

# CHAPTER 1

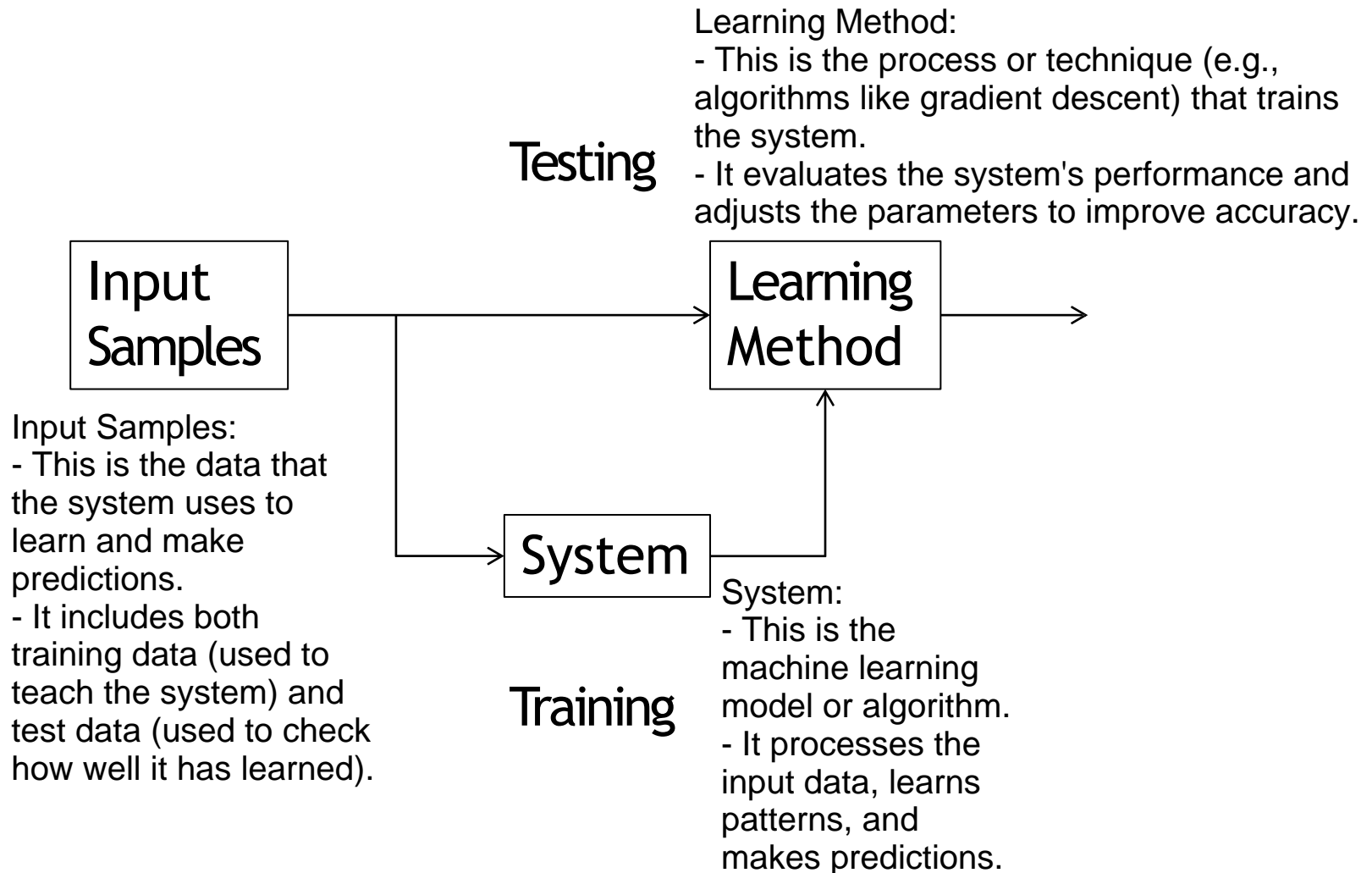
## •INTRODUCTION

### 1.4 Building a Machine Learning Algorithm

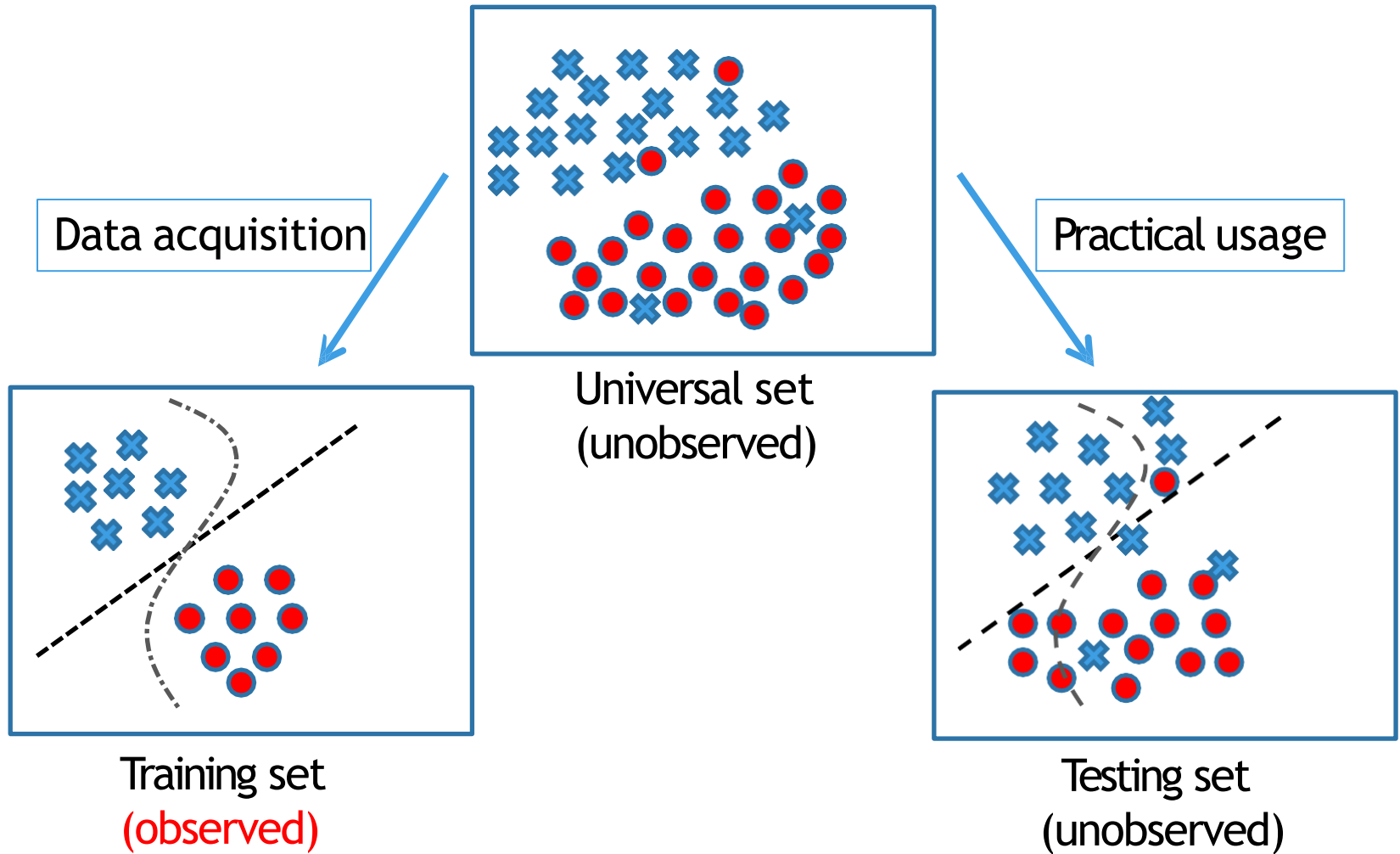
# What is machine learning?

- A branch of **artificial intelligence**, concerned with the design and development of algorithms that allow computers to evolve behaviors based on empirical data.
- As intelligence requires knowledge, it is necessary for the computers to acquire knowledge.

# Learning system model

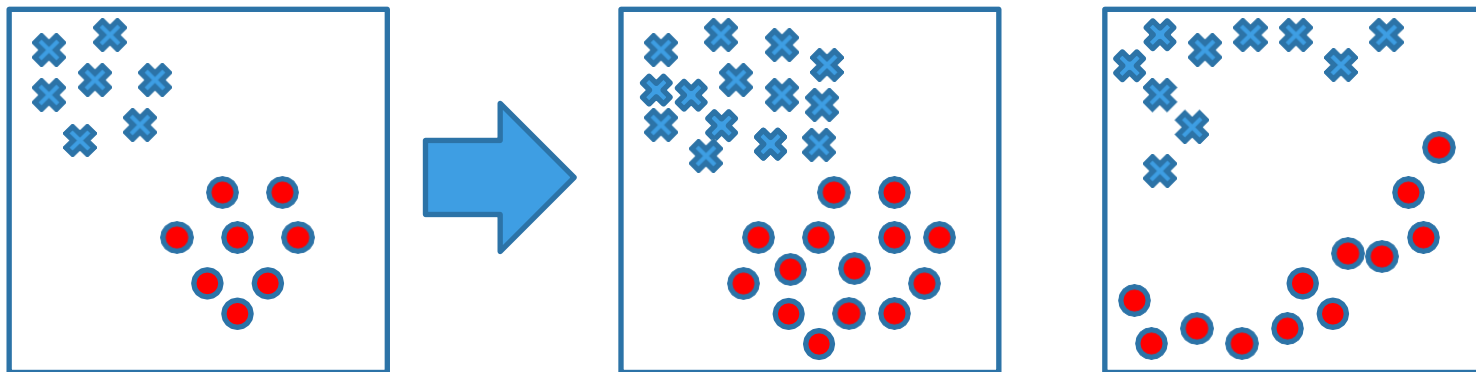


# Training and testing

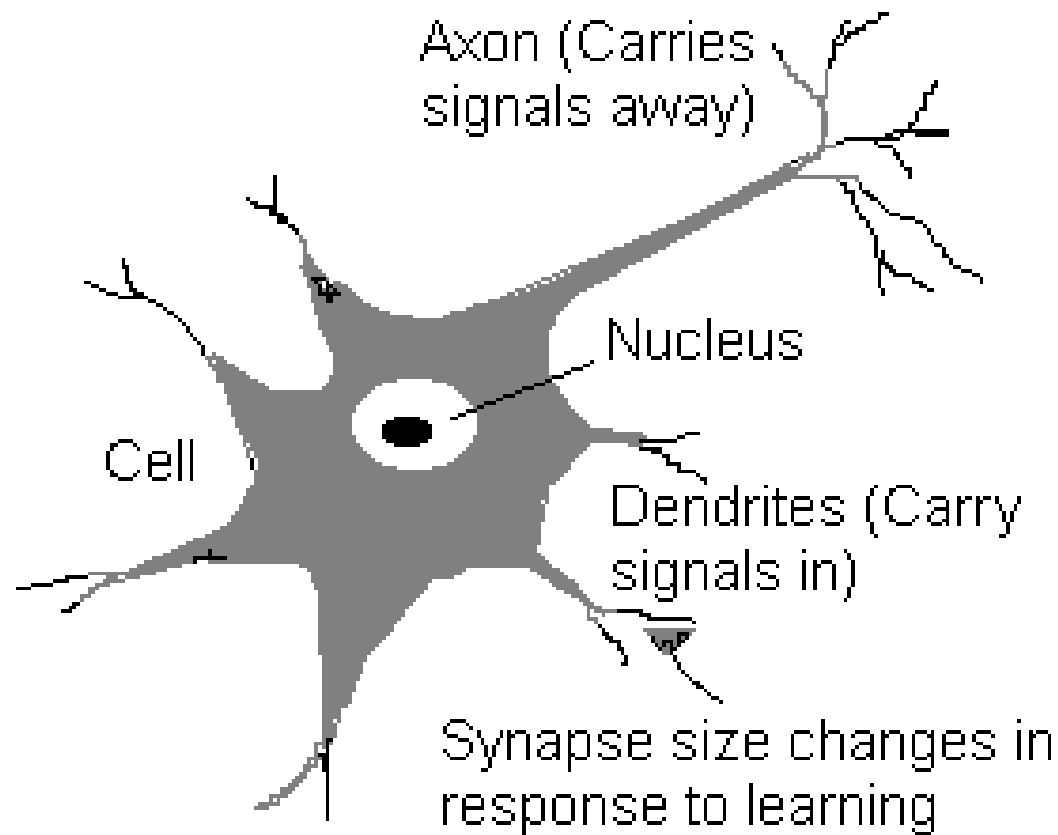


# Training and testing

- ❑ Training is the process of making the system able to learn.
- ❑ No rule:
  - ❑ Training set and testing set come from the same distribution
  - ❑ Need to make some assumptions or bias

