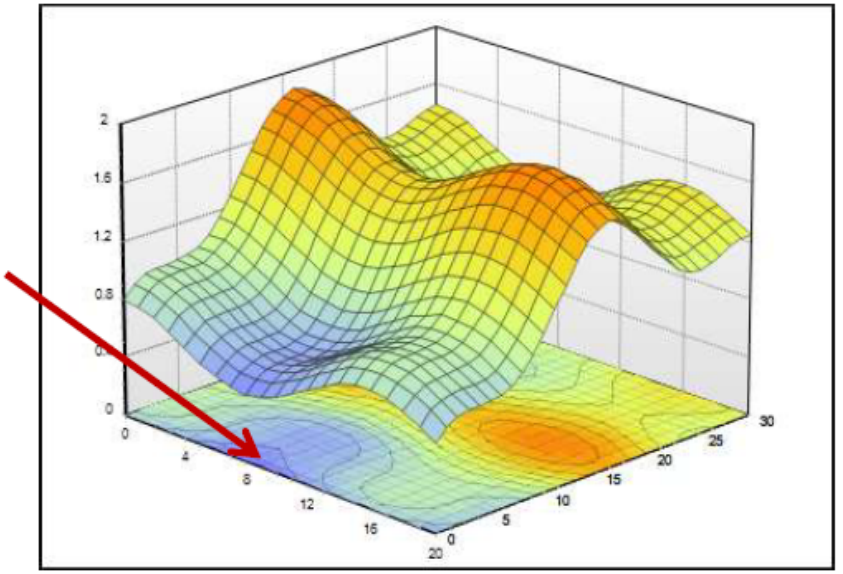
$$f''(x_{soln}) = \frac{df'(x_{soln})}{dx}$$

- If $f''(x_{soln})$ is positive x_{soln} is a minimum, otherwise it is a maximum

Optimization: Recap

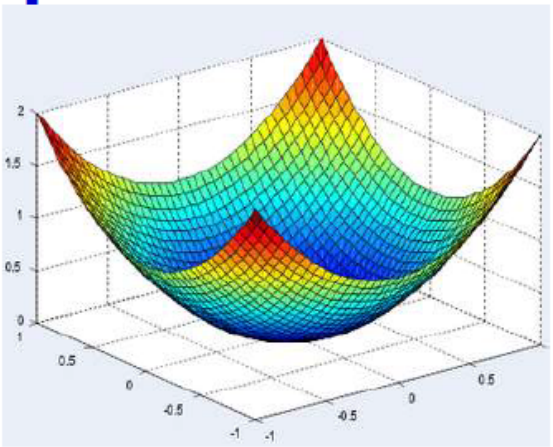
What about functions of multiple variables?





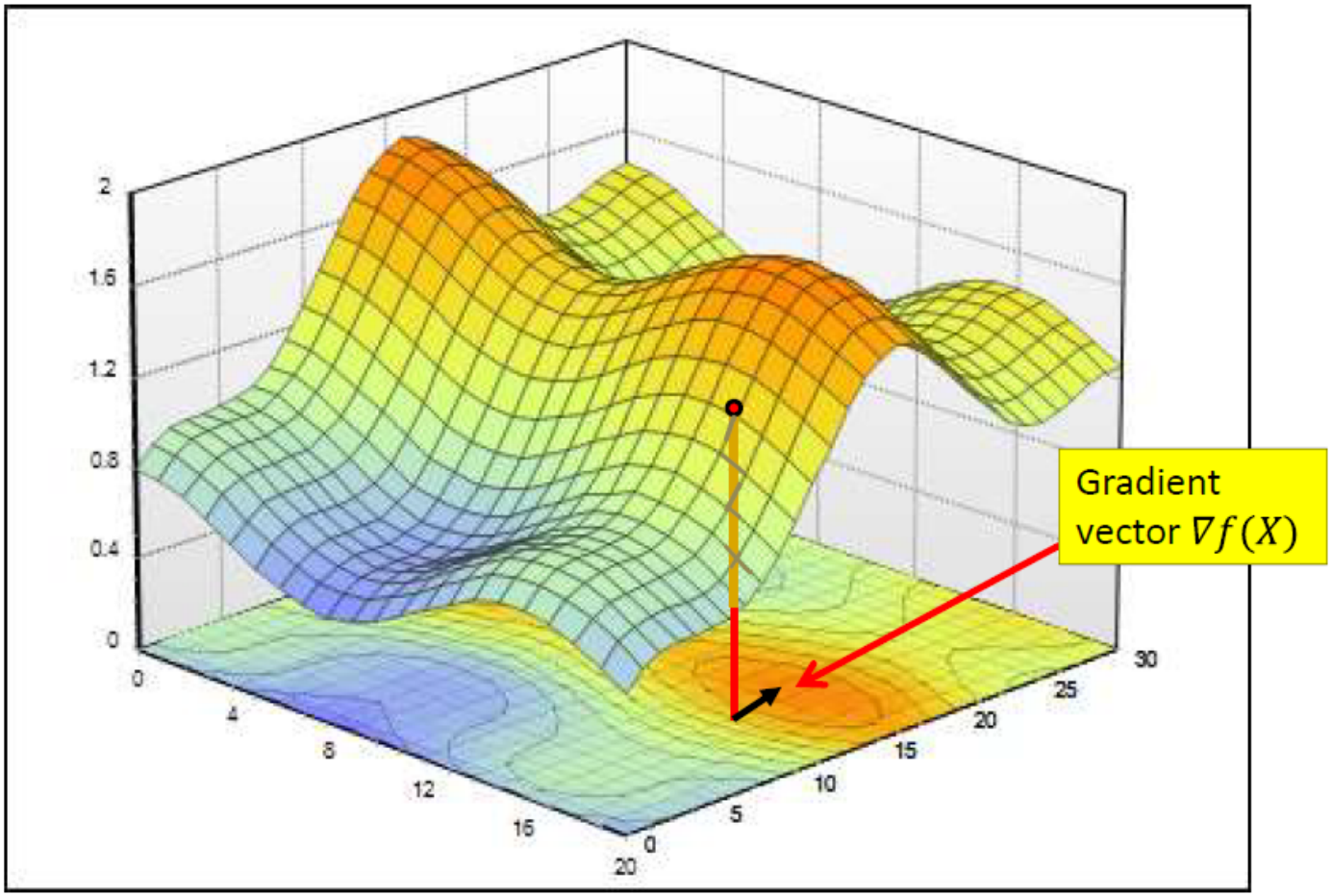
Gradients of scalar functions with multi-variate inputs

- Consider $f(X) = f(x_1, x_2, \ldots, x_n)$

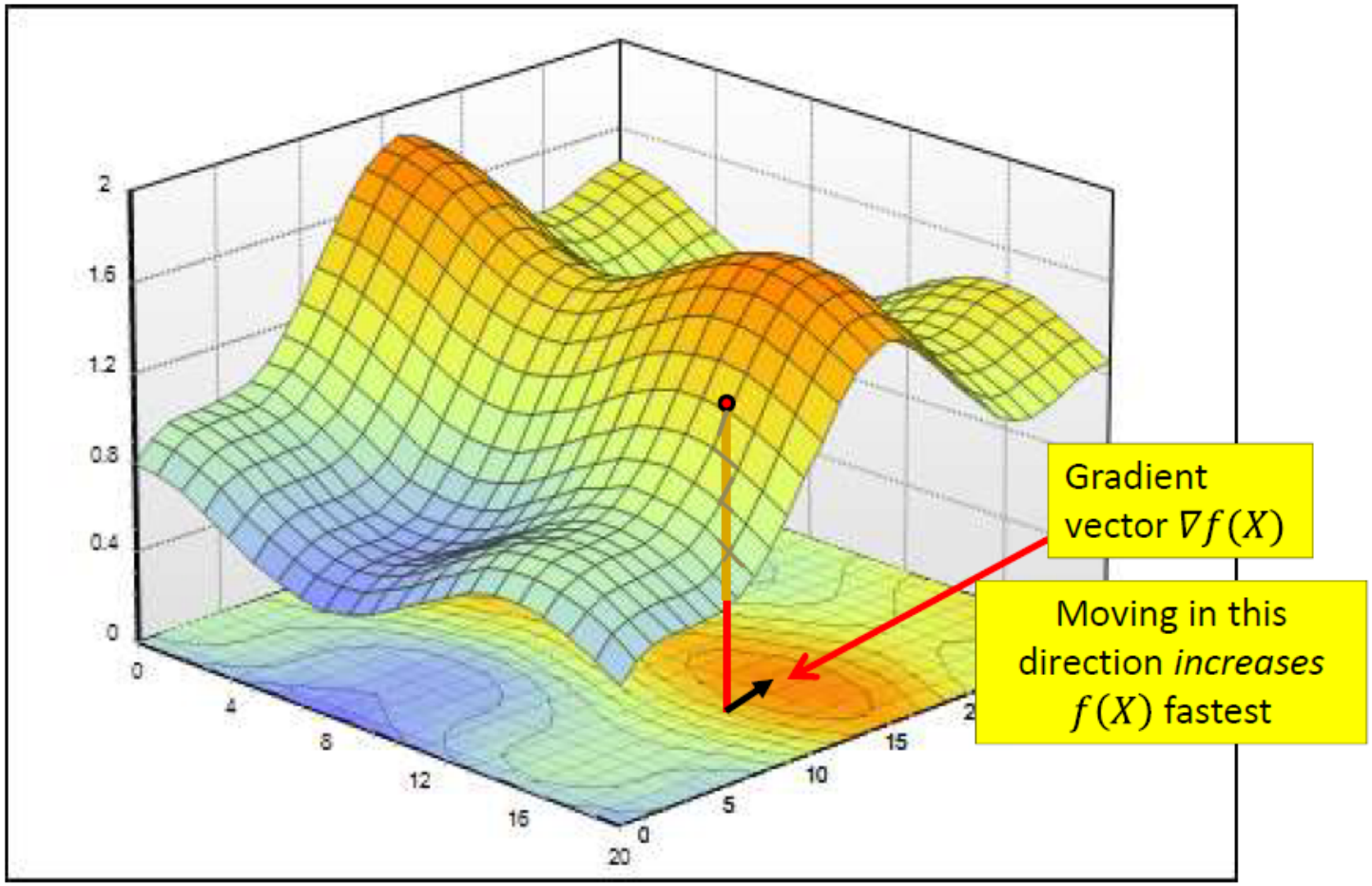


$$\nabla f(X) = \left[\frac{\partial f(X)}{\partial x_1} \quad \frac{\partial f(X)}{\partial x_2} \quad \ldots \quad \frac{\partial f(X)}{\partial x_n}\right]$$

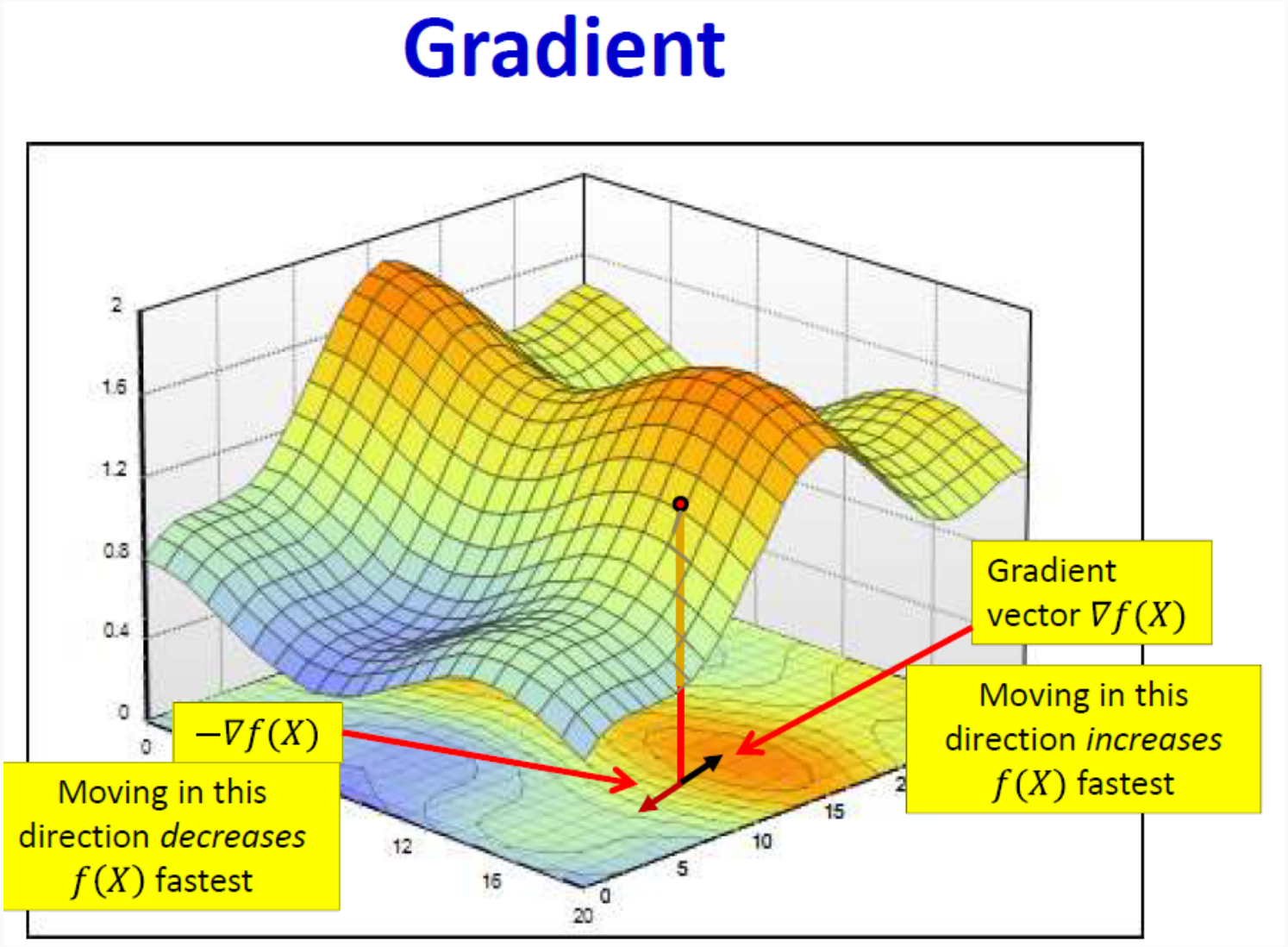
Gradient


2
1.6
1.2
0.8
0.4
0
0
4
8
12
16
20
0
5
10
15
20
25
30
Gradient
vector $\nabla f(X)$

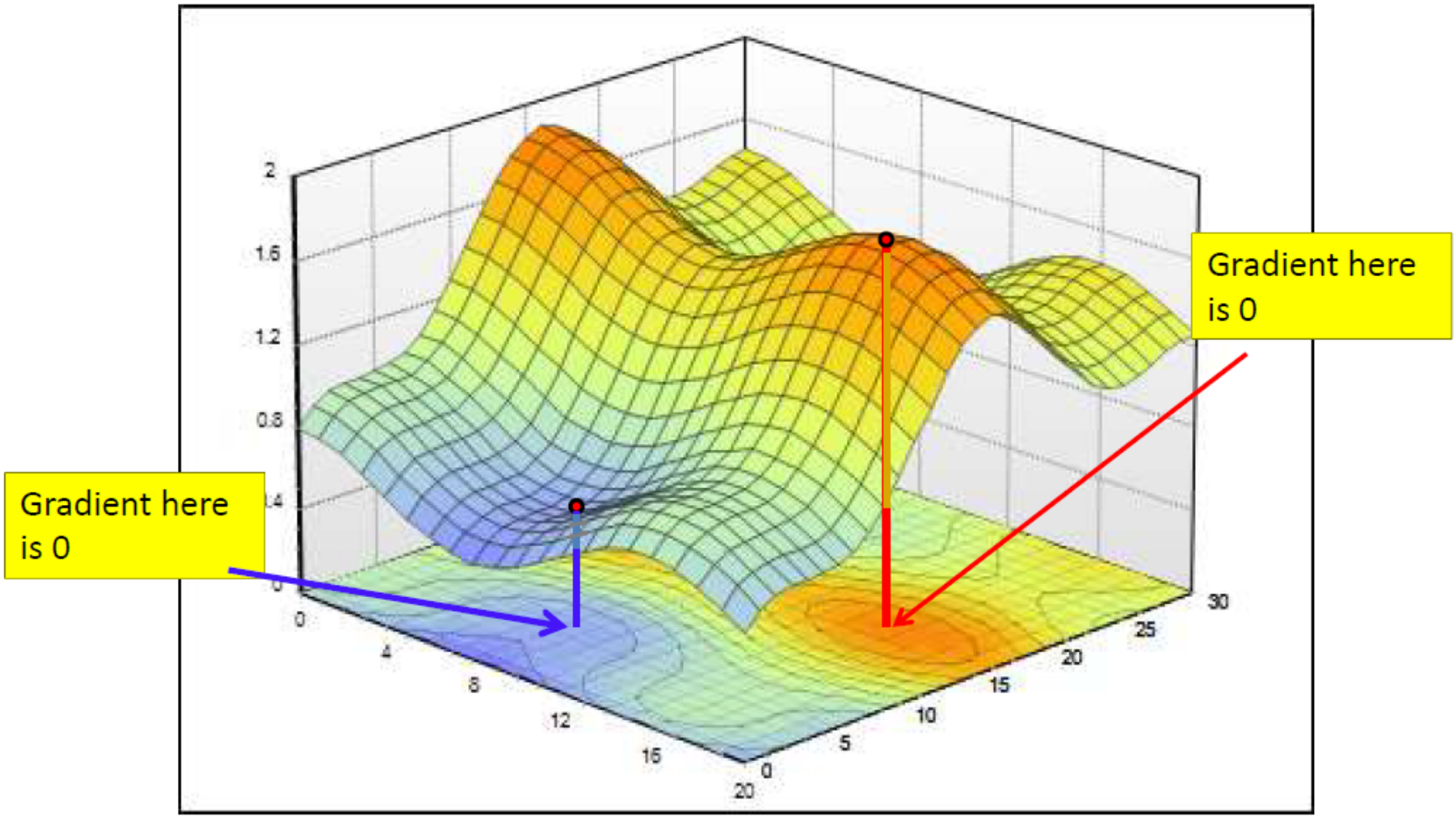
Gradient


2
1.6
1.2
0.8
0.4
0
0
4
8
12
16
20
0
5
10
15
Gradient vector $\nabla f(X)$
Moving in this direction *increases* $f(X)$ fastest

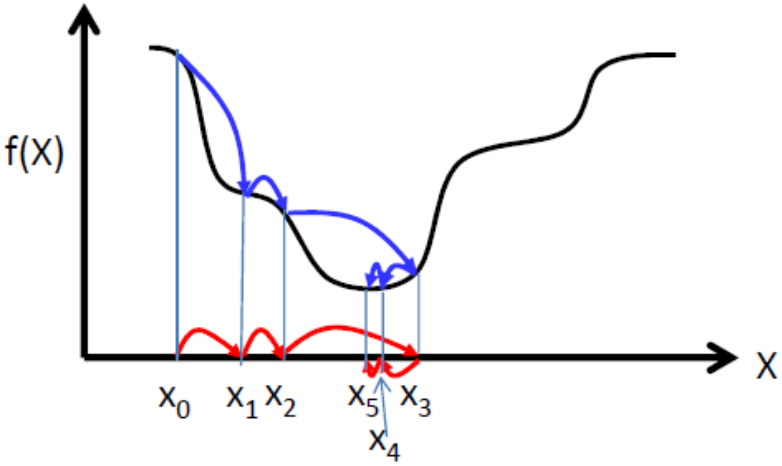
Optimization: Recap

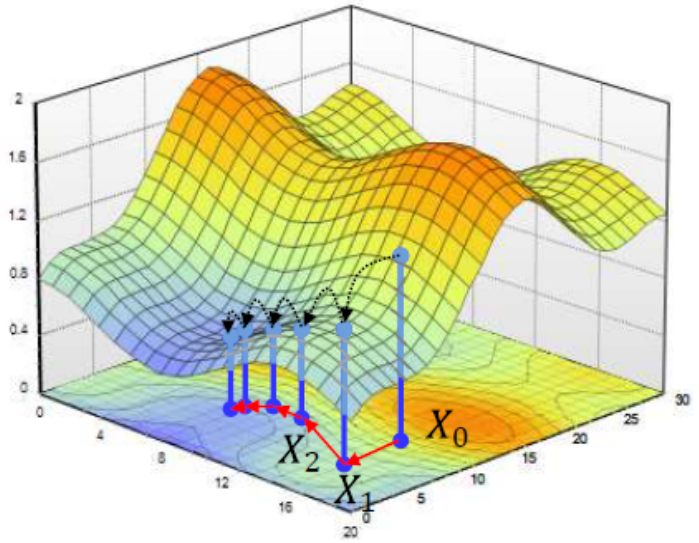


Gradient


Gradient here is 0
Gradient here is 0
2
1.6
1.2
0.8
0
4
8
12
16
20
0
5
10
15
20
25
30

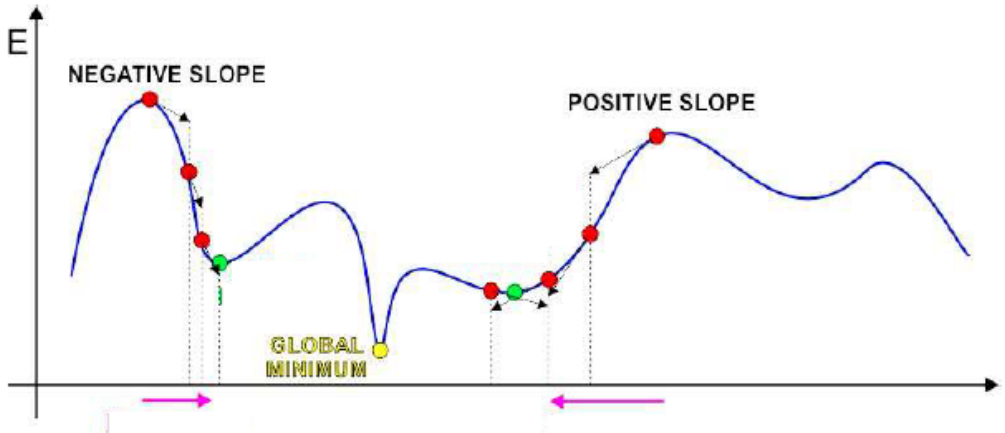
Iterative solutions





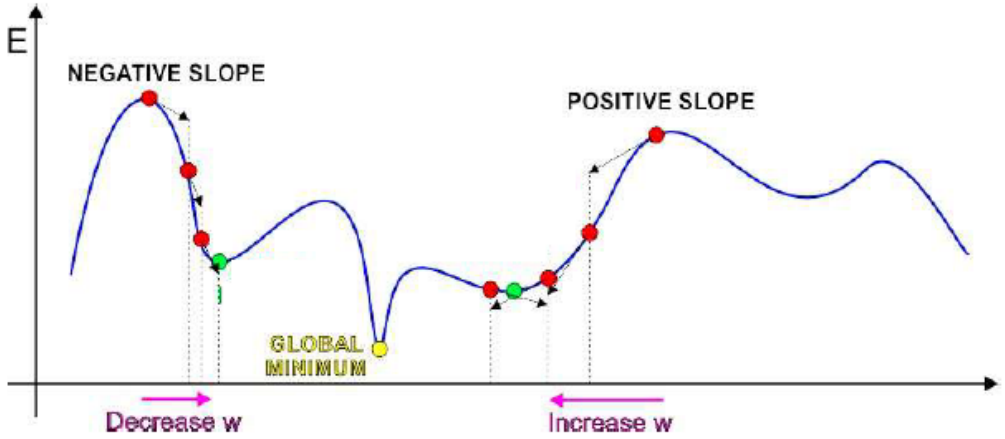
- Iterative solutions
  - Start from an initial guess X_0 for the optimal X
  - Update the guess towards a (hopefully) "better" value of $f(X)$
  - Stop when $f(X)$ no longer decreases
- Problems:
  - Which direction to step in
  - How big must the steps be

The Approach of Gradient Descent



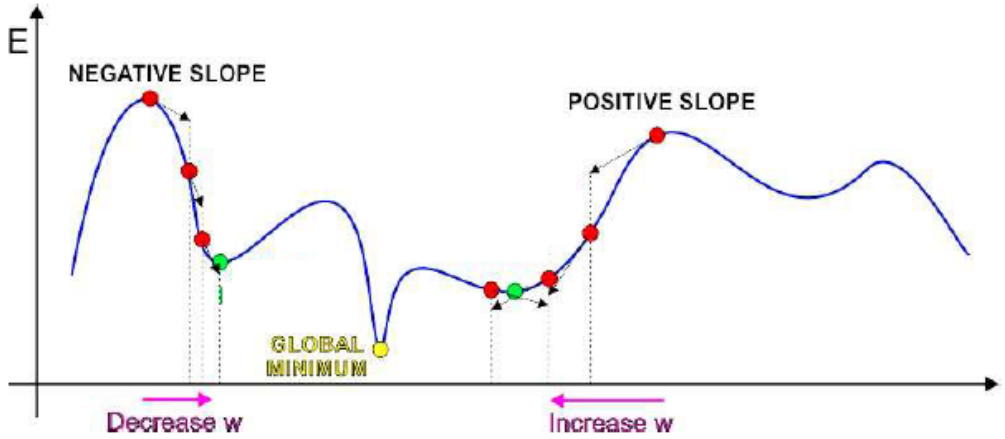
- Iterative solution:
  - Start at some point
  - Find direction in which to shift this point to decrease error
    - This can be found from the derivative of the function
      - A positive derivative → moving left decreases error
      - A negative derivative → moving right decreases error
  - Shift point in this direction

The Approach of Gradient Descent



- Iterative solution: Trivial algorithm
  - Initialize x^0
  - While $f'(x^k) \neq 0$
    - $x^{k+1} = x^k - sign\left(f'(x^k)\right).step$
  - Identical to previous algorithm

