CS 291A: Deep Learning for NLP

Neural Networks: Basics

William Wang

UCSB Computer Science

william@cs.ucsb.edu

Slides adapted from Y. V. Chen and H. Lee.

Announcements

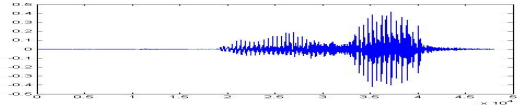
Project proposals are due today to ke00@ucsb.edu.

Homework 1 is out today.

1. You will implement W2V and Glove.
2. **Start early**: if you have not used CodaLab before (register with your umail).
3. If you have not implemented SGD before, it will take time for you to get things right.

Learning ≈ Looking for a Function

Speech Recognition

$f($  $) =$ “你好”

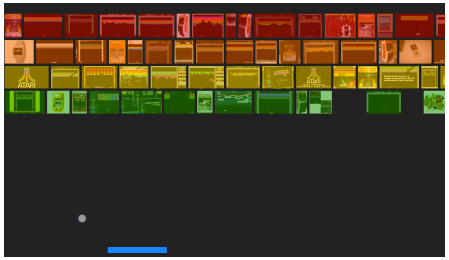
Handwritten Recognition

$f($  $) =$ “2”

Weather forecast

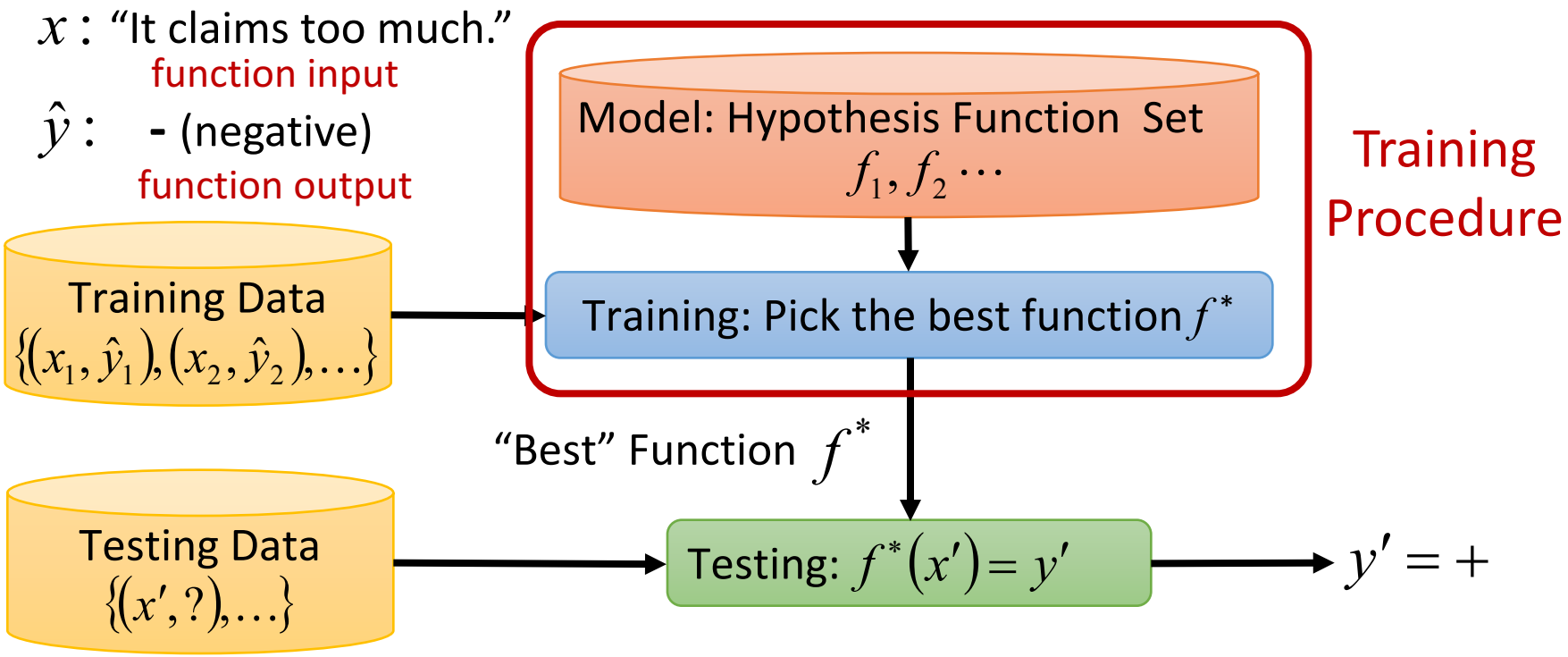
$f($ Thursday $) =$ “  Saturday”

Play video games

$f($  $) =$ “move left”

How to learn the function (a.k.a. train the model)?

Machine Learning Framework



Training is to pick the best function given the observed data
Testing is to predict the label using the learned function

Classification Task

Sentiment Analysis

"this is awesome!" ⟶ +

"this sucks…" ⟶ -

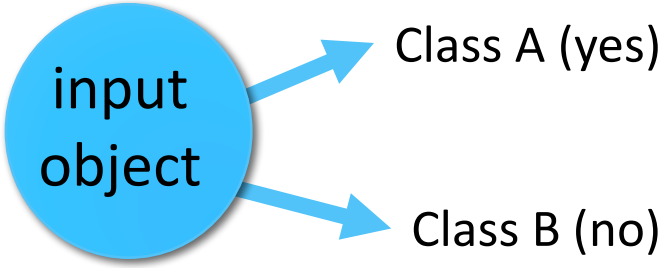
Speech Phoneme Recognition

 ⟶ /h/

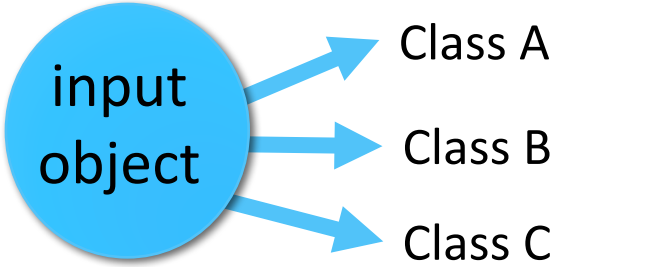
Handwritten Recognition

 ⟶ 2

→ Binary Classification



→ Multi-class Classification



Some cases are not easy to be formulated as classification problems

Target Function

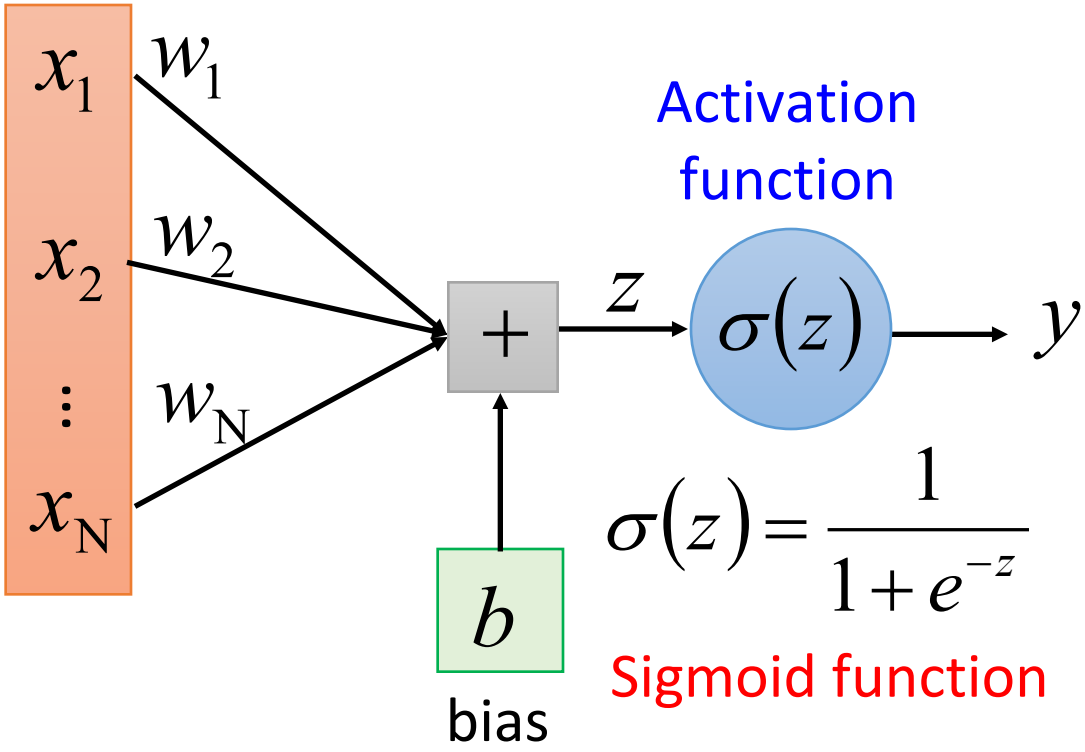
Classification Task

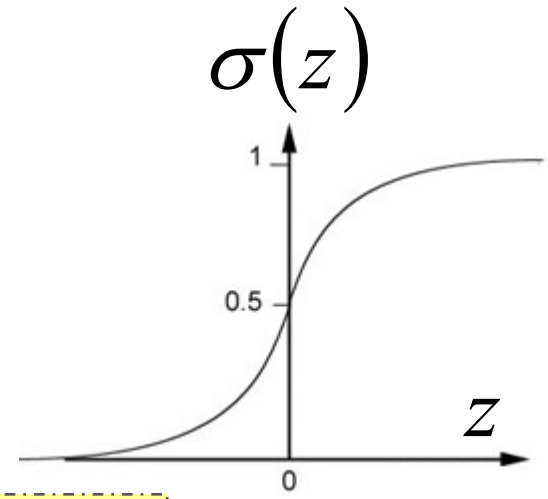
$$f(x) = y \quad \Longrightarrow \quad f : R^N \rightarrow R^M$$

- x: input object to be classified → a N-dim vector
- y: class/label → a M-dim vector