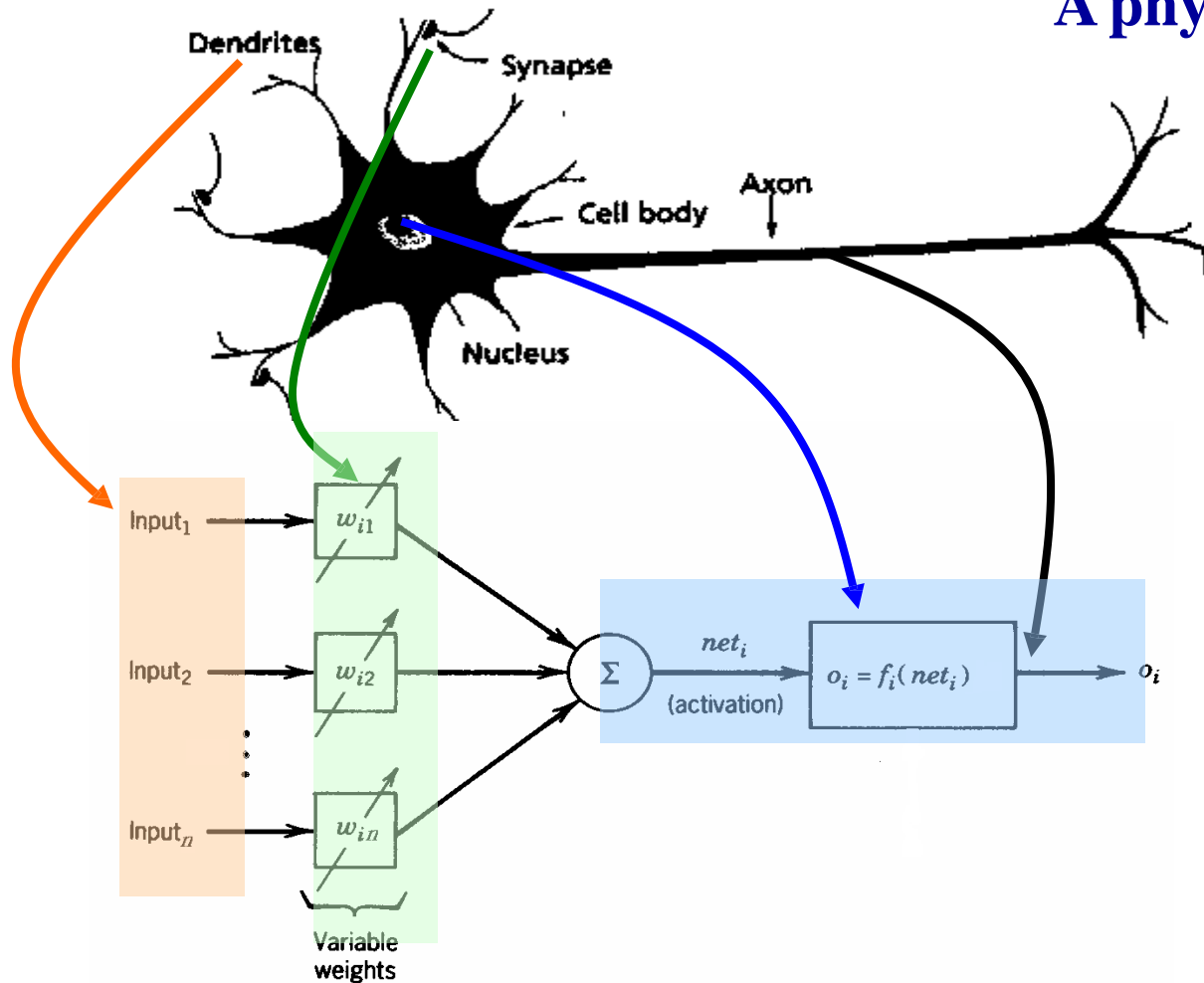


## Biological Neurons



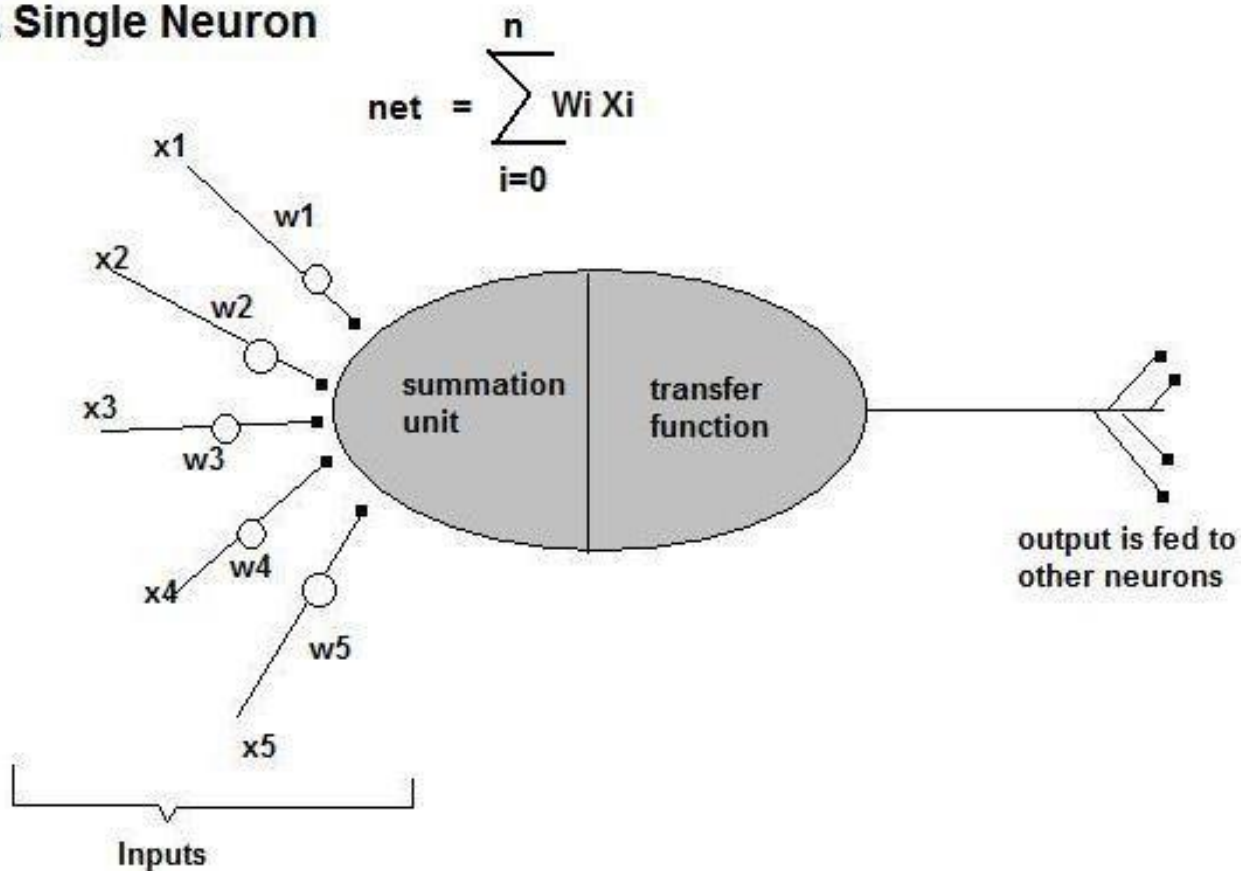
# Artificial Neurons

A physical neuron



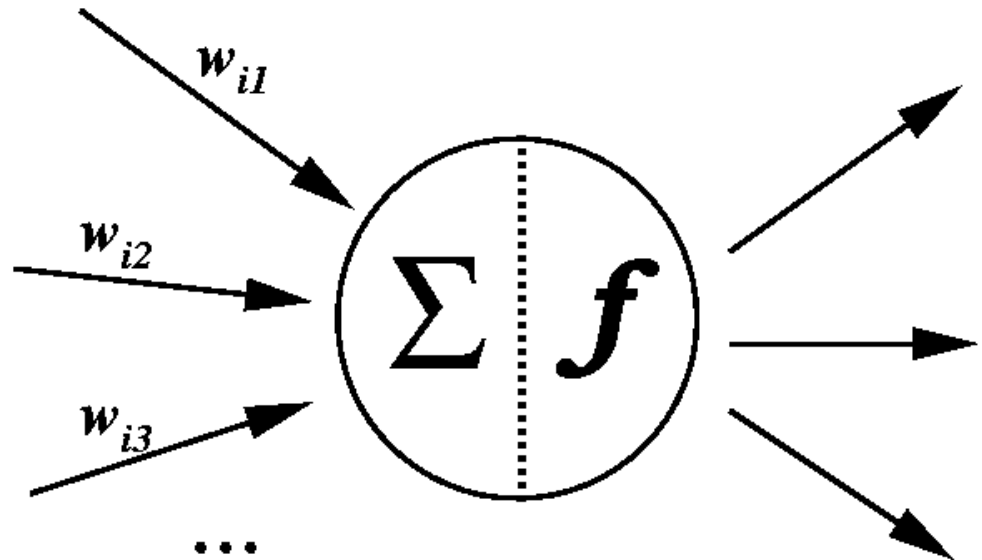
# ARTIFICIAL NEURON MODEL

## A Single Neuron





ARTIFICIAL  
NEURON  
MODEL



$$y_i = f(\text{net}_i)$$

# Basic elements of neural networks are:

- ❑ Input nodes
- ❑ Weights
- ❑ Activation function
- ❑ Total signal reaching at output are given by:

$$\text{Output (y)} = \sum_{i=1}^n W_i X_i$$

- ❑ Output may be more than one it depends on number of neuron i.e. multiple neuron will cause multiple output.

## Types of Learning

### 1.2.2 Supervised and Unsupervised Learning

- **Supervised (inductive) learning**
  - Training data includes desired outputs
- **Unsupervised learning**
  - Training data does not include desired outputs
- **Semi-supervised learning**
  - Training data includes a few desired outputs
- **Reinforcement learning**
  - Rewards from sequence of actions

# Performance

---

- [?] Several factors affecting the performance:
  - [?] Types of training provided
    - [?] The form and extent of any initial background knowledge
  - [?] The type of feedback provided
  - [?] The learning algorithms used

# Algorithms

---

- ❑ The success of machine learning system also depends on the algorithms.
- ❑ The algorithms control the search to find and build the knowledge structures.
- ❑ The learning algorithms should extract useful information from training examples.

# Algorithms

## ☐ Supervised learning

- ☐ Prediction
- ☐ Classification (discrete labels), Regression (real values)

## ☐ Unsupervised learning

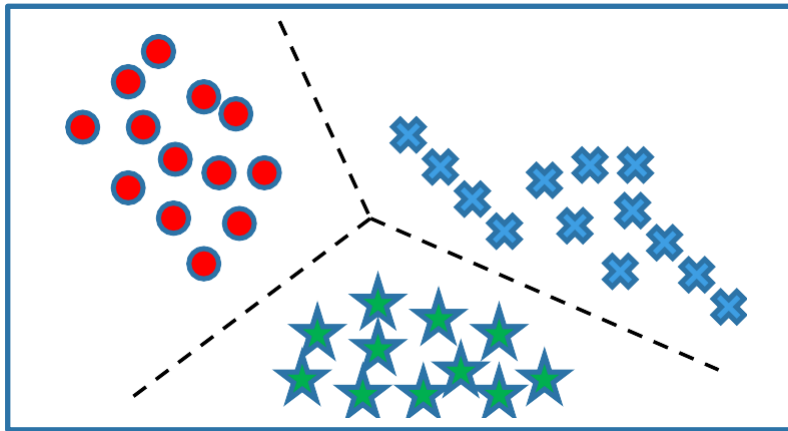
- ☐ Clustering
- ☐ Probability distribution estimation
- ☐ Finding association (in features)
- ☐ Dimension reduction

## ☐ Semi-supervised learning

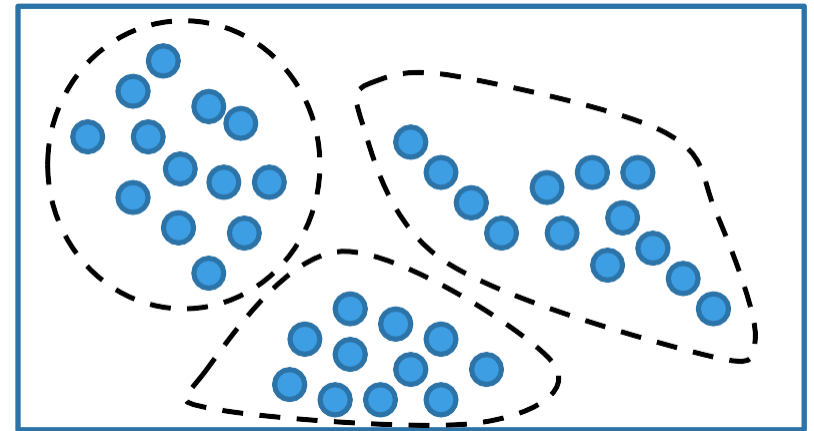
## ☐ Reinforcement learning

- ☐ Decision making (robot, chess machine)