The Approach of Gradient Descent



- Iterative solution: Trivial algorithm
  - Initialize x_0
  - While $f'(x^k) \neq 0$
    - $x^{k+1} = x^k - \eta^k f'(x^k)$
  - η^k is the "step size"

Gradient descent/ascent (multivariate)

- The gradient descent/ascent method to find the minimum or maximum of a function f iteratively
  - To find a *maximum* move *in the direction of the gradient*

$$x^{k+1} = x^k + \eta^k \nabla f\left(x^k\right)^T$$

  - To find a *minimum* move *exactly opposite the direction of the gradient*

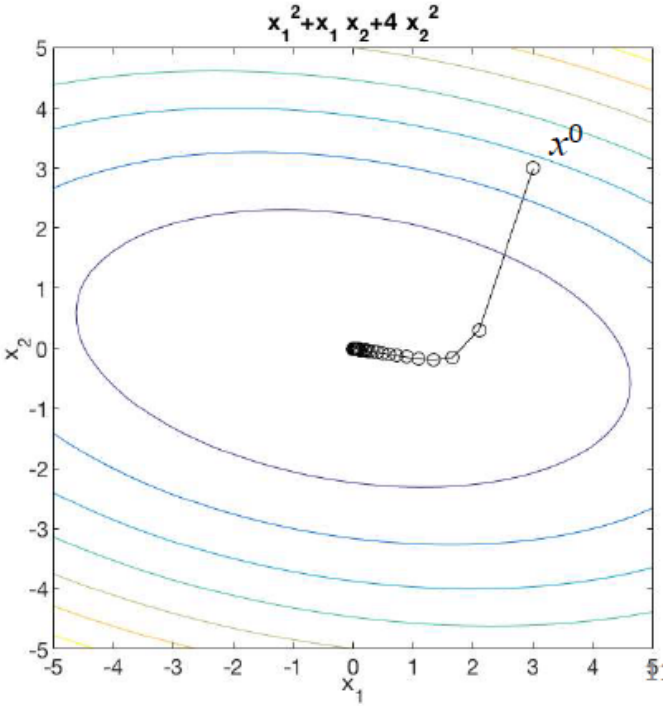
$$x^{k+1} = x^k - \eta^k \nabla f\left(x^k\right)^T$$

- Many solutions to choosing step size η^k

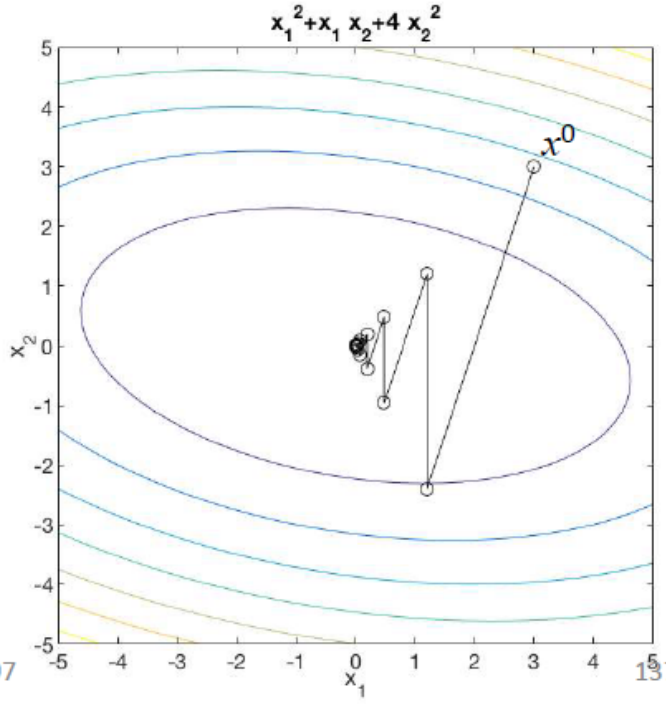
Influence of step size example (constant step size)

$$f(x_1, x_2) = (x_1)^2 + x_1 x_2 + 4(x_2)^2 \qquad x^{initial} = \begin{bmatrix} 3 \\ 3 \end{bmatrix}$$

$\eta = 0.1$



$\eta = 0.2$



ERM: Back to NN

- Empirical risk minimization
  - framework to design learning algorithms

$$\arg\min_{\boldsymbol{\theta}} \frac{1}{T} \sum_{t} l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$$

  - $l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$ is a loss function

- Learning is cast as optimization
  - ideally, we'd optimize classification error, but it's not smooth
  - loss function is a surrogate for what we truly should optimize (e.g. upper bound)

$\boldsymbol{\theta}$ ($\boldsymbol{\theta} \equiv \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \ldots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\}$)

Gradient Descent Algorithm

Algorithm 2: Gradient descent Algorithm

Input: Input Data, $\theta = \{\mathbf{W^{(1)}}, \mathbf{b^{(1)}}, \mathbf{W^{(2)}}, \mathbf{b^{(2)}}, \ldots, \mathbf{W^{(L+1)}}, \mathbf{b^{(L+1)}}\}$

Require: learning rate η, and stopping criteria $\Delta\theta$, and $\delta = 1e^{-8}$

while *untill stopping criteria* $> \delta$ **do**

compute the gradient of the loss function

$\frac{\partial J}{\partial \theta} = \frac{1}{T}\sum_t \bigtriangledown l(\mathbf{f}(\mathbf{x_t}, \mathbf{y_t})$

$\theta_{it} := \theta_{it-1} - \eta\frac{\partial J}{\partial \theta}$

$\Delta\theta = \theta_{it} - \theta_{it-1}$

$it = it + 1$

end

Output: Learned weight vectors θ of the Network

- once you have learned the weights, you can use this model to predict the outcome of the testing data or new input data

Multilayer NN -SGD

Topics: stochastic gradient descent (SGD)

- Algorithm that performs updates after each example
  - initialize $\boldsymbol{\theta}$ ($\boldsymbol{\theta} \equiv \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \ldots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\}$)
  - for N iterations
    - for each training example $(\mathbf{x}^{(t)}, y^{(t)})$
      - ✓ $\Delta = -\nabla_{\boldsymbol{\theta}} l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)}) - \lambda \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$
      - ✓ $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \Delta$

    } training epoch = iteration over **all** examples

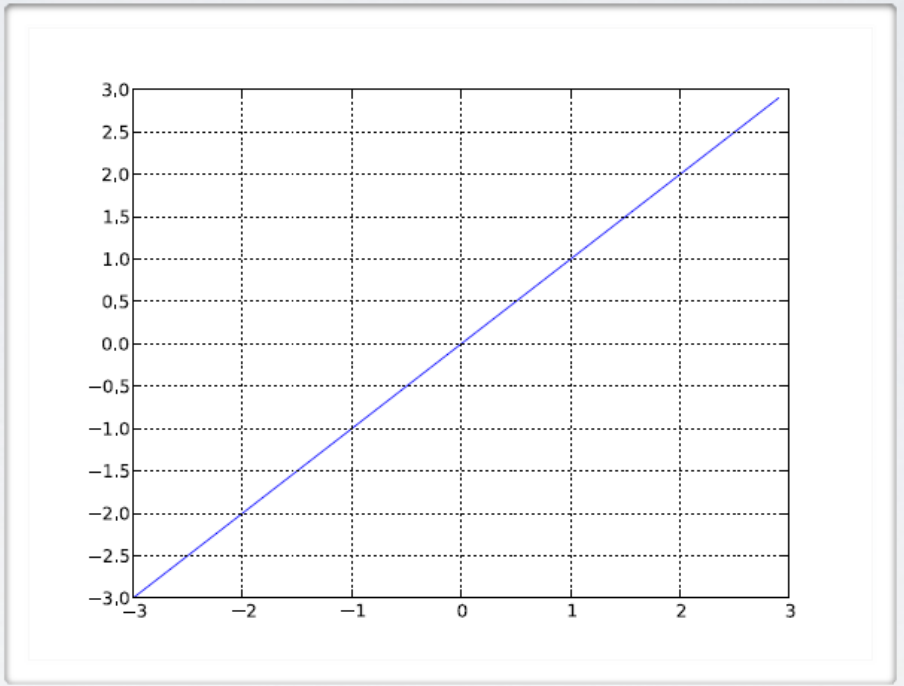
- To apply this algorithm to neural network training, we need
  - the loss function $l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
  - a procedure to compute the parameter gradients $\nabla_{\boldsymbol{\theta}} l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
  - the regularizer $\Omega(\boldsymbol{\theta})$ (and the gradient $\nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$)
  - initialization method

Gradient of Activation Functions

Topics: linear activation function gradient

- Partial derivative:

$$g'(a) = 1$$



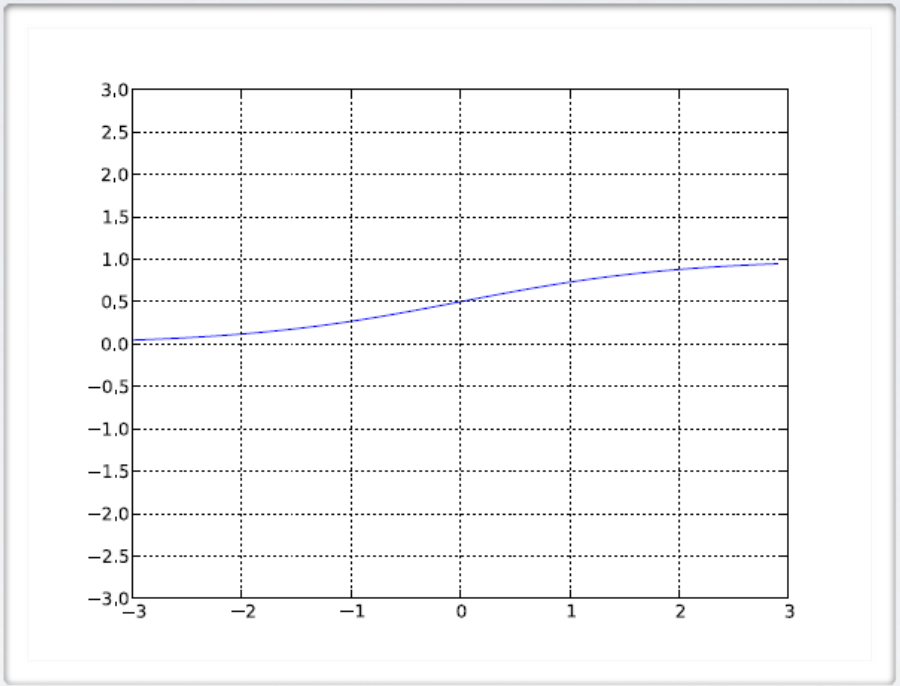
$$g(a) = a$$

Gradient of Activation Functions

Topics: sigmoid activation function gradient

- Partial derivative:

$$g'(a) = g(a)(1 - g(a))$$



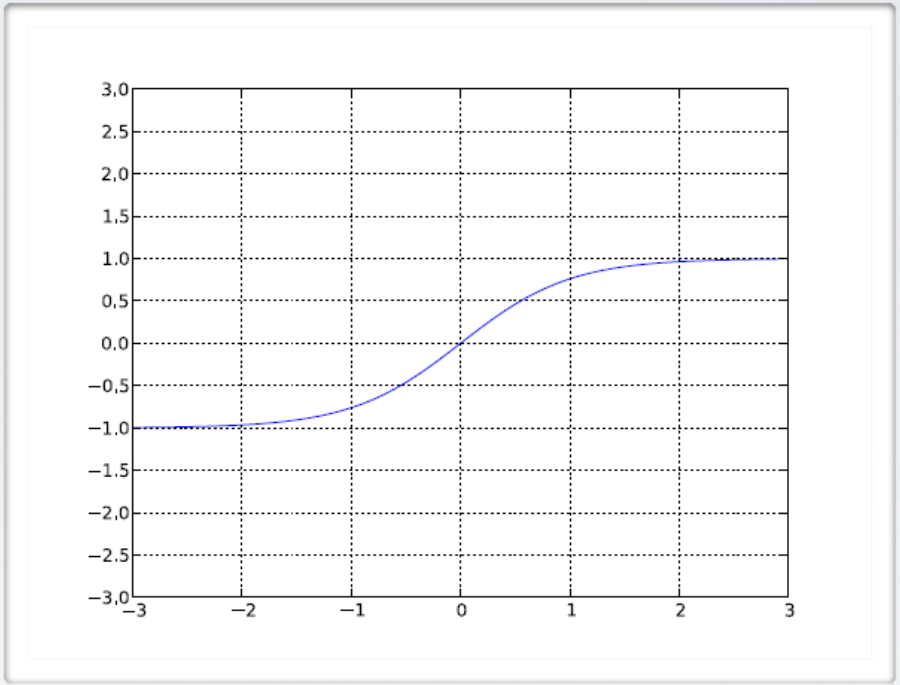
$$g(a) = \text{sigm}(a) = \frac{1}{1+\exp(-a)}$$

Gradient of Activation Functions

Topics: tanh activation function gradient

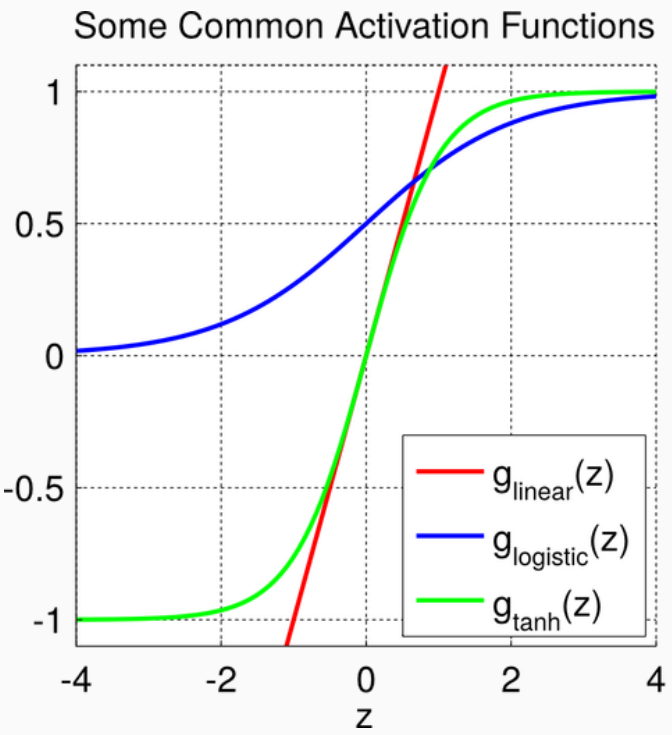
- Partial derivative:

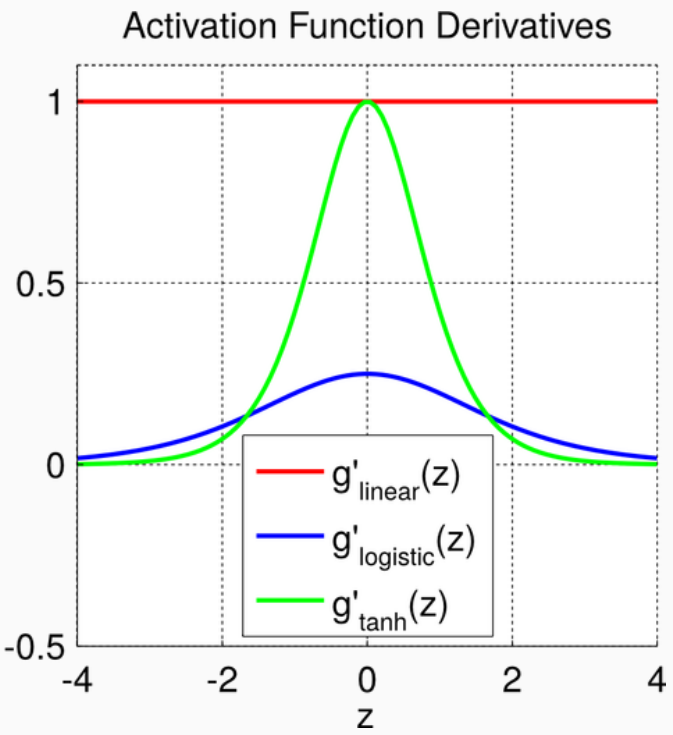
$$g'(a) = 1 - g(a)^2$$



$$g(a) = \tanh(a) = \frac{\exp(a) - \exp(-a)}{\exp(a) + \exp(-a)} = \frac{\exp(2a) - 1}{\exp(2a) + 1}$$

Gradient of Activation Functions





Output gradient

Topics: loss gradient at output pre-activation

- Partial derivative:

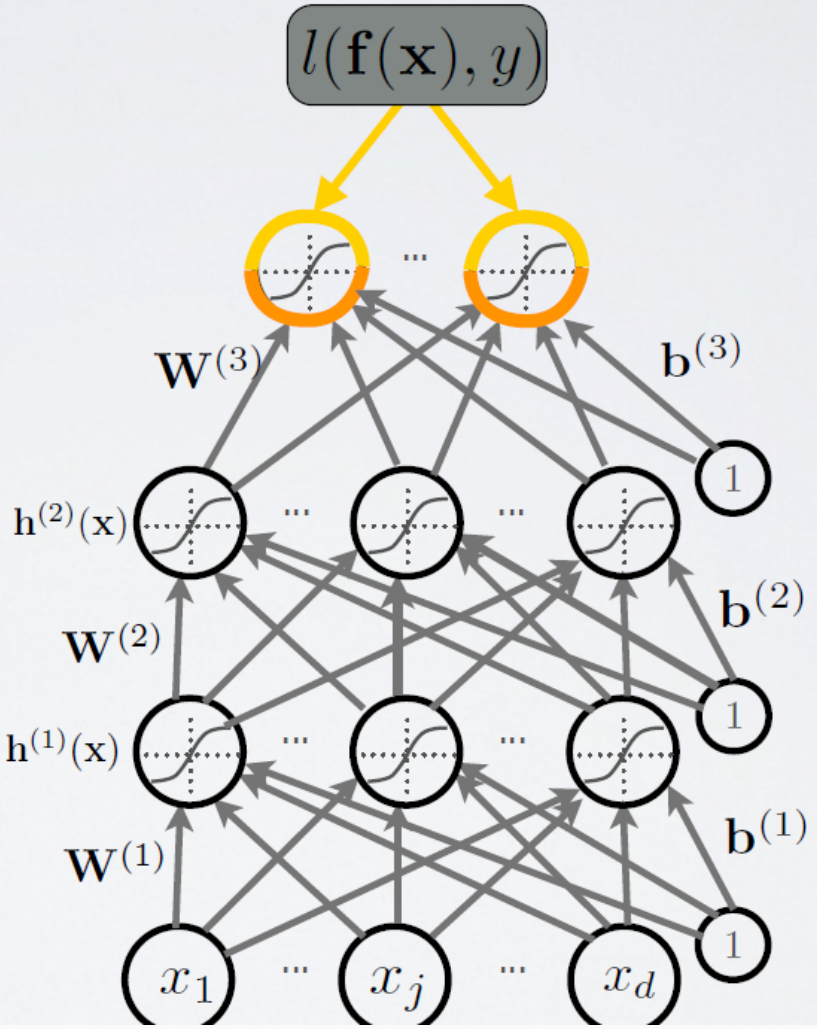
$$\frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} - \log f(\mathbf{x})_y$$

$$= \quad -\left(1_{(y=c)} - f(\mathbf{x})_c\right)$$

- Gradient:

$$\nabla_{\mathbf{a}^{(L+1)}(\mathbf{x})} - \log f(\mathbf{x})_y$$

$$= \quad -\left(\mathbf{e}(y) - \mathbf{f}(\mathbf{x})\right)$$



Output gradient

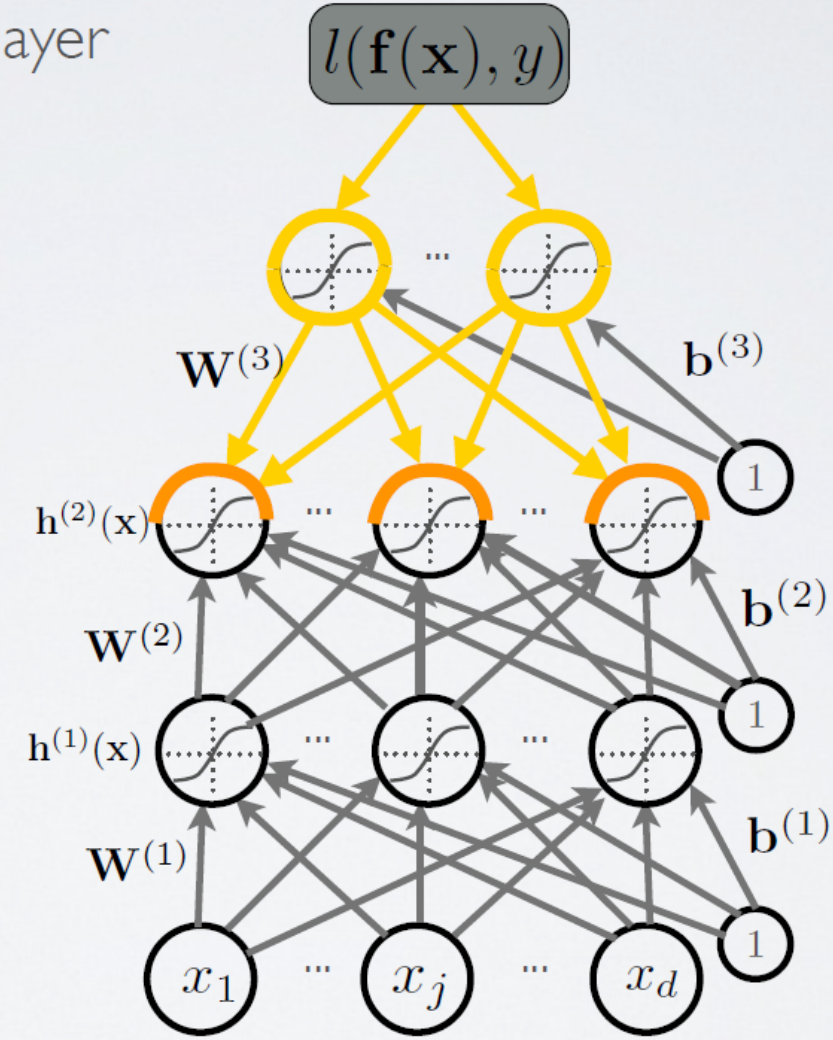
$$
\begin{aligned}
& \frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} - \log f(\mathbf{x})_y \\
= \quad & \frac{-1}{f(\mathbf{x})_y} \frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} f(\mathbf{x})_y \\
= \quad & \frac{-1}{f(\mathbf{x})_y} \frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} \text{softmax}(\mathbf{a}^{(L+1)}(\mathbf{x}))_y \\
= \quad & \frac{-1}{f(\mathbf{x})_y} \frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} \frac{\exp(a^{(L+1)}(\mathbf{x})_y)}{\sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'})} \\
= \quad & \frac{-1}{f(\mathbf{x})_y} \left(\frac{\frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} \exp(a^{(L+1)}(\mathbf{x})_y)}{\sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'})} - \frac{\exp(a^{(L+1)}(\mathbf{x})_y) \left(\frac{\partial}{\partial a^{(L+1)}(\mathbf{x})_c} \sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'}) \right)}{\left(\sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'}) \right)^2} \right) \\
= \quad & \frac{-1}{f(\mathbf{x})_y} \left(\frac{1_{(y=c)} \exp(a^{(L+1)}(\mathbf{x})_y)}{\sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'})} - \frac{\exp(a^{(L+1)}(\mathbf{x})_y)}{\sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'})} \frac{\exp(a^{(L+1)}(\mathbf{x})_c)}{\sum_{c'} \exp(a^{(L+1)}(\mathbf{x})_{c'})} \right) \\
= \quad & \frac{-1}{f(\mathbf{x})_y} \left(1_{(y=c)} \text{softmax}(\mathbf{a}^{(L+1)}(\mathbf{x}))_y - \text{softmax}(\mathbf{a}^{(L+1)}(\mathbf{x}))_y \, \text{softmax}(\mathbf{a}^{(L+1)}(\mathbf{x}))_c \right) \\
= \quad & \frac{-1}{f(\mathbf{x})_y} \left(1_{(y=c)} f(\mathbf{x})_y - f(\mathbf{x})_y \, f(\mathbf{x})_c \right) \\
= \quad & - \left(1_{(y=c)} - f(\mathbf{x})_c \right)
\end{aligned}
$$

$$
\frac{\partial \frac{g(x)}{h(x)}}{\partial x} = \frac{\partial g(x)}{\partial x} \frac{1}{h(x)} - \frac{g(x)}{h(x)^2} \frac{\partial h(x)}{\partial x}
$$

Hidden layer gradient

Topics: loss gradient at hidden layer

- ... this is getting complicated!!