Assume both x and y can be represented as fixed-size vectors

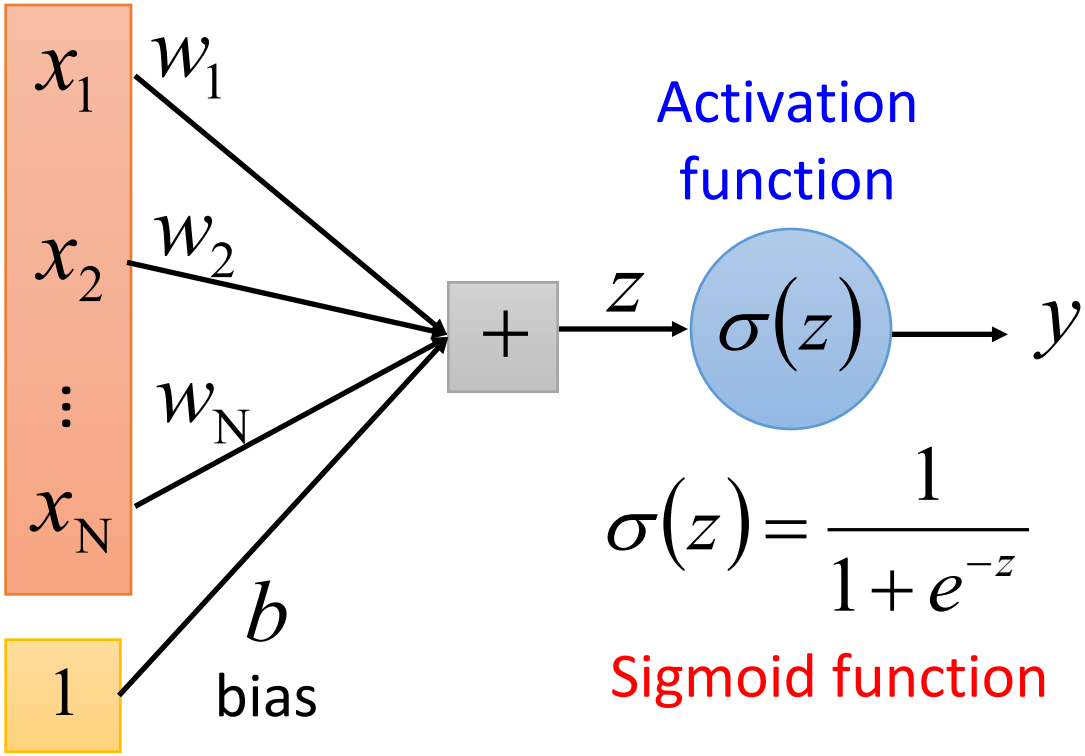
A Single Neuron

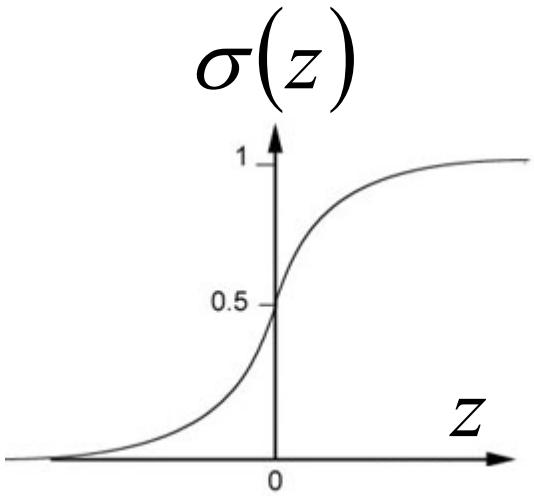




Each neuron is a very simple function

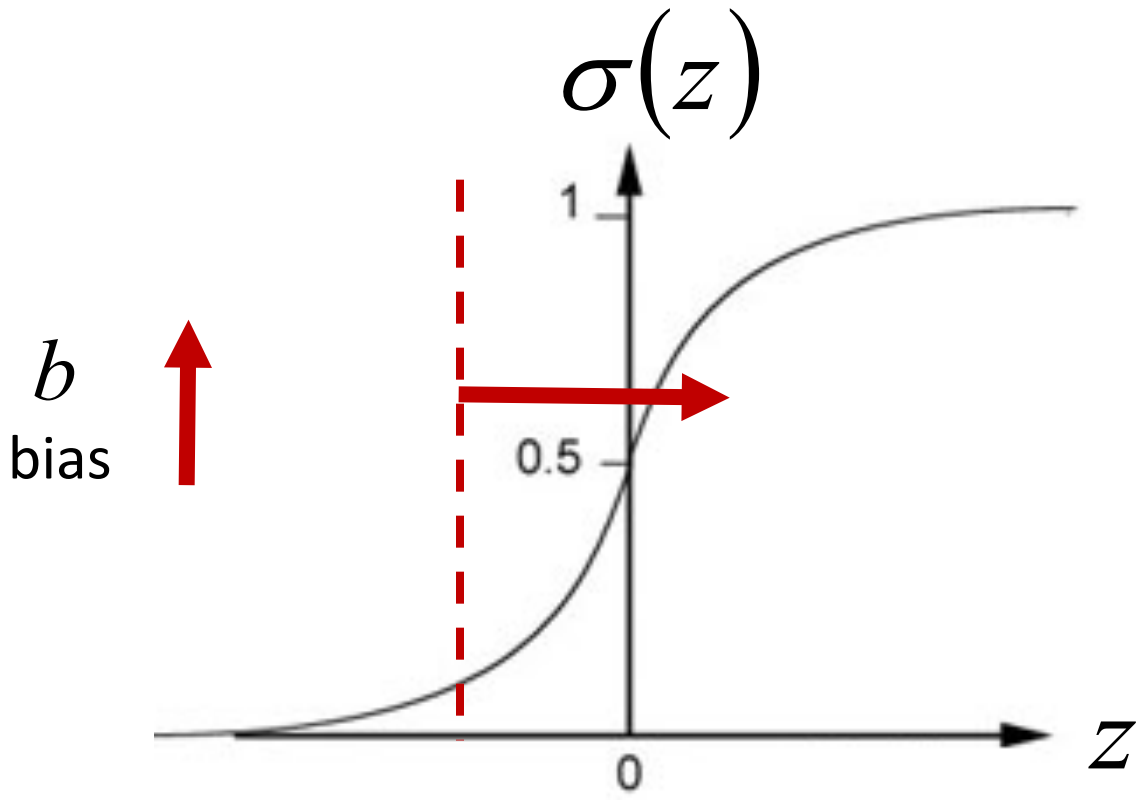
A Single Neuron





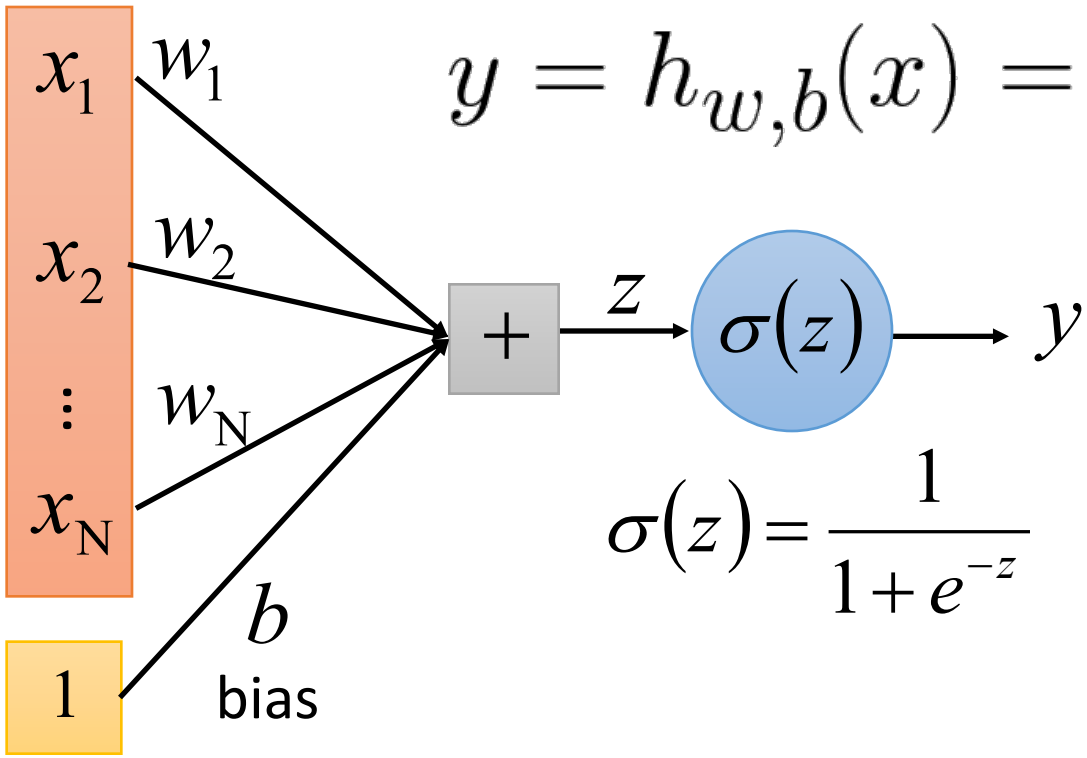
The bias term is an “always on” feature

Why Bias?



The bias term gives a class prior

Model Parameters of A Single Neuron



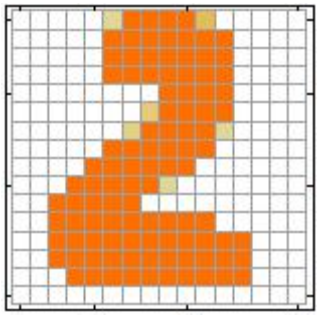
$$y = h_{w,b}(x) = \sigma(w^T x + b)$$

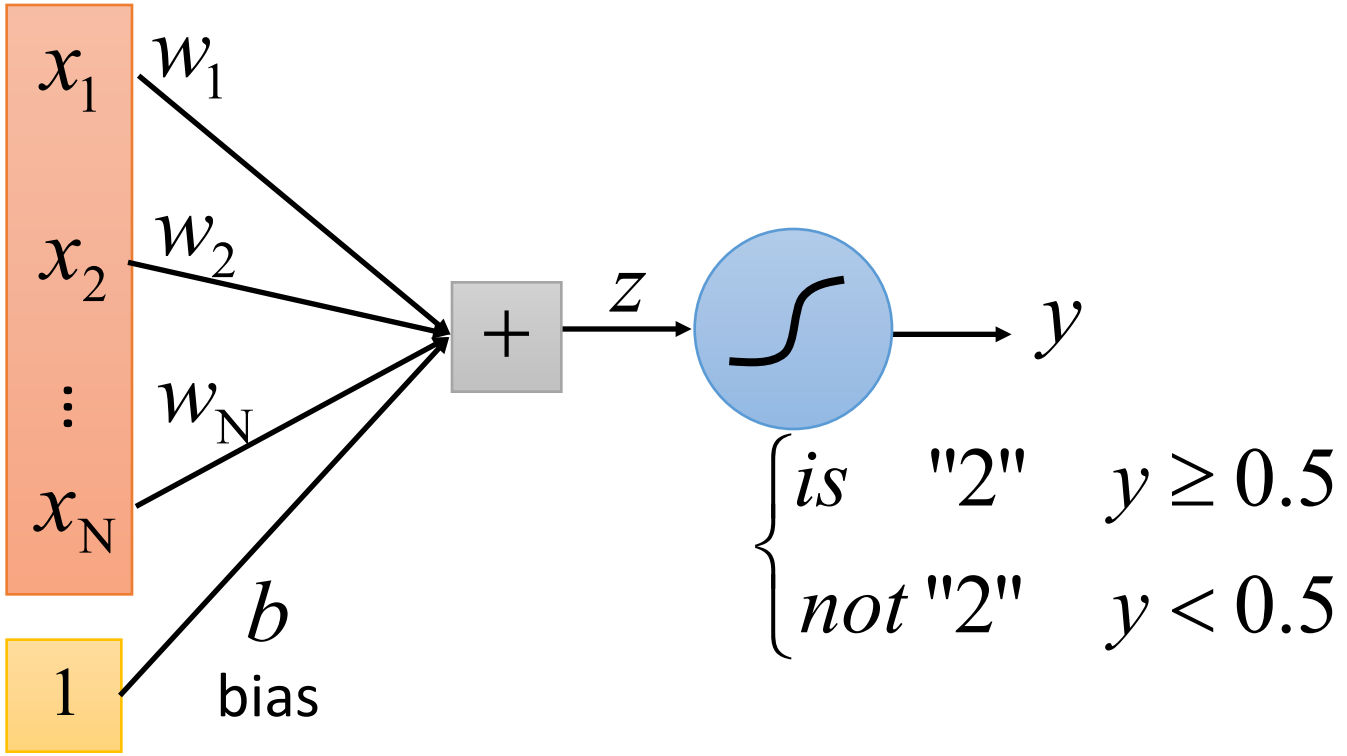
$$\sigma(z) = \frac{1}{1+e^{-z}}$$

w, b are the parameters of this neuron

A Single Neuron

$$f : R^N \rightarrow R^M$$



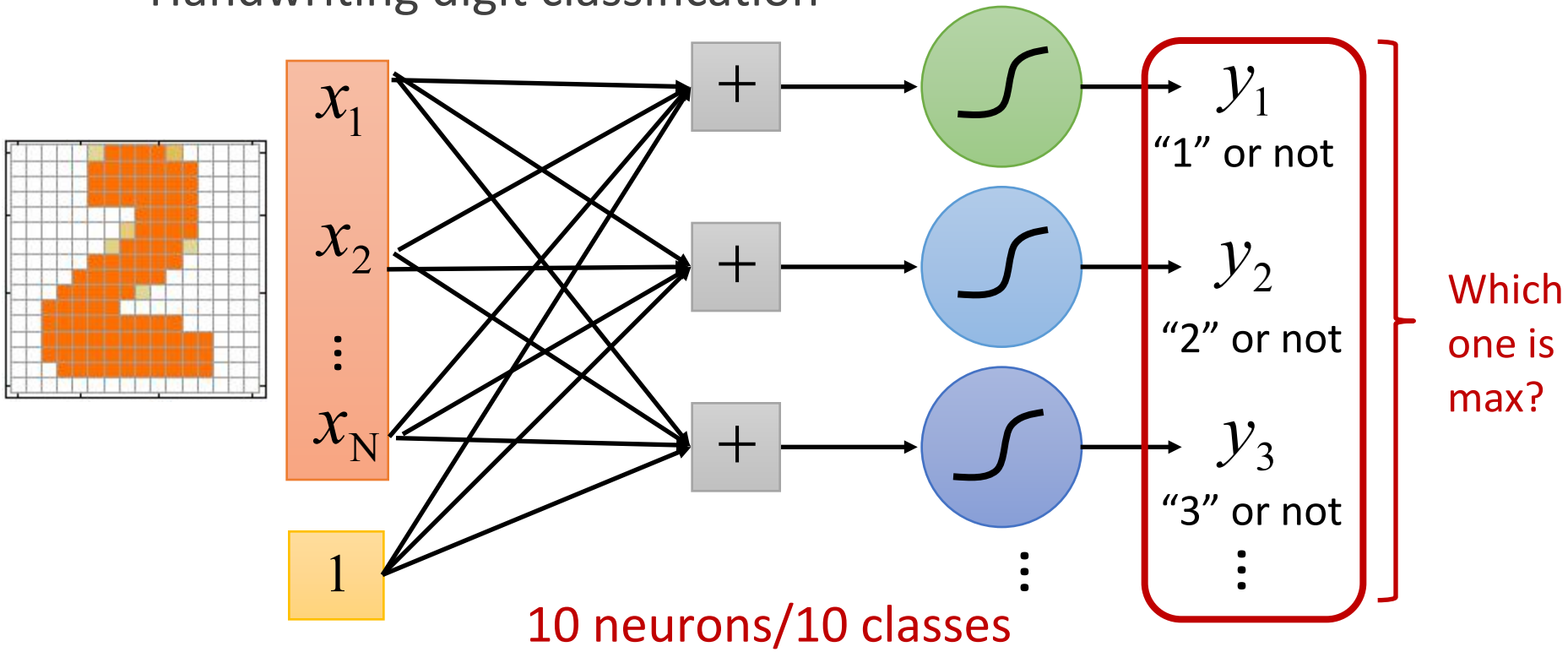


A single neuron can only handle binary classification

A Layer of Neurons

$$f : R^N \rightarrow R^M$$

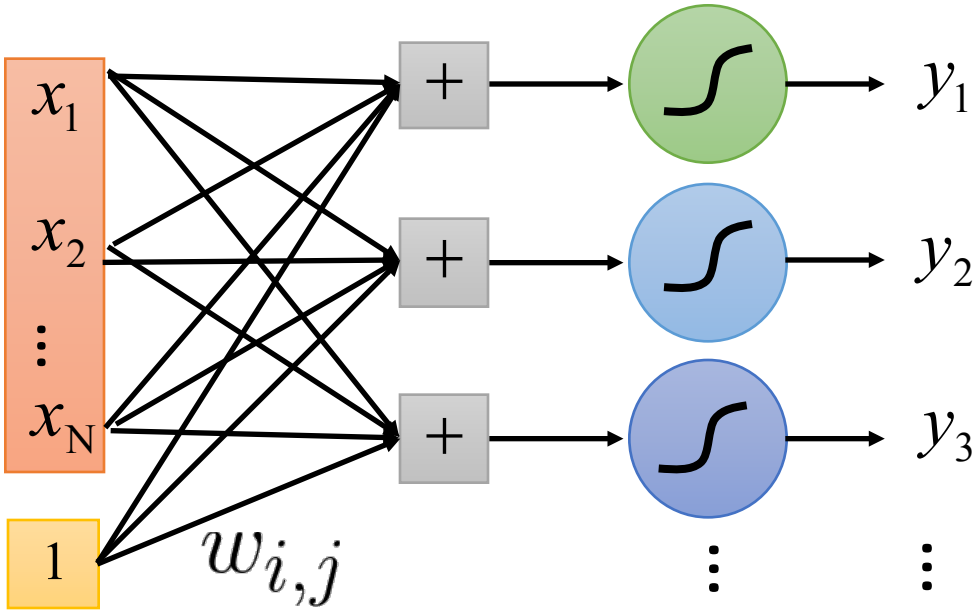
Handwriting digit classification

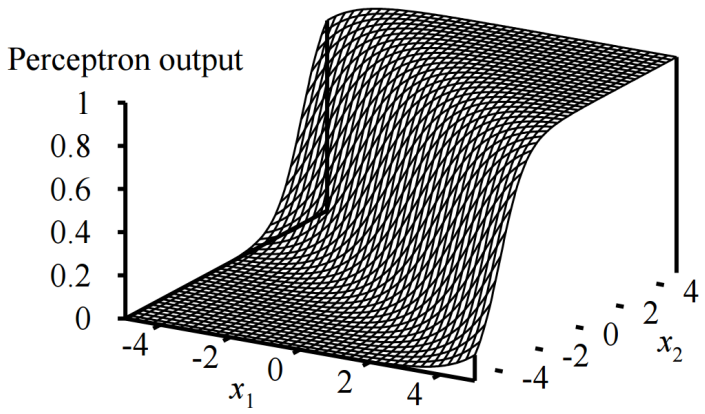


A layer of neurons can handle multiple possible output, and the result depends on the max one