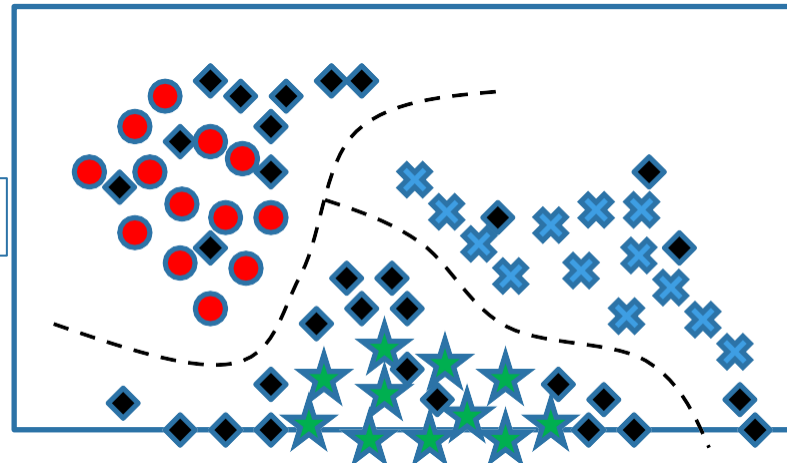
# Algorithms



Supervised learning



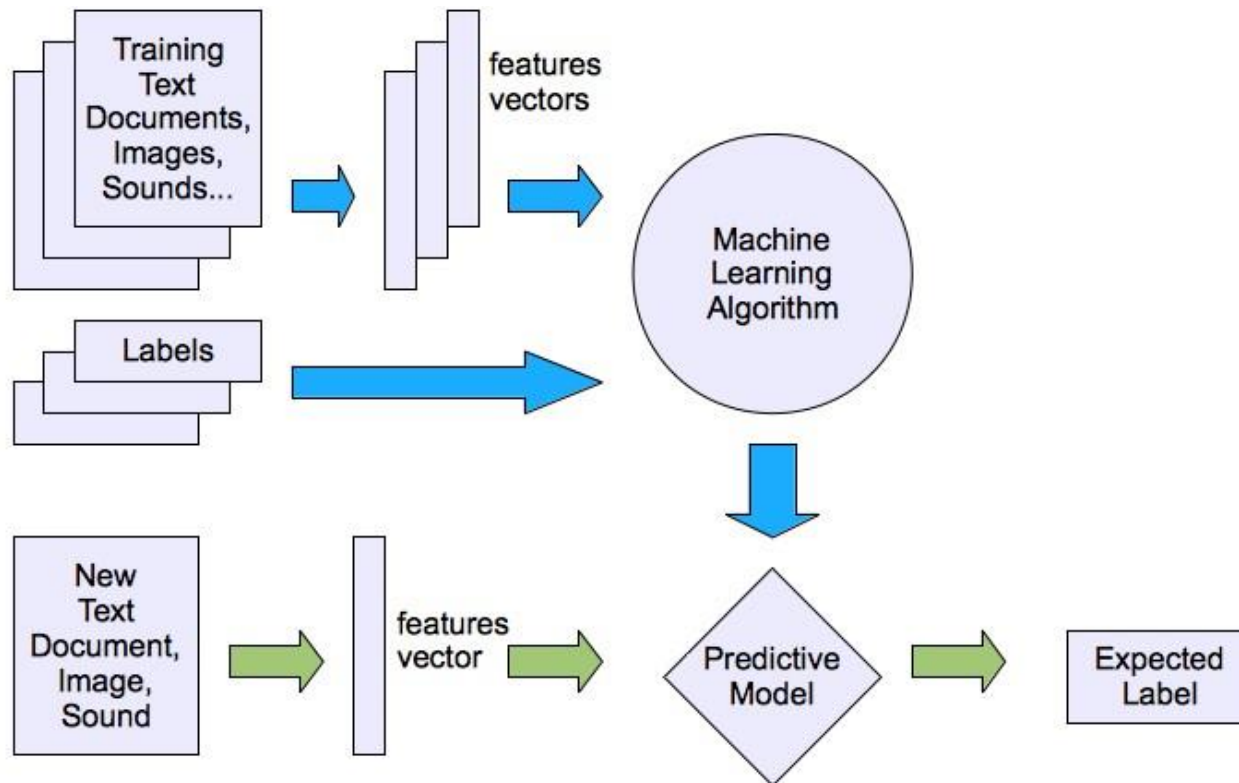
Unsupervised learning



Semi-supervised learning

# Machine learning structure

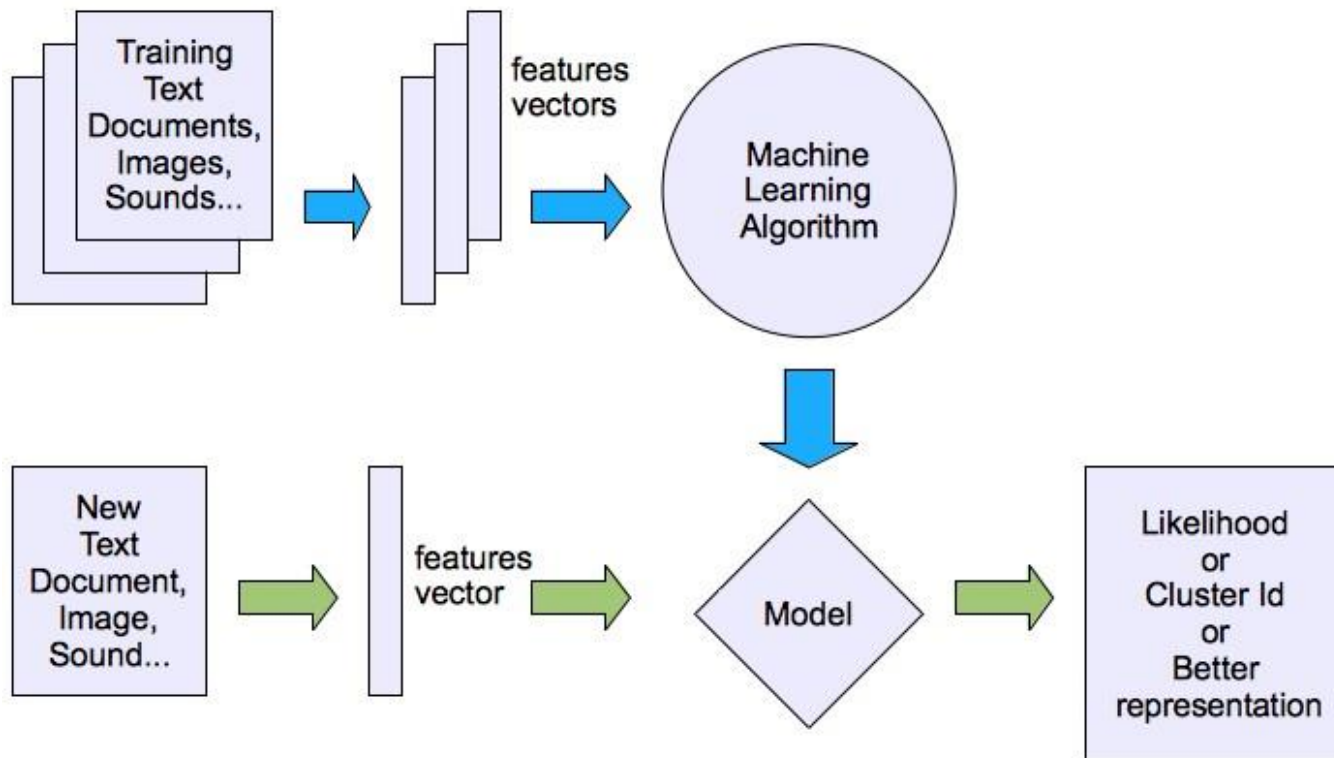
## ? Supervised learn





# Machine learning structure

## ? Unsupervised learning



## 2.2 Deep Feedforward Networks – Example of Ex OR

### DEFINITIONS OF NEURAL NETWORKS

#### **According to Nigrin (1993), p. 11:**

A neural network is a circuit composed of a very large number of simple processing elements that are neurally based. Each element operates only on local information.

Furthermore each element operates asynchronously; thus there is no overall system clock.

#### **According to Zurada (1992):**

Artificial neural systems, or neural networks, are physical cellular systems which can acquire, store and utilize experiential knowledge.

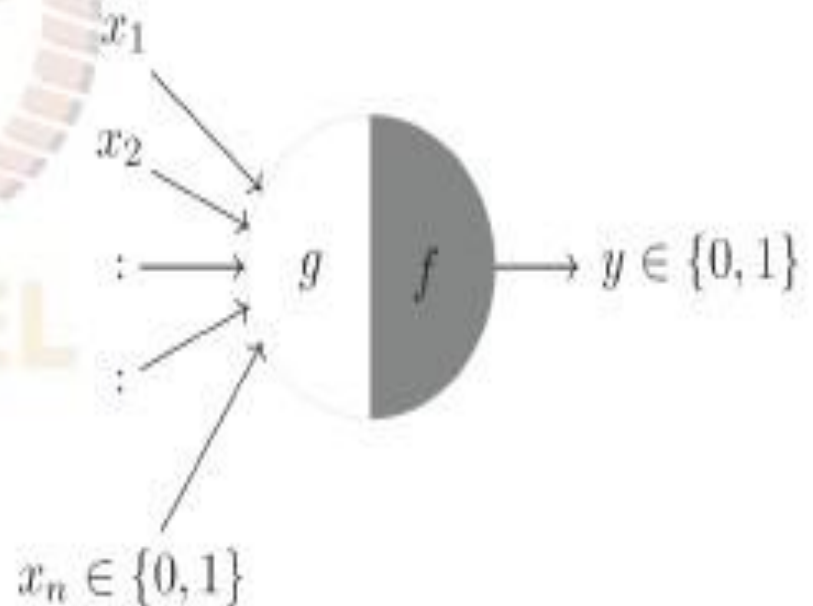
# McCulloch-Pitts Neuron

- McCulloch (neuroscientist) and Pitts (logician) proposed a highly simplified computational model of the neuron (1943)
- $g$  aggregates the inputs and the function  $f$  takes a decision based on this aggregation
- The inputs can be excitatory or inhibitory
- $y = 0$  if any  $x_i$  is inhibitory, else

$$g(x_1, x_2, \dots, x_n) = g(\mathbf{x}) = \sum_{i=1}^n x_i$$

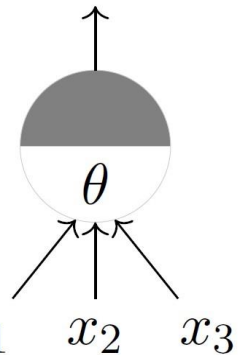
$$y = f(g(\mathbf{x})) = 1 \text{ if } g(\mathbf{x}) \geq \theta$$
$$= 0 \text{ if } g(\mathbf{x}) < \theta$$

- $\theta$  is a thresholding parameter



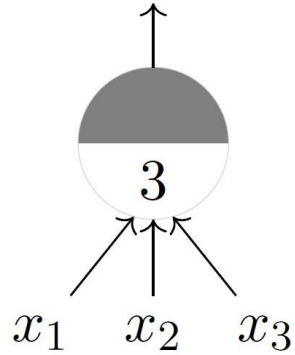
# McCulloch-Pitts Neuron

$y \in \{0, 1\}$



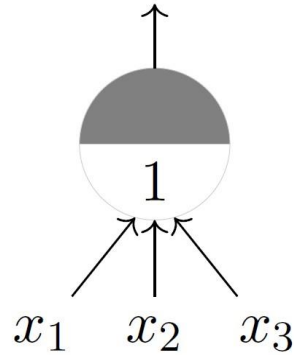
A McCulloch-Pitts Unit

$y \in \{0, 1\}$



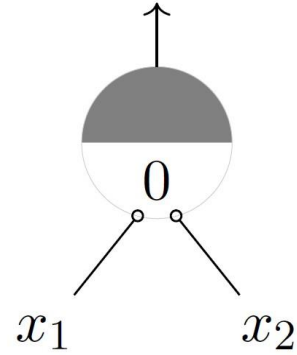
AND function

$y \in \{0, 1\}$



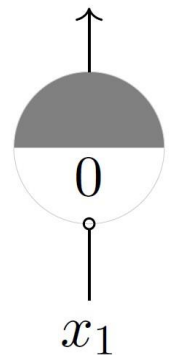
OR function

$y \in \{0, 1\}$



NOR function

$y \in \{0, 1\}$



NOT function



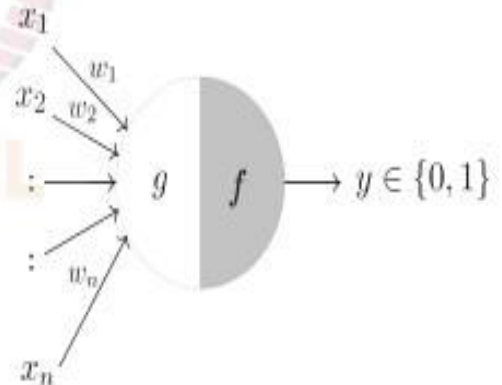
- Feedforward MP networks can compute any Boolean function  $f : \{0, 1\}^n \rightarrow \{0, 1\}$
- Recursive MP networks can simulate any Deterministic Finite Automaton (DFA) (S

# Perceptrons

- Frank Rosenblatt, an American psychologist, proposed the perceptron model (1958)
- Later refined and carefully analyzed by Minsky and Papert (1969)
- A more general computational model than McCulloch-Pitts neurons
- Inputs are no longer limited to boolean values

$$g(x_1, x_2, \dots, x_n) = g(\mathbf{x}) = \sum_{i=1}^n w_i * x_i$$

$$y = f(g(\mathbf{x})) = 1 \text{ if } g(\mathbf{x}) \geq \theta$$
$$= 0 \text{ if } g(\mathbf{x}) < \theta$$



# Perceptrons

- Frank Rosenblatt, an American psychologist, proposed the perceptron model (1958)
- Later refined and carefully analyzed by Minsky and Papert (1969)
- A more general computational model than McCulloch-Pitts neurons
- Inputs are no longer limited to boolean values