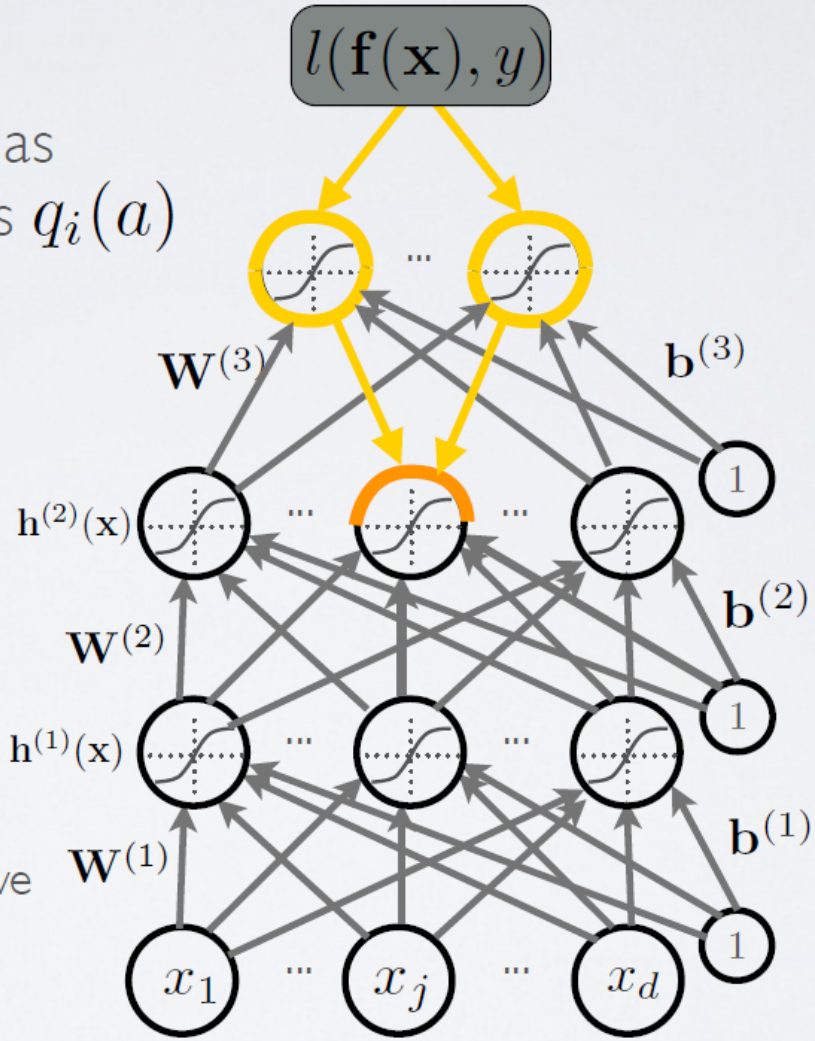
Hidden layer gradient

Topics: chain rule

- If a function $p(a)$ can be written as a function of intermediate results $q_i(a)$ then we have:

$$\frac{\partial p(a)}{\partial a} = \sum_i \frac{\partial p(a)}{\partial q_i(a)} \frac{\partial q_i(a)}{\partial a}$$

- We can invoke it by setting
  - a to a unit in layer
  - $q_i(a)$ to a pre-activation in the layer above
  - $p(a)$ is the loss function



Hidden layer gradient

Topics: loss gradient at hidden layers

- Partial derivative:

$$\frac{\partial}{\partial h^{(k)}(\mathbf{x})_j} - \log f(\mathbf{x})_y$$

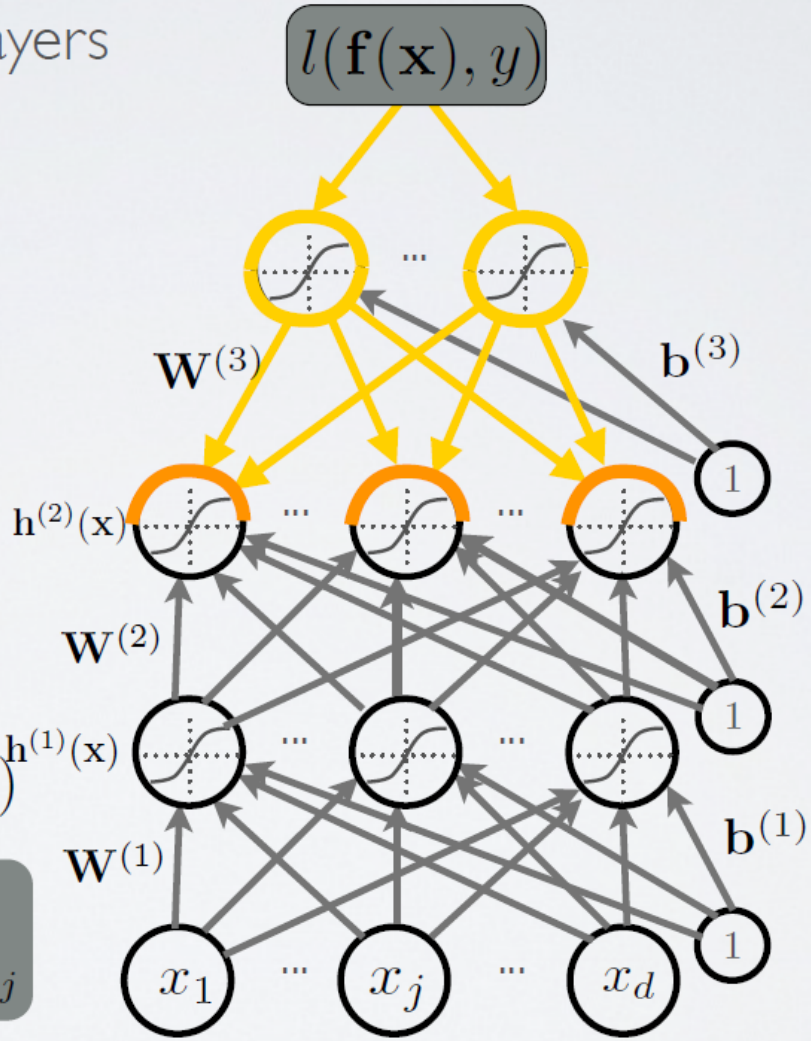
$$= \sum_i \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k+1)}(\mathbf{x})_i} \frac{\partial a^{(k+1)}(\mathbf{x})_i}{\partial h^{(k)}(\mathbf{x})_j}$$

$$= \sum_i \frac{\partial - \log f(\mathbf{x})_y}{\partial a^{(k+1)}(\mathbf{x})_i} W^{(k+1)}_{i,j}$$

$$= (\mathbf{W}^{k+1}_{\cdot,j})^\top (\nabla_{\mathbf{a}^{k+1}(\mathbf{x})} - \log f(\mathbf{x})_y)$$

REMINDER

$$a^{(k)}(\mathbf{x})_i = b^{(k)}_i + \sum_j W^{(k)}_{i,j} h^{(k-1)}(\mathbf{x})_j$$



Hidden layer gradient

Topics: loss gradient at hidden layers

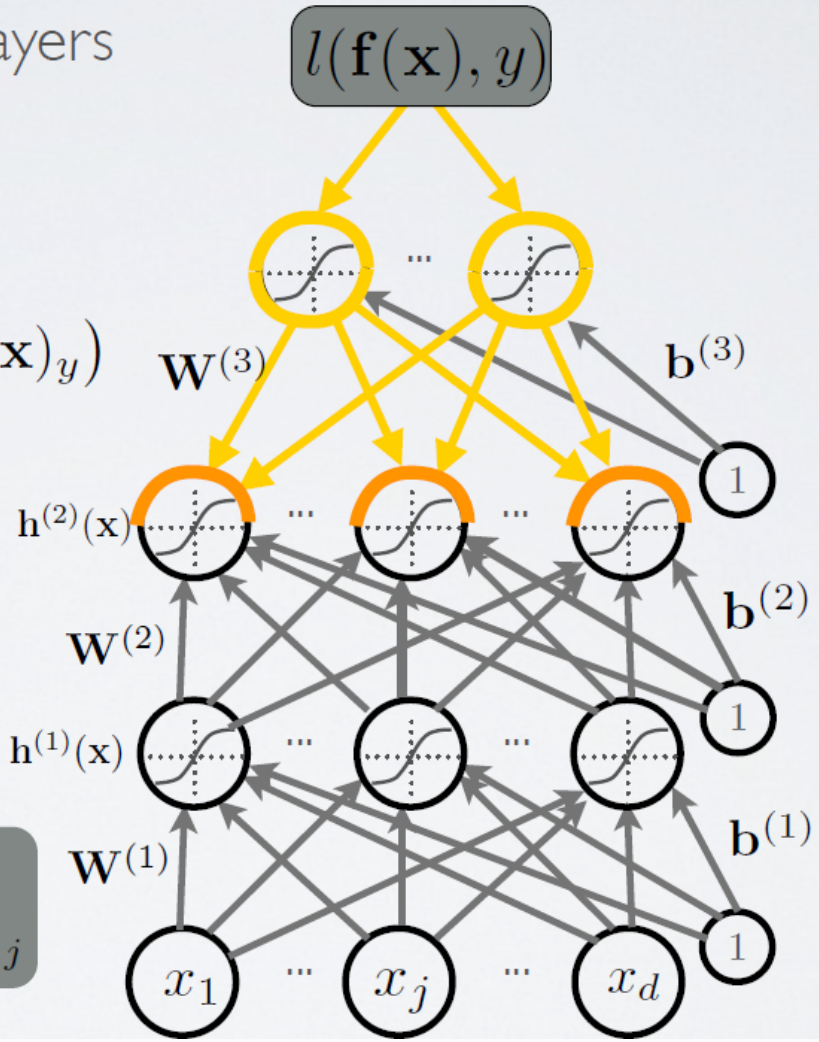
- Gradient:

$$\nabla_{\mathbf{h}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y$$

$$= \mathbf{W}^{(k+1)\top} \left(\nabla_{\mathbf{a}^{(k+1)}(\mathbf{x})} - \log f(\mathbf{x})_y \right)$$

REMINDER

$$a^{(k)}(\mathbf{x})_i = b_i^{(k)} + \sum_j W_{i,j}^{(k)} h^{(k-1)}(\mathbf{x})_j$$



Hidden layer gradient

Topics: loss gradient at hidden layers pre-activation

- Partial derivative:

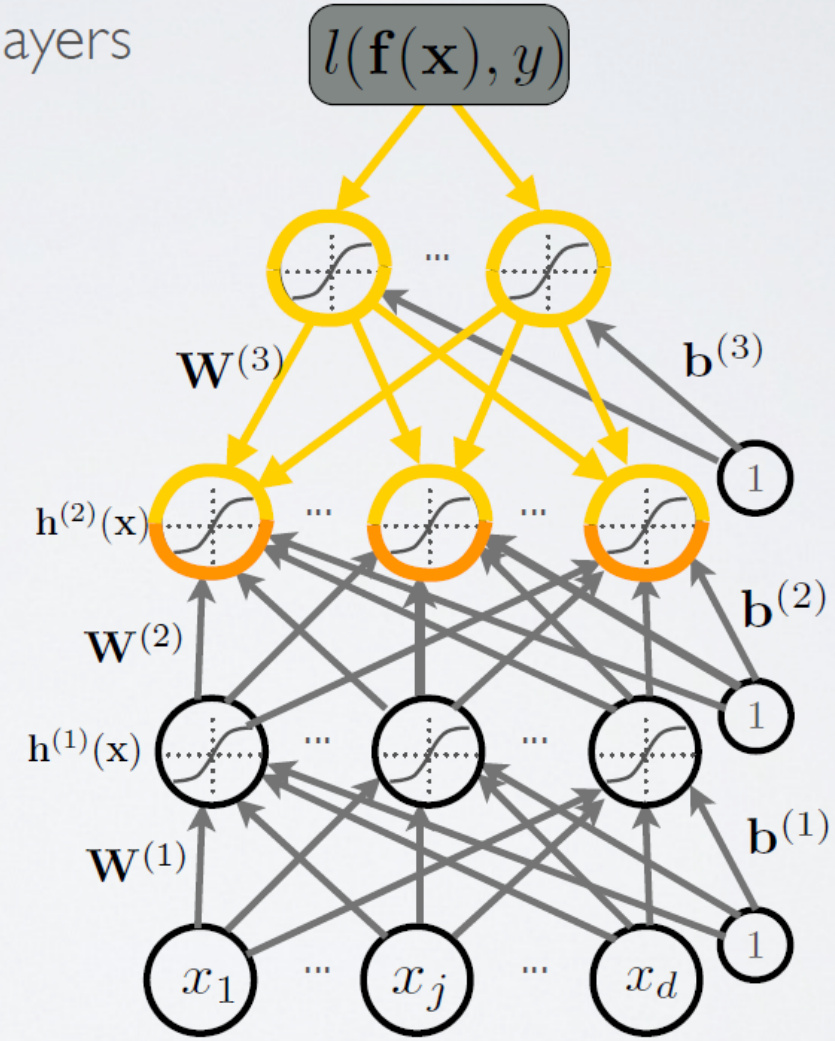
$$\frac{\partial}{\partial a^{(k)}(\mathbf{x})_j} - \log f(\mathbf{x})_y$$

$$= \frac{\partial - \log f(\mathbf{x})_y}{\partial h^{(k)}(\mathbf{x})_j} \frac{\partial h^{(k)}(\mathbf{x})_j}{\partial a^{(k)}(\mathbf{x})_j}$$

$$= \frac{\partial - \log f(\mathbf{x})_y}{\partial h^{(k)}(\mathbf{x})_j} g'(a^{(k)}(\mathbf{x})_j)$$

REMINDER

$$h^{(k)}(\mathbf{x})_j = g(a^{(k)}(\mathbf{x})_j)$$



Hidden layer gradient

Topics: loss gradient at hidden layers pre-activation

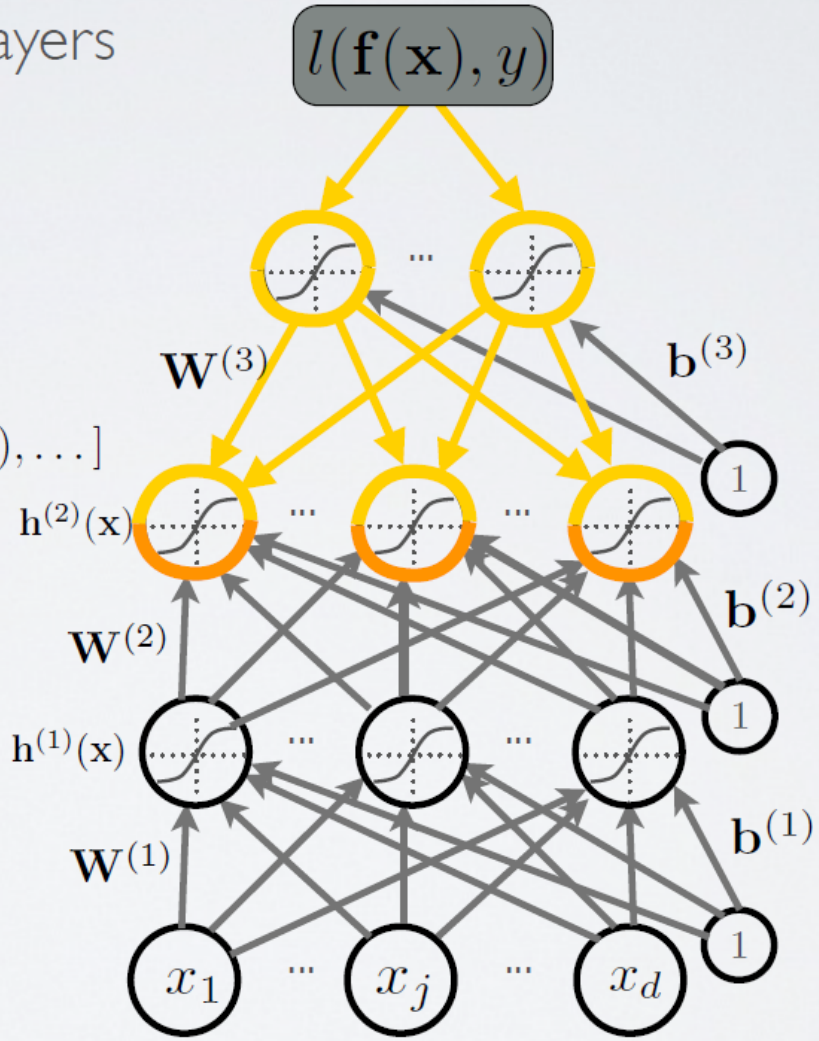
- Gradient:

$$\begin{aligned}
&\nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y \\
= \quad &\left(\nabla_{\mathbf{h}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y\right)^\top \nabla_{\mathbf{a}^{(k)}(\mathbf{x})} \mathbf{h}^{(k)}(\mathbf{x}) \\
= \quad &\left(\nabla_{\mathbf{h}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y\right) \odot [\ldots, g'(a^{(k)}(\mathbf{x})_j), \ldots]
\end{aligned}$$

element-wise product

REMINDER

$$h^{(k)}(\mathbf{x})_j = g(a^{(k)}(\mathbf{x})_j)$$



Backpropagation

Topics: backpropagation algorithm

- This assumes a forward propagation has been made before
  - compute output gradient (before activation)

$$\nabla_{\mathbf{a}^{(L+1)}(\mathbf{x})} - \log f(\mathbf{x})_y \Longleftarrow -(\mathbf{e}(y) - \mathbf{f}(\mathbf{x}))$$

  - for k from $L+1$ to 1
    - compute gradients of hidden layer parameter

$$\nabla_{\mathbf{W}^{(k)}} - \log f(\mathbf{x})_y \Longleftarrow \left(\nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y\right) \mathbf{h}^{(k-1)}(\mathbf{x})^\top$$

$$\nabla_{\mathbf{b}^{(k)}} - \log f(\mathbf{x})_y \Longleftarrow \nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y$$

    - compute gradient of hidden layer below

$$\nabla_{\mathbf{h}^{(k-1)}(\mathbf{x})} - \log f(\mathbf{x})_y \Longleftarrow \mathbf{W}^{(k)\top} \left(\nabla_{\mathbf{a}^{(k)}(\mathbf{x})} - \log f(\mathbf{x})_y\right)$$

    - compute gradient of hidden layer below (before activation)

$$\nabla_{\mathbf{a}^{(k-1)}(\mathbf{x})} - \log f(\mathbf{x})_y \Longleftarrow \left(\nabla_{\mathbf{h}^{(k-1)}(\mathbf{x})} - \log f(\mathbf{x})_y\right) \odot [\ldots, g'(a^{(k-1)}(\mathbf{x})_j), \ldots]$$

Regularization

Topics: empirical risk minimization, regularization

- Empirical risk minimization
  - framework to design learning algorithms

$$\arg\min_{\boldsymbol{\theta}} \frac{1}{T} \sum_{t} l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)}) + \lambda \Omega(\boldsymbol{\theta})$$

  - $l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$ is a loss function
  - $\Omega(\boldsymbol{\theta})$ is a regularizer (penalizes certain values of $\boldsymbol{\theta}$)
- Learning is cast as optimization
  - ideally, we'd optimize classification error, but it's not smooth
  - loss function is a surrogate for what we truly should optimize (e.g. upper bound)

Regularization

Topics: stochastic gradient descent (SGD)

- Algorithm that performs updates after each example
  - initialize $\boldsymbol{\theta}$ ($\boldsymbol{\theta} \equiv \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \ldots, \mathbf{W}^{(L+1)}, \mathbf{b}^{(L+1)}\}$)
  - for N iterations
    - for each training example $(\mathbf{x}^{(t)}, y^{(t)})$
      - ✓ $\Delta = -\nabla_{\boldsymbol{\theta}} l(f(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)}) - \lambda \nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$
      - ✓ $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \, \Delta$

    } training epoch = iteration over **all** examples

- To apply this algorithm to neural network training, we need
  - the loss function $l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
  - a procedure to compute the parameter gradients $\nabla_{\boldsymbol{\theta}} l(\mathbf{f}(\mathbf{x}^{(t)}; \boldsymbol{\theta}), y^{(t)})$
  - the regularizer $\Omega(\boldsymbol{\theta})$ (and the gradient $\nabla_{\boldsymbol{\theta}} \Omega(\boldsymbol{\theta})$)
  - initialization method

Regularization

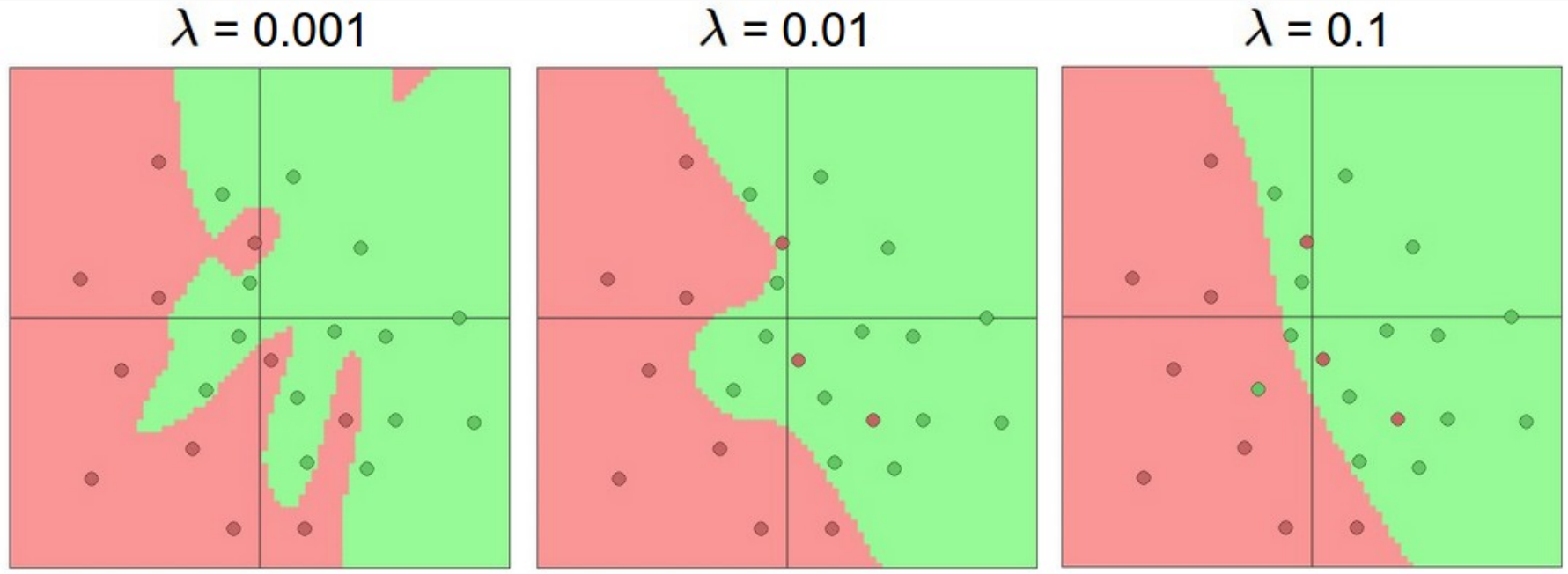
Topics: L2 regularization

$$\Omega(\boldsymbol{\theta}) = \sum_k \sum_i \sum_j \left(W_{i,j}^{(k)}\right)^2 = \sum_k ||\mathbf{W}^{(k)}||_F^2$$

- Gradient: $\nabla_{\mathbf{W}^{(k)}} \Omega(\boldsymbol{\theta}) = 2\mathbf{W}^{(k)}$

- Only applied on weights, not on biases (weight decay)
- Can be interpreted as having a Gaussian prior over the weights

Regularization



Regularization

Topics: L1 regularization

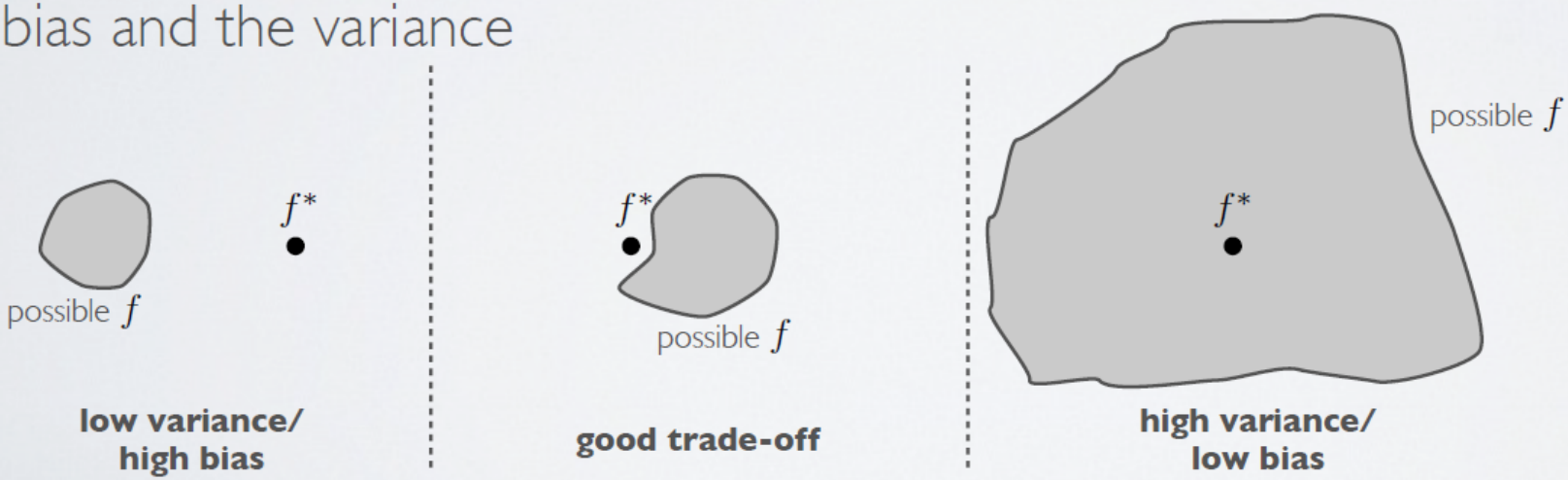
$$\Omega(\boldsymbol{\theta}) = \sum_k \sum_i \sum_j |W_{i,j}^{(k)}|$$

- Gradient: $\nabla_{\mathbf{W}^{(k)}} \Omega(\boldsymbol{\theta}) = \text{sign}(\mathbf{W}^{(k)})$
  - where $\text{sign}(\mathbf{W}^{(k)})_{i,j} = 1_{\mathbf{W}_{i,j}^{(k)} > 0} - 1_{\mathbf{W}_{i,j}^{(k)} < 0}$
- Also only applied on weights
- Unlike L2, L1 will push certain weights to be exactly 0
- Can be interpreted as having a Laplacian prior over the weights