A Layer of Neurons – Perceptron

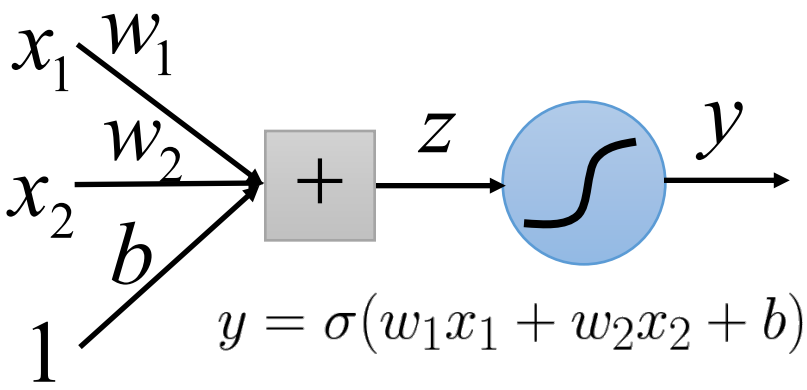
Output units all operate separately – no shared weights





Adjusting weights moves the location, orientation, and steepness of cliff

Piazza Poll: Expressiveness of Perceptron

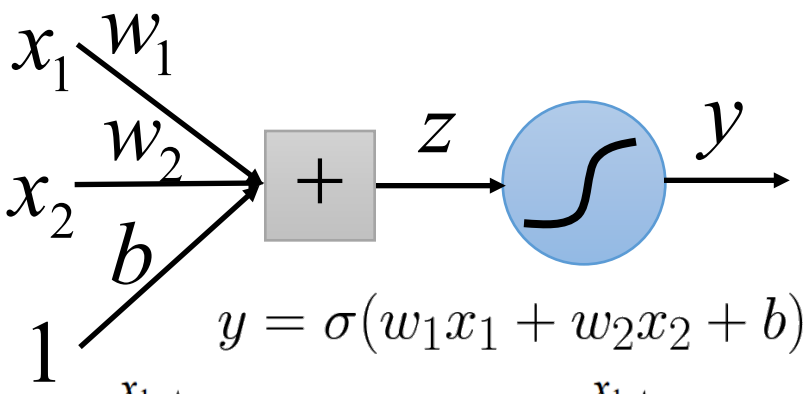


$$\begin{cases} yes & y \geq threshold \\ no & y < threshold \end{cases}$$

Input		Output
A	B	
0	0	0
0	1	1
1	0	1
1	1	0

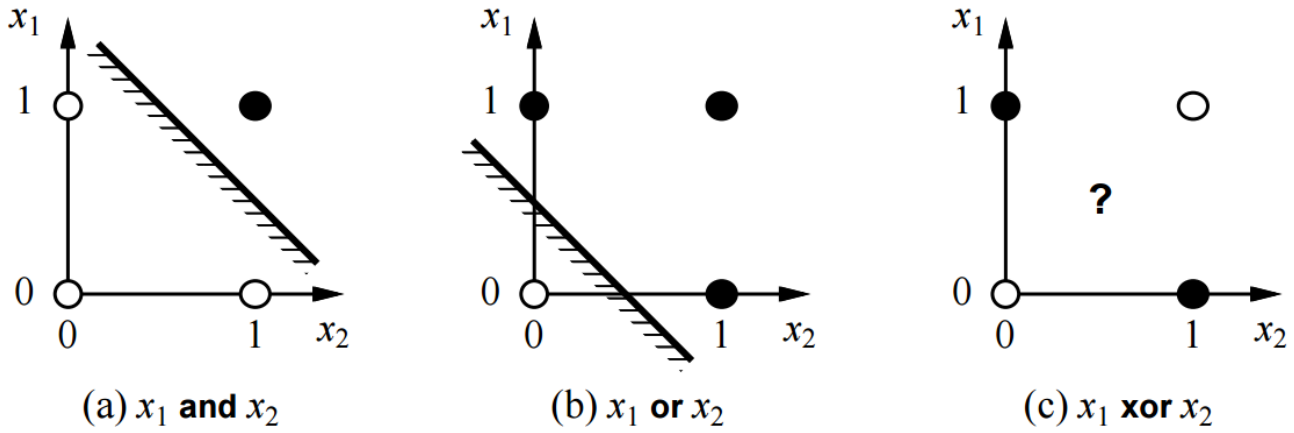
Piazza Poll: can a perceptron learn an **XOR** function?

Expressiveness of Perceptron



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

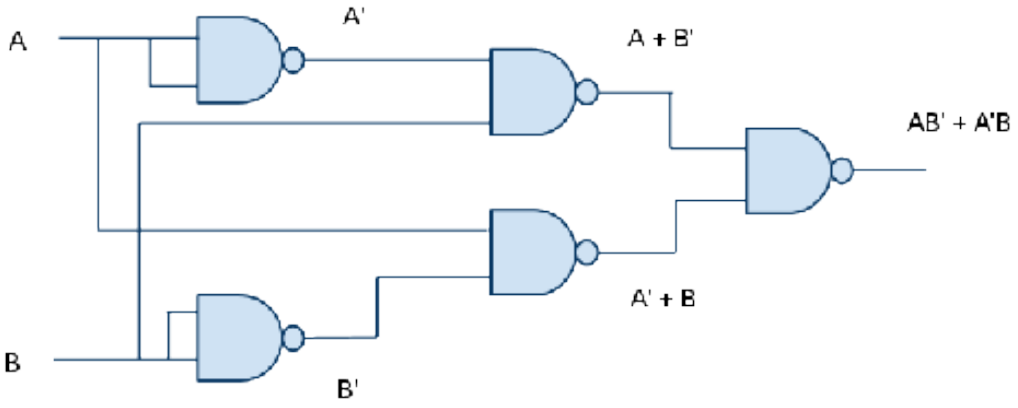
$$\begin{cases} yes & y \geq threshold \\ no & y < threshold \end{cases}$$



A perceptron can represent AND, OR, NOT, etc., but not XOR → linear separator

How to Implement XOR?

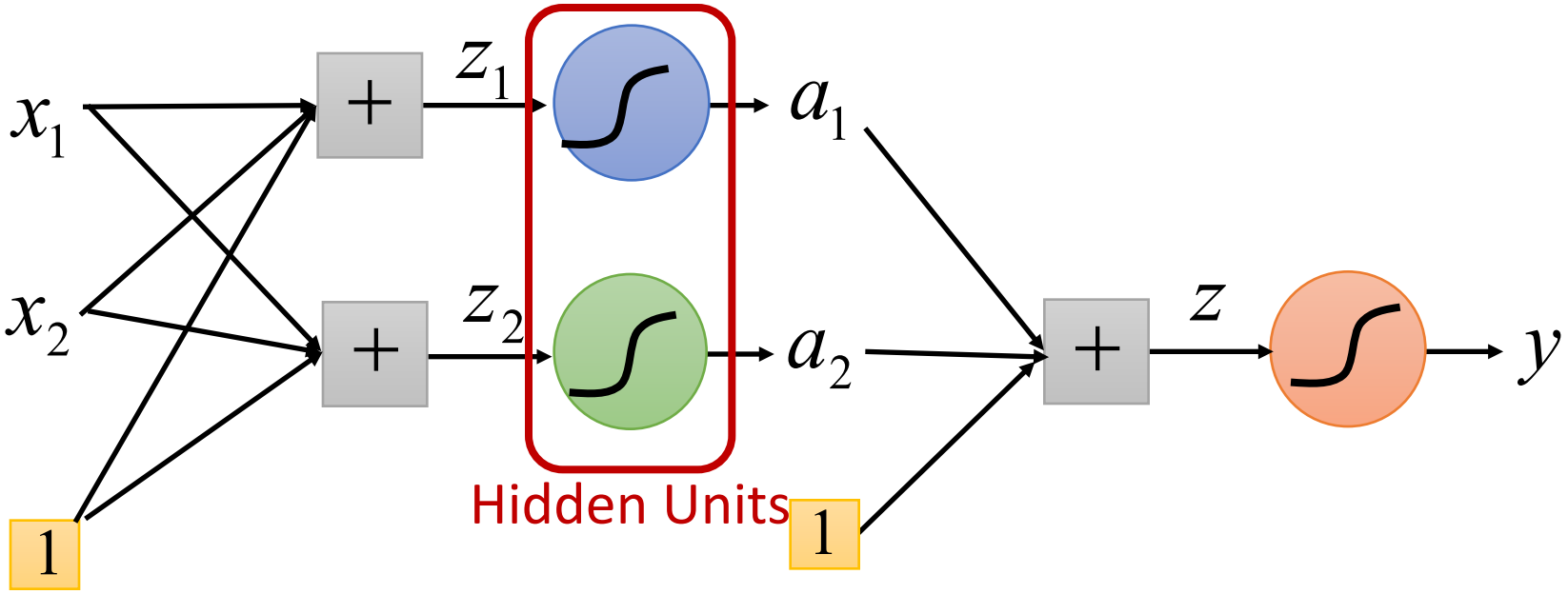
Input		Output
A	B	
0	0	0
0	1	1
1	0	1
1	1	0



A xor B = AB’ + A’B

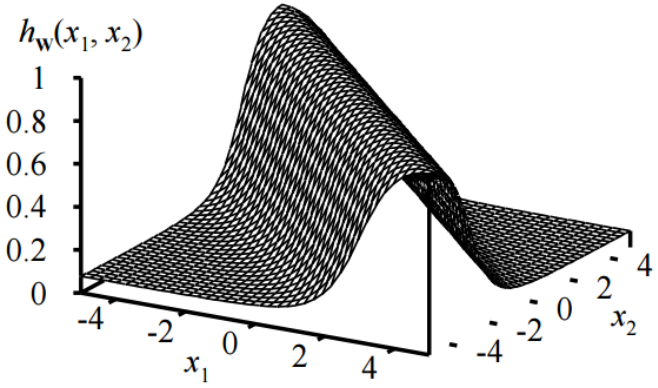
Multiple operations can produce more complicate output

Neural Networks – Multi-Layer Perceptron



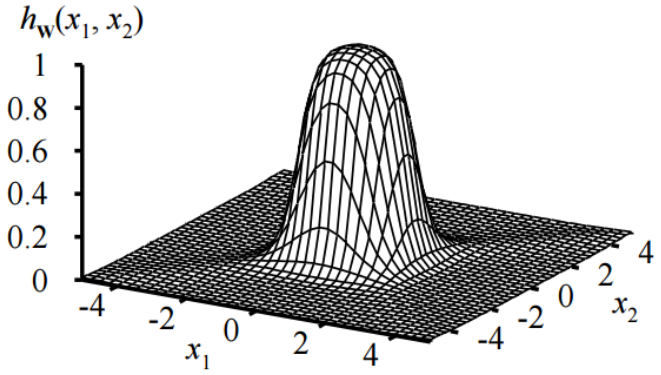
Expressiveness of Multi-Layer Perceptron

Continuous function w/ 2 layers



Combine two opposite-facing threshold functions to make a ridge

Continuous function w/ 3 layers



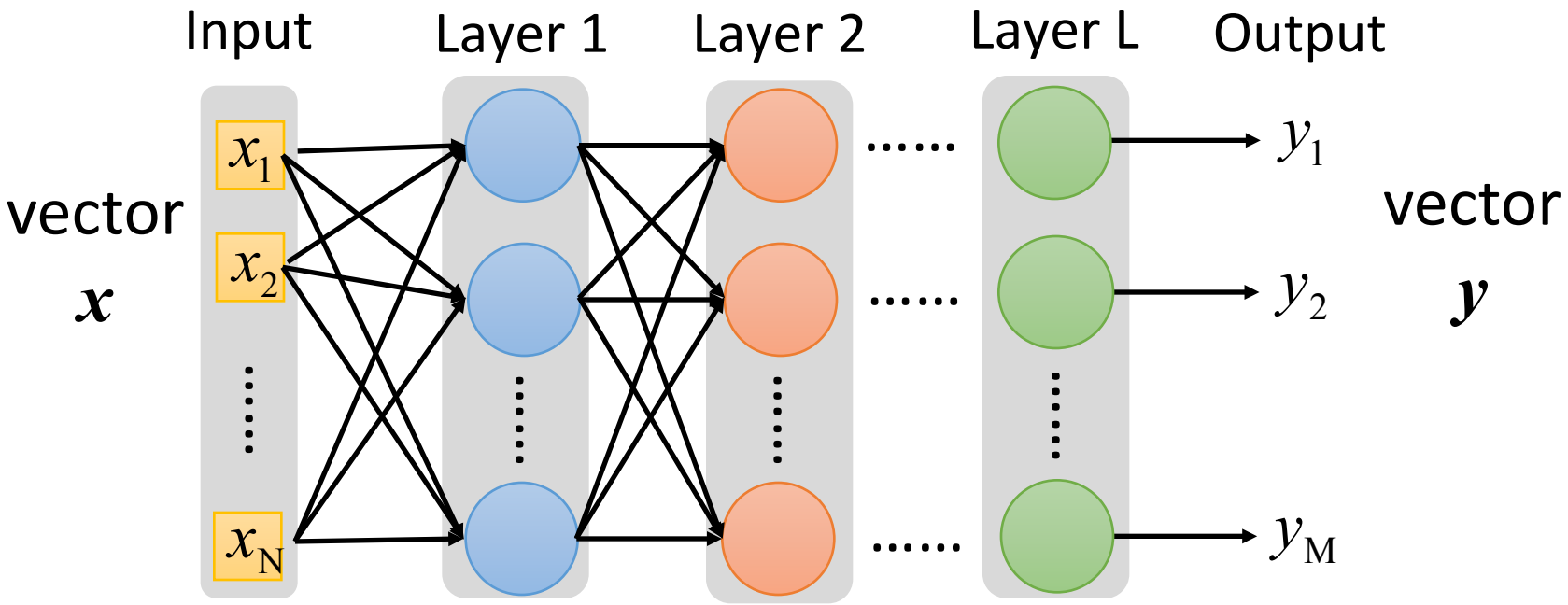
Combine two perpendicular ridges to make a bump

→ Add bumps of various sizes and locations to fit any surface

multiple layers enhance the model expression
→ the model can approximate more complex functions

Deep Neural Networks (DNN) $f : R^N \rightarrow R^M$

Fully connected feedforward network



Deep NN: multiple hidden layers

Learning as optimization

Main idea: gradient-based learning.