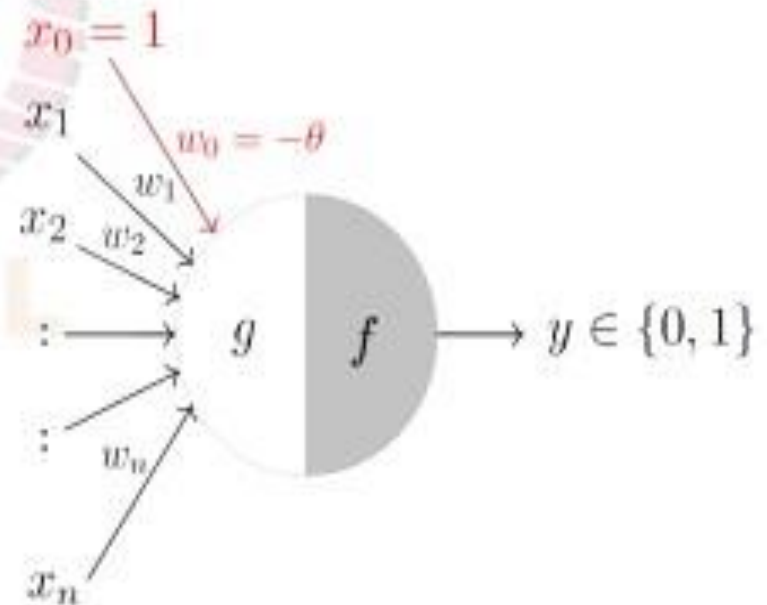
- A more accepted convention

$$g(x_1, x_2, \dots, x_n) = g(\mathbf{x}) = \sum_{i=0}^n w_i * x_i$$

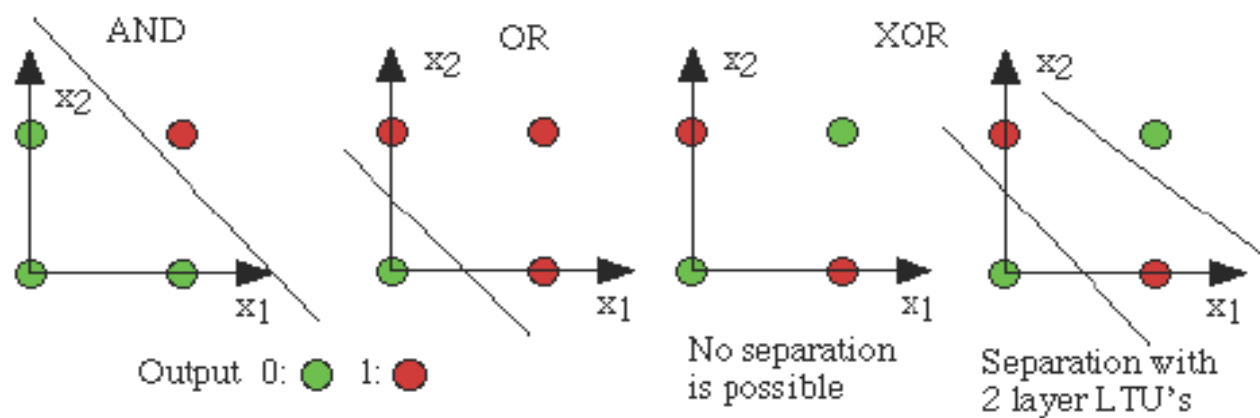
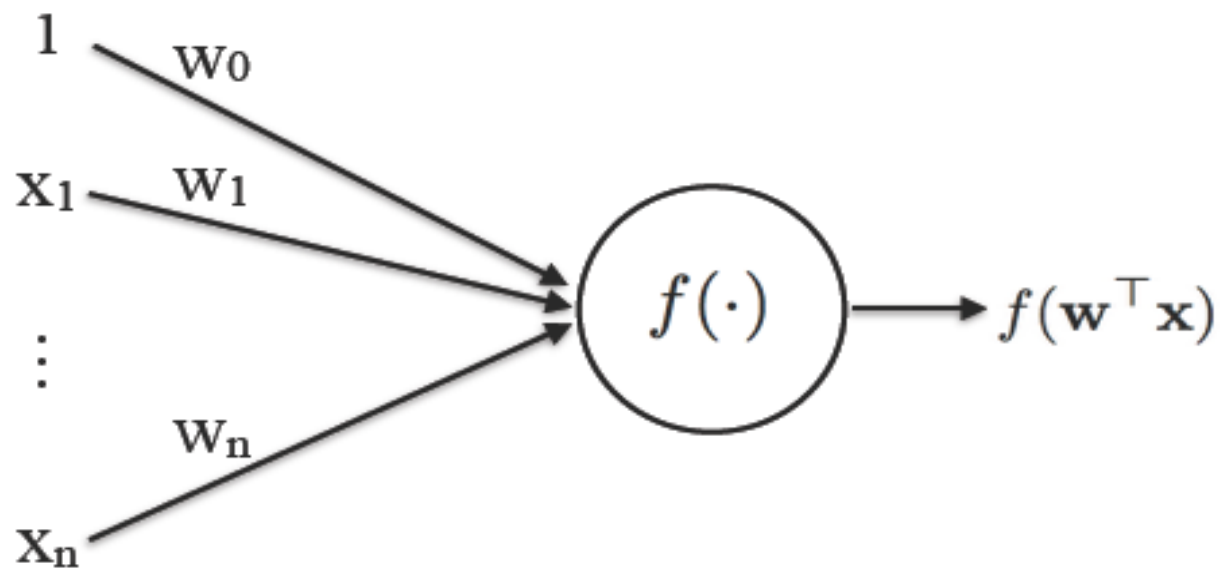
$$y = f(g(\mathbf{x})) = 1 \text{ if } g(\mathbf{x}) \geq 0$$

$$= 0 \text{ if } g(\mathbf{x}) < 0$$

where  $x_0 = 1$  and  $w_0 = -\theta$



F



# The X O R Problem

$x_1$	$x_2$	XOR	
0	0	0	$w_0 + \sum_{i=1}^2 w_i x_i < 0$
1	0	1	$w_0 + \sum_{i=1}^2 w_i x_i \geq 0$
0	1	1	$w_0 + \sum_{i=1}^2 w_i x_i \geq 0$
1	1	0	$w_0 + \sum_{i=1}^2 w_i x_i < 0$

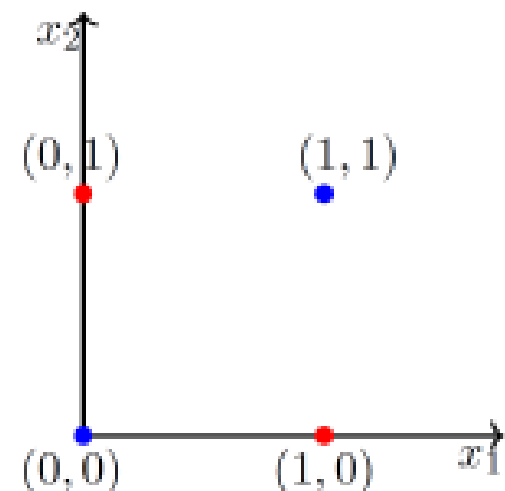
$$w_0 + w_1 \cdot 0 + w_2 \cdot 0 < 0 \implies w_0 < 0$$

$$w_0 + w_1 \cdot 1 + w_2 \cdot 0 \geq 0 \implies w_1 > -w_0$$

$$w_0 + w_1 \cdot 0 + w_2 \cdot 1 \geq 0 \implies w_2 > -w_0$$

$$w_0 + w_1 \cdot 1 + w_2 \cdot 1 < 0 \implies w_1 + w_2 < -w_0$$

- The fourth condition contradicts conditions 2 and 3
- No solution possible satisfying this set of inequalities



- Indeed you can see that it is impossible to draw a line which separates the red points from the blue points



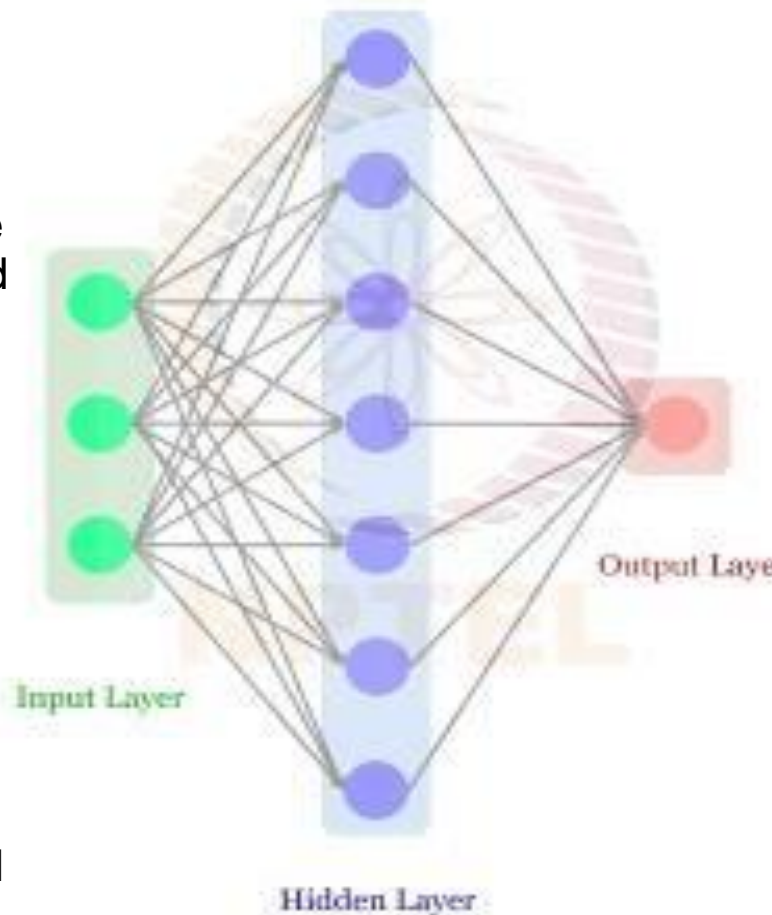
# Multi-Layer Perceptrons

This diagram illustrates the structure of an MLP, which includes:

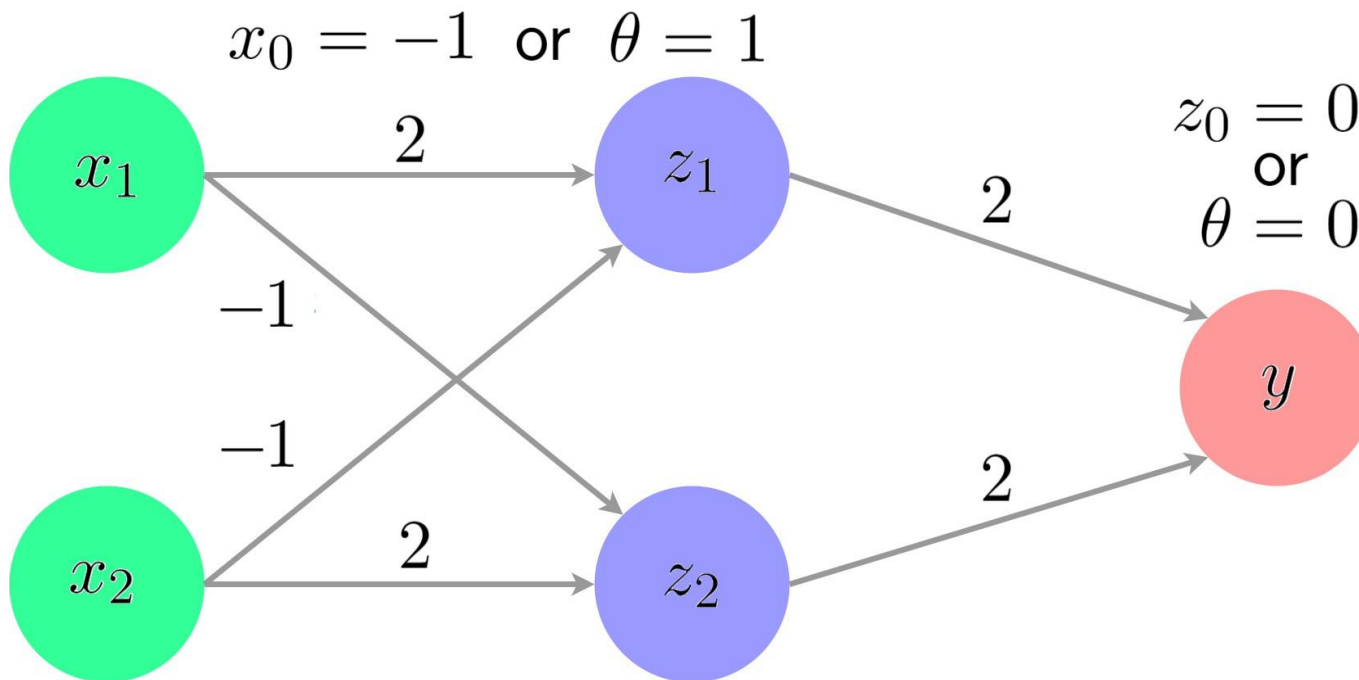
1. Input Layer: Takes input values (features) such as  $x_1$ ,  $x_2$  and  $x_3$ .
2. Hidden Layers: Intermediate layers (like Hidden Layer 1 and 2) with neurons that process the inputs to detect complex patterns.
3. Output Layer: Produces the final result of the model.

Key idea: Each neuron in a layer is connected to every neuron in the next layer through weights and biases.

Purpose: The MLP is designed to handle complex tasks by learning non-linear relationships in data.



# Solving XOR with Multi-Layer Perceptrons



A neural network solves the XOR problem, which is non-linearly separable and cannot be solved by a single-layer perceptron. Using a multi-layer perceptron (MLP) with a hidden layer, the network applies a non-linear transformation to the inputs. This is achieved through weighted sums, biases, and an activation function (e.g., sigmoid), creating a new space where the XOR outputs (0 or 1) become linearly separable. The output layer combines the transformed outputs from the hidden layer to produce the final result. Through training, the network learns the optimal weights and biases, enabling it to perfectly model the XOR logic.

$(x_1, x_2)$	$(z_1, z_2)$	$y$
(0,0)	(0,0)	0
(0,1)	(0,0)	1
(1,0)	(1,0)	1
(1,1)	(0,0)	0

# Going Beyond Binary Inputs and Outputs

## Question

- What about arbitrary functions of the form  $y = f(x)$  where  $x \in \mathbb{R}^n$  (instead of  $\{0, 1\}^n$ ) and  $y \in \mathbb{R}$  (instead of  $\{0, 1\}$ )?
- Can we use the same perceptron model to represent such functions?

## Answers:

1. In this case, the perceptron model can still be used, but it requires modification. The perceptron originally handles binary outputs ( $\{0, 1\}$ ), so applying it to arbitrary real-valued functions involves generalizing the activation function to work with continuous outputs. This leads to models like the linear perceptron or regression models.
2. The classic perceptron with a step activation function cannot represent real-valued outputs directly. However, by replacing the step function with a differentiable activation function (like the sigmoid or ReLU), the model can approximate arbitrary functions. This concept forms the basis of artificial neural networks used in modern machine learning.

# Need of activation function

## 1. Introducing Non-Linearity:

- Without activation functions, a neural network would simply be a linear combination of inputs and weights, limited to modeling only linear relationships.

## 2. Learning Complex Representations:

- Activation functions enable neurons to learn more intricate representations of data.

## 3. Decision-Making Capabilities:

- Activation functions introduce decision-making power to neurons.
- Based on the activation function's output, a neuron can decide whether to "fire" or not, influencing the flow of information through the network and ultimately shaping the model's output.

## 4. Controlling Output Range:

- Activation functions can regulate the output range of neurons.
- Some activation functions, like sigmoid and tanh, constrain outputs between 0 and 1 or -1 and 1, making them suitable for probability-related tasks.
- Others, like ReLU, allow for unbounded positive outputs, useful for representing real-valued quantities.