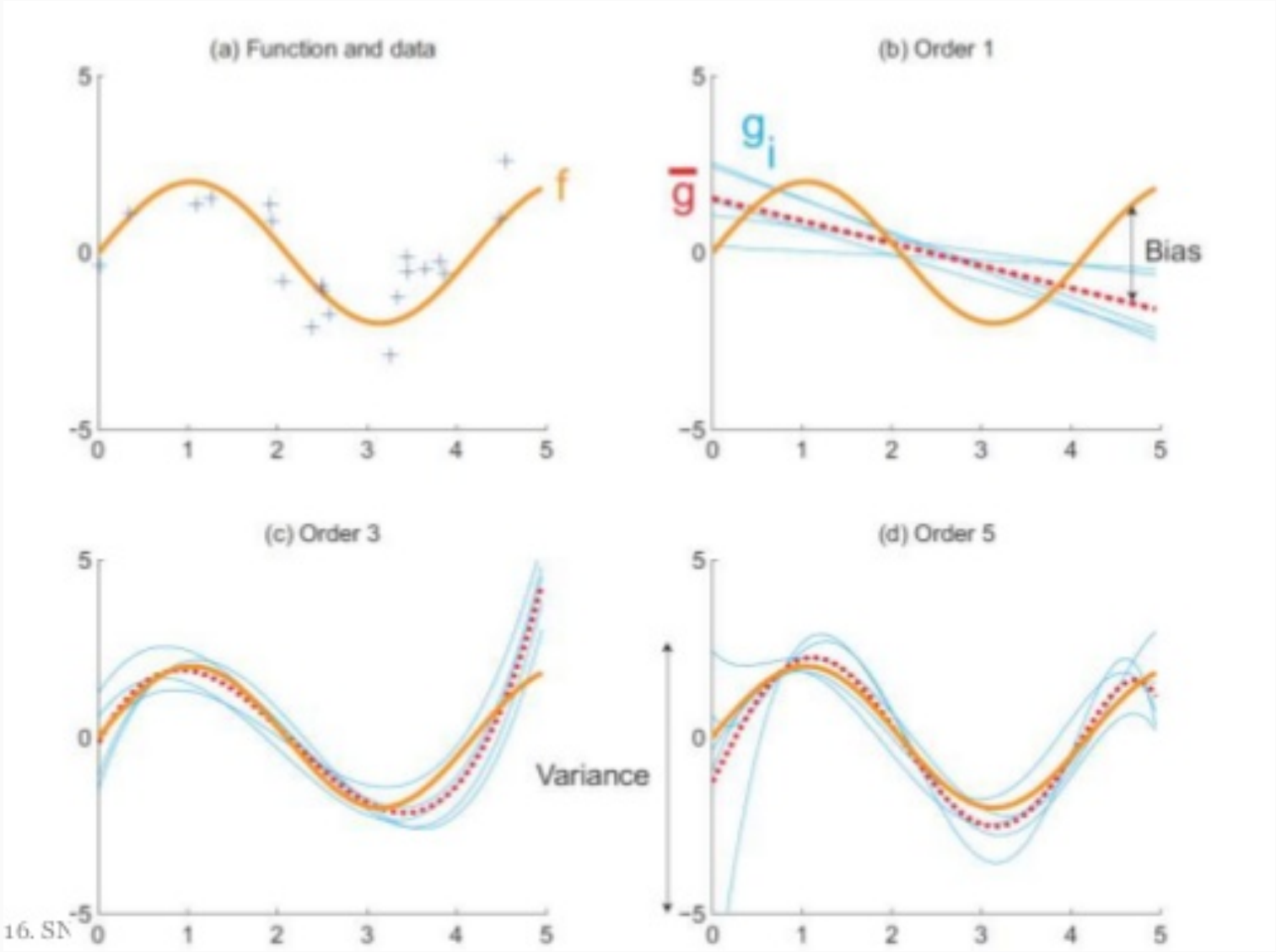
Regularization-Bias Variance

Topics: bias-variance trade-off

- Variance of trained model: does it vary a lot if the training set changes
- Bias of trained model: is the average model close to the true solution
- Generalization error can be seen as the sum of the (squared) bias and the variance

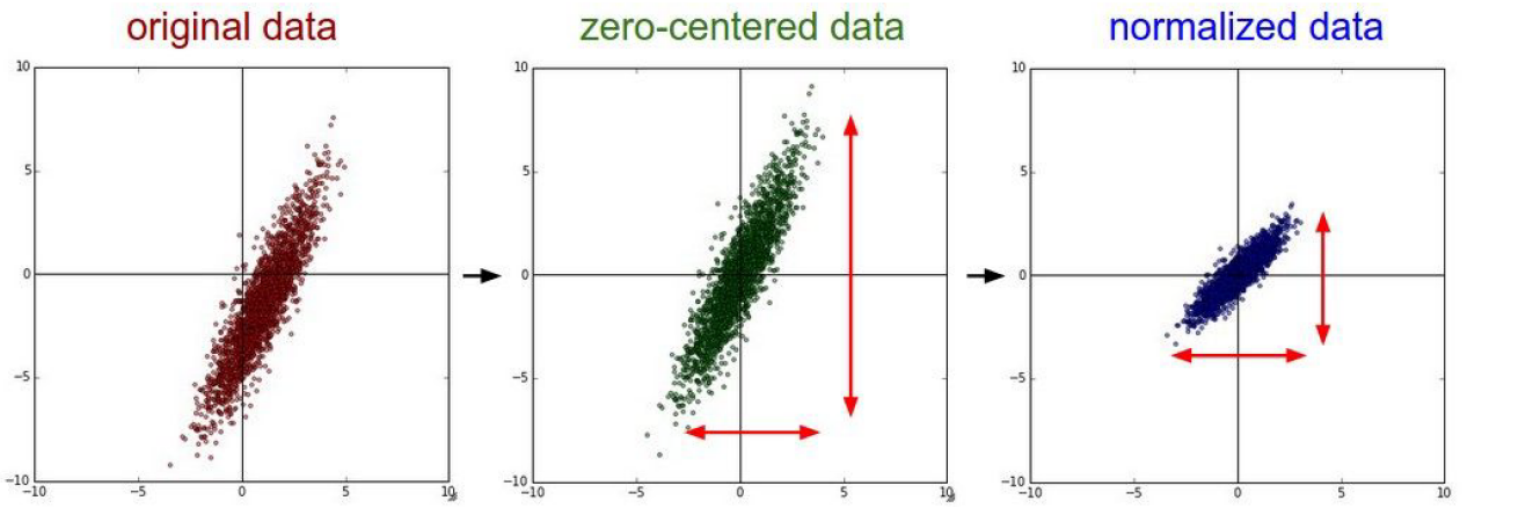


Regularization-Bias Variance



Data preprocessing:Z score

Zero mean Unit Variance



(Assume X [NxD] is data matrix,
each example in a row)

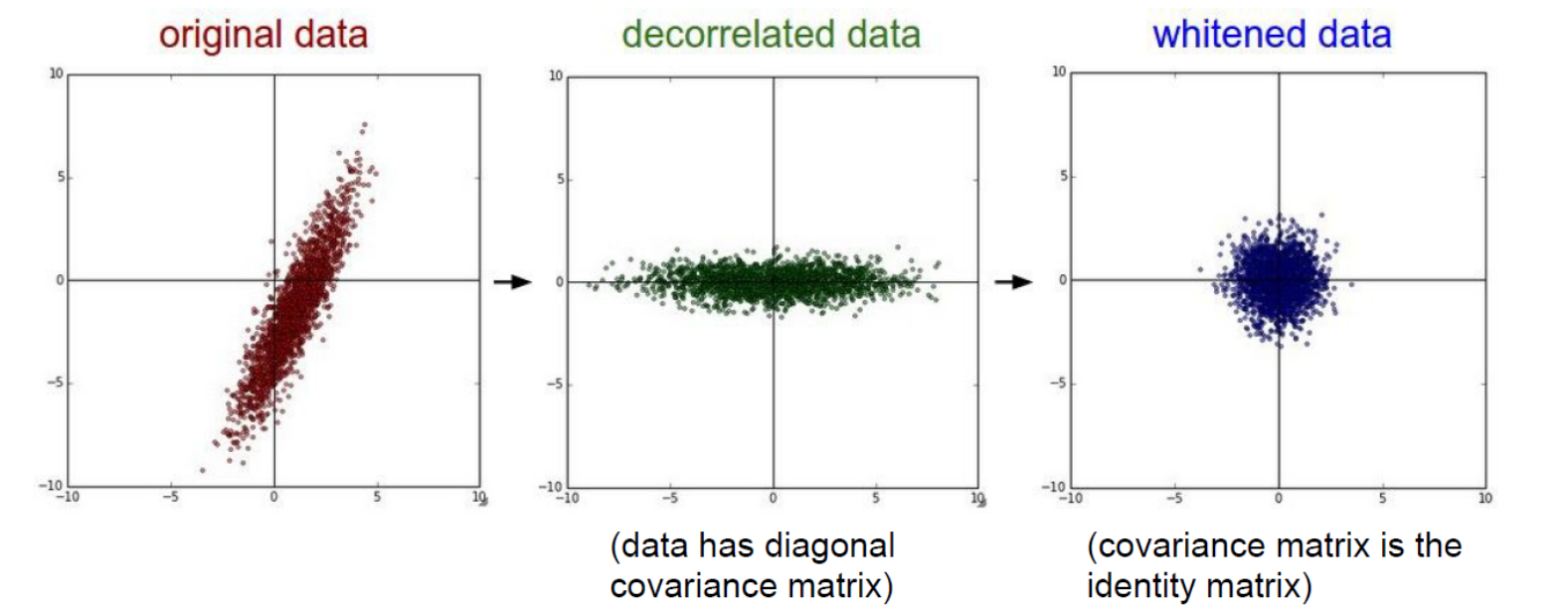
Stand

scaler function in scikit-learn:

preprocessing.*StandardScaler*().*fit*(*X*)

Data preprocessing: Whitening

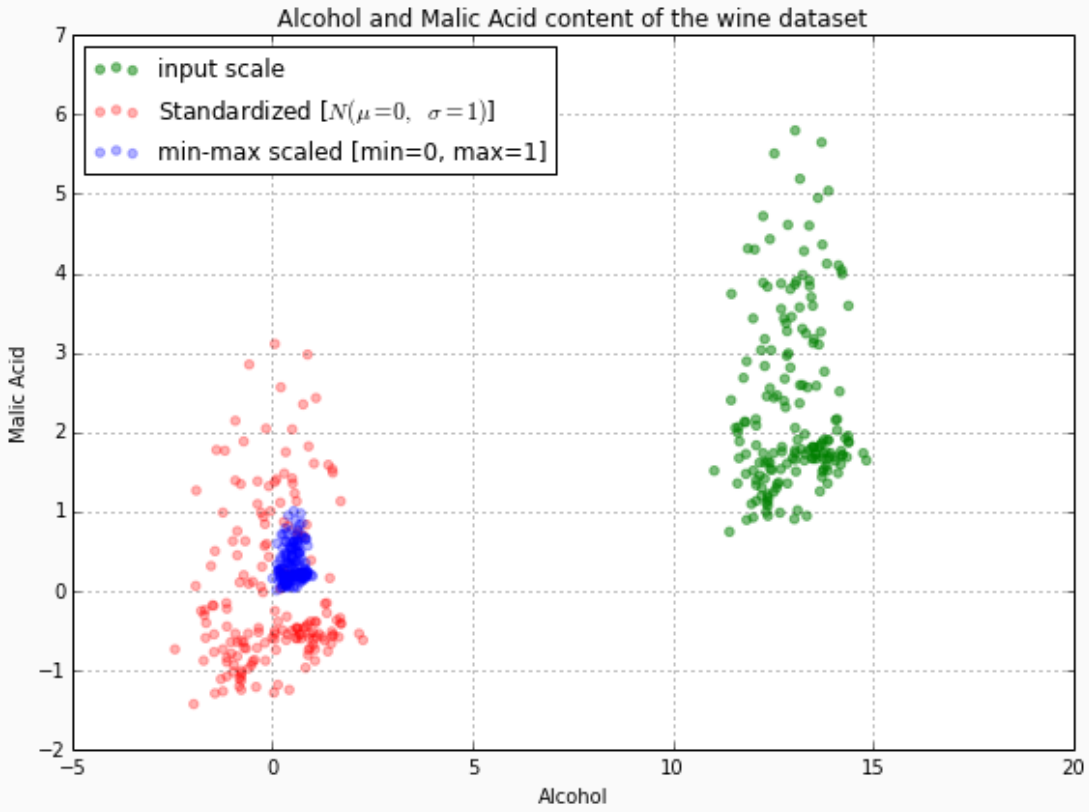
PCA data whitening



In scikit-learn:

$sklearn.decomposition.PCA(n\_components = None, copy = True, whiten = True)$

Data preprocessing: Min Max Scaling

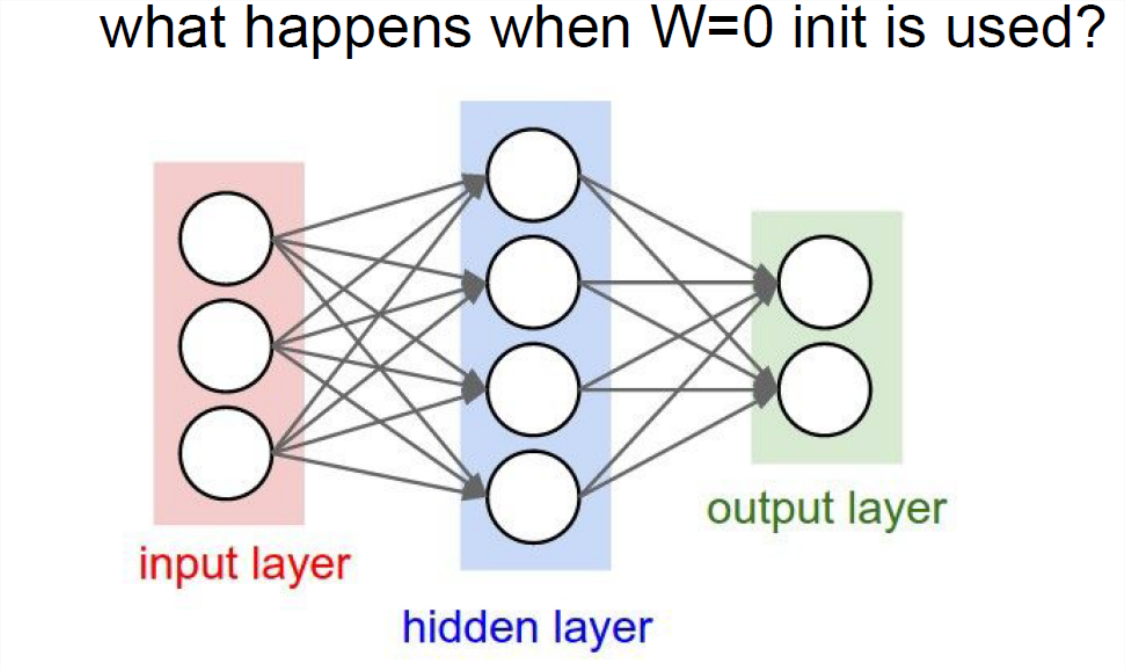


Min Max scaling In scikit-learn:

sklearn.preprocessing.MinMaxScaler(*feature_range* $= (0, 1)$)

Weight initialization: Zero/Constant Weights

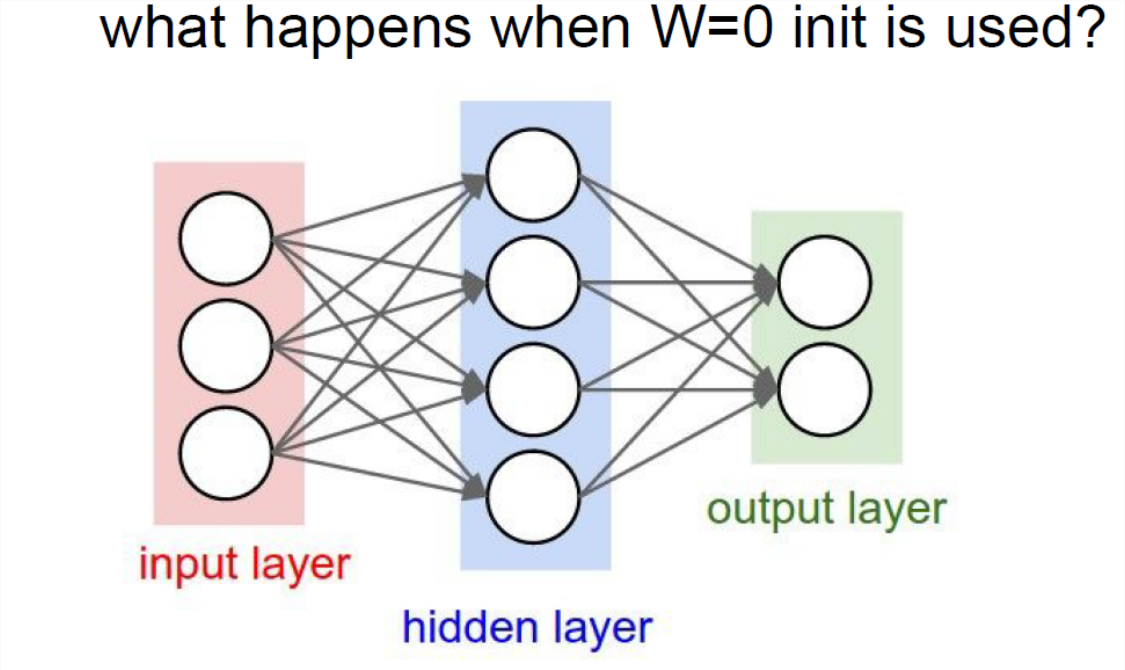
Zero weights



- what happens when the constant weight is initialized?
- output activations are same at different nodes - less diverse

Weight initialization: Zero/Constant Weights

Zero weights



- what happens when the constant weight is initialized?
- output activations are same at different nodes - less diverse

Weight initialization: random number

Random weights

- First idea: **Small random numbers**
(gaussian with zero mean and 1e-2 standard deviation)

```
W = 0.01* np.random.randn(D,H)
```

Works ~okay for small networks, but can lead to non-homogeneous distributions of activations across the layers of a network.

- for the deep networks, it does not work well

Weight initialization: random number

Random weights

Lets look at some activation statistics

E.g. 10-layer net with 500 neurons on each layer, using tanh non-linearities, and initializing as described in last slide.

```
# assume some unit gaussian 10-D input data
D = np.random.randn(1000, 500)
hidden_layer_sizes = [500]*10
nonlinearities = ['tanh']*len(hidden_layer_sizes)
```

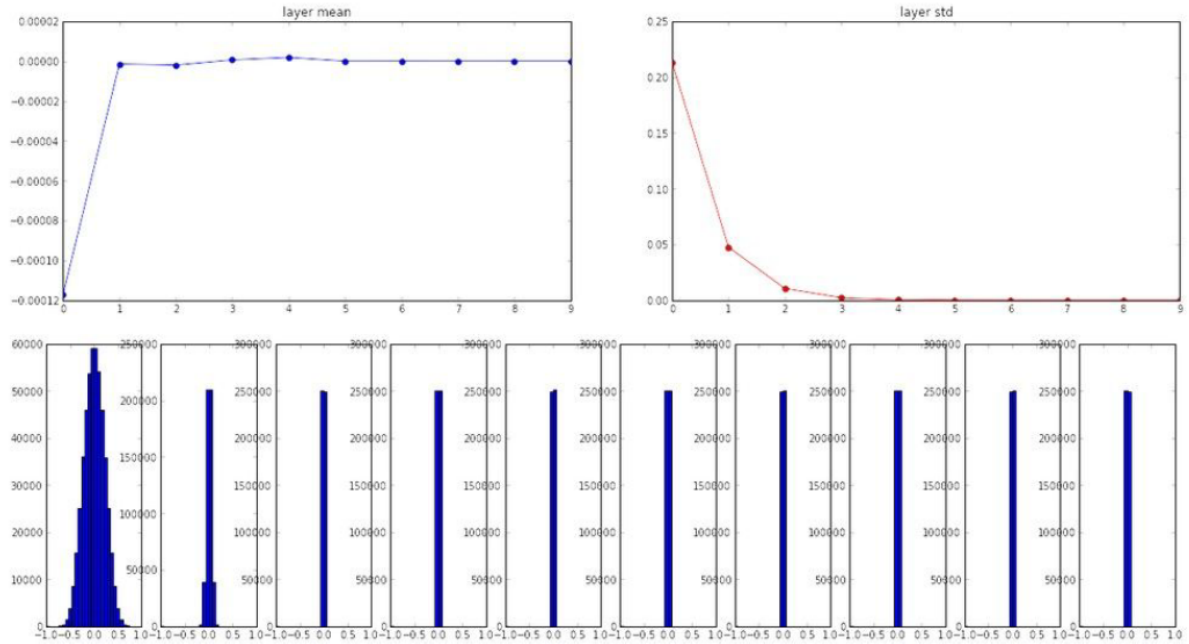
```
act = {'relu':lambda x:np.maximum(0,x), 'tanh':lambda x:np.tanh(x)}
Hs = {}
for i in xrange(len(hidden_layer_sizes)):
    X = D if i == 0 else Hs[i-1] # input at this layer
    fan_in = X.shape[1]
    fan_out = hidden_layer_sizes[i]
    W = np.random.randn(fan_in, fan_out) * 0.01 # layer initialization

    H = np.dot(X, W) # matrix multiply
    H = act[nonlinearities[i]](H) # nonlinearity
    Hs[i] = H # cache result on this layer
```

```
# look at distributions at each layer
print 'input layer had mean %f and std %f' % (np.mean(D), np.std(D))
layer_means = [np.mean(H) for i,H in Hs.iteritems()]
layer_stds = [np.std(H) for i,H in Hs.iteritems()]
for i,H in Hs.iteritems():
    print 'hidden layer %d had mean %f and std %f' % (i+1, layer_means[i], layer_stds[i])

# plot the means and standard deviations
plt.figure()
plt.subplot(121)
plt.plot(Hs.keys(), layer_means, 'ob-')
plt.title('layer mean')
plt.subplot(122)
plt.plot(Hs.keys(), layer_stds, 'or-')
plt.title('layer std')

# plot the raw distributions
plt.figure()
for i,H in Hs.iteritems():
    plt.subplot(1,len(Hs),i+1)
    plt.hist(H.ravel(), 30, range=(-1,1))
```

Weight initialization: random number

Random weights

```
input layer had mean 0.000927 and std 0.998388
hidden layer 1 had mean -0.000117 and std 0.213081
hidden layer 2 had mean -0.000001 and std 0.047551
hidden layer 3 had mean -0.000002 and std 0.010630
hidden layer 4 had mean 0.000001 and std 0.002378
hidden layer 5 had mean 0.000002 and std 0.000532
hidden layer 6 had mean -0.000000 and std 0.000119
hidden layer 7 had mean 0.000000 and std 0.000026
hidden layer 8 had mean -0.000000 and std 0.000006
hidden layer 9 had mean 0.000000 and std 0.000001
hidden layer 10 had mean -0.000000 and std 0.000000
```



All activations become zero!

Q: think about the backward pass. What do the gradients look like?

Hint: think about backward pass for a W*X gate.