Given an objective function, we use optimization techniques to find parameters such that they maximize the function value.

Problem Statement

Given a loss function and several model parameter sets

- Loss function: $C(\theta)$
- Model parameter sets: $\{\theta_1, \theta_2, \cdots\}$

Find a model parameter set that minimizes $C(\theta)$

How to solve this optimization problem?

1) Brute force – enumerate all possible θ

2) Calculus – $\dfrac{\partial C(\theta)}{\partial \theta} = 0$

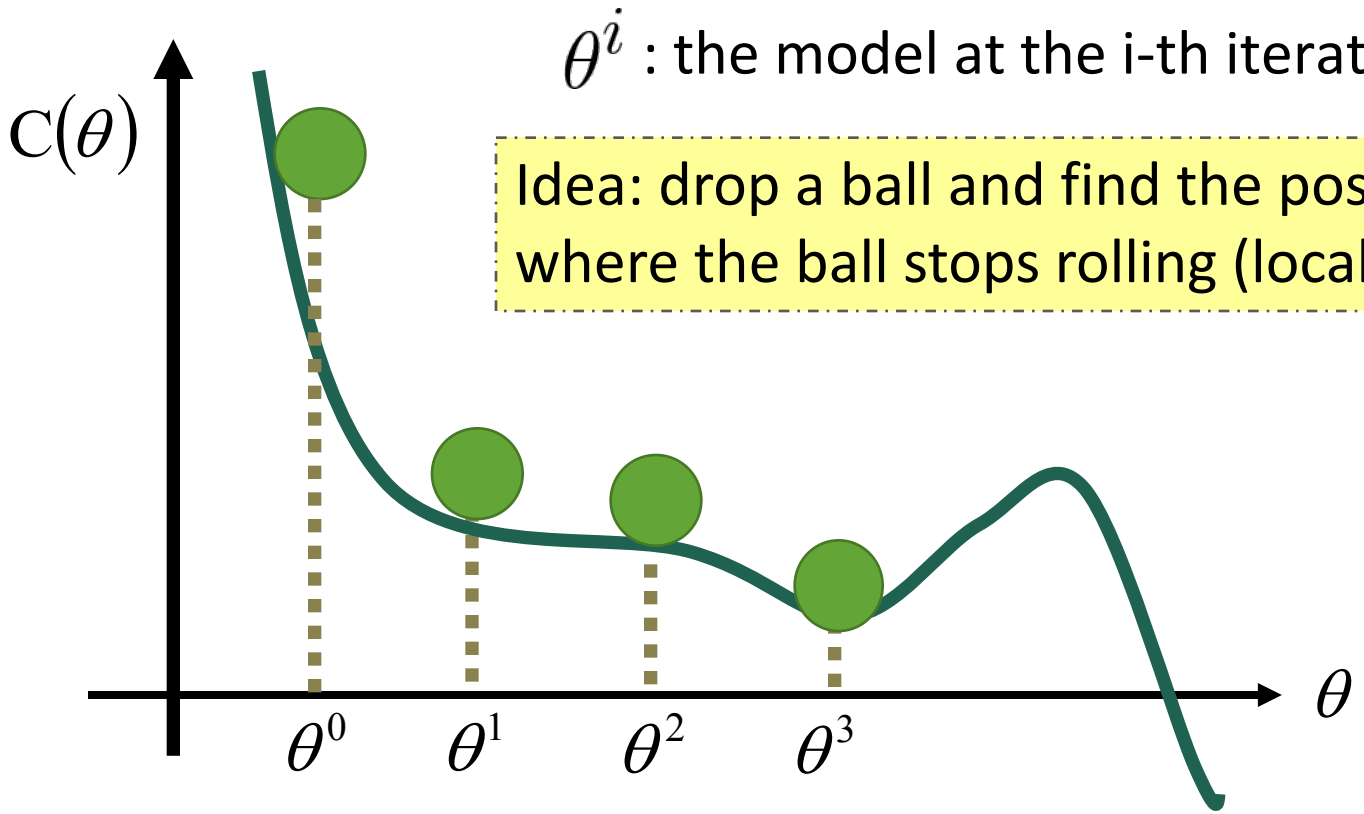
Issue: whole space of $C(\theta)$ is unknown

Gradient Descent for Optimization

Assume that θ has only one variable

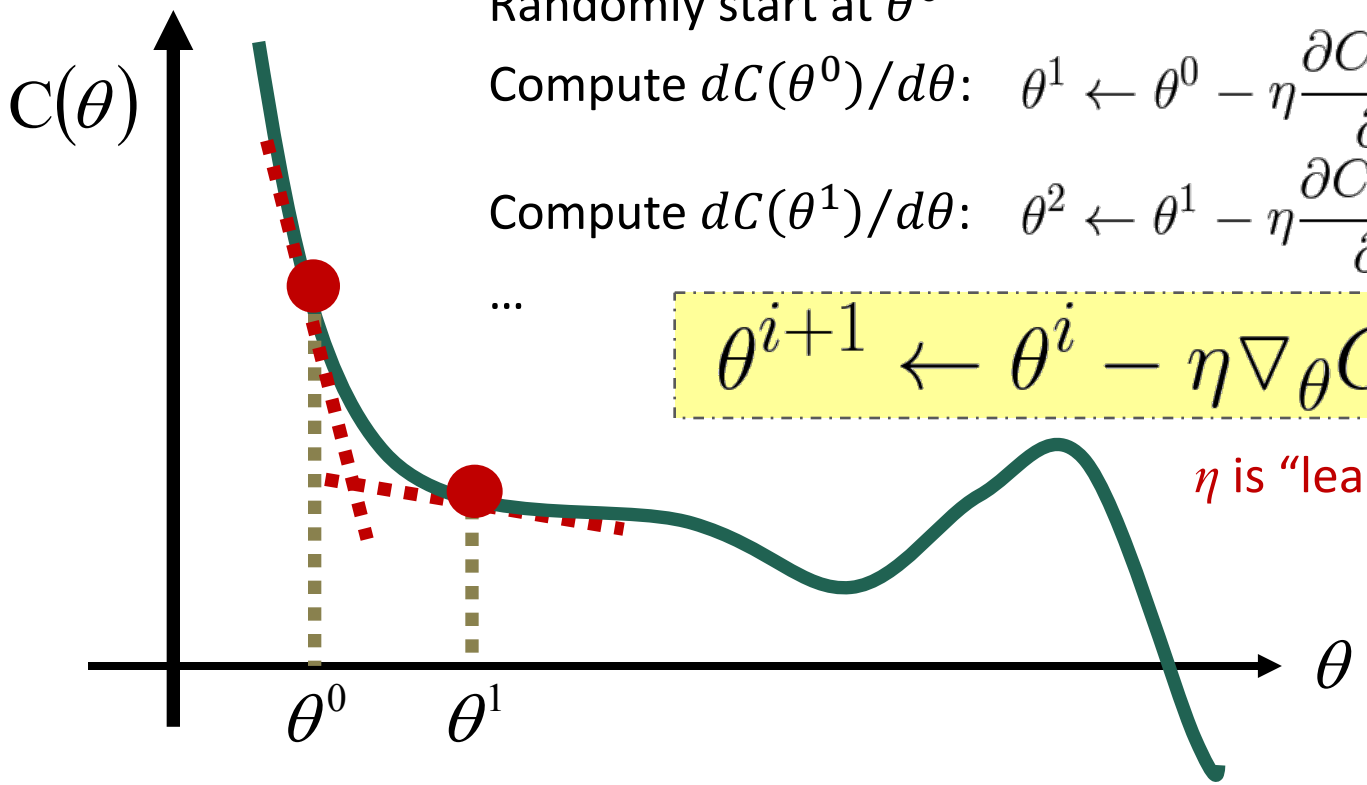
θ^i : the model at the i-th iteration

Idea: drop a ball and find the position where the ball stops rolling (local minima)



Gradient Descent for Optimization

Assume that θ has only one variable



Randomly start at θ^0

Compute $dC(\theta^0)/d\theta$: $\theta^1 \leftarrow \theta^0 - \eta \frac{\partial C(\theta^0)}{\partial \theta}$

Compute $dC(\theta^1)/d\theta$: $\theta^2 \leftarrow \theta^1 - \eta \frac{\partial C(\theta^1)}{\partial \theta}$

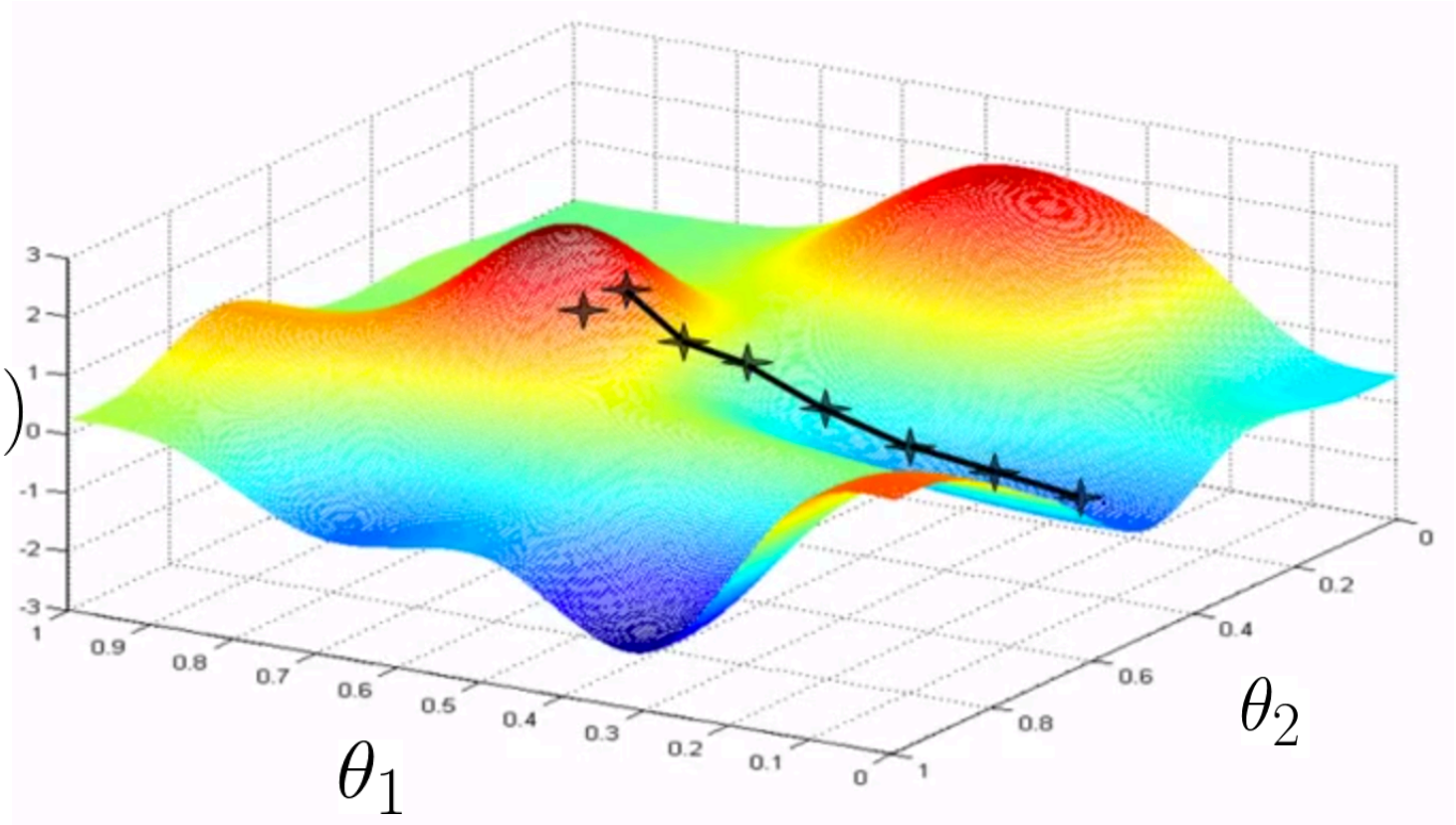
…

$$\theta^{i+1} \leftarrow \theta^i - \eta \nabla_\theta C(\theta^i)$$

η is "learning rate"

Gradient Descent for Optimization

Assume that θ has two variables $\{\theta_1, \theta_2\}$



Gradient Descent for Optimization

Assume that θ has two variables $\{\theta_1, \theta_2\}$

- Randomly start at θ^0: $\theta^0 = \begin{bmatrix} \theta_1^0 \\ \theta_2^0 \end{bmatrix}$
- Compute the gradients of $C(\theta)$ at θ^0: $\nabla_\theta C(\theta^0) = \begin{bmatrix} \frac{\partial C(\theta_1^0)}{\partial \theta_1} \\ \frac{\partial C(\theta_2^0)}{\partial \theta_2} \end{bmatrix}$
- Update parameters:

$$\begin{bmatrix} \theta_1^1 \\ \theta_2^1 \end{bmatrix} = \begin{bmatrix} \theta_1^0 \\ \theta_2^0 \end{bmatrix} - \eta \begin{bmatrix} \frac{\partial C(\theta_1^0)}{\partial \theta_1} \\ \frac{\partial C(\theta_2^0)}{\partial \theta_2} \end{bmatrix}$$

$$\theta^{i+1} \leftarrow \theta^i - \eta \nabla_\theta C(\theta^i)$$

- Compute the gradients of $C(\theta)$ at θ^1: $\nabla_\theta C(\theta^1) = \begin{bmatrix} \frac{\partial C(\theta_1^1)}{\partial \theta_1} \\ \frac{\partial C(\theta_2^1)}{\partial \theta_2} \end{bmatrix}$

Stochastic Gradient Descent (SGD)