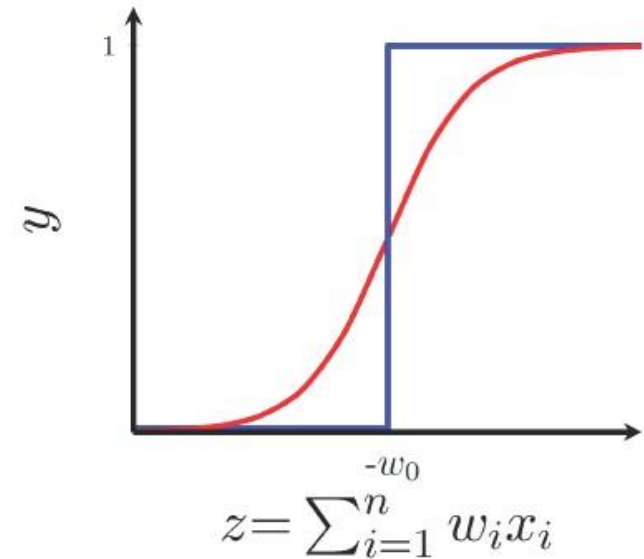
# Need for activation function

## Sigmoid function








- We could use any logistic function to obtain a smoother output function than a step function
- One form is the sigmoid function:

$$y = \frac{1}{1 + e^{-(w_0 + \sum_{i=0}^n w_i x_i)}}$$

- No longer a sharp transition at the threshold  $-w_0$
- Also, output is no longer binary but a real value between 0 and 1 which can be interpreted as a probability
- Unlike the step function, this one is smooth, continuous at  $-w_0$  and most importantly **differentiable**

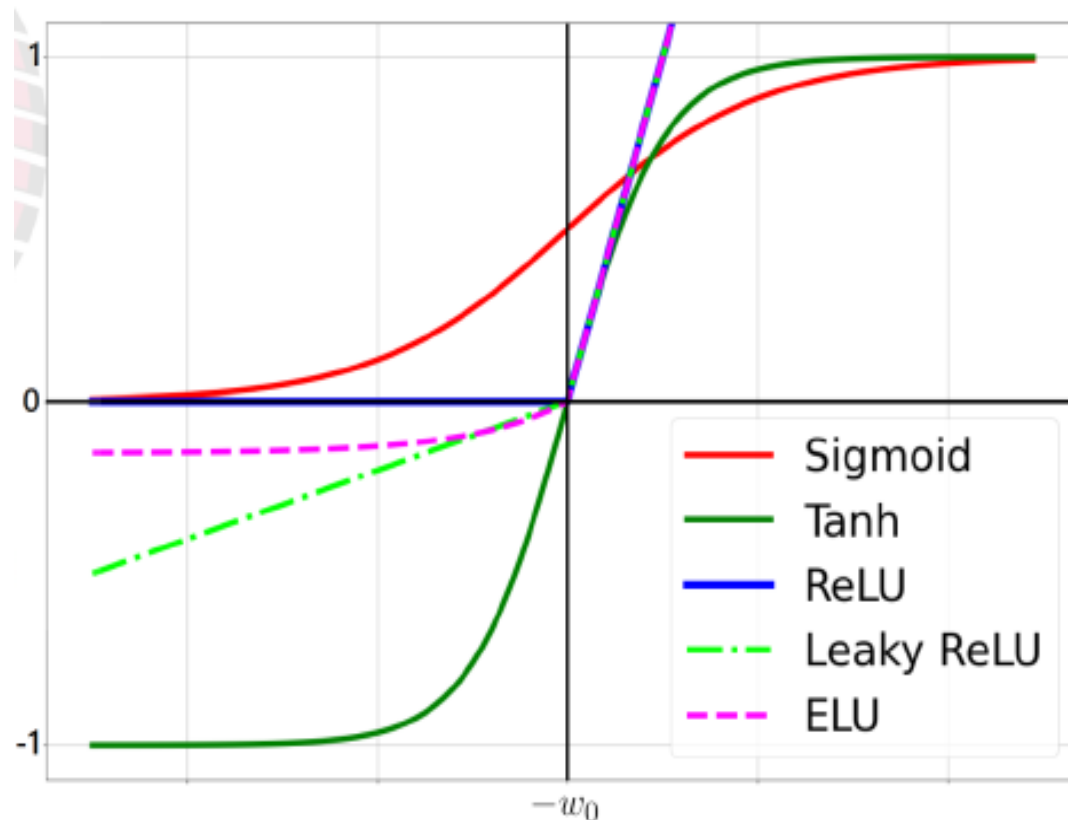


# Activation Function

Identity		$f(x) = x$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Logistic (a.k.a. Soft step)		$f(x) = \frac{1}{1 + e^{-x}}$
TanH		$f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$
ArcTan		$f(x) = \tan^{-1}(x)$
Softsign <sup>[7][8]</sup>		$f(x) = \frac{x}{1 +  x }$
Rectified linear unit (ReLU) <sup>[9]</sup>		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$

# Activation Function

- **Leaky ReLU:**  $y = \max(\alpha z, z), \alpha \in (0, 1)$
- **Exponential Linear Unit (ELU):**  
 $y = \max(\alpha(e^z - 1), z), \text{ where } \alpha > 0$

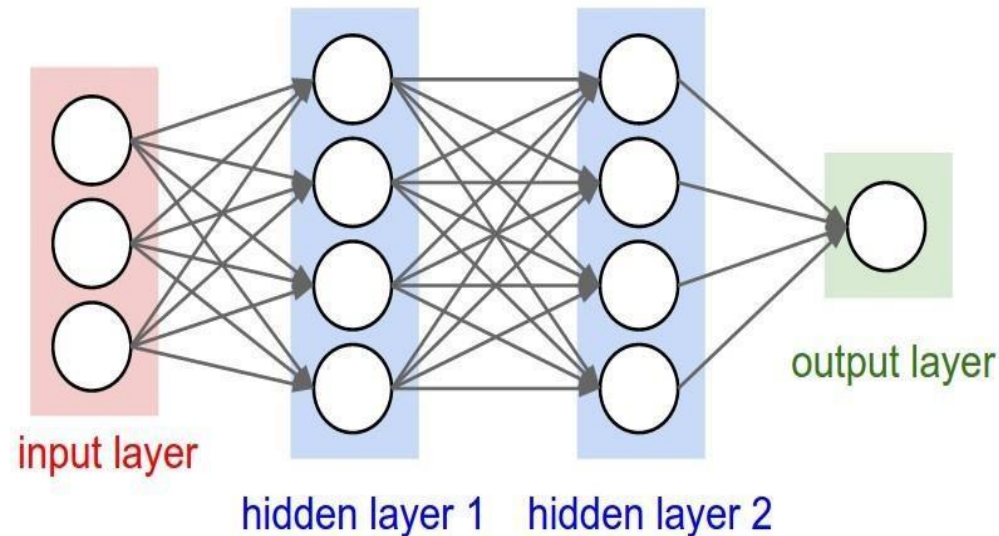


# **Feedforward Neural Networks and Backpropagation**



# Feedforward Networks

A feedforward neural network, also called a multi-layer perceptron, is a collection of neurons, organized in *layers*.

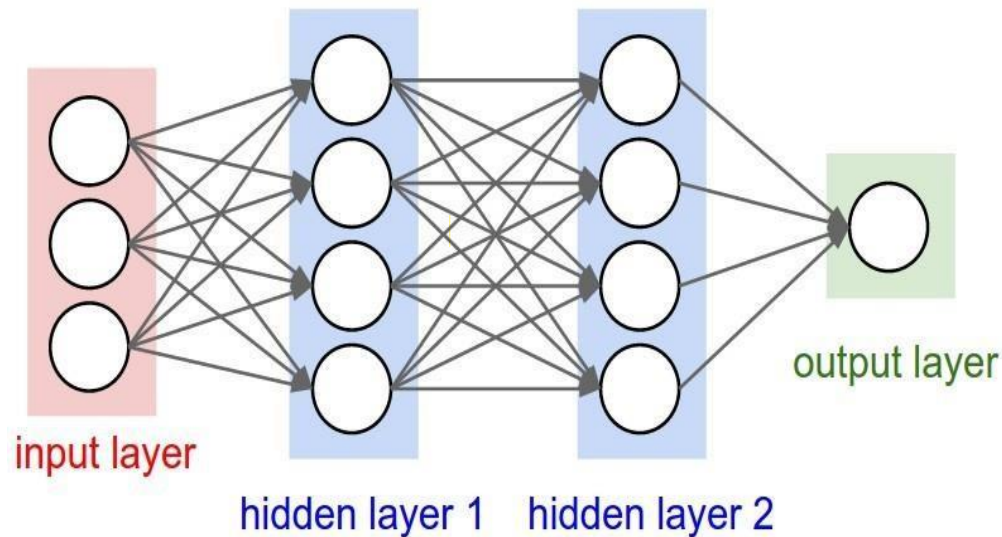


It is used to approximate some function  $f^*$ . For instance,  $f^*$  could be a classifier that maps an input vector  $\mathbf{x}$  to a category  $\mathbf{y}$ .

The neurons are arranged in the form of a directed acyclic graph i.e., the information only flows in one direction - input  $\mathbf{x}$  to output  $\mathbf{y}$ . Hence the term **feedforward**.

- How Do FNNs Work?
- Imagine a network of interconnected artificial neurons, each capable of processing information and making simple decisions. In an FNN, these neurons are organized in layers:
- Input layer: Receives the raw data.
- Hidden layers: Perform computations and extract features from the data. There can be one or several hidden layers depending on the complexity of the problem.
- Output layer: Produces the final prediction or result.

# Feedforward Networks



The number of layers in the network (excluding the input layer) is known as depth

Each neuron can be seen as a **vector-to-scalar** function which takes a vector of inputs

from the previous layer and computes a scalar value.

Above network can be seen as a composition of functions  $y = f^{(3)}(f^{(2)}(f^{(1)}(x)))$ ,  $f^{(1)}$  being the first hidden layer,  $f^{(2)}$  being the second and  $f^{(3)}$  being the final output layer.

- How information travels through the Forward Network:

- 1. Inputs:

- Each neuron in the input layer receives a specific element from the data point.

- 2. Weighted Sum:

- Each hidden neuron sums the products of its inputs with their corresponding weights (coefficients).

- 3. Activation:

- The neuron applies an activation function (e.g., sigmoid, ReLU) to the weighted sum to introduce non-linearity.

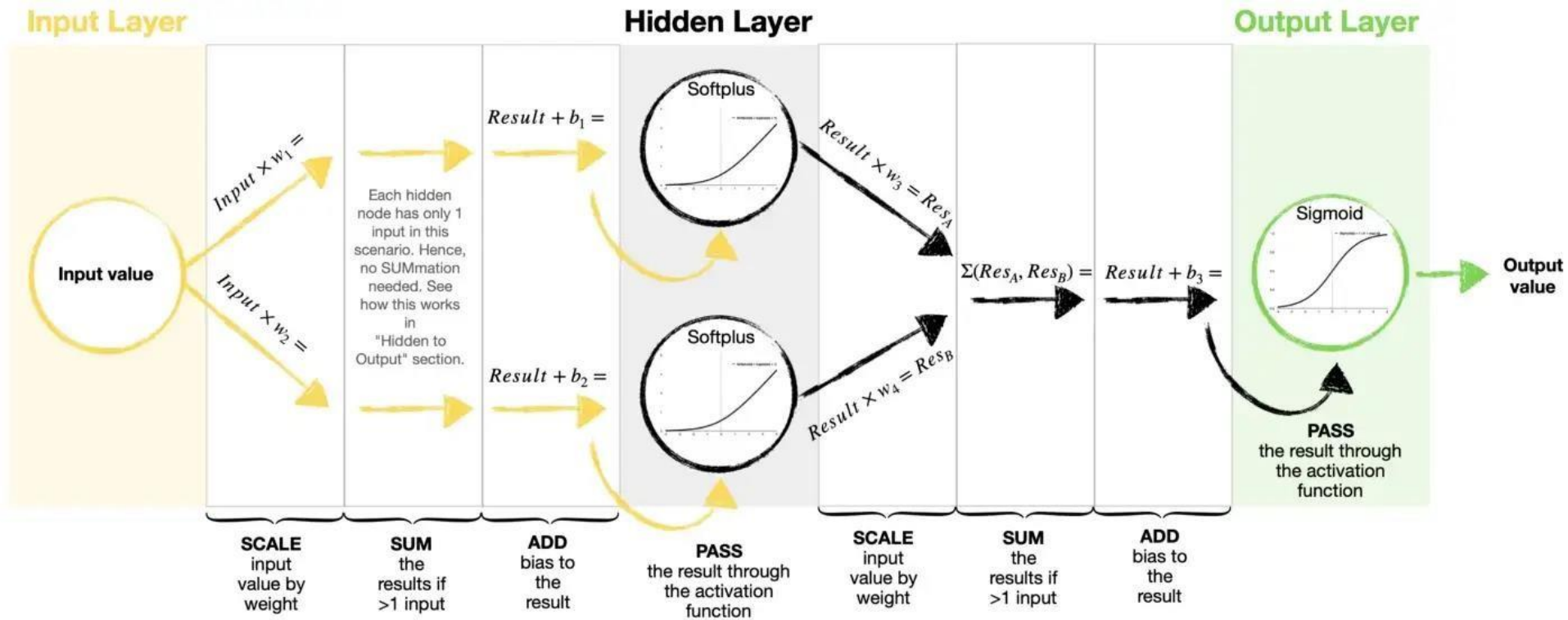
- 4. Output:

- The output neuron combines the activated signals from the hidden layer and performs its own activation, producing the final prediction.