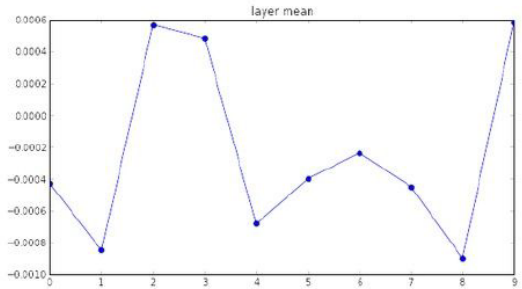
Weight initialization: random number

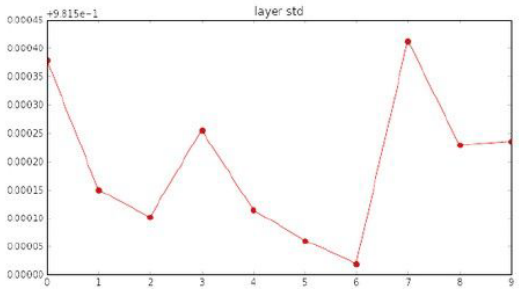
Random weights

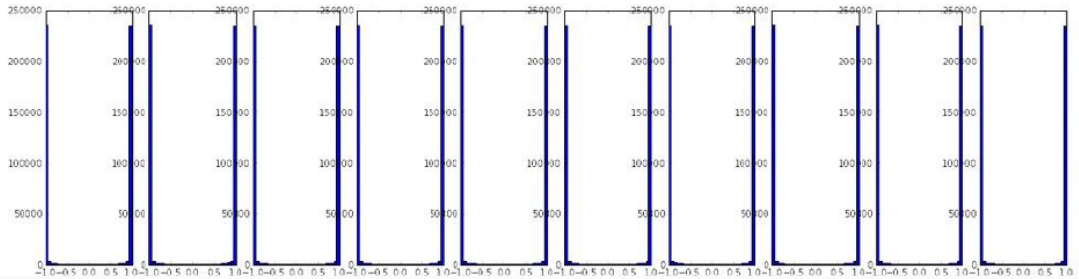
```
W = np.random.randn(fan_in, fan_out) * 1.0 # layer initialization
```

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean -0.000430 and std 0.981879
hidden layer 2 had mean -0.000849 and std 0.981649
hidden layer 3 had mean 0.000566 and std 0.981601
hidden layer 4 had mean 0.000483 and std 0.981755
hidden layer 5 had mean -0.000682 and std 0.981614
hidden layer 6 had mean -0.000401 and std 0.981560
hidden layer 7 had mean -0.000237 and std 0.981520
hidden layer 8 had mean -0.000448 and std 0.981913
hidden layer 9 had mean -0.000899 and std 0.981728
hidden layer 10 had mean 0.000584 and std 0.981736
```

*1.0 instead of *0.01







Almost all neurons completely saturated, either -1 and 1. Gradients will be all zero.

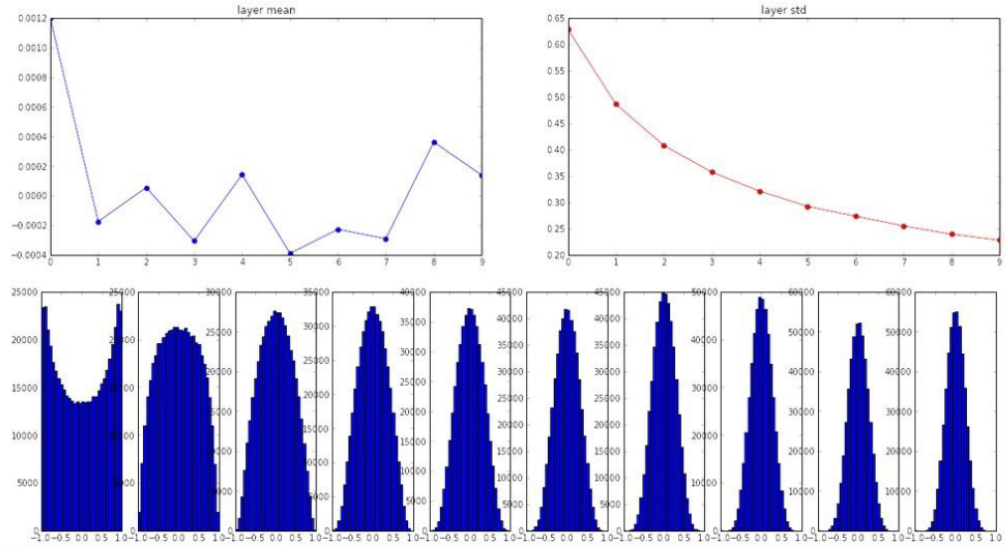
Weight initialization: Xavier

Random weights: Xavier

```
input layer had mean 0.001800 and std 1.001311
hidden layer 1 had mean 0.001198 and std 0.627953
hidden layer 2 had mean -0.000175 and std 0.486051
hidden layer 3 had mean 0.000055 and std 0.407723
hidden layer 4 had mean -0.000306 and std 0.357108
hidden layer 5 had mean 0.000142 and std 0.320917
hidden layer 6 had mean -0.000389 and std 0.292116
hidden layer 7 had mean -0.000228 and std 0.273387
hidden layer 8 had mean -0.000291 and std 0.254935
hidden layer 9 had mean 0.000361 and std 0.239266
hidden layer 10 had mean 0.000139 and std 0.228008
```

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

"Xavier initialization"
[Glorot et al., 2010]



Reasonable initialization.
(Mathematical derivation assumes linear activations)

- scale the weights according the input dimension of the data
- more number of inputs $\rightarrow$ smaller weights
- less number of inputs $\rightarrow$ larger weights

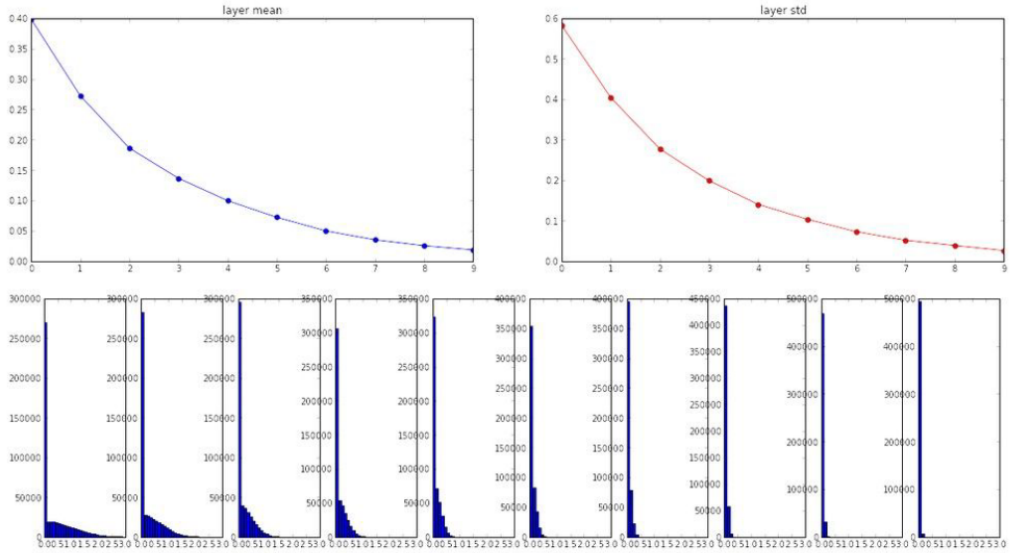
Weight initialization: Xavier-ReLU

Random weights: Xavier-ReLU

```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.398623 and std 0.582273
hidden layer 2 had mean 0.272352 and std 0.403795
hidden layer 3 had mean 0.186076 and std 0.276912
hidden layer 4 had mean 0.136442 and std 0.198685
hidden layer 5 had mean 0.099568 and std 0.140299
hidden layer 6 had mean 0.072234 and std 0.103280
hidden layer 7 had mean 0.049775 and std 0.072748
hidden layer 8 had mean 0.035138 and std 0.051572
hidden layer 9 had mean 0.025404 and std 0.038583
hidden layer 10 had mean 0.018408 and std 0.026076
```

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in) # layer initialization
```

but when using the ReLU nonlinearity it breaks.



- scale the weights according the input dimension of the data
- For the ReLU activation function, it fails: decrease in std is much more rapid

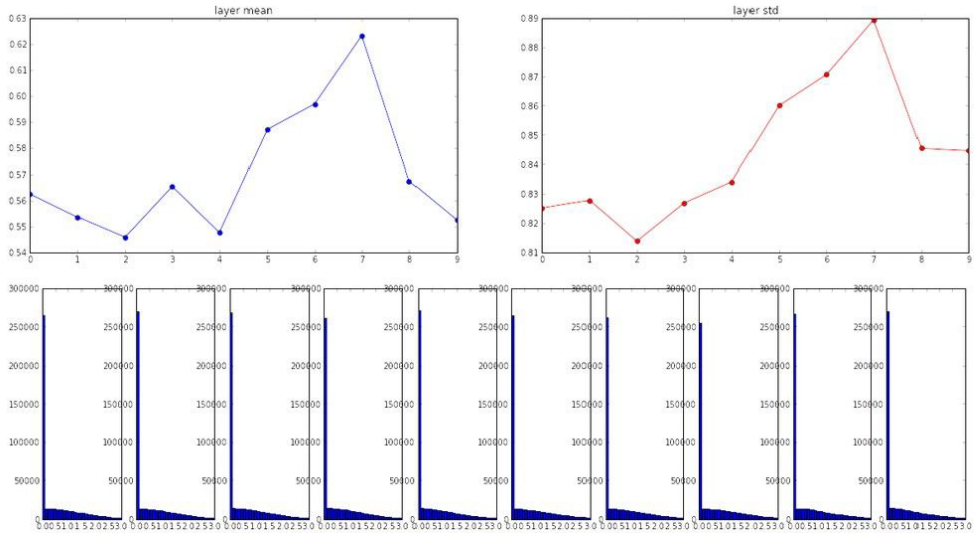
Weight initialization: Xavier-ReLU

Random weights: Xavier-ReLU

```
input layer had mean 0.000501 and std 0.999444
hidden layer 1 had mean 0.562488 and std 0.825232
hidden layer 2 had mean 0.553614 and std 0.827835
hidden layer 3 had mean 0.545867 and std 0.813855
hidden layer 4 had mean 0.565396 and std 0.826902
hidden layer 5 had mean 0.547678 and std 0.834092
hidden layer 6 had mean 0.587103 and std 0.860035
hidden layer 7 had mean 0.596867 and std 0.870610
hidden layer 8 had mean 0.623214 and std 0.889348
hidden layer 9 had mean 0.567498 and std 0.845357
hidden layer 10 had mean 0.552531 and std 0.844523
```

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in/2) # layer initialization
```

He et al., 2015
(note additional /2)



- **In each layers ReLU half the variance**
- **need to account for extra 2**
- **ensures your distribtion in the layers**

- scale the weights according the input dimension of the data
- For the ReLU activation function, it fails: decrease in std is much more rapid

Weight initialization: Uniform

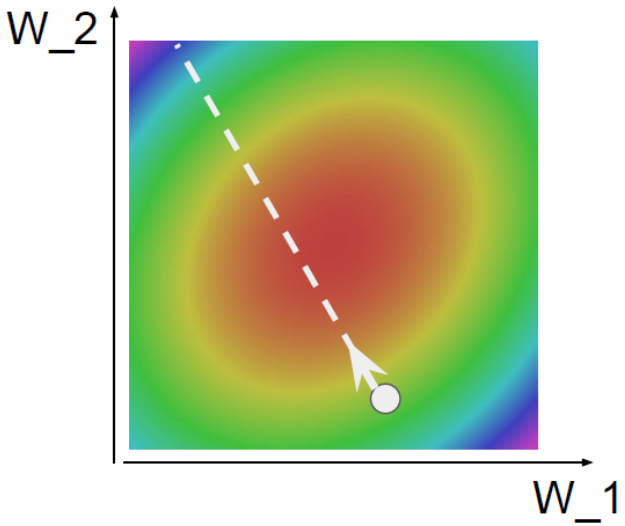
Random weights: Uniform

- Can't initialize all weights to the same value
  - we can show that all hidden units in a layer will always behave the same
  - need to break symmetry
- Recipe: sample $\mathbf{W}_{i,j}^{(k)}$ from $U\left[-b, b\right]$ where $b = \frac{\sqrt{6}}{\sqrt{H_k + H_{k-1}}}$ (size of $\mathbf{h}^{(k)}(\mathbf{x})$)
  - the idea is to sample around 0 but break symmetry
  - other values of b could work well (not an exact science) (see Glorot & Bengio, 2010)

- b depends on number of input and output neurons

Optimization

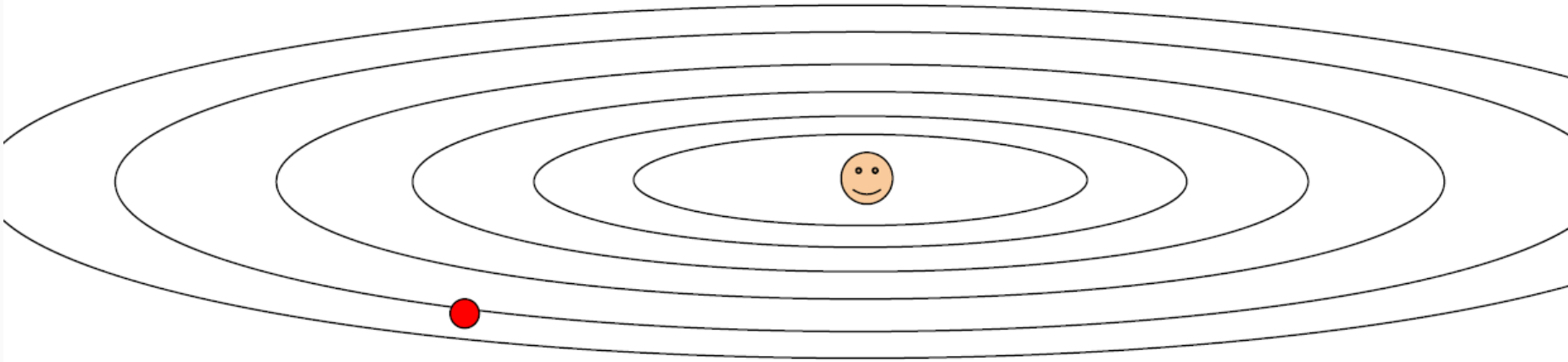
```
# Vanilla Gradient Descent

while True:
  weights_grad = evaluate_gradient(loss_fun, data, weights)
  weights += - step_size * weights_grad # perform parameter update
```



Problems with SGD

Problems with SGD:

What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?

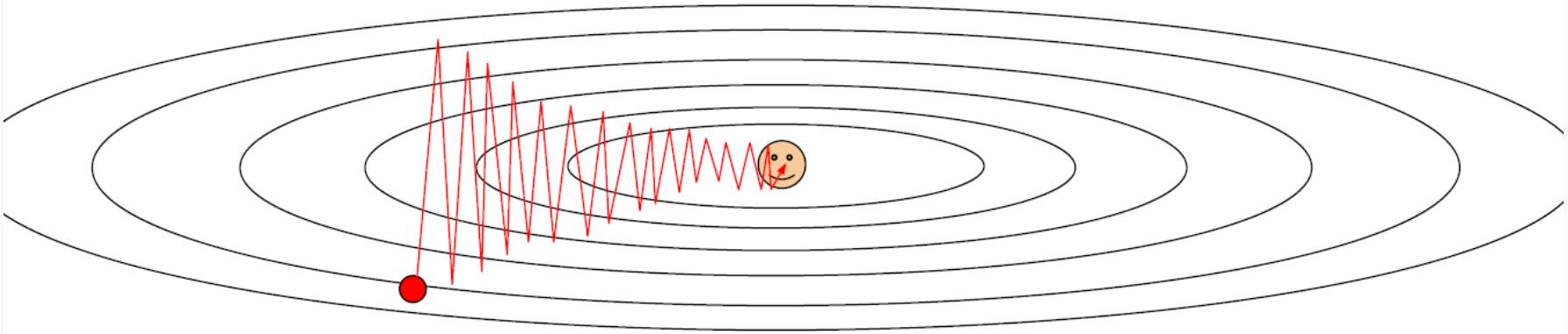


Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

Problems with SGD

Problems with SGD:

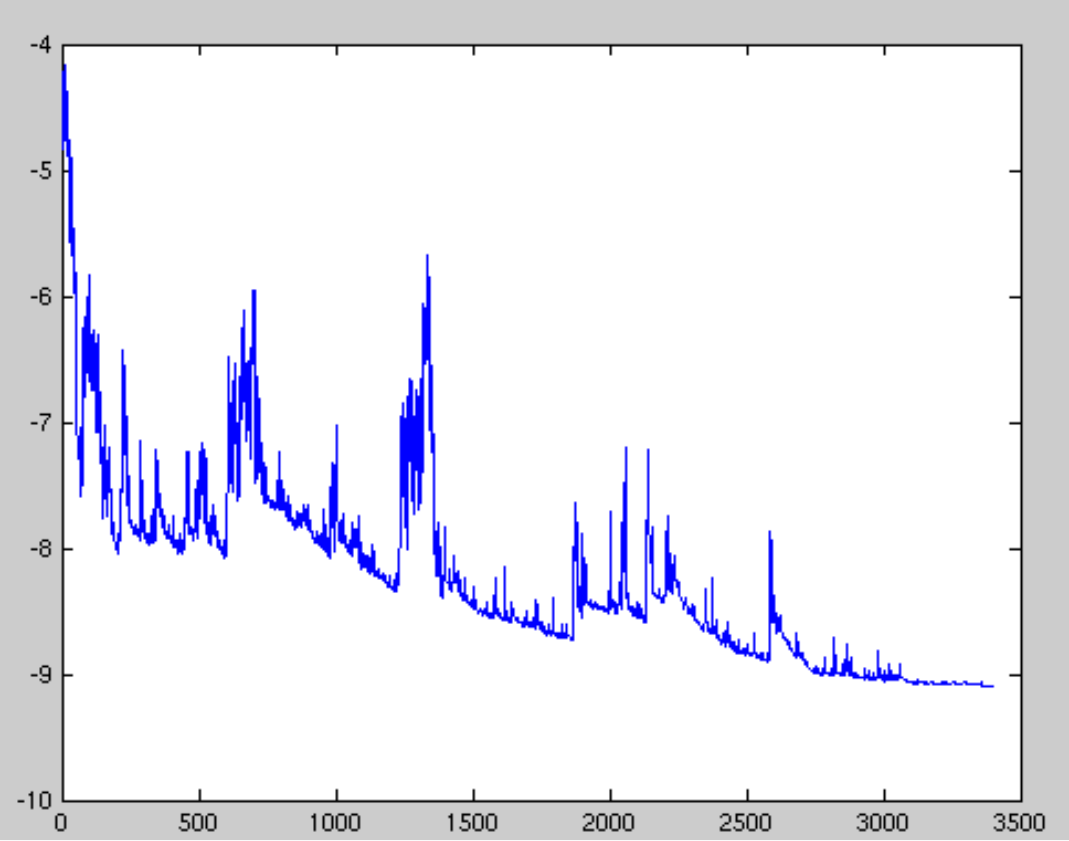
What if loss changes quickly in one direction and slowly in another?
What does gradient descent do?
Very slow progress along shallow dimension, jitter along steep direction



Loss function has high **condition number**: ratio of largest to smallest singular value of the Hessian matrix is large

Problems with SGD

Problems with SGD:

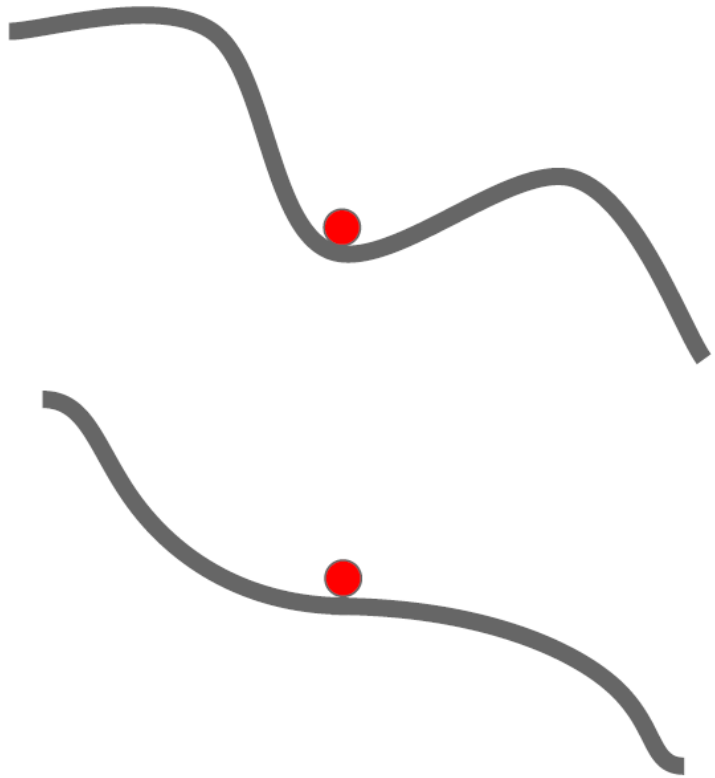


Problems with SGD

Problems with SGD:

What if the loss function has a **local minima** or **saddle point**?

Zero gradient, gradient descent gets stuck



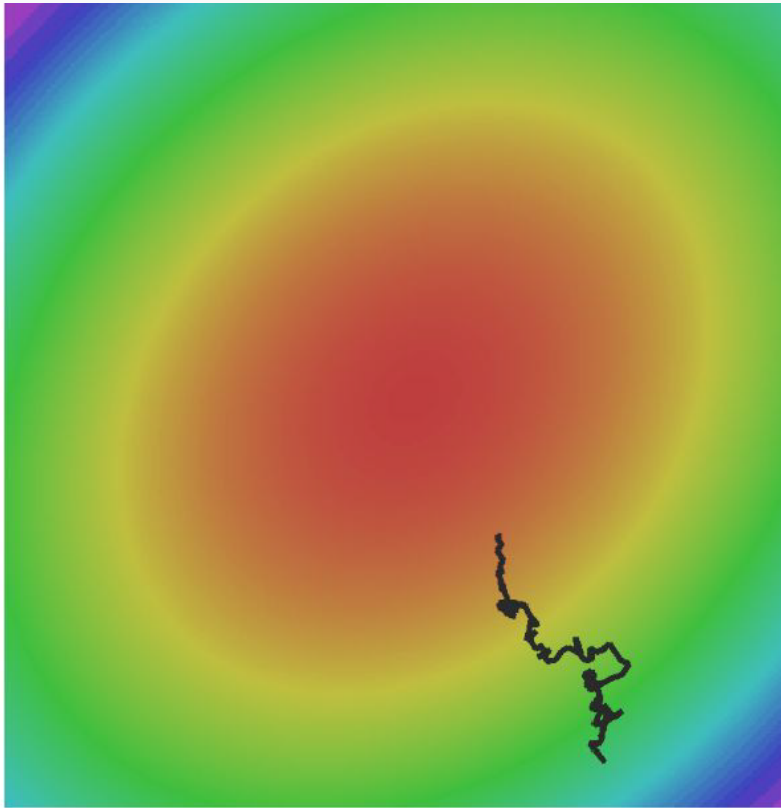
Problems with SGD

Problems with SGD:

Our gradients come from minibatches so they can be noisy!

$$L(W) = \frac{1}{N} \sum_{i=1}^{N} L_i(x_i, y_i, W)$$

$$\nabla_W L(W) = \frac{1}{N} \sum_{i=1}^{N} \nabla_W L_i(x_i, y_i, W)$$



SGD + Momentum

SGD

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

```
while True:
  dx = compute_gradient(x)
  x += learning_rate * dx
```

SGD+Momentum

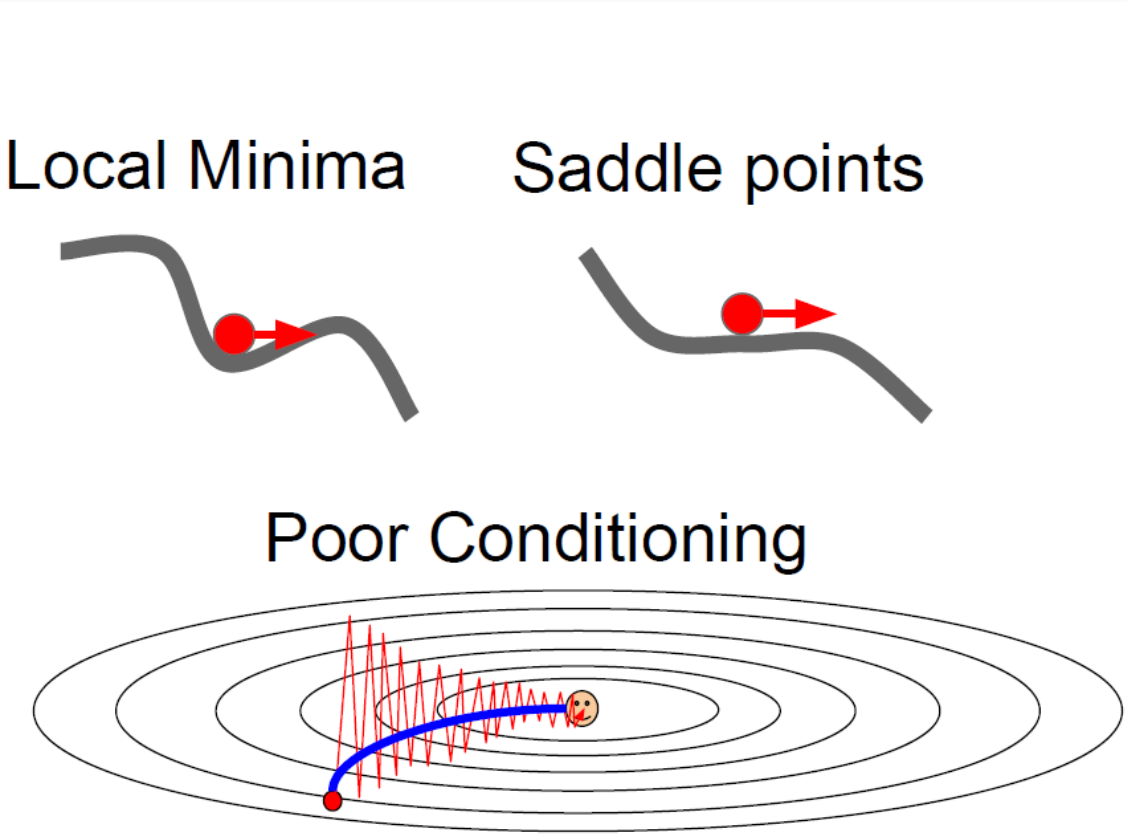
$$v_{t+1} = \rho v_t + \nabla f(x_t)$$

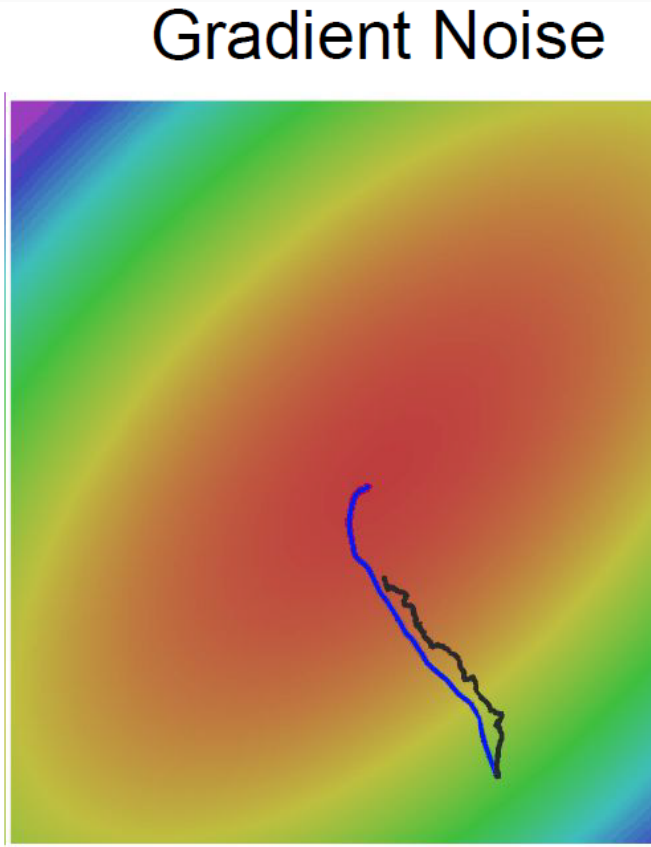
$$x_{t+1} = x_t - \alpha v_{t+1}$$