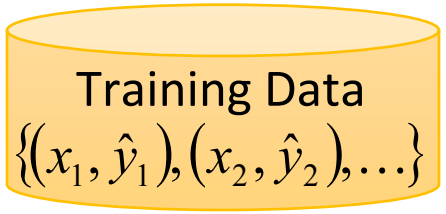
Gradient Descent

$$\theta^{i+1} = \theta^i - \eta \nabla C(\theta^i) \quad \nabla C(\theta^i) = \frac{1}{K} \sum_k \nabla C_k(\theta^i)$$

Stochastic Gradient Descent (SGD)

- Pick a training sample x_k

$$\theta^{i+1} = \theta^i - \eta \nabla C_k(\theta^i)$$



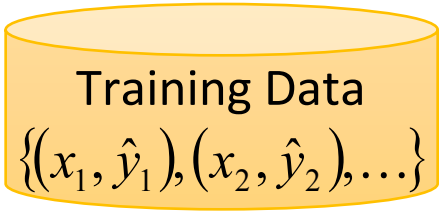
- If all training samples have same probability to be picked

$$E[\nabla C_k(\theta^i)] = \frac{1}{K} \sum_k \nabla C_k(\theta^i)$$

The model can be updated after seeing one training sample → faster

Epoch Definition

When running SGD, the model starts θ^0



pick x_1 $\theta^1 = \theta^0 - \eta \nabla C_1(\theta^0)$

pick x_2 $\theta^2 = \theta^1 - \eta \nabla C_2(\theta^1)$

⋮ ⋮

pick x_k $\theta^k = \theta^{k-1} - \eta \nabla C_k(\theta^{k-1})$

⋮ ⋮

pick x_K $\theta^K = \theta^{K-1} - \eta \nabla C_K(\theta^{K-1})$

see all training samples once

→ one epoch

---

pick x_1 $\theta^{K+1} = \theta^K - \eta \nabla C_1(\theta^K)$

Mini-Batch SGD

Gradient Descent

$$\theta^{i+1} = \theta^i - \eta \frac{1}{K} \sum_k \nabla C_k(\theta^i)$$

Use all K samples in each iteration

Stochastic Gradient Descent (SGD)

- Pick a training sample x_k

$$\theta^{i+1} = \theta^i - \eta \nabla C_k(\theta^i)$$

Use 1 samples in each iteration

Mini-Batch SGD

- Pick a set of B training samples as a batch b

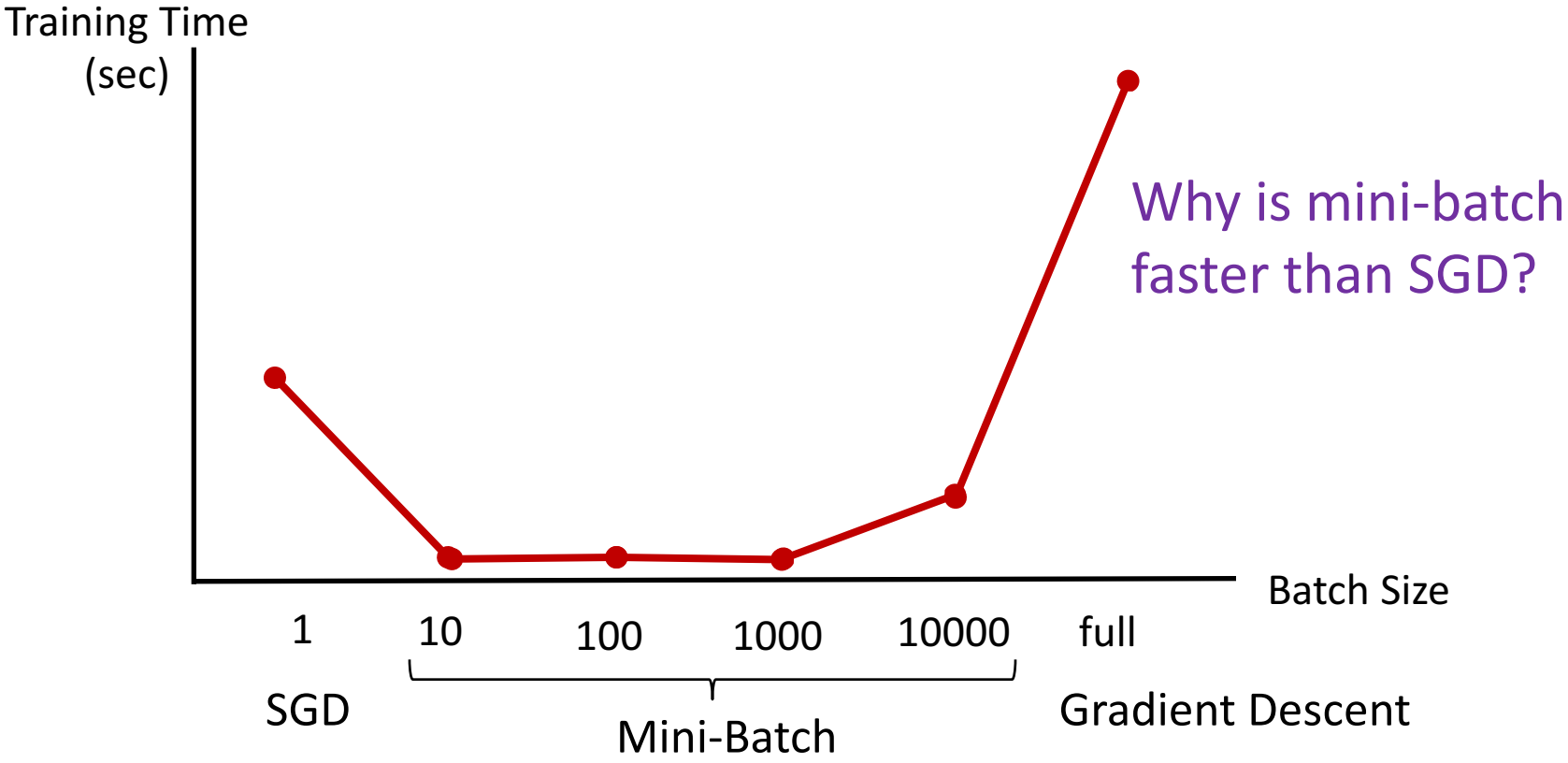
B is "batch size"

$$\theta^{i+1} = \theta^i - \eta \frac{1}{B} \sum_{x_k \in b} \nabla C_k(\theta^i)$$

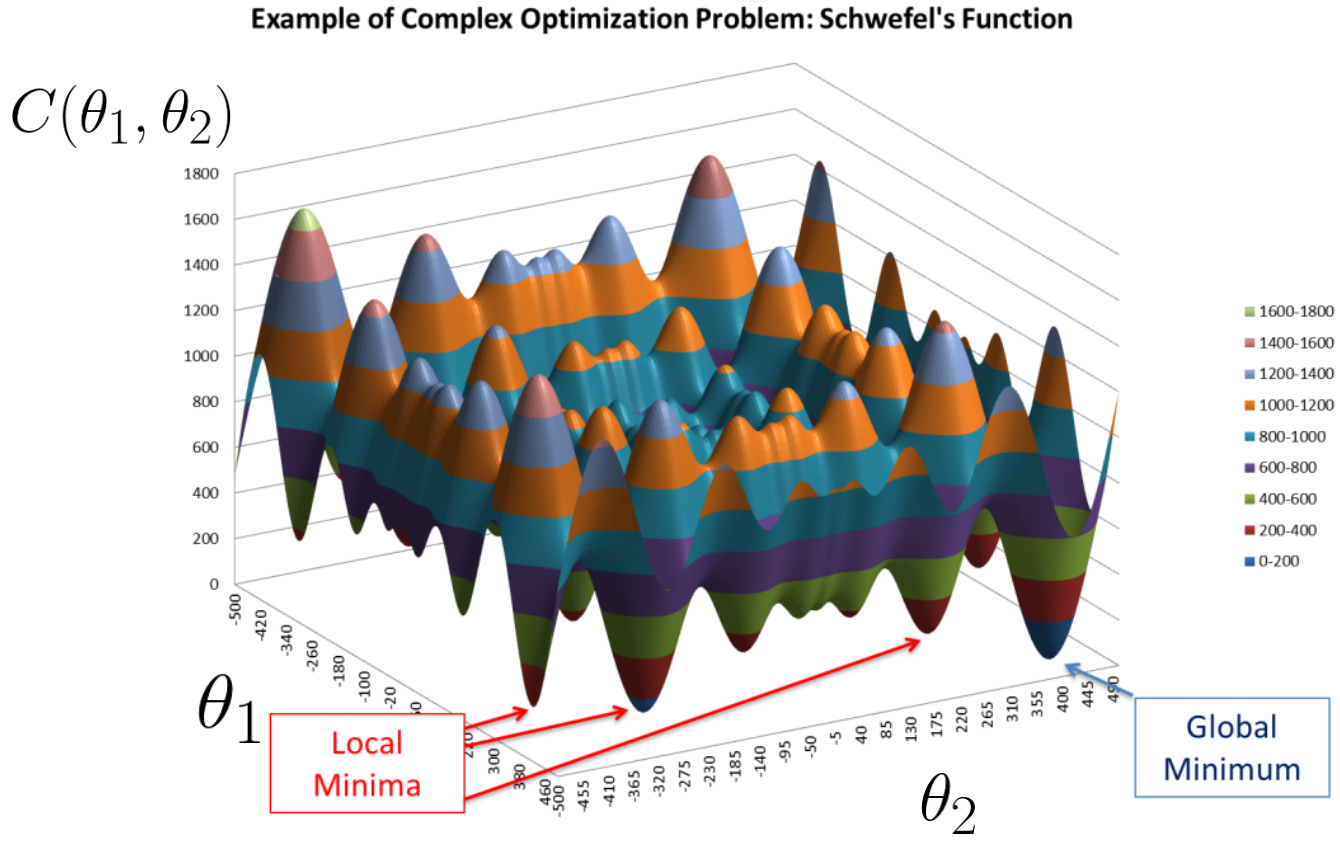
Use all B samples in each iteration

Gradient Descent v.s. SGD v.s. Mini-Batch

Training speed: mini-batch > SGD > Gradient Descent



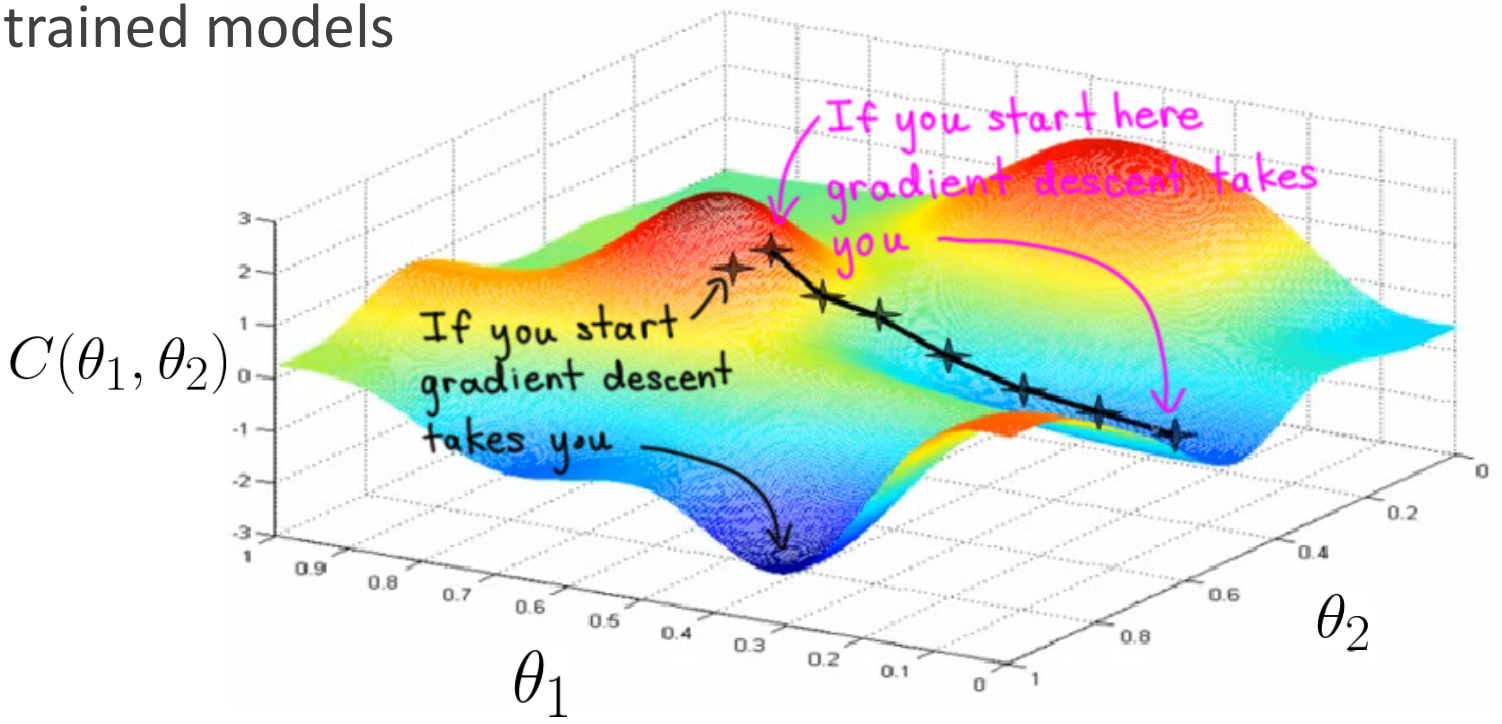
Issue: Local Optima / Saddle Points



Neural networks has no guarantee for obtaining global optimal solution

Initialization

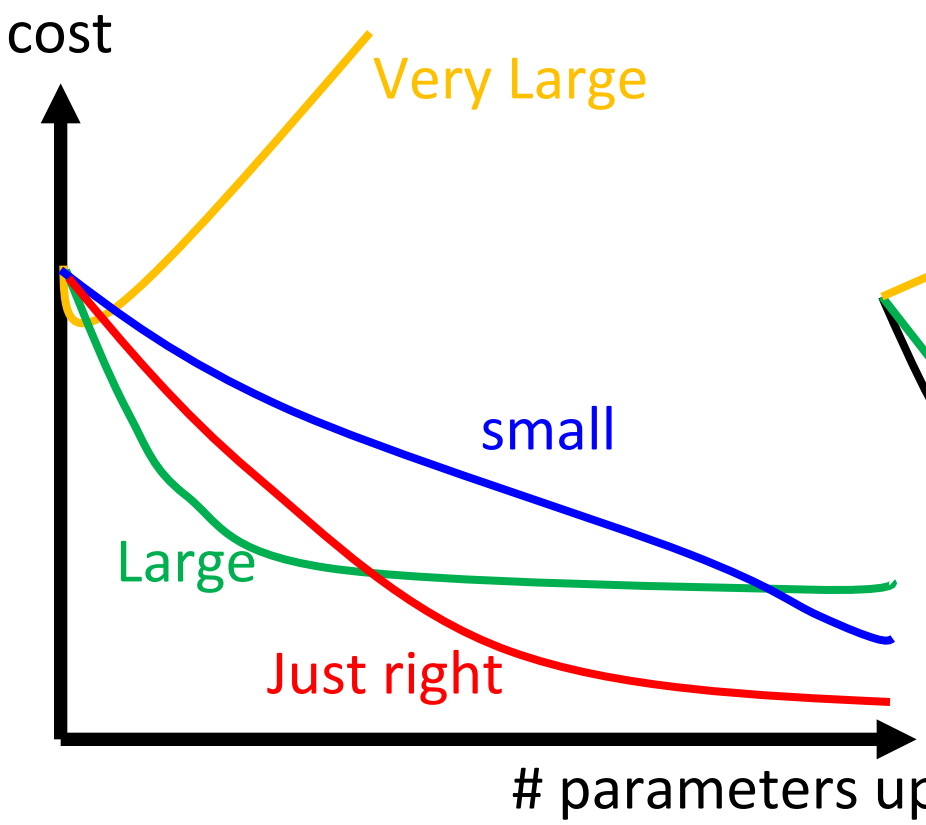
Different initialization parameters may result in different trained models