# Feedforward Networks

- To approximate some function  $f^*$ , we are generally given noisy estimates of  $f^*(\mathbf{x})$  at different points, in the form of a dataset  $\{\mathbf{x}_i, y_i\}_{i=1}^M$ .
- Our neural network defines a function  $y = f(\mathbf{x}; \theta)$ . Our goal is to learn the parameters (weights and biases)  $\theta$  such that  $f$  best approximates  $f^*$ .
- How to find the values of the parameters i.e., train the network?
- **Gradient Descent**, the go-to method to train neural networks

# Feedforward Networks



## 2.3 Gradient-Based Learning

### Gradient Descent

- Gradient descent is an optimization algorithm used to adjust the weights and biases of the network, ultimately leading it to learn and improve its performance on a specific task.
- How it works:
- 1. Objective Function and Error:
- We start with an objective function, which quantifies the network's performance on the training data. Examples include mean squared error for regression or cross-entropy for classification.
- During training, the network predicts outputs for each data point, and these predictions are compared to the actual targets. The difference between the prediction and the target is called the error.

# Gradient Descent

- 2. Gradient Calculation:
  - Gradient descent relies on the concept of a gradient, which essentially tells us how quickly the error changes with respect to each weight and bias in the network.
  - By calculating the gradients for each parameter, we understand how changing that parameter would affect the overall error.
- 3. Weight and Bias Update:
  - The core idea of gradient descent is to take small steps in the direction that reduces the error.
  - We use the calculated gradients to update the weights and biases of the network. The update typically follows a rule like  $\text{weight} = \text{weight} - \text{learning\_rate} * \text{gradient}$ , where `learning_rate` determines the step size.

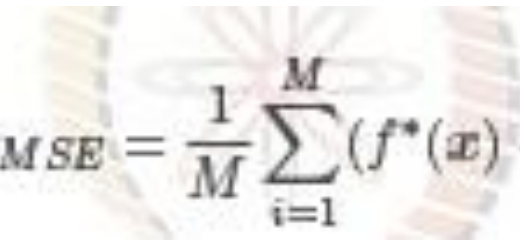


# Gradient Descent

- 4. Iteration and Optimization:
- This process of calculating gradients, updating weights, and computing predictions is repeated iteratively over the entire training data multiple times (epochs).
- With each iteration, the network gradually adjusts its parameters, learning the underlying patterns in the data and improving its ability to make accurate predictions.

# Gradient Descent: A 1 D Example

- Neural networks are usually trained by minimizing a loss function, such as mean square error:


$$Loss_{MSE} = \frac{1}{M} \sum_{i=1}^M (f^*(\mathbf{x}) - f(\mathbf{x}; \theta))^2$$

- Let us consider a simple 1D example, where we try to minimize the function  $f(x) = \frac{1}{2} x^2$ .  
Specifically, we find out the value  $x^*$  gives the smallest value for  $f(x)$  i.e.,  $f(x^*)$ .

$$x^* = \arg \min_x f(x)$$

# Gradient Descent: A 1D Example

- We can obtain the slope of the function  $f(x)$  at  $x$  by taking its derivative i.e.,  $f'(x)$ .

# Gradient Descent: A 1 D Example

- We can obtain the slope of the function  $f(x)$  at  $x$  by taking its derivative i.e.,  $f'(x)$ .
- This means, if we give a very small *push* to  $x$  in the direction (sign) of the slope, we're sure that the function will increase.

$$f(x + p \cdot \text{sign}(f'(x))) > f(x) \text{ for an infinitesimally small } p$$

1. The slope: The slope of a function at a particular point tells us whether the function is increasing or decreasing at that spot. To find the slope, we calculate the derivative of the function, written as  $f'(x)$ .
2. Direction of movement:  
If you move a little in the direction of the slope (positive gradient), the function value increases. If you move in the opposite direction of the slope (negative gradient), the function value decreases.
3. How gradient descent works:  
Start at a random point  $x$  on the graph.  
Use the slope  $f'(x)$  to figure out which direction to move.  
Take small steps in the direction of the negative slope (because you're trying to go downhill to reach the lowest point).  
Repeat this process until you reach the minimum of the function.

# Gradient Descent: A 1 D Example

- We can obtain the slope of the function  $f(x)$  at  $x$  by taking its derivative i.e.,  $f'(x)$ .
- This means, if we give a very small *push* to  $x$  in the direction (sign) of the slope, we're sure that the function will increase.

$$f(x + p \cdot \text{sign}(f'(x))) > f(x) \text{ for an infinitesimally small } p$$

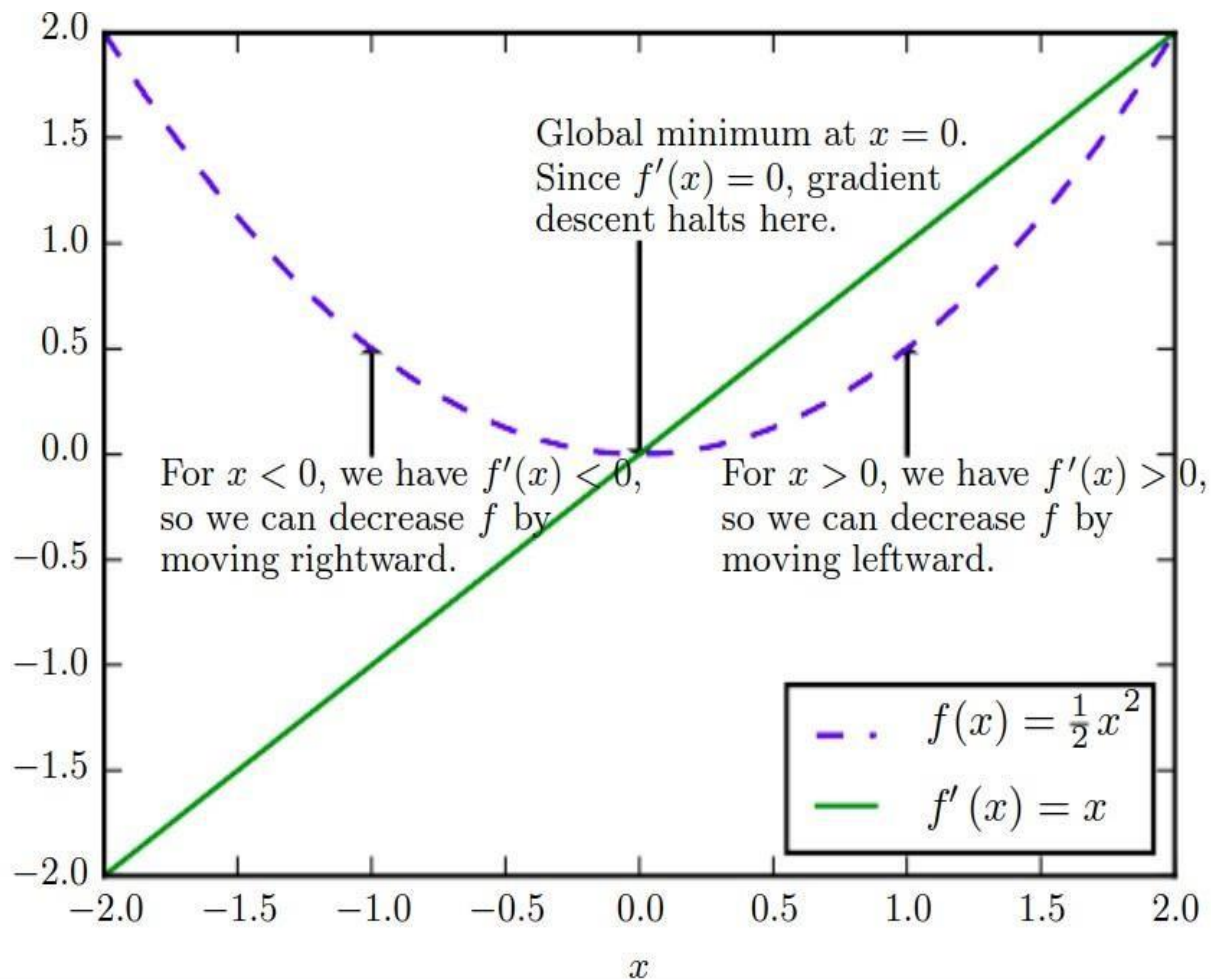
- The reverse is also true i.e.,

$$f(x - p \cdot \text{sign}(f'(x))) < f(x) \text{ for an infinitesimally small } p$$

- This forms the basis for gradient descent - we start off at a random  $x$ , and take small steps in the direction of the **negative** gradient.

- If the slope ( $f'(x)$ ) is positive, the function is increasing. To minimize the function, you should move in the opposite direction, which is to the left (towards lower values of  $x$ ).
- If the slope ( $f'(x)$ ) is negative, the function is decreasing. To minimize the function, you should move in the opposite direction again, which is to the right (towards higher values of  $x$ ).

## Gradient Descent: A 1D Example



# Gradient Descent 1D Example

Imagine a hiker descending a hilly terrain to reach the lowest point (valley). Gradient descent mimics this process to find the minimum of a function.

## Key concepts:

- Function:** The hill you're navigating, represented by a mathematical equation (e.g.,  $f(x) = x^2$ ).
- Gradient:** The slope of the hill at any point, indicating the direction of steepest ascent or descent.
- Learning rate:** The hiker's step size, controlling how far they move in each iteration.

## Steps:

- 1.Start: Choose an initial position (x-value) on the hill.
- 2.Calculate gradient: Determine the slope (derivative) of the function at this position.
- 3.Take a step: Move in the direction opposite to the gradient (downhill), with a step size determined by the learning rate.
- 4.Repeat: Calculate the gradient at the new position and take another step, continuing until you reach the valley (or a close approximation).g

# Example

Function:

- We're working with the function  $f(x) = x^2$ , which represents a parabolic curve (like a hill) with its minimum at  $x = 0$ .

Initial Position:

- We start at  $x = 3$ , meaning the hiker is standing on the hill at a point where the value of  $f(x)$  is 9.

Learning Rate:

- The learning rate is set to 0.1, which controls how large each step the hiker takes will be.

Iteration 1:

1. Calculate gradient:

- The gradient of  $f(x) = x^2$  is  $2x$ .
- At  $x = 3$ , the gradient is 6 (the slope of the hill is positive and steep).

2. Take a step:

- Since we want to move downhill, we take a step in the opposite direction of the gradient.
- The step size is  $-0.1 * 6$  (learning rate multiplied by gradient), which equals -0.6.

3. New position:

- The hiker's new position becomes  $x = 3 - 0.6 = 2.4$ .



# Example...

Iteration 2:

1. Calculate gradient:

- At  $x = 2.4$ , the gradient of  $f(x) = x^2$  is 4.8.

2. Take a step:

- The step size is  $-0.1 * 4.8 = -0.48$ .

3. New position:

- The hiker moves to  $x = 2.4 - 0.48 = 1.92$ .

Repeat iterations:

- This process continues, with the hiker taking steps based on the gradient, gradually moving closer to the minimum of the function.
- The iterations stop when the hiker reaches a point where the gradient is very close to zero, indicating they're at (or very near) the bottom of the valley.

## Why Negative Gradient?

- The gradient of the loss function points in the direction of the steepest increase in loss.
- We want to minimize the loss, so we take a step in the opposite direction, which is achieved by multiplying the gradient by  $-1$ .

Imagine a landscape:

- The loss function forms a landscape with valleys (minima) and hills (maxima).
- The negative gradient points us down the steepest slope of the hill towards the valley.
- By iteratively taking steps in this direction, we eventually reach the bottom of the valley (minimum loss), improving the network's accuracy.

# Why Negative Gradient?

Benefits of using the negative gradient:

- Helps find the optimal parameters that minimize the loss function.
- Enables efficient learning by guiding the network towards better predictions.
- Forms the basis for various optimization algorithms used in neural network training.
- the negative gradient in neural networks plays a critical role in guiding the learning process and adjusting parameters to minimize the loss function, ultimately leading to improved network performance and accurate predictions.

# Why Negative Gradient?

- Consider the multivariate case, since while training neural networks, the loss function we minimize is parametrized by multiple weights,  $\boldsymbol{\vartheta}$
- For simplicity, we denote our loss function as  $L(\boldsymbol{\vartheta})$ . Our aim is to find the weight vector  $\boldsymbol{\vartheta}$  which minimizes  $L(\boldsymbol{\vartheta})$
- Let  $\mathbf{u}$ , a unit vector, be the direction that takes us to the minimum, i.e.:

$$\begin{aligned} & \min_{\mathbf{u}, \mathbf{u}^T \mathbf{u} = 1} \mathbf{u}^T \nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) \\ &= \min_{\mathbf{u}, \mathbf{u}^T \mathbf{u} = 1} \|\mathbf{u}\|_2 \|\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta})\|_2 \cos \beta \end{aligned}$$

- Since  $\|\mathbf{u}\|_2 = 1$ , we can minimize the above function when  $\beta = 180^\circ$ , i.e. when  $\mathbf{u}$  is the direction of **negative** gradient

# How to use Gradient Descent

We can use Gradient Descent to train neural networks as follows:

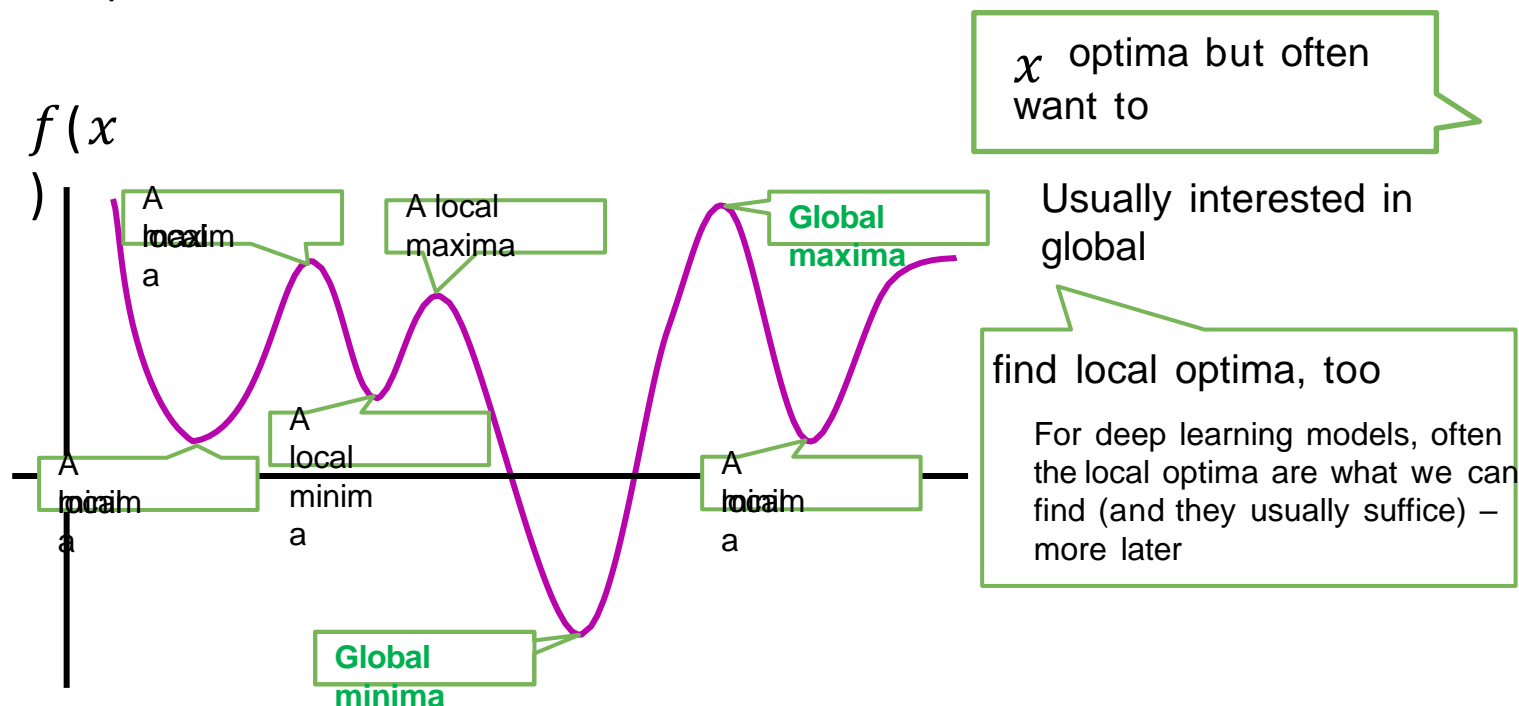
- Start with a random weight vector  $\theta$ .
- Compute the loss function over the dataset, i.e.,  $L(\theta)$  with the current network, using a suitable loss function such as mean-squared error
- Compute the gradients of the loss function with respect to each weight value  $\frac{\delta L}{\delta \theta_i}$ .
- Update the weights as follows, where  $\eta$  is the learning rate i.e., the amount by which the weight is changed in each step:

$$\theta_i^{\text{next}} = \theta_i^{\text{curr}} - \eta \frac{\delta L}{\delta \theta_i^{\text{curr}}}$$

We can repeat the above steps until the gradient is zero.

# Functions and their optima

- Many ML problems require us to optimize a function  $f$  of some variable(s)  $x$
- For simplicity, assume  $f$  is a scalar-valued function of a scalar  $x$  ( $f: \mathbb{R} \rightarrow \mathbb{R}$ )



- Any function has one/more optima (maxima, minima), and maybe saddle points
- Finding the optima or saddles requires derivatives/gradients of the function

# Derivatives

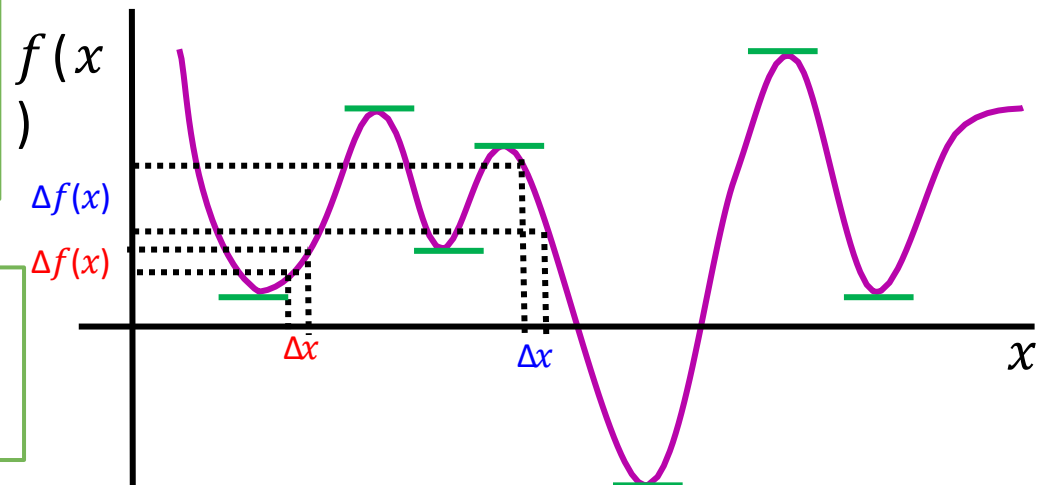
- Magnitude of derivative at a point is the rate of change of the func at that point

$$\frac{df(x)}{dx} = \lim_{\Delta x \rightarrow 0} \frac{\Delta f(x)}{\Delta x}$$

Sign is also important: Positive derivative means  $f$  is **increasing** at  $x$  if we increase the value of  $x$  by a very small amount; negative derivative means it is **decreasing**

Understanding how  $f$  changes its value as we change  $x$  is helpful to understand optimization (minimization/maximization)

algorithms

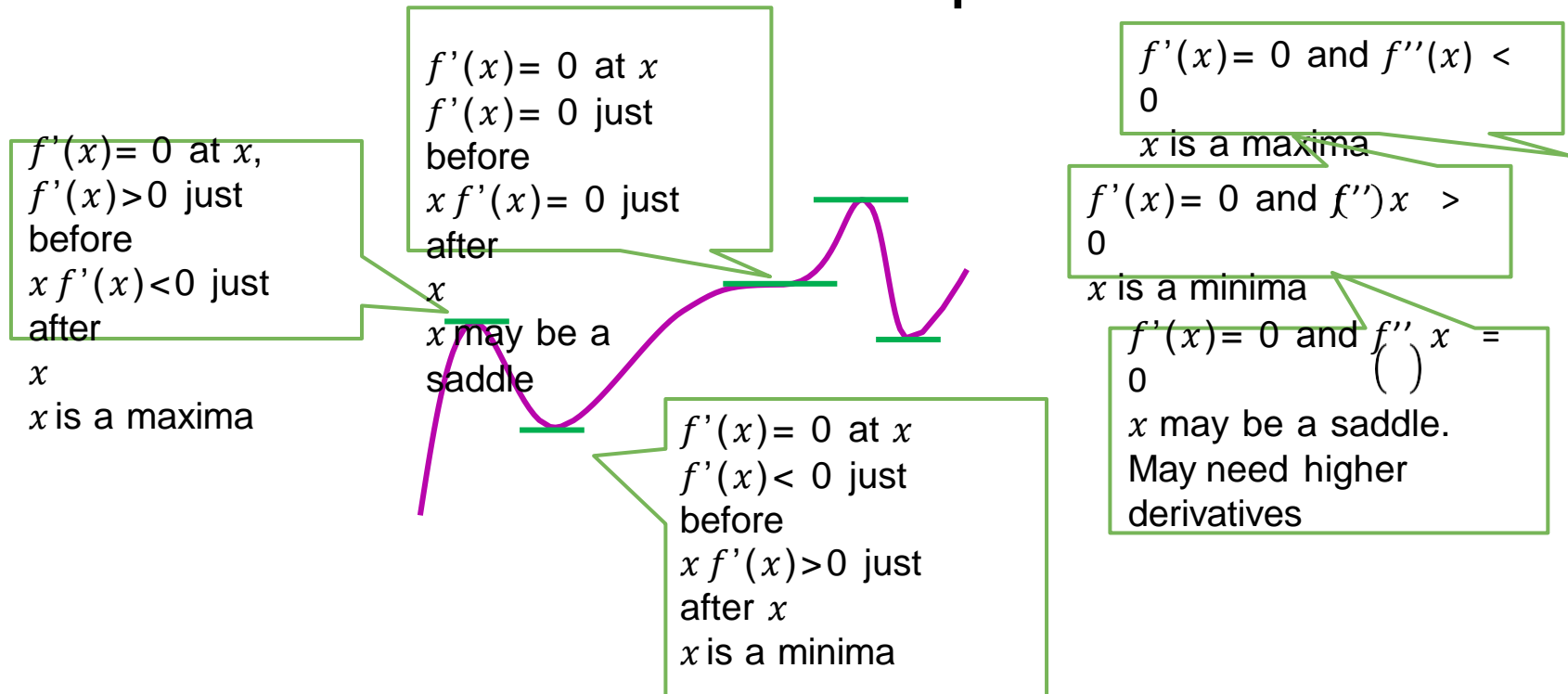


- Derivative becomes zero at stationary points (optima or saddle points)
  - The function becomes “flat” ( $\Delta f / \Delta x = 0$  if we change  $x$  by a very little at such points)
  - These are the points where the function has its maxima/minima (unless they are saddles)

# Derivatives

48

- How the derivative itself changes tells us about the function's optima

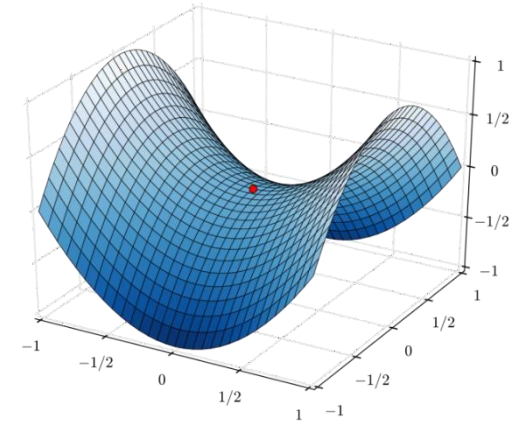
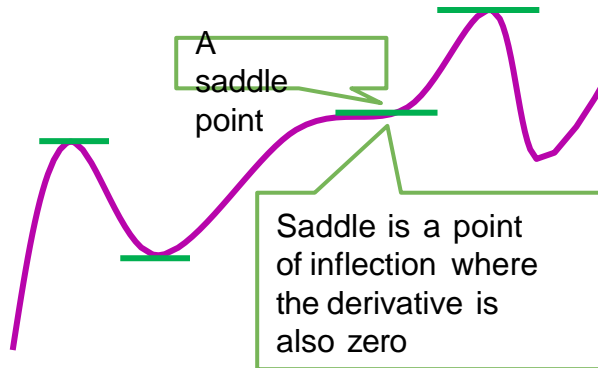


- The second derivative  $f''(x)$  can provide this information



# Saddle Points

- Points where derivative is zero but are neither minima nor maxima 49

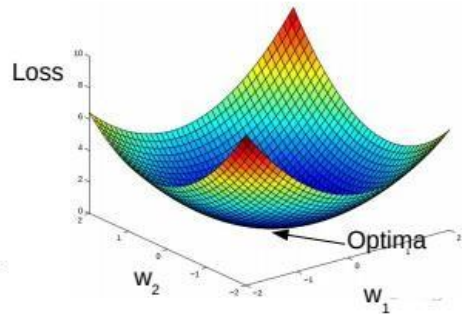
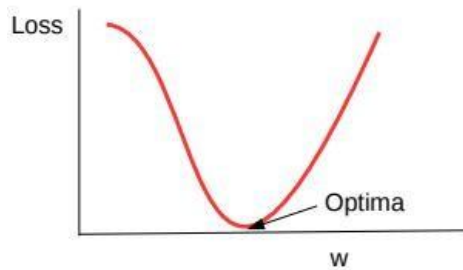


- Saddle points are very common for loss functions of deep learning models
  - Need to be handled carefully during optimization
- Second or higher derivative may help identify if a stationary point is a saddle

# Convex and Non-Convex Functions

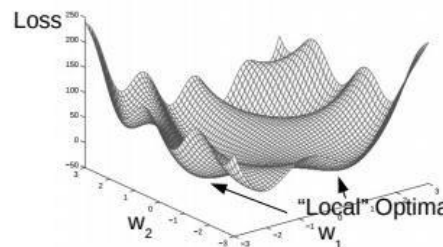
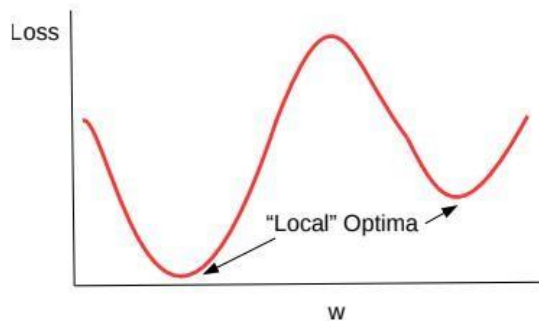
50

- A function being optimized can be either **convex** or **non-convex**



Convex functions are bowl-shaped. They have a unique optima (minima)

Negative of a convex function is called a **concave** function, which also has a unique optima (maxima)



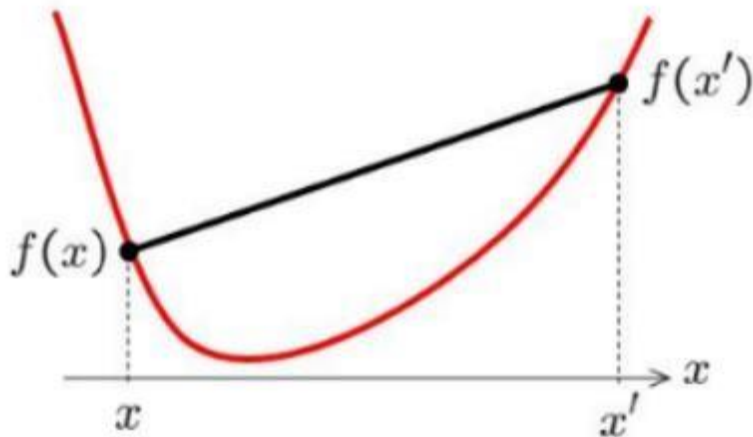
Non-convex functions have multiple minima. Usually harder to optimize as compared to convex functions