```
vx = 0
while True:
  dx = compute_gradient(x)
  vx = rho * vx + dx
  x += learning_rate * vx
```

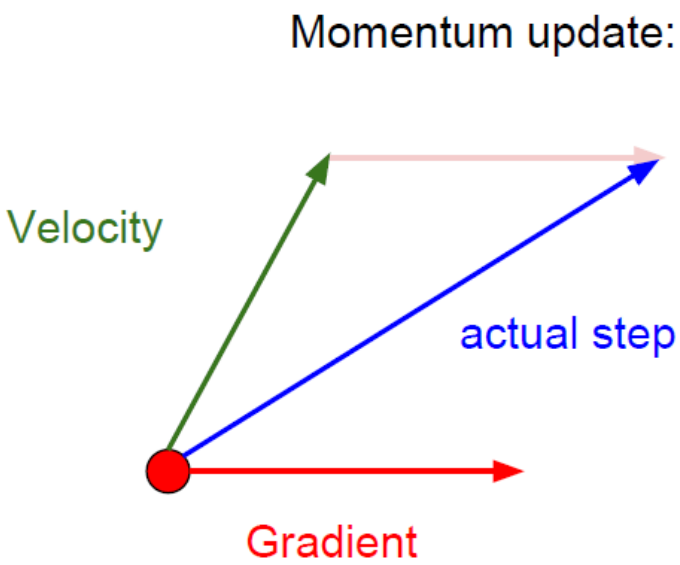
- Build up “velocity” as a running mean of gradients
- Rho gives “friction”; typically rho=0.9 or 0.99

SGD with Momentum



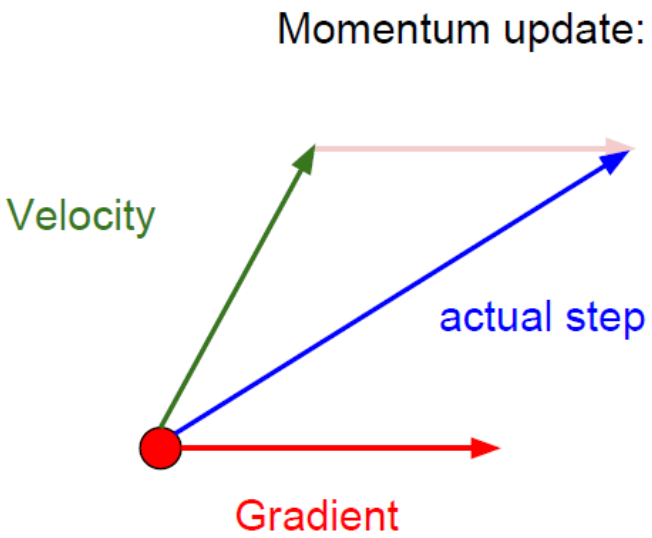


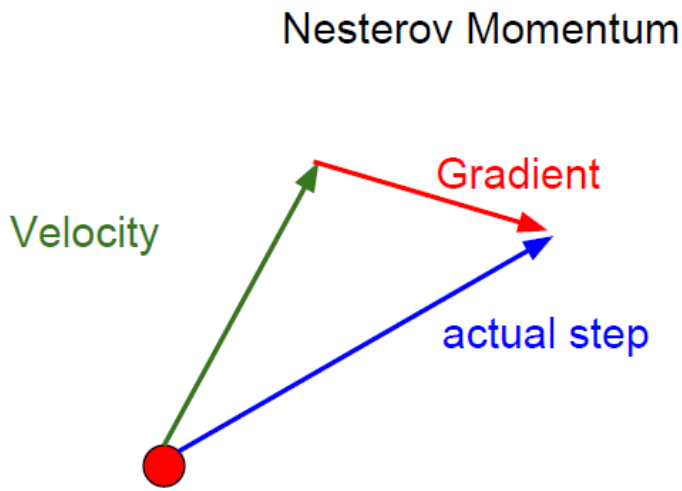
SGD with Momentum



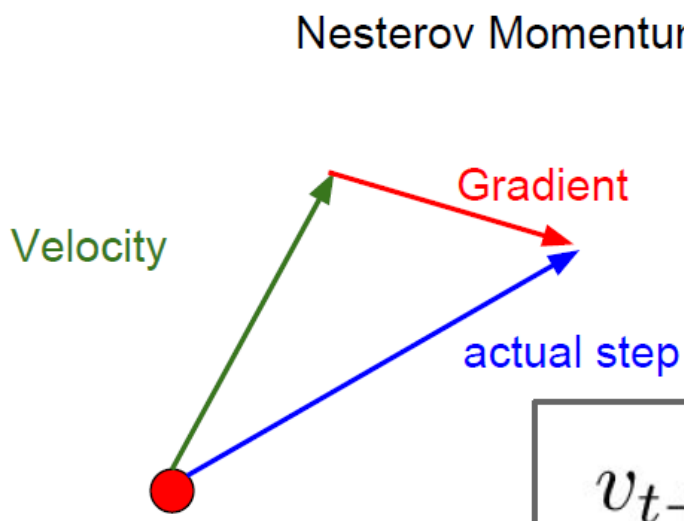
SGD with Nestrov Momentum

Nesterov Momentum



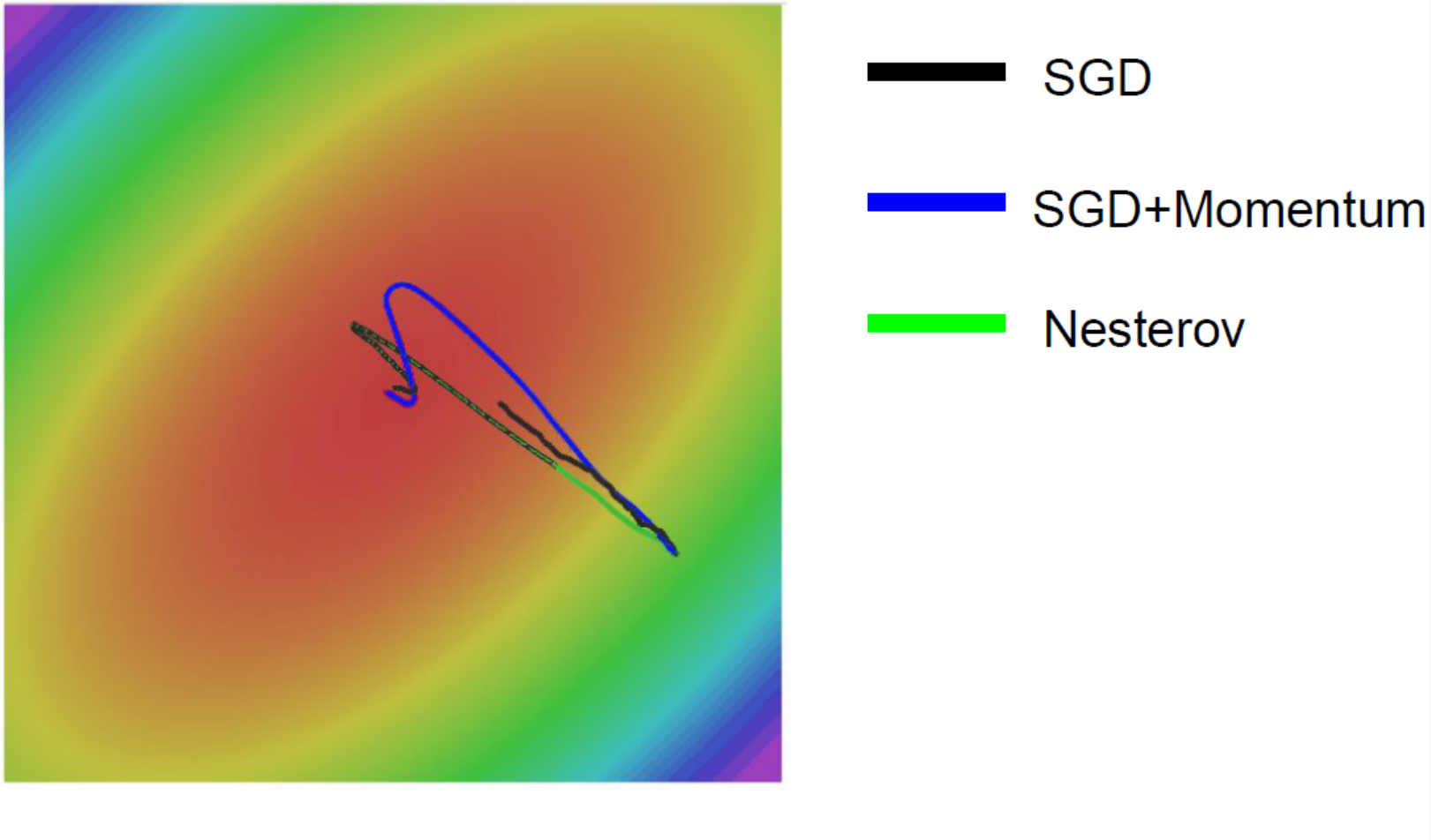


SGD with Nestrov Momentum



$$v_{t+1} = \rho v_t - \alpha \nabla f(x_t + \rho v_t)$$
$$x_{t+1} = x_t + v_{t+1}$$

SGD with Nestrov Momentum



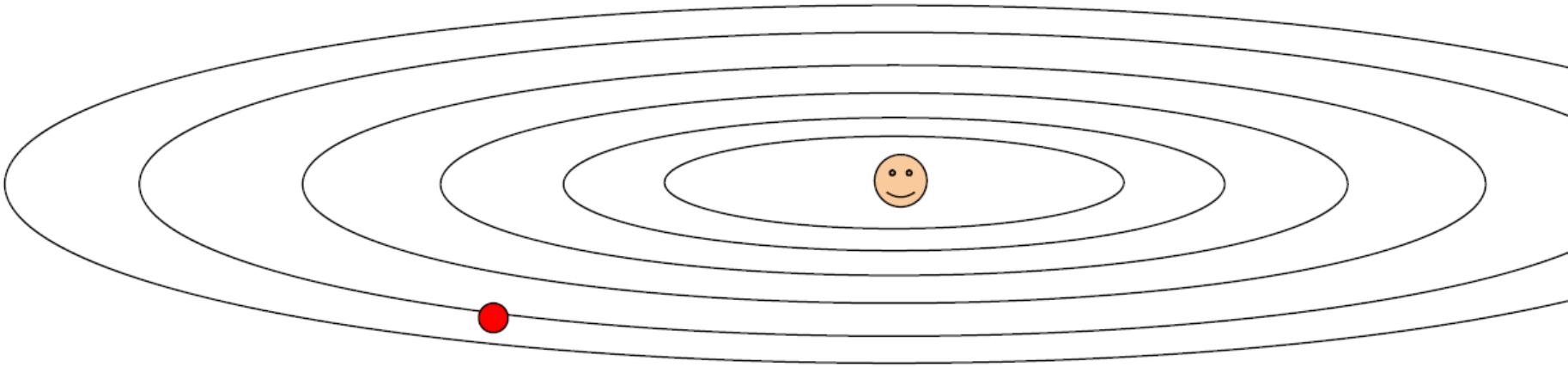
AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

- Adagrad uses second order momentum update
- Adagrad performs per parameter update

AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```



Q: What happens with AdaGrad?

RMSProp

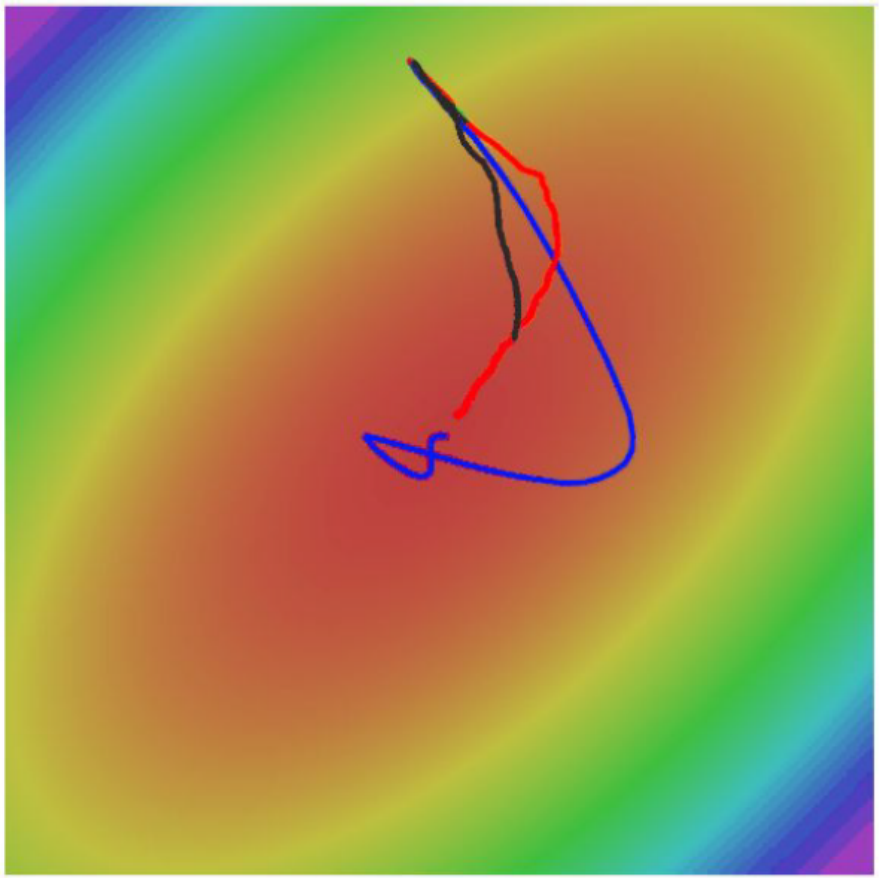
AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSProp

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSProp



SGD

SGD+Momentum

RMSProp

Adam

```
first_moment = 0
second_moment = 0
while True:
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  x -= learning_rate * first_moment / (np.sqrt(second_moment) + 1e-7))
```

Sort of like RMSProp with momentum

Momentum

AdaGrad / RMSProp

Adam

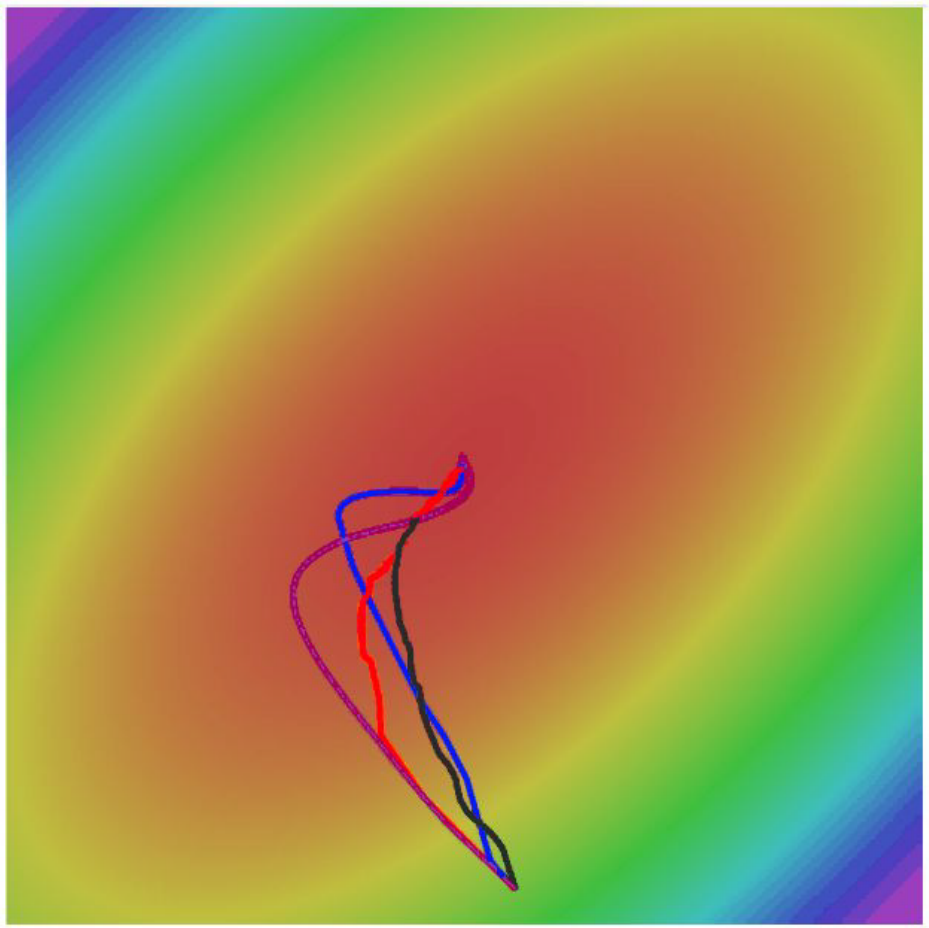
```
first_moment = 0
second_moment = 0
for t in range(num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

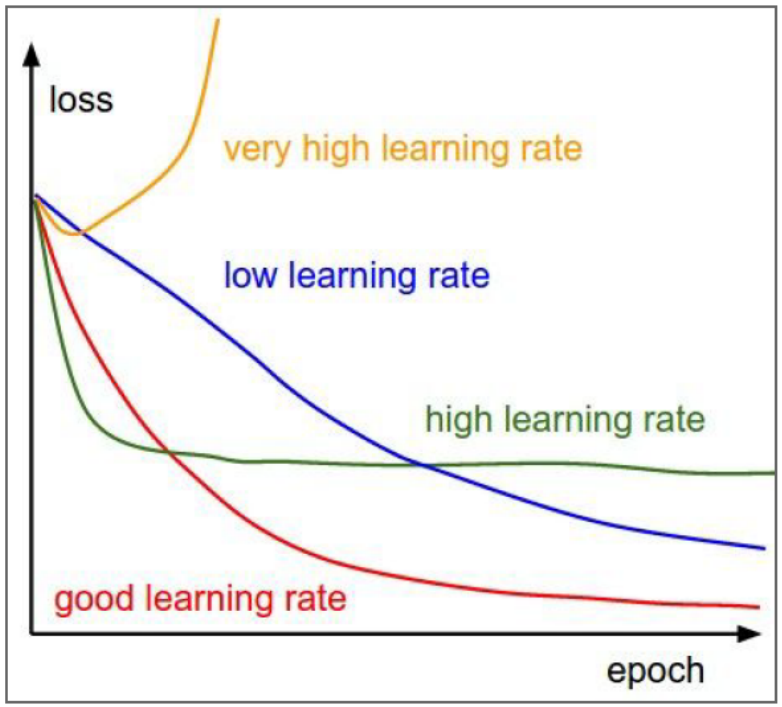
AdaGrad / RMSProp

Adam



Choice of Learning rate

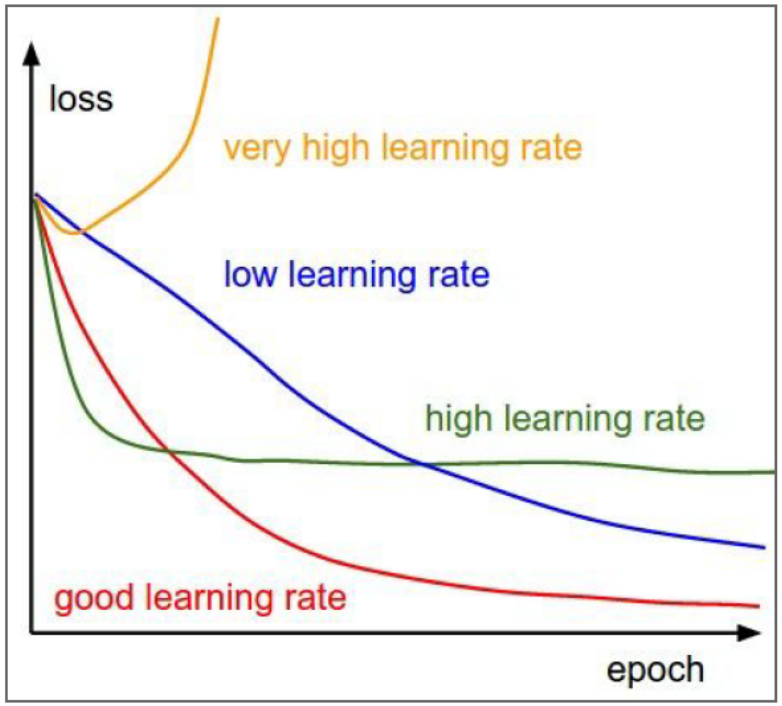
SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



Q: Which one of these learning rates is best to use?

Learning rate decay

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



=> Learning rate decay over time!

step decay:
e.g. decay learning rate by half every few epochs.

exponential decay:

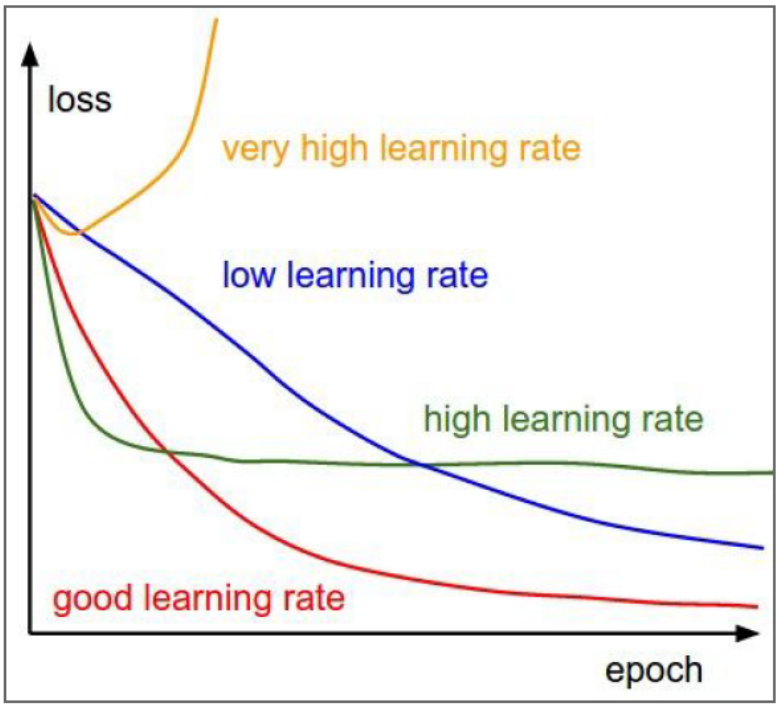
$$\alpha = \alpha_0 e^{-kt}$$

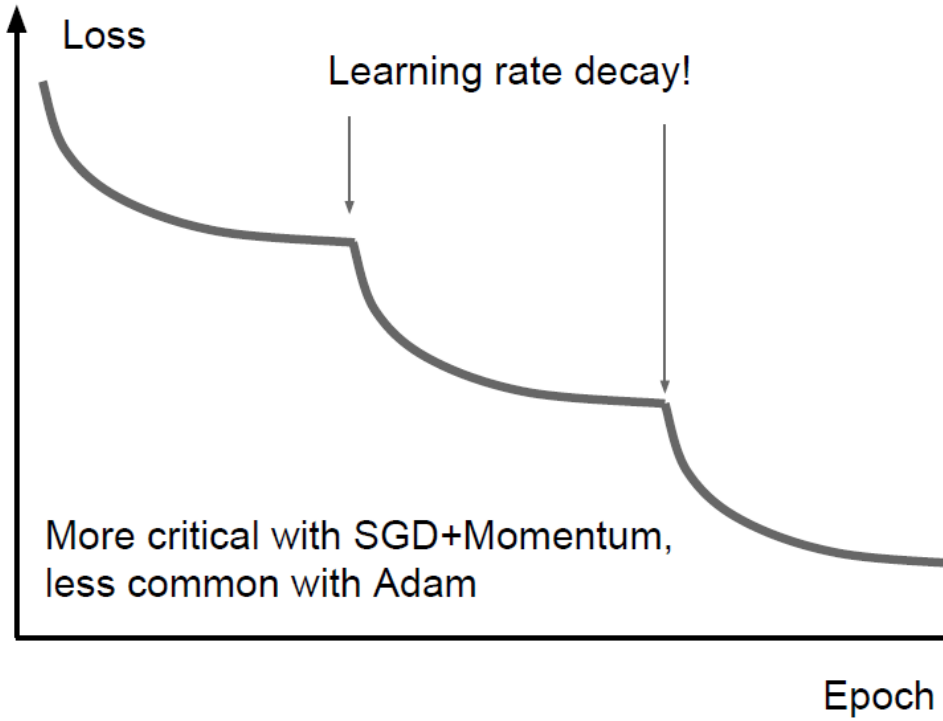
1/t decay:

$$\alpha = \alpha_0 / (1 + kt)$$

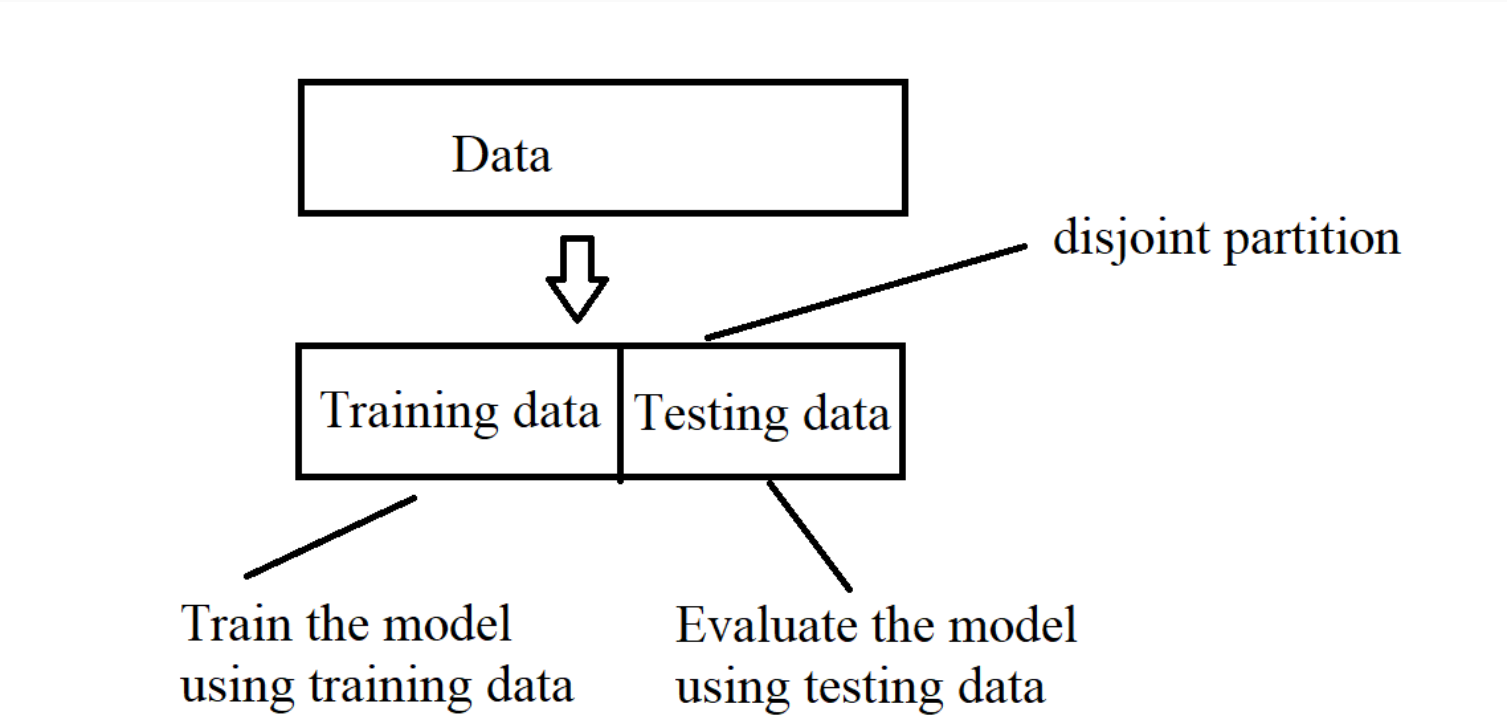
Learning rate decay

SGD, SGD+Momentum, Adagrad, RMSProp, Adam all have **learning rate** as a hyperparameter.