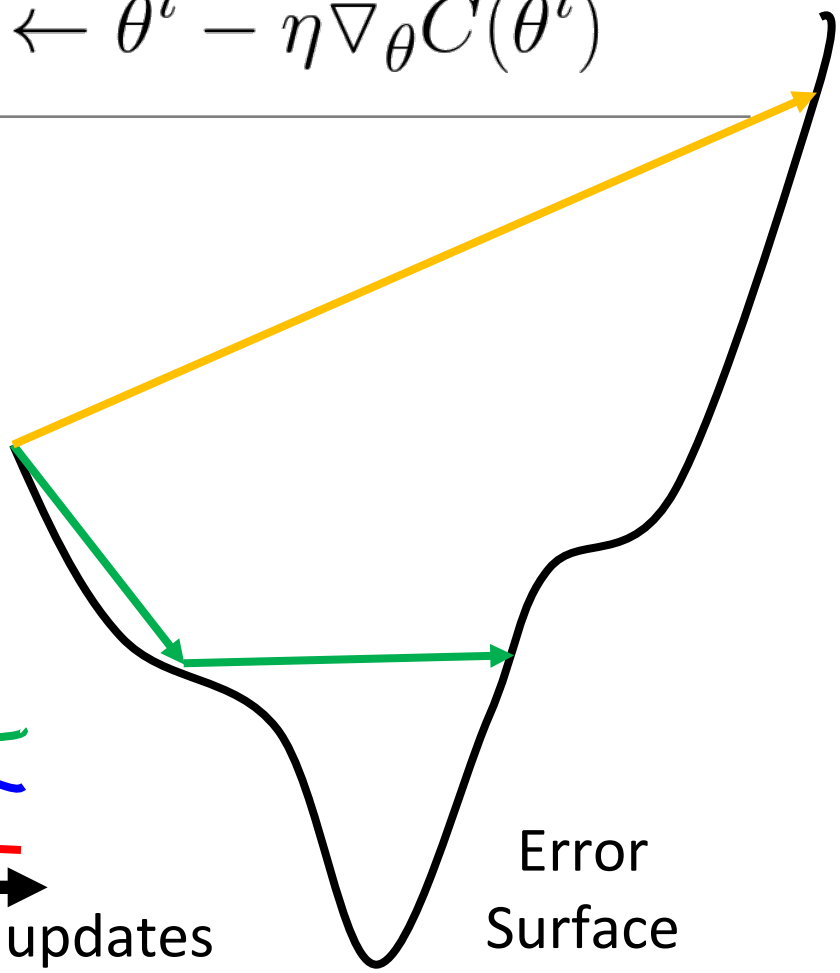


Do not initialize the parameters equally → set them randomly

Learning Rate

$$\theta^{i+1} \leftarrow \theta^{i} - \eta \nabla_{\theta} C(\theta^{i})$$





Learning rate should be set carefully

Overfitting

Possible solutions

- more training samples
- some tips: dropout, etc.

How to train a deep neural network?

Main idea: gradient-based learning.

Challenges:

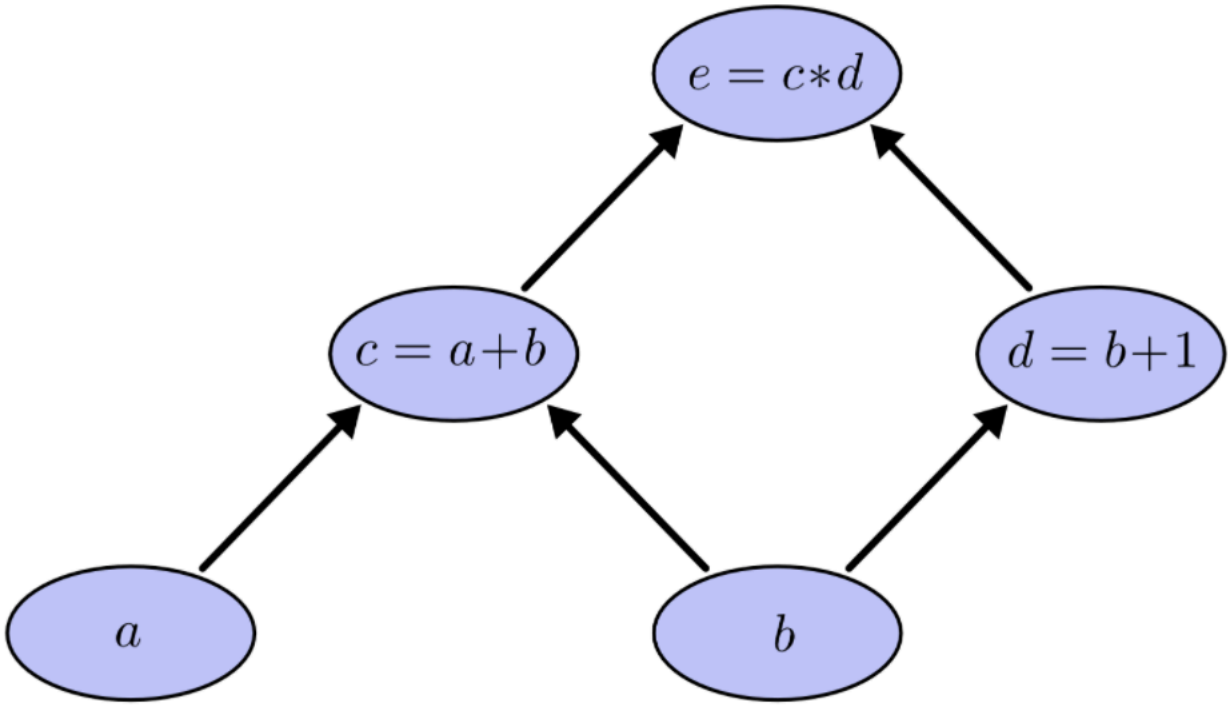
- There are too many parameters.
- There are strong dependencies among parameters in different layers.

Solution: backpropagation.

Computational Graphs

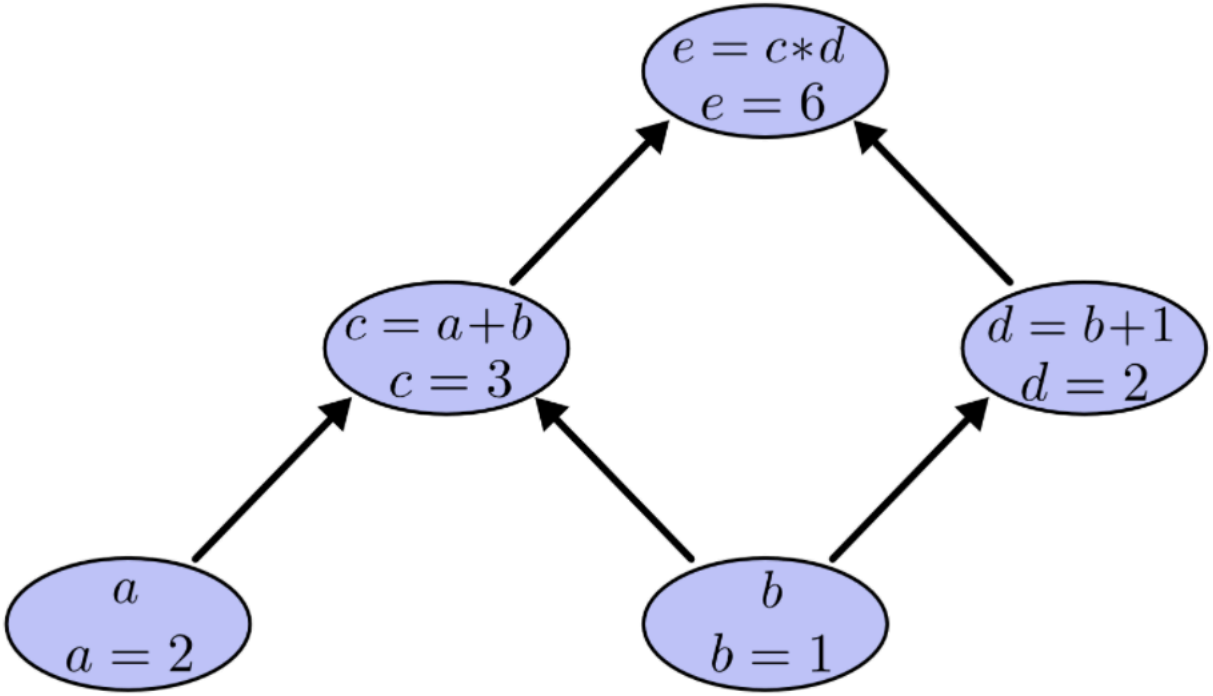
e=(a+b)*(b+1)

- c=a+b
- d=b+1
- e=c*d



Computational Graphs

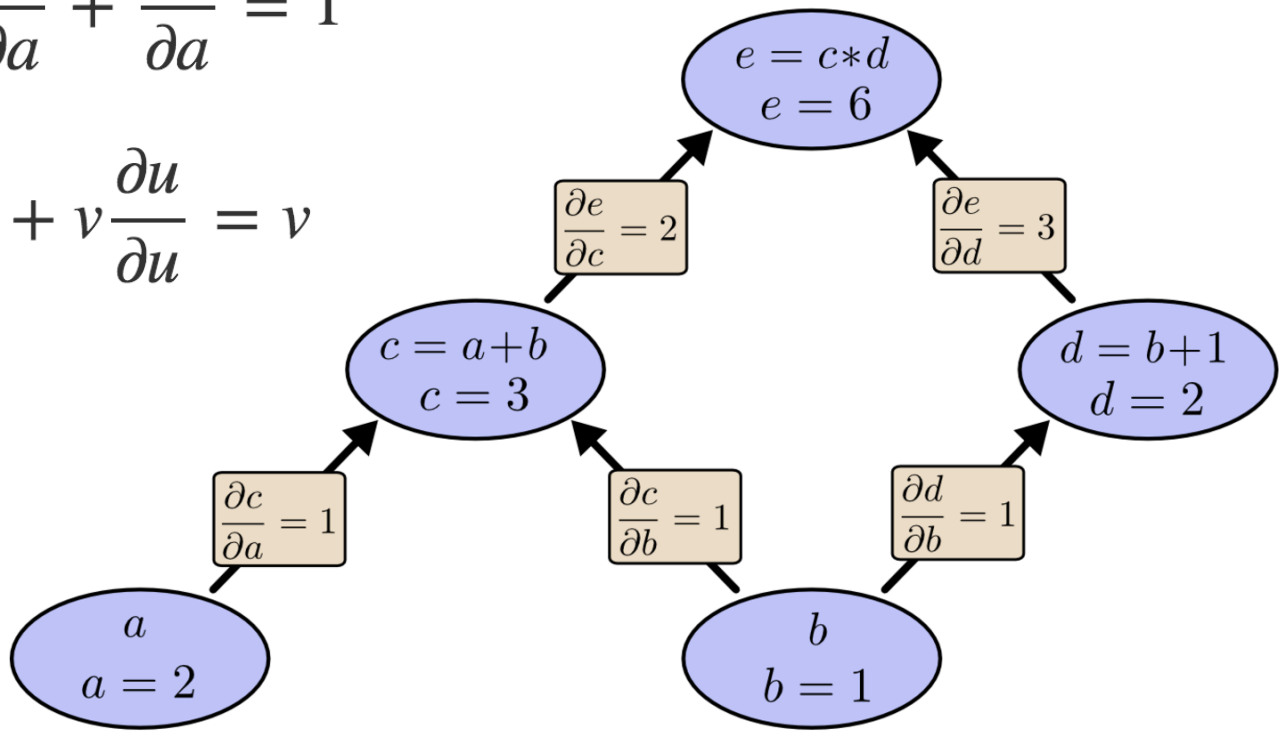
If we let a = 2 and b = 1



Sum and Produce Rules: Derivatives on Computational Graphs

$$\frac{\partial}{\partial a}(a+b) = \frac{\partial a}{\partial a} + \frac{\partial b}{\partial a} = 1$$

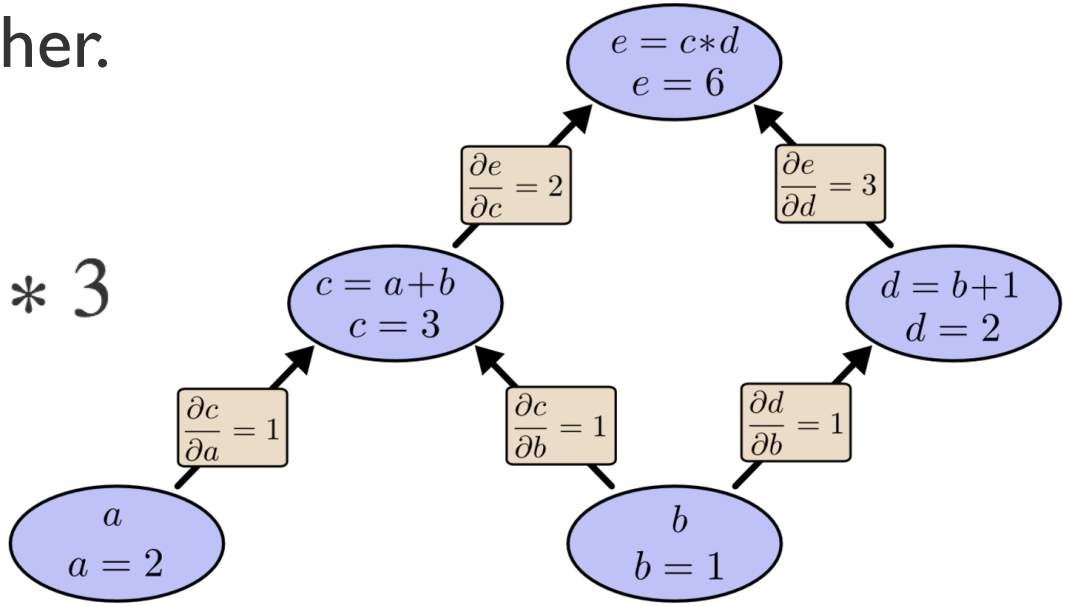
$$\frac{\partial}{\partial u}uv = u\frac{\partial v}{\partial u} + v\frac{\partial u}{\partial u} = v$$



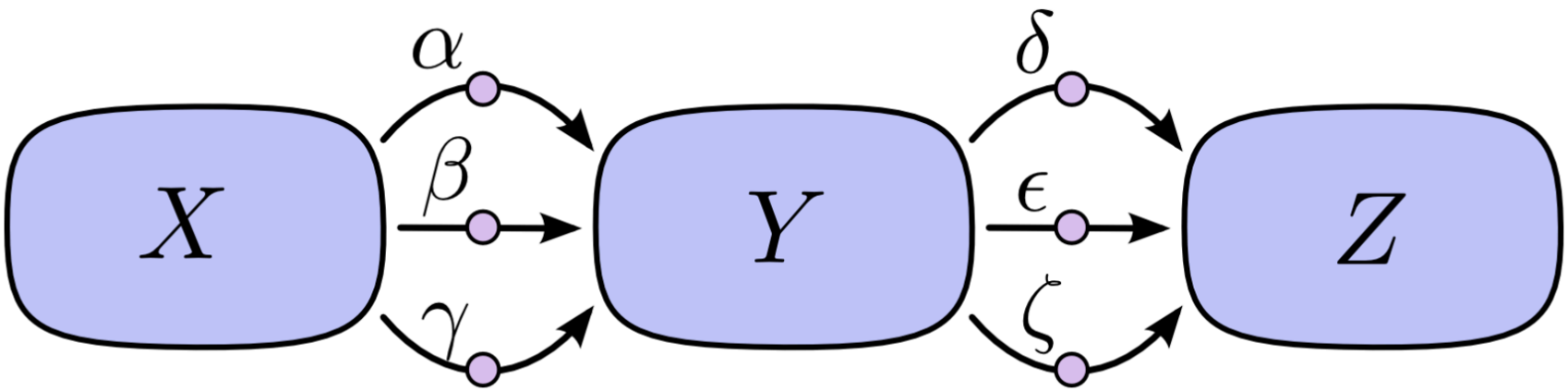
General Rule:
Derivatives on Computational Graphs

1) sum over all possible paths from one node to the other
2) multiplying the derivatives on each edge of the path together.

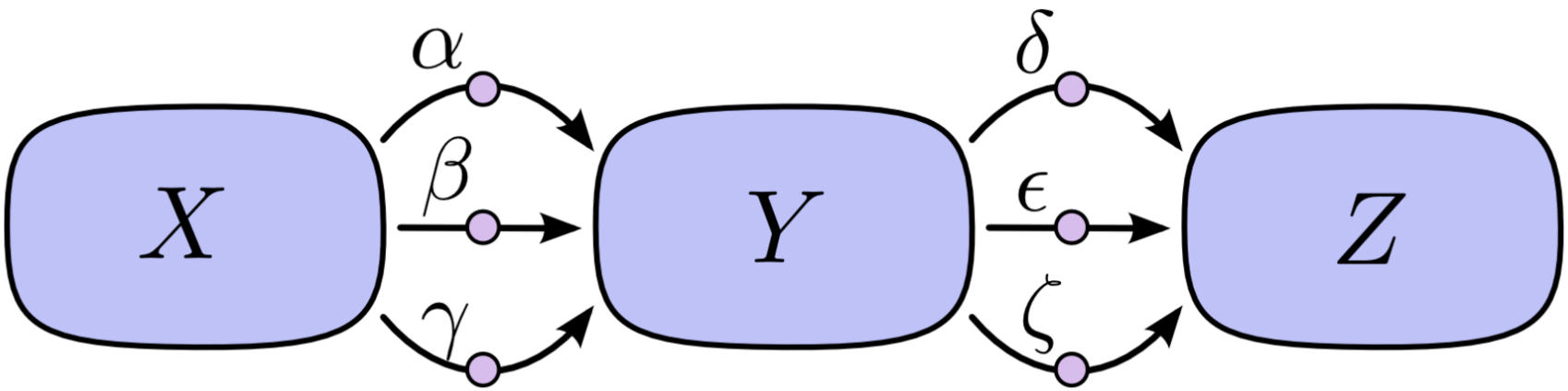
$$\frac{\partial e}{\partial b} = 1 * 2 + 1 * 3$$



What is $\frac{\partial Z}{\partial X}$?

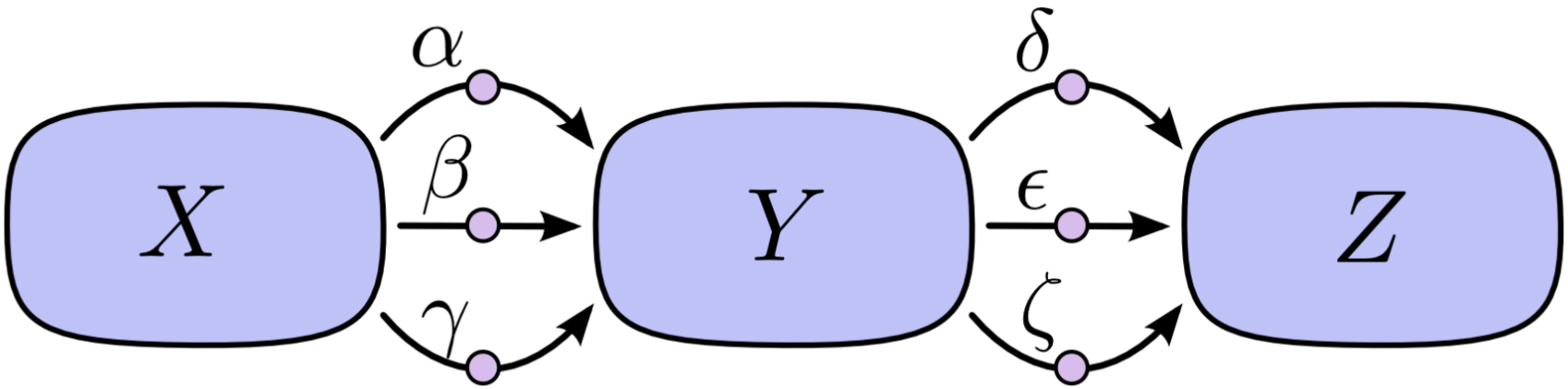


Summing over all paths



$$\frac{\partial Z}{\partial X} = \alpha\delta + \alpha\epsilon + \alpha\zeta + \beta\delta + \beta\epsilon + \beta\zeta + \gamma\delta + \gamma\epsilon + \gamma\zeta$$

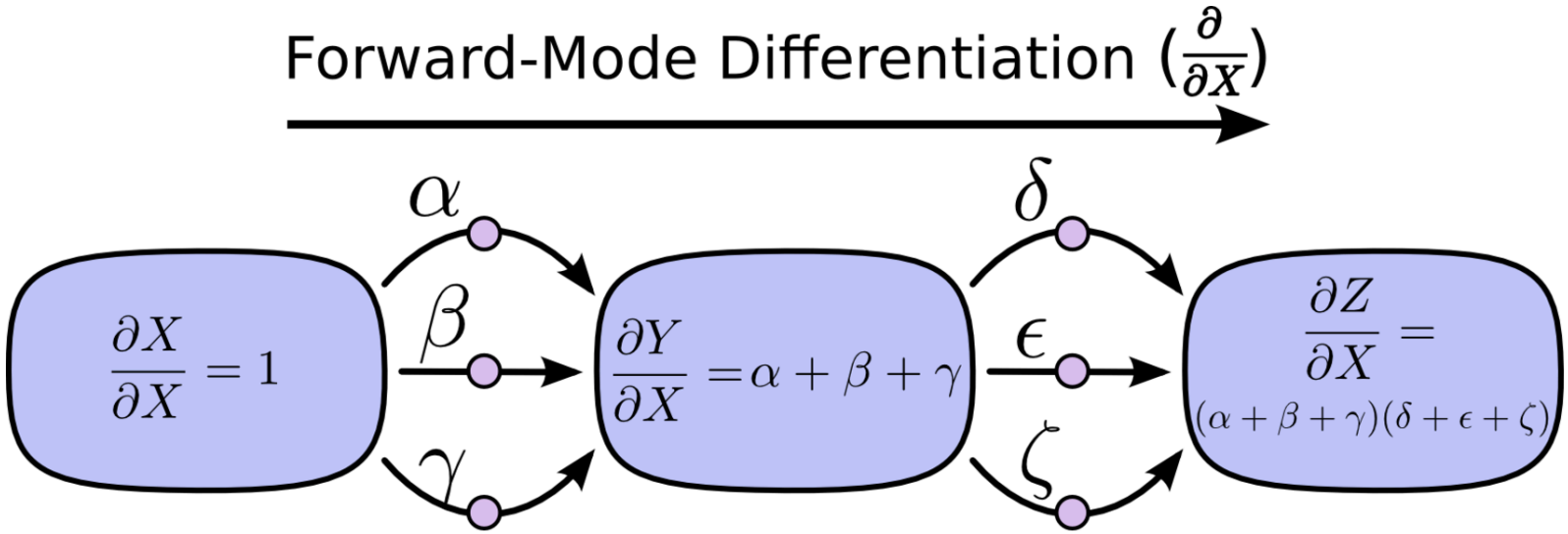
Efficient solution: forward differentiation



$$\frac{\partial Z}{\partial X} = \alpha\delta + \alpha\epsilon + \alpha\zeta + \beta\delta + \beta\epsilon + \beta\zeta + \gamma\delta + \gamma\epsilon + \gamma\zeta$$

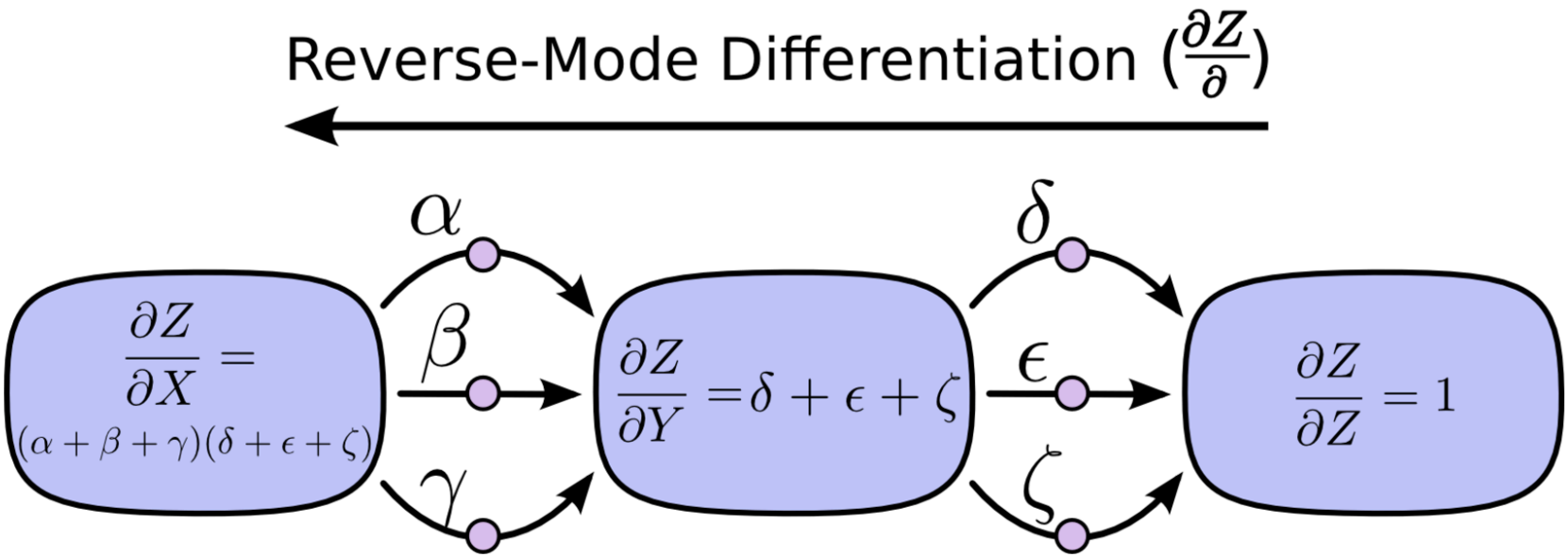
$$\frac{\partial Z}{\partial X} = (\alpha + \beta + \gamma)(\delta + \epsilon + \zeta)$$