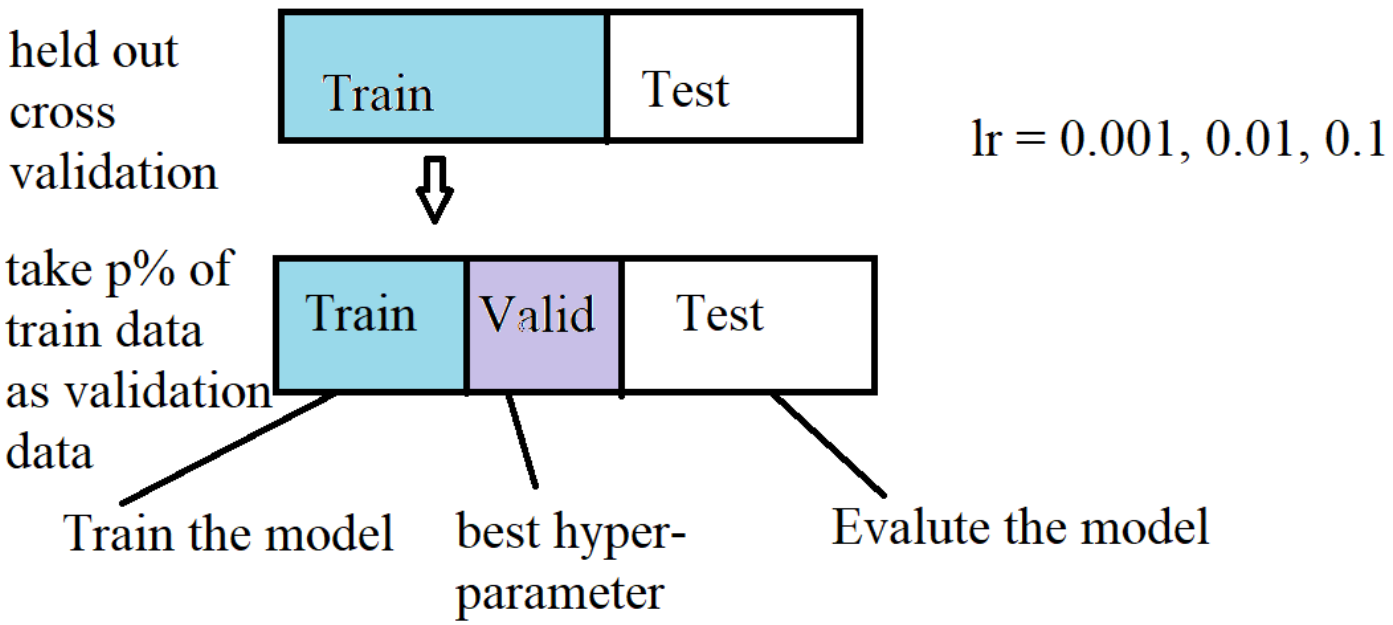


Cross validation:train test split



- this strategy suits when there is no hyper-parameters in the classifier
- hyperparameter - non-trainable parameter
- Example: Bayes classifier (parametric models)

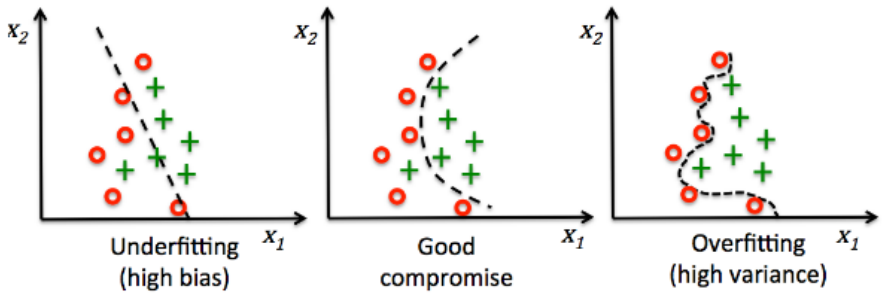
Cross validation: held-out

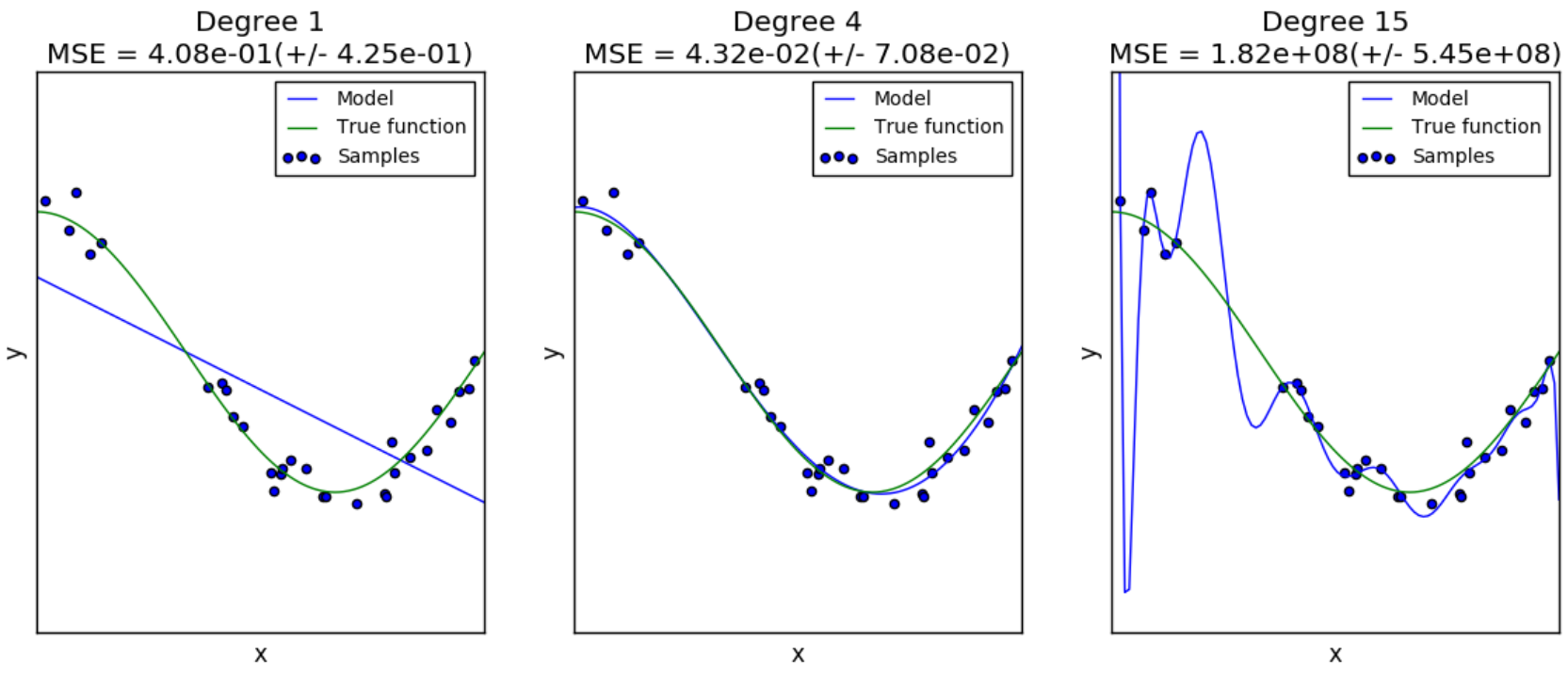


Cross validation: K-fold

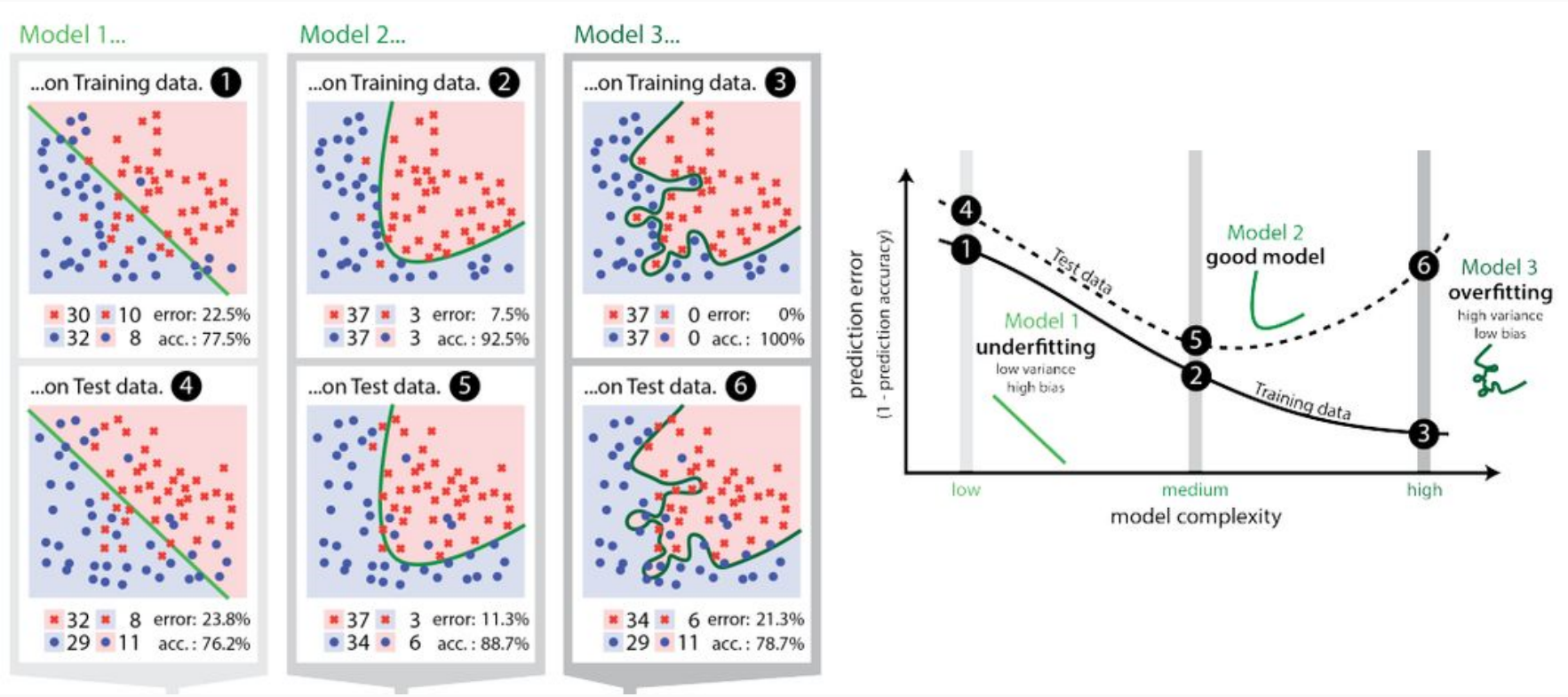
- helps to access the robustness of the classifier when there is a change in the input data
- useful in the determining the best choice of hyperparameter

Cross validation: Over fitting



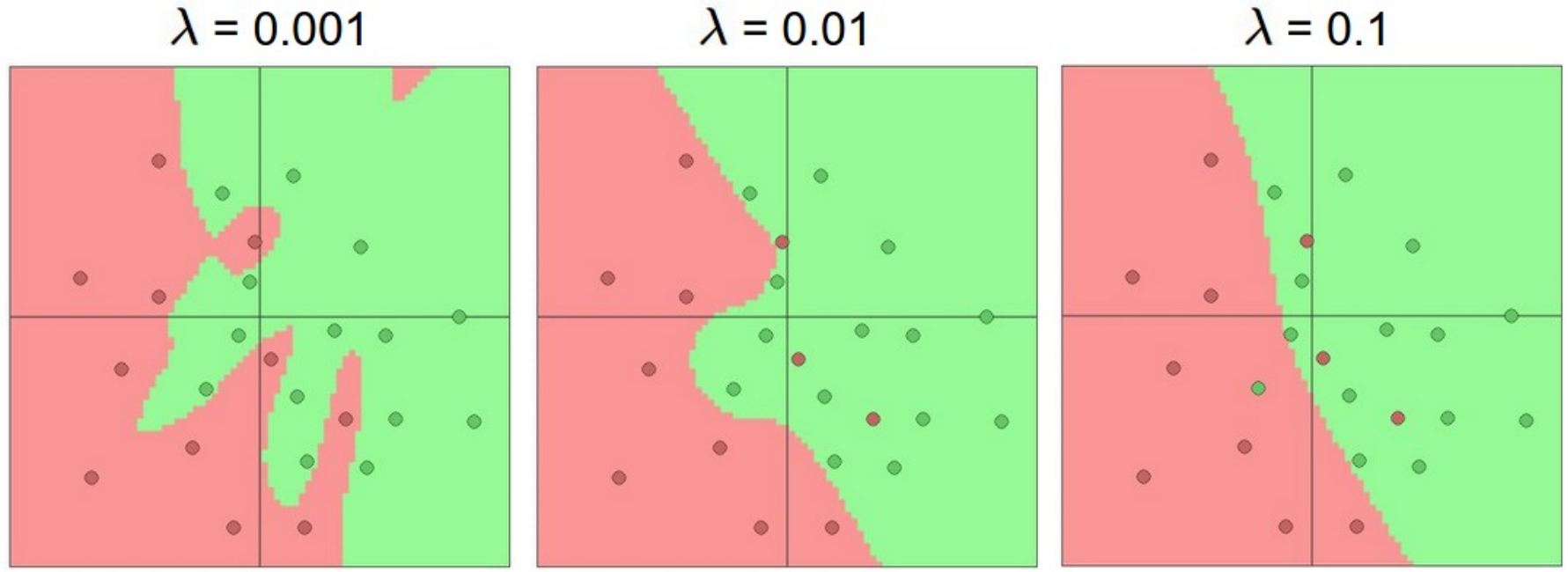


Cross validation: Over fitting



- two class classification problem, Low Generalization error
- training data could be corrupted by noise
- regularization techniques to prevent over fitting : L2, L1

Cross validation: Over fitting



- two class classification problem, Low Generalization error
- regularization techniques to prevent over fitting : L2, L1

Cross validation: tuning hyperparameter

Lets try to train now…

Tip: Make sure that you can overfit very small portion of the training data

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```

The above code:

- take the first 20 examples from CIFAR-10
- turn off regularization (reg = 0.0)
- use simple vanilla 'sgd'

- hyperparameters : learning rate α, regularization parameter λ

Cross validation: tuning hyperparameter

Lets try to train now…

Tip: Make sure that you can overfit very small portion of the training data

Very small loss, train accuracy 1.00, nice!

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of cl
trainer = ClassifierTrainer()
X_tiny = X_train[:20] # take 20 examples
y_tiny = y_train[:20]
best_model, stats = trainer.train(X_tiny, y_tiny, X_tiny, y_tiny,
                                  model, two_layer_net,
                                  num_epochs=200, reg=0.0,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = False,
                                  learning_rate=1e-3, verbose=True)
```

```
Finished epoch 1 / 200: cost 2.302603, train: 0.400000, val 0.400000, lr 1.000000e-03
Finished epoch 2 / 200: cost 2.302258, train: 0.450000, val 0.450000, lr 1.000000e-03
Finished epoch 3 / 200: cost 2.301849, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 4 / 200: cost 2.301196, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 5 / 200: cost 2.300044, train: 0.650000, val 0.650000, lr 1.000000e-03
Finished epoch 6 / 200: cost 2.297864, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 7 / 200: cost 2.293595, train: 0.600000, val 0.600000, lr 1.000000e-03
Finished epoch 8 / 200: cost 2.285096, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 9 / 200: cost 2.268094, train: 0.550000, val 0.550000, lr 1.000000e-03
Finished epoch 10 / 200: cost 2.234787, train: 0.500000, val 0.500000, lr 1.000000e-0
Finished epoch 11 / 200: cost 2.173187, train: 0.500000, val 0.500000, lr 1.000000e-0
Finished epoch 12 / 200: cost 2.076862, train: 0.500000, val 0.500000, lr 1.000000e-0
Finished epoch 13 / 200: cost 1.974090, train: 0.400000, val 0.400000, lr 1.000000e-0
Finished epoch 14 / 200: cost 1.895885, train: 0.400000, val 0.400000, lr 1.000000e-0
Finished epoch 15 / 200: cost 1.820876, train: 0.450000, val 0.450000, lr 1.000000e-0
Finished epoch 16 / 200: cost 1.737430, train: 0.450000, val 0.450000, lr 1.000000e-0
Finished epoch 17 / 200: cost 1.642356, train: 0.500000, val 0.500000, lr 1.000000e-0
Finished epoch 18 / 200: cost 1.535239, train: 0.600000, val 0.600000, lr 1.000000e-0
Finished epoch 19 / 200: cost 1.421527, train: 0.600000, val 0.600000, lr 1.000000e-0
...
Finished epoch 195 / 200: cost 0.002694, train: 1.000000, val 1.000000, lr 1.00000
Finished epoch 196 / 200: cost 0.002674, train: 1.000000, val 1.000000, lr 1.00000
Finished epoch 197 / 200: cost 0.002655, train: 1.000000, val 1.000000, lr 1.00000
Finished epoch 198 / 200: cost 0.002635, train: 1.000000, val 1.000000, lr 1.00000
Finished epoch 199 / 200: cost 0.002617, train: 1.000000, val 1.000000, lr 1.00000
Finished epoch 200 / 200: cost 0.002597, train: 1.000000, val 1.000000, lr 1.00000
finished optimization. best validation accuracy: 1.000000
```

Cross validation: tuning hyperparameter

Lets try to train now…

Start with small regularization and find learning rate that makes the loss go down.

loss not going down: learning rate too low

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, numb
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e-6, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.302576, train: 0.080000, val 0.103000, lr 1.000
Finished epoch 2 / 10: cost 2.302582, train: 0.121000, val 0.124000, lr 1.000
Finished epoch 3 / 10: cost 2.302558, train: 0.119000, val 0.138000, lr 1.000
Finished epoch 4 / 10: cost 2.302519, train: 0.127000, val 0.151000, lr 1.000
Finished epoch 5 / 10: cost 2.302517, train: 0.158000, val 0.171000, lr 1.000
Finished epoch 6 / 10: cost 2.302518, train: 0.179000, val 0.172000, lr 1.000
Finished epoch 7 / 10: cost 2.302466, train: 0.180000, val 0.176000, lr 1.000
Finished epoch 8 / 10: cost 2.302452, train: 0.175000, val 0.185000, lr 1.000
Finished epoch 9 / 10: cost 2.302459, train: 0.206000, val 0.192000, lr 1.000
Finished epoch 10 / 10: cost 2.302420, train: 0.190000, val 0.192000, lr 1.00
finished optimization. best validation accuracy: 0.192000
```

Loss barely changing: Learning rate is probably too low

Notice train/val accuracy goes to 20% though, what's up with that? (remember this is softmax)

Cross validation: tuning hyperparameter

Lets try to train now…

Start with small regularization and find learning rate that makes the loss go down.

loss not going down:
learning rate too low
loss exploding:
learning rate too high

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of cla
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=1e6, verbose=True)
```

```
/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:50: RuntimeWarning: divide
countered in log
  data_loss = -np.sum(np.log(probs[range(N), y])) / N
/home/karpathy/cs231n/code/cs231n/classifiers/neural_net.py:48: RuntimeWarning: invali
ountered in subtract
  probs = np.exp(scores - np.max(scores, axis=1, keepdims=True))
Finished epoch 1 / 10: cost nan, train: 0.091000, val 0.087000, lr 1.000000e+06
Finished epoch 2 / 10: cost nan, train: 0.095000, val 0.087000, lr 1.000000e+06
Finished epoch 3 / 10: cost nan, train: 0.100000, val 0.087000, lr 1.000000e+06
```

cost: NaN almost always means high learning rate...

Cross validation: tuning hyperparameter

Lets try to train now…

Start with small regularization and find learning rate that makes the loss go down.

loss not going down:
learning rate too low
loss exploding:
learning rate too high

```
model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, numbe
trainer = ClassifierTrainer()
best_model, stats = trainer.train(X_train, y_train, X_val, y_val,
                                  model, two_layer_net,
                                  num_epochs=10, reg=0.000001,
                                  update='sgd', learning_rate_decay=1,
                                  sample_batches = True,
                                  learning_rate=3e-3, verbose=True)
```

```
Finished epoch 1 / 10: cost 2.186654, train: 0.308000, val 0.306000, lr 3.0000
Finished epoch 2 / 10: cost 2.176230, train: 0.330000, val 0.350000, lr 3.0000
Finished epoch 3 / 10: cost 1.942257, train: 0.376000, val 0.352000, lr 3.0000
Finished epoch 4 / 10: cost 1.827868, train: 0.329000, val 0.310000, lr 3.0000
Finished epoch 5 / 10: cost inf, train: 0.128000, val 0.128000, lr 3.000000e-0
Finished epoch 6 / 10: cost inf, train: 0.144000, val 0.147000, lr 3.000000e-0
```

3e-3 is still too high. Cost explodes….

=> Rough range for learning rate we should be cross-validating is somewhere [1e-3 … 1e-5]

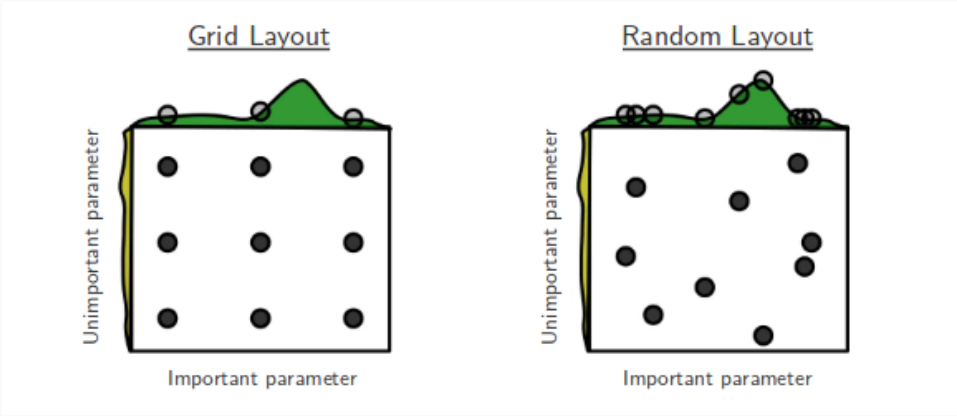
Cross-validation strategy

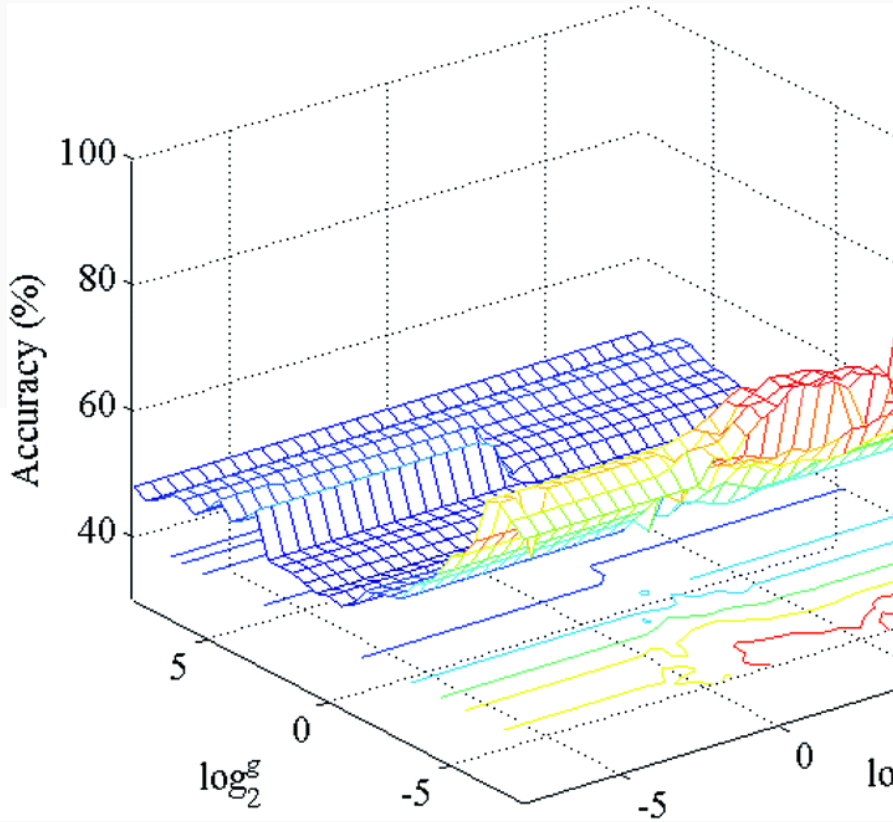
coarse -> fine cross-validation in stages

First stage: only a few epochs to get rough idea of what params work
Second stage: longer running time, finer search
… (repeat as necessary)

Tip for detecting explosions in the solver:
If the cost is ever > 3 * original cost, break out early

Cross validation: tuning hyperparameter-Grid Search





Cross validation: tuning hyperparameter

For example: run coarse search for 5 epochs

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)

      trainer = ClassifierTrainer()
      model = init_two_layer_model(32*32*3, 50, 10) # input size, hidden size, number of classes
      trainer = ClassifierTrainer()
      best_model_local, stats = trainer.train(X_train, y_train, X_val, y_val,
                                              model, two_layer_net,
                                              num_epochs=5, reg=reg,
                                              update='momentum', learning_rate_decay=0.9,
                                              sample_batches = True, batch_size = 100,
                                              learning_rate=lr, verbose=False)
```

note it's best to optimiz
in log space!

```
val_acc: 0.412000, lr: 1.405206e-04, reg: 4.793564e-01, (1 / 100)
val_acc: 0.214000, lr: 7.231888e-06, reg: 2.321281e-04, (2 / 100)
val_acc: 0.208000, lr: 2.119571e-06, reg: 8.011857e+01, (3 / 100)
val_acc: 0.196000, lr: 1.551131e-05, reg: 4.374936e-05, (4 / 100)
val_acc: 0.079000, lr: 1.753300e-05, reg: 1.200424e+03, (5 / 100)
val_acc: 0.223000, lr: 4.215128e-05, reg: 4.196174e+01, (6 / 100)
val_acc: 0.441000, lr: 1.750259e-04, reg: 2.110807e-04, (7 / 100)
val_acc: 0.241000, lr: 6.749231e-05, reg: 4.226413e+01, (8 / 100)
val_acc: 0.482000, lr: 4.296863e-04, reg: 6.642555e-01, (9 / 100)
val_acc: 0.079000, lr: 5.401602e-06, reg: 1.599828e+04, (10 / 100)
val_acc: 0.154000, lr: 1.618508e-06, reg: 4.925252e-01, (11 / 100)
```

nice

Cross validation: tuning hyperparameter

Now run finer search...

```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-5, 5)
      lr = 10**uniform(-3, -6)
```

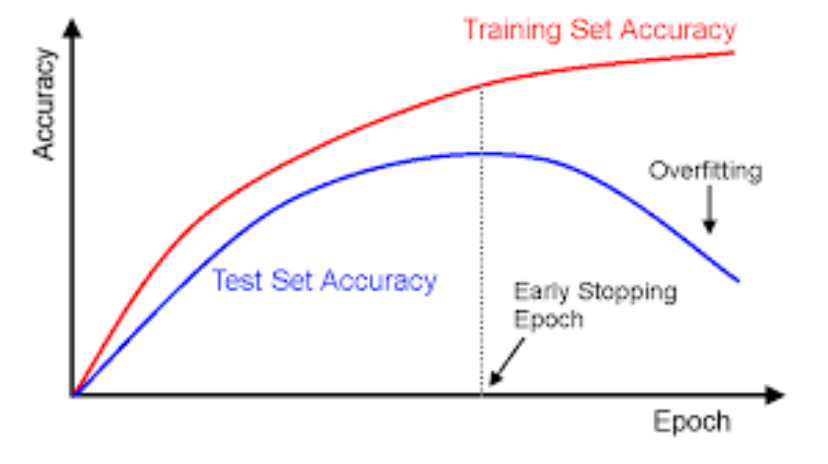
adjust range

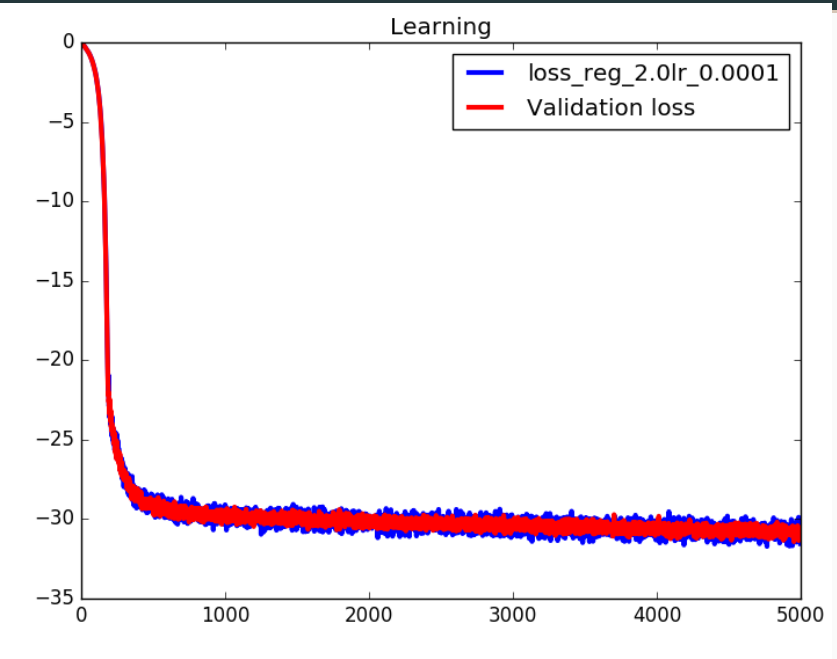
```
max_count = 100
for count in xrange(max_count):
      reg = 10**uniform(-4, 0)
      lr = 10**uniform(-3, -4)
```

```
val_acc: 0.527000, lr: 5.340517e-04, reg: 4.097824e-01, (0 / 100)
val_acc: 0.492000, lr: 2.279484e-04, reg: 9.991345e-04, (1 / 100)
val_acc: 0.512000, lr: 8.680827e-04, reg: 1.349727e-02, (2 / 100)
val_acc: 0.461000, lr: 1.028377e-04, reg: 1.220193e-02, (3 / 100)
val_acc: 0.460000, lr: 1.113730e-04, reg: 5.244309e-02, (4 / 100)
val_acc: 0.498000, lr: 9.477776e-04, reg: 2.001293e-03, (5 / 100)
val_acc: 0.469000, lr: 1.484369e-04, reg: 4.328313e-01, (6 / 100)
val_acc: 0.522000, lr: 5.586261e-04, reg: 2.312685e-04, (7 / 100)
val_acc: 0.530000, lr: 5.808183e-04, reg: 8.259964e-02, (8 / 100)
val_acc: 0.489000, lr: 1.979168e-04, reg: 1.010889e-04, (9 / 100)
val_acc: 0.490000, lr: 2.036031e-04, reg: 2.406271e-03, (10 / 100)
val_acc: 0.475000, lr: 2.021162e-04, reg: 2.287807e-01, (11 / 100)
val_acc: 0.460000, lr: 1.135527e-04, reg: 3.905040e-02, (12 / 100)
val_acc: 0.515000, lr: 6.947668e-04, reg: 1.562808e-02, (13 / 100)
val_acc: 0.531000, lr: 9.471549e-04, reg: 1.433895e-03, (14 / 100)
val_acc: 0.509000, lr: 3.140888e-04, reg: 2.857518e-01, (15 / 100)
val_acc: 0.514000, lr: 6.438349e-04, reg: 3.033781e-01, (16 / 100)
val_acc: 0.502000, lr: 3.921784e-04, reg: 2.707126e-04, (17 / 100)
val_acc: 0.509000, lr: 9.752279e-04, reg: 2.850865e-03, (18 / 100)
val_acc: 0.500000, lr: 2.412048e-04, reg: 4.997821e-04, (19 / 100)
val_acc: 0.466000, lr: 1.319314e-04, reg: 1.189915e-02, (20 / 100)
val_acc: 0.516000, lr: 8.039527e-04, reg: 1.528291e-02, (21 / 100)
```

53% - relatively good for a 2-layer neural net with 50 hidden neurons

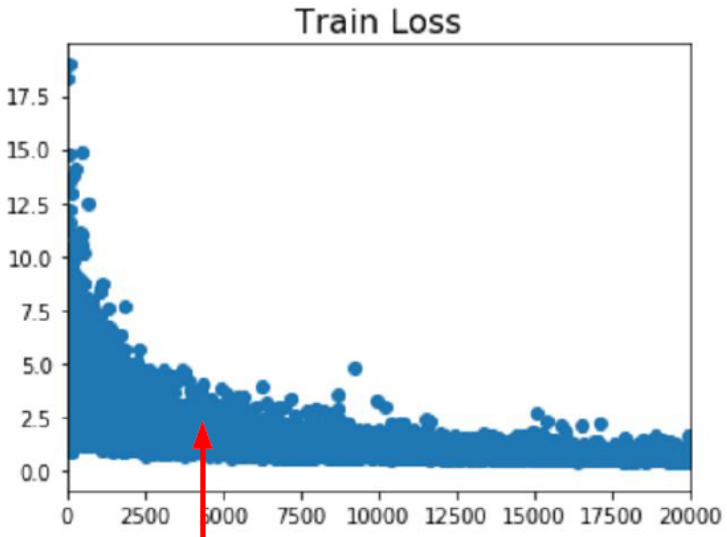
Cross validation: Early stopping in NN

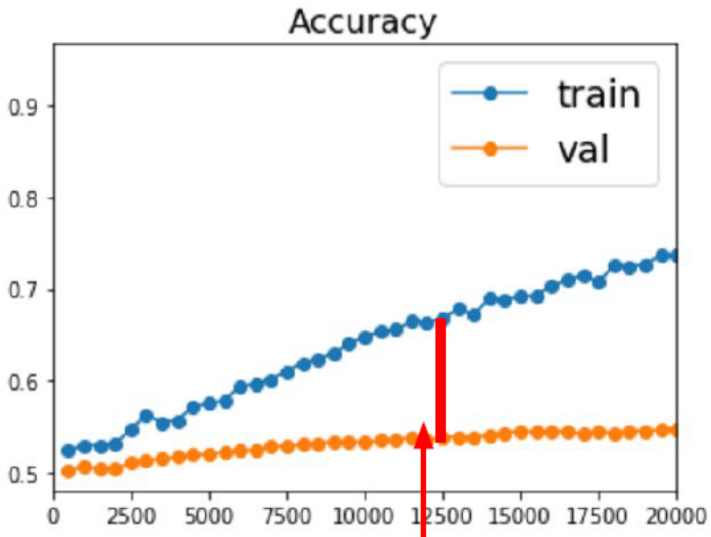




- Why does test accuracy decreases?
- this behaviour is not we expect
- It is not easy to have the right side plot always
- to overcome this you can use regularization, but some times still this problem exists

Cross validation: Early stopping in NN





Better optimization algorithms help reduce training loss

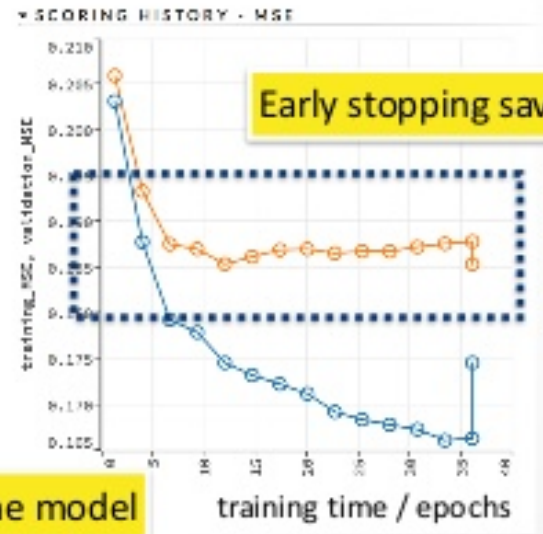
But we really care about error on new data - how to reduce the gap?

Cross validation: Early stopping in NN

Tip #4: Use Early Stopping! (On by default)

Before: trains too long, but at least **overwrite_with_best_model=true** prevents overfitting (returns the model with lowest validation error)

Now: specify additional convergence criterion: E.g. **stopping_rounds=5, stopping_metric="MSE", stopping_tolerance=1e-3,** to stop as soon as the moving average (length 5) of the validation MSE does not improve by at least 0.1% for 5 consecutive scoring events



Higgs dataset

Use Flow to inspect the model

Batch Normalization

[Ioffe and Szegedy, 2015]

"you want unit gaussian activations? just make them so."

consider a batch of activations at some layer.
To make each dimension unit gaussian, apply:

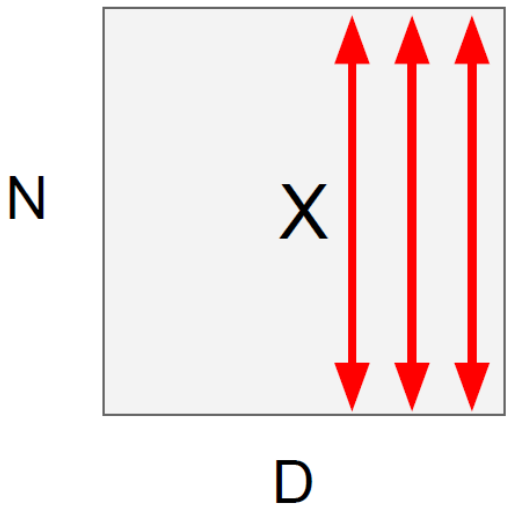
$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

this is a vanilla differentiable function...

Batch Normalization

[Ioffe and Szegedy, 2015]

"you want unit gaussian activations?
just make them so."



1. compute the empirical mean and variance independently for each dimension.

2. Normalize

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

Batch Normalization

[Ioffe and Szegedy, 2015]

Normalize:

$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathrm{E}[x^{(k)}]}{\sqrt{\mathrm{Var}[x^{(k)}]}}$$

And then allow the network to squash the range if it wants to:

$$y^{(k)} = \gamma^{(k)}\widehat{x}^{(k)} + \beta^{(k)}$$

Note, the network can learn:

$$\gamma^{(k)} = \sqrt{\mathrm{Var}[x^{(k)}]}$$

$$\beta^{(k)} = \mathrm{E}[x^{(k)}]$$

to recover the identity mapping.

Batch Normalization

[Ioffe and Szegedy, 2015]

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization in a funny way, and slightly reduces the need for dropout, maybe

Batch Normalization

[Ioffe and Szegedy, 2015]

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

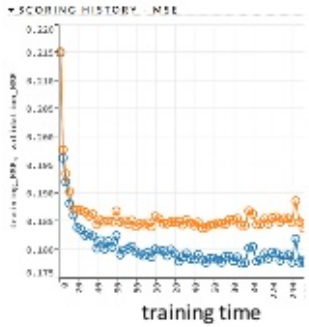
Note: at test time BatchNorm layer functions differently:

The mean/std are not computed based on the batch. Instead, a single fixed empirical mean of activations during training is used.

(e.g. can be estimated during training with running averages)

Dropout

Tip #7: Use Regularization



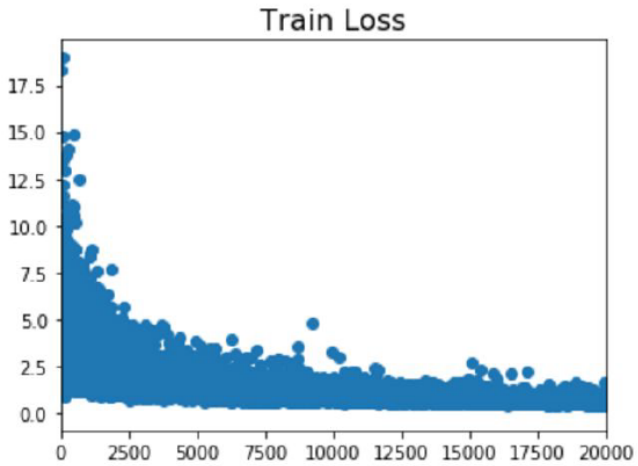
Rectifier, hidden=[50,50]

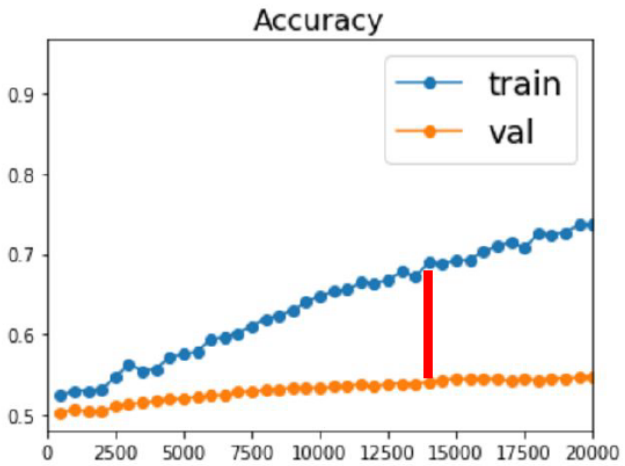
RectifierWithDropout, hidden=[50,50],
l1=1e-4, l2=1e-4, hidden_dropout_ratio=[0.2,0.3]

Overfitting is easy, generalization is art

Higgs dataset

Dropout



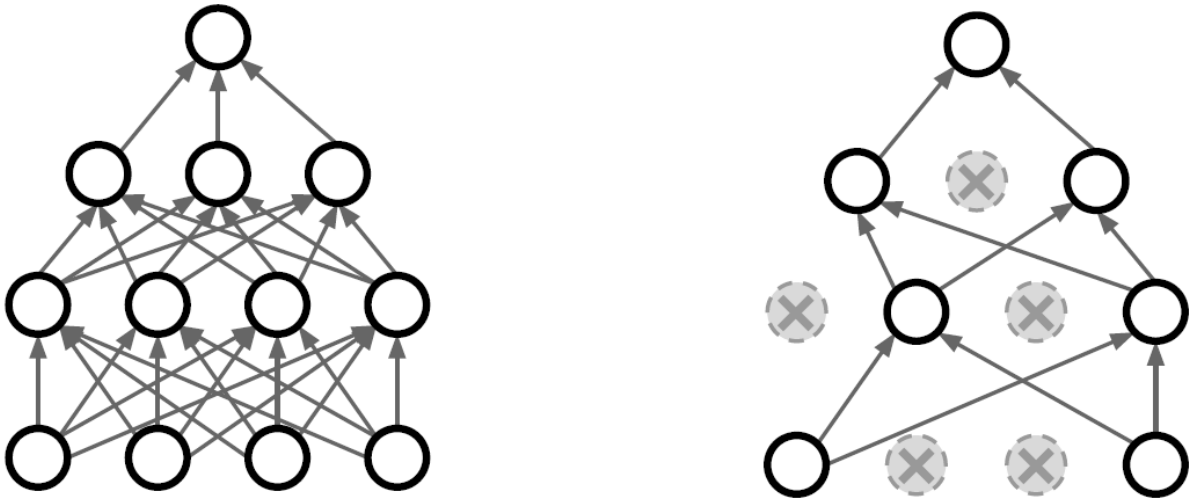


Regularization

Regularization: Dropout

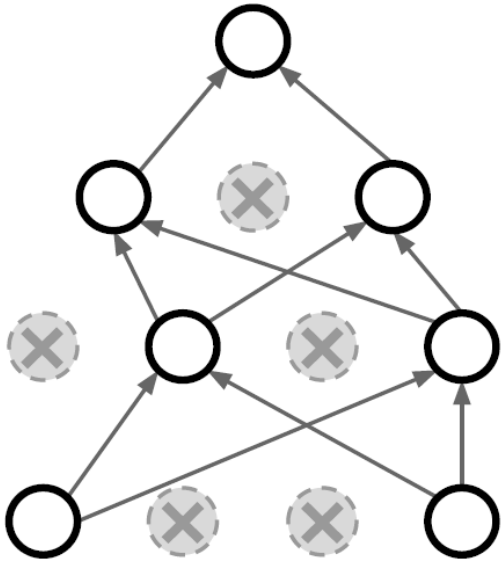
In each forward pass, randomly set some neurons to zero

Probability of dropping is a hyperparameter; 0.5 is common



Regularization: Dropout

How can this possibly be a good idea?



Forces the network to have a redundant representation;
Prevents co-adaptation of features

Dropout: Test time

Dropout makes our output random!

Output (label) Input (image) Random mask

$$y = f_W(x, z)$$

Want to "average out" the randomness at test-time

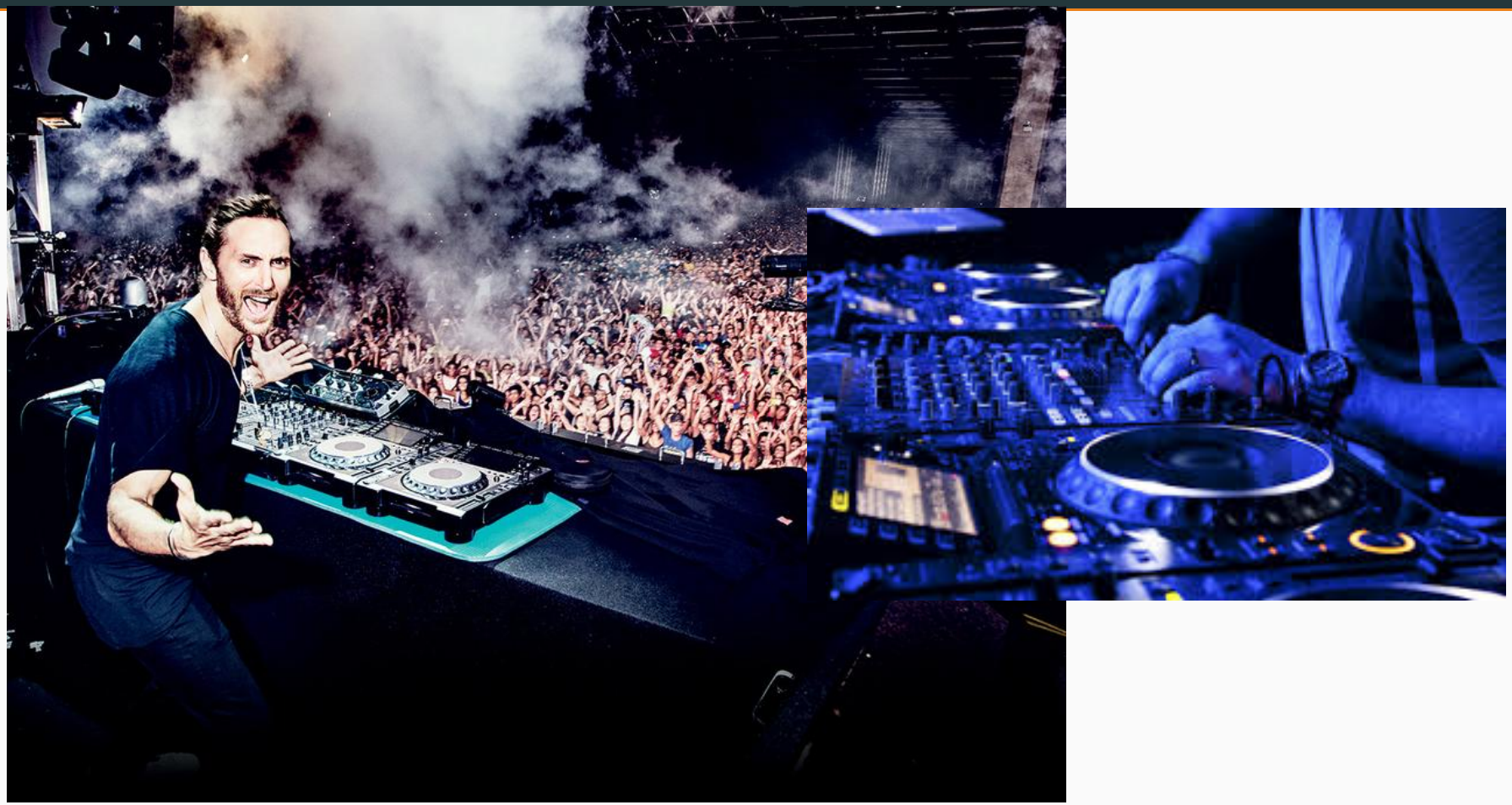
$$y = f(x) = E_z\big[f(x,z)\big] = \int p(z) f(x,z) dz$$

But this integral seems hard …

Recommended Settings

- if you are dealing with images use **ReLU** activation function
- For non-image datasets, first try with **ReLU**, if performance is not satisfactory, then use **sigmoid or tanh** activation function
- scale your data with zero mean and unit variance with **sigmoid and tanh** activation function
- Use **SGD+Nestrov momentum or Adam** optimizer to train your network. For Adam usually, recommended default **learning rate is** $1e-3$
- always monitor the validation loss or accuracy, when the validation loss is not decreasing anymore or starts to increase, stop training with Early stopping criteria

Recommended Settings



- **There is no one good strategy to get the best results. Always play with settings !!!**