Efficient solution: forward differentiation

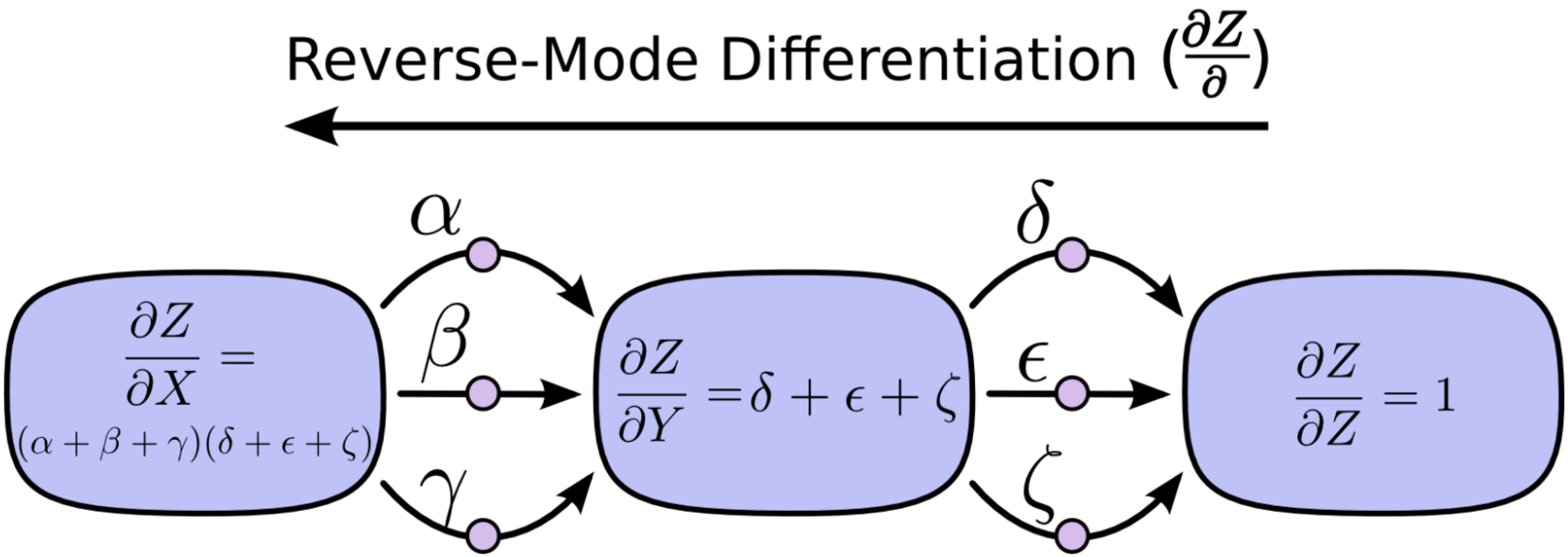


Forward-mode differentiation tracks how one input affects every node.

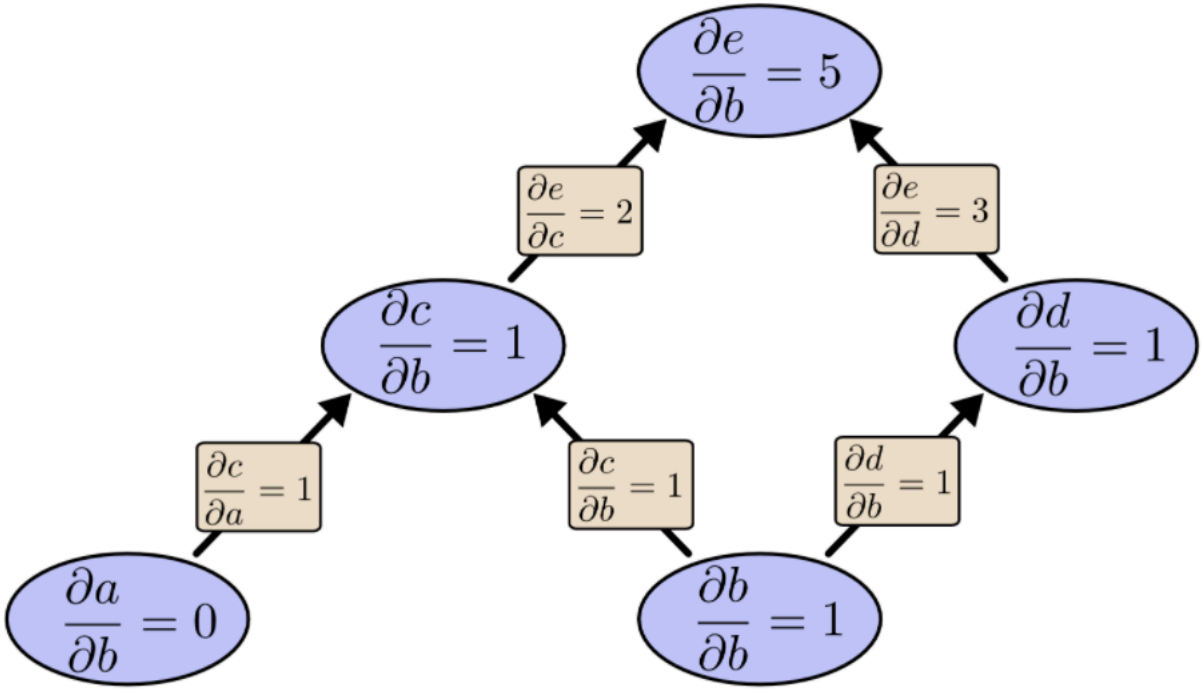
Reverse-mode differentiation



Piazza Poll: Why reverse-mode differentiation?

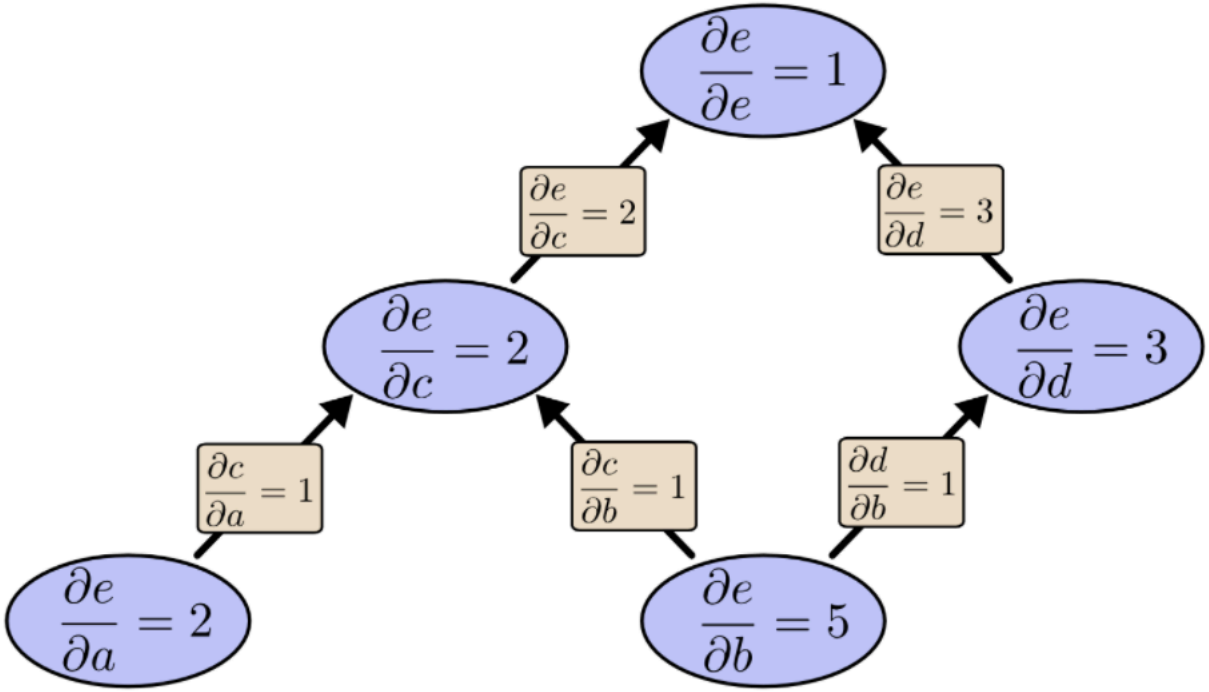


Forward propagation



Forward propagation tells you how to adjust the input.

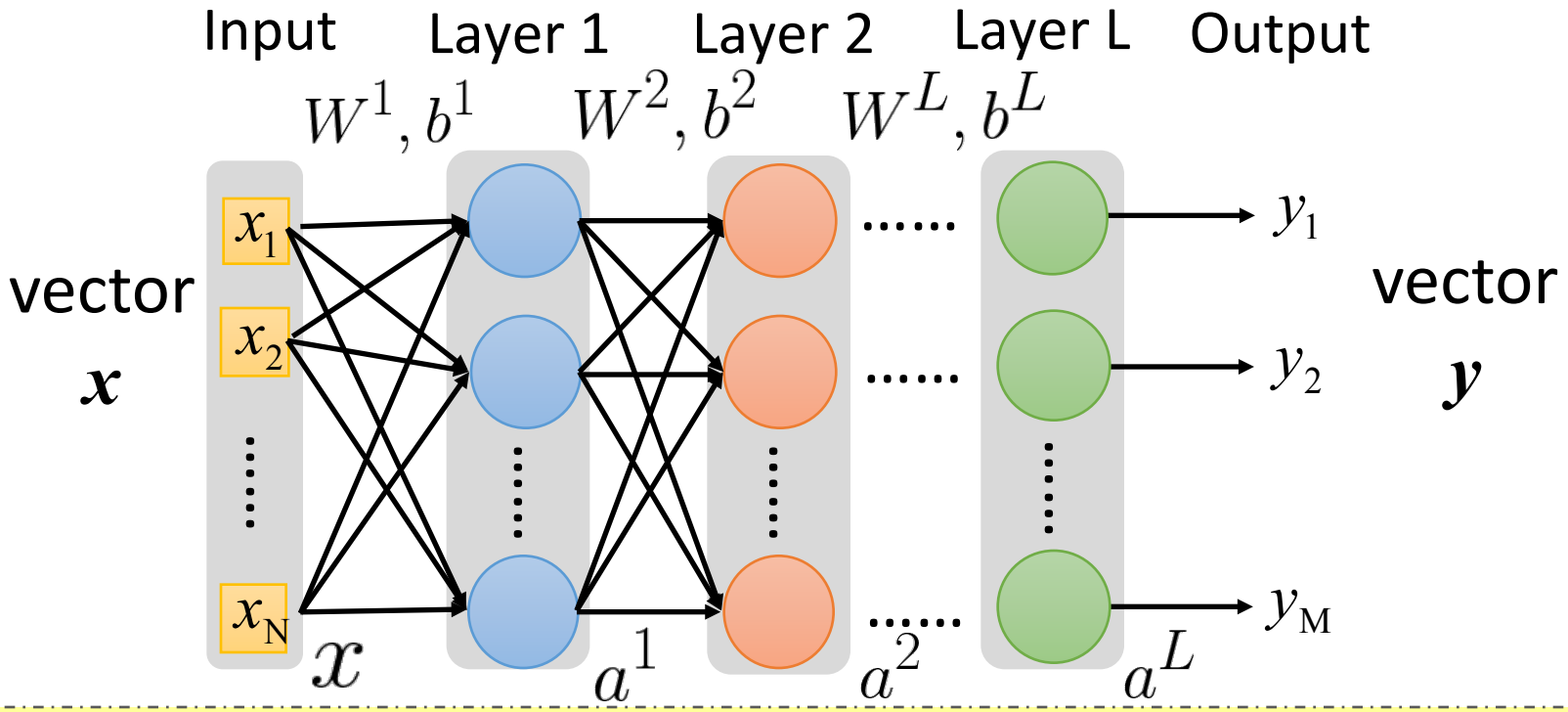
Why backprop?



We need to adjust the weight of each node.

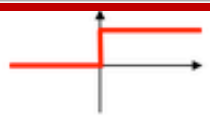
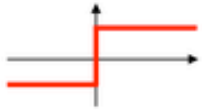
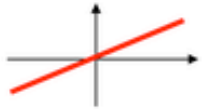
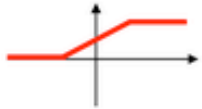
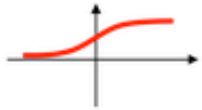
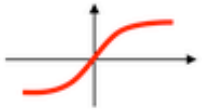
Neural Network Formulation $f : R^N \rightarrow R^M$

Fully connected feedforward network



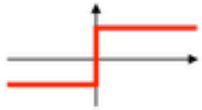
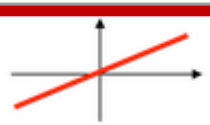
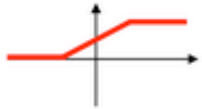
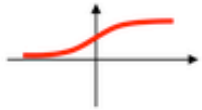
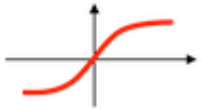
$$y = f(x) = \sigma(W^L \cdots \sigma(W^2 \sigma(W^1 x + b^1) + b^2) \cdots + b^L)$$

Activation Function $\sigma(\cdot)$

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Multi-layer NN	

bounded function

Activation Function $\sigma(\cdot)$

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Multi-layer NN	

boolean

linear

non-linear

Non-Linear Activation Function

Sigmoid

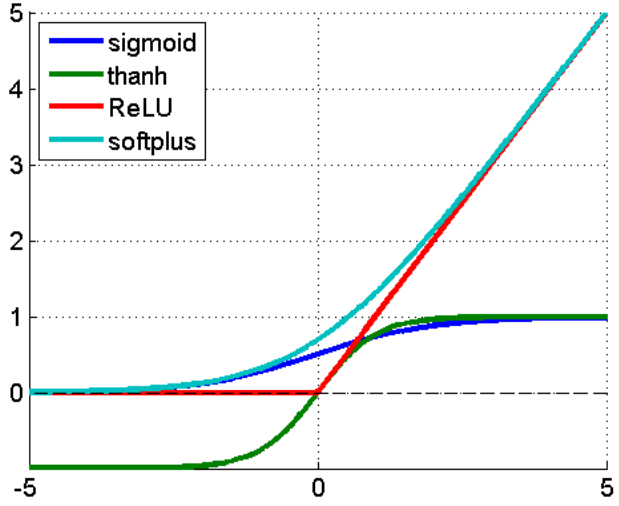
$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}$$

Tanh

$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

Rectified Linear Unit (ReLU)

$$\text{ReLU}(x) = \max(x, 0)$$



Non-linear functions are frequently used in neural net

Why Non-Linearity?

Function approximation

- ***Without non-linearity***, deep neural networks work the same as linear transform

$$W_1(W_2 \cdot x) = (W_1 W_2)x = Wx$$

- ***With non-linearity***, networks with more layers can approximate more complex function