Loss functions of most deep learning models are non-convex

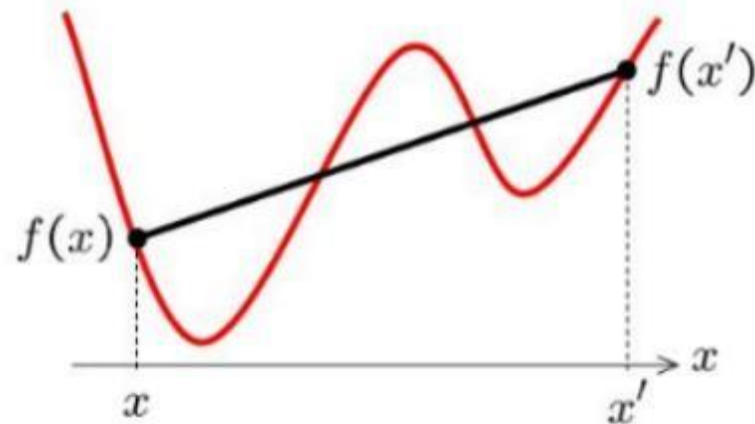
## Convex Functions

- Informally,  $f(x)$  is convex if all of its chords\* lie above the function everywhere

Convex Function

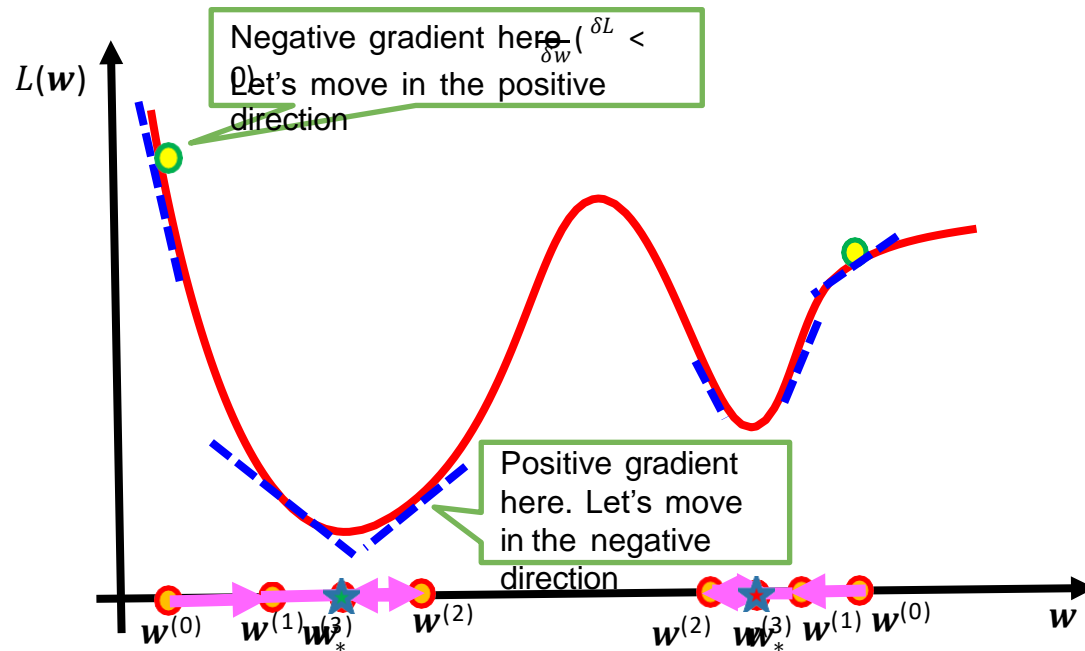


Non-convex Function



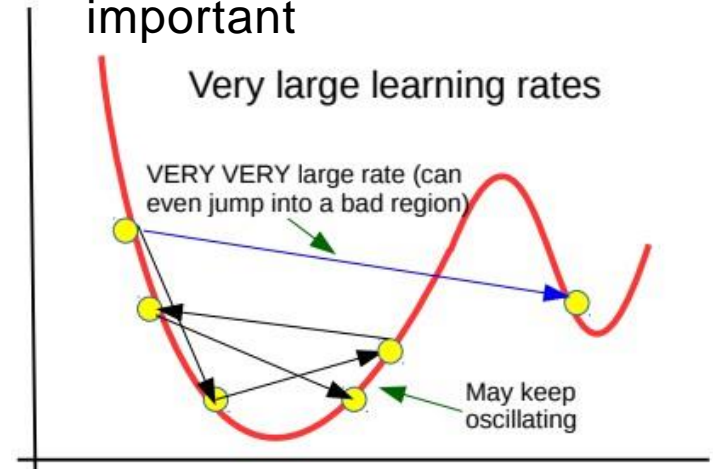
\* A chord is simply a straight line segment that connects two points on the graph of a function.

# Gradient Descent: An Illustration

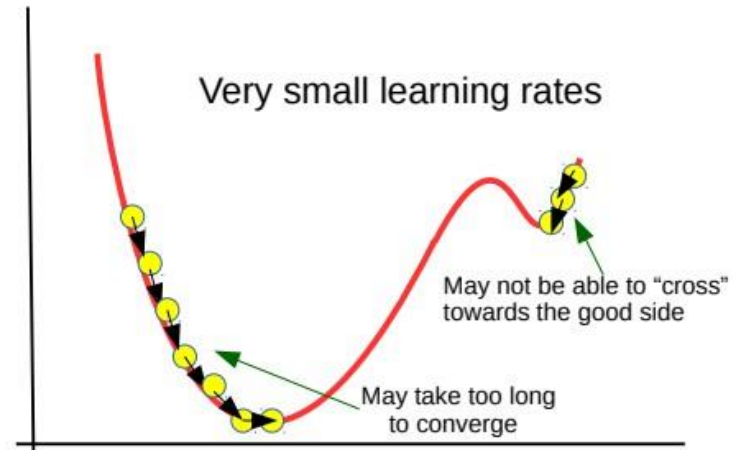


Learning rate is very important

Very large learning rates



Very small learning rates

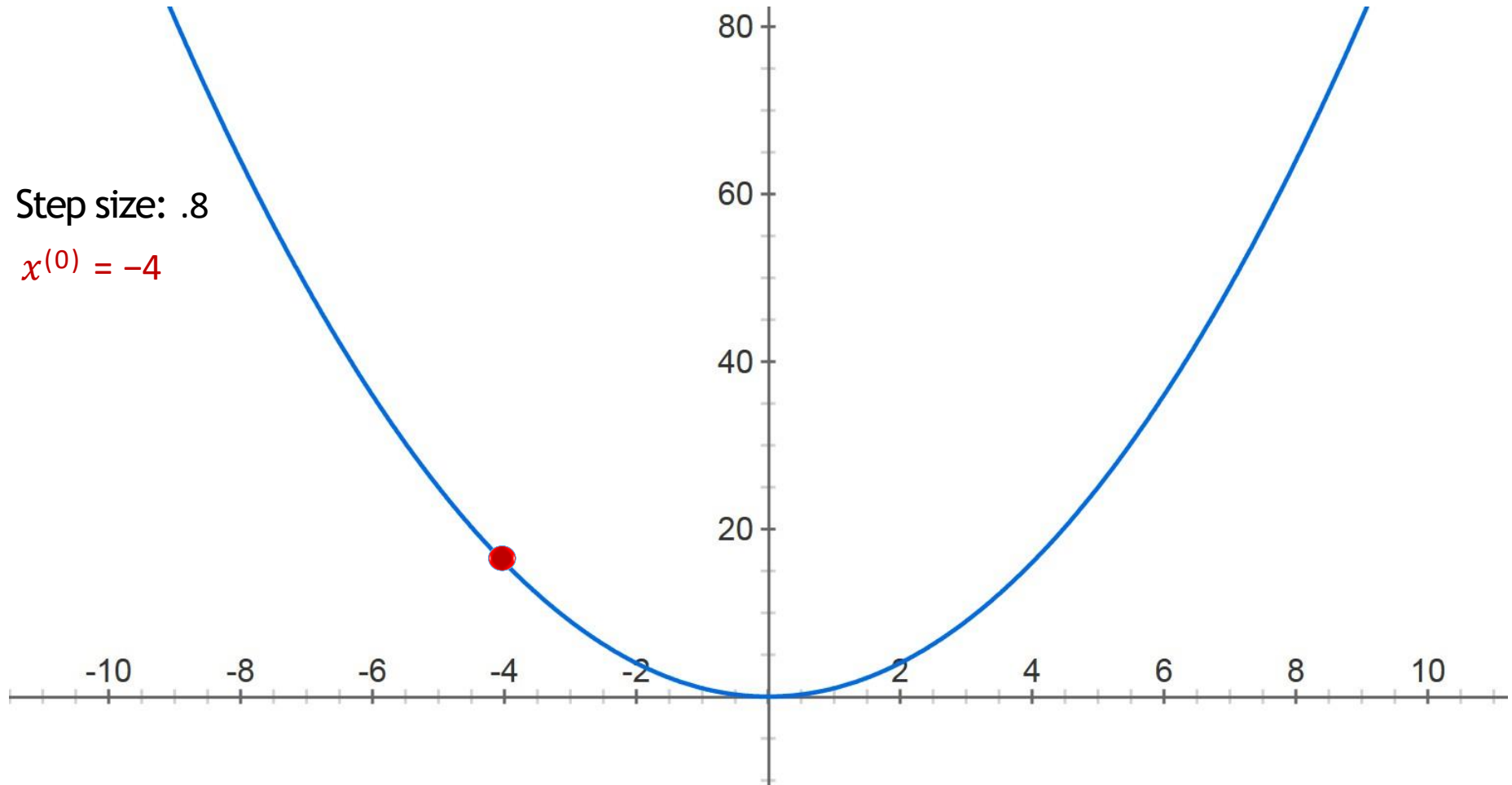


# Gradient Descent

$$f(x) = x^2 \quad \text{Gradient is } f'(x) = 2x$$

Step size: .8

$$x^{(0)} = -4$$



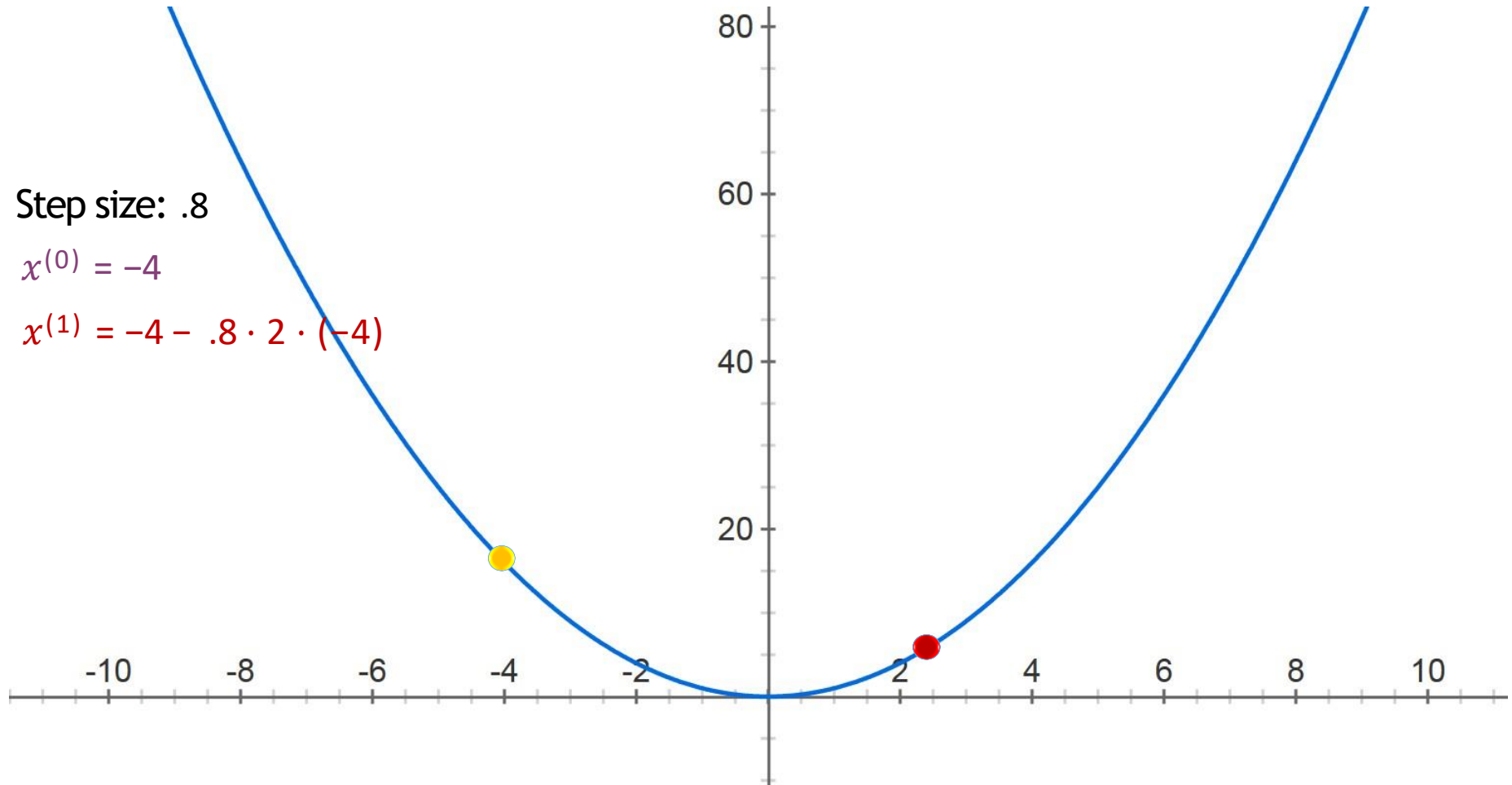
# Gradient Descent

$$f(x) = x^2$$

Step size: .8

$$x^{(0)} = -4$$

$$x^{(1)} = -4 - .8 \cdot 2 \cdot (-4)$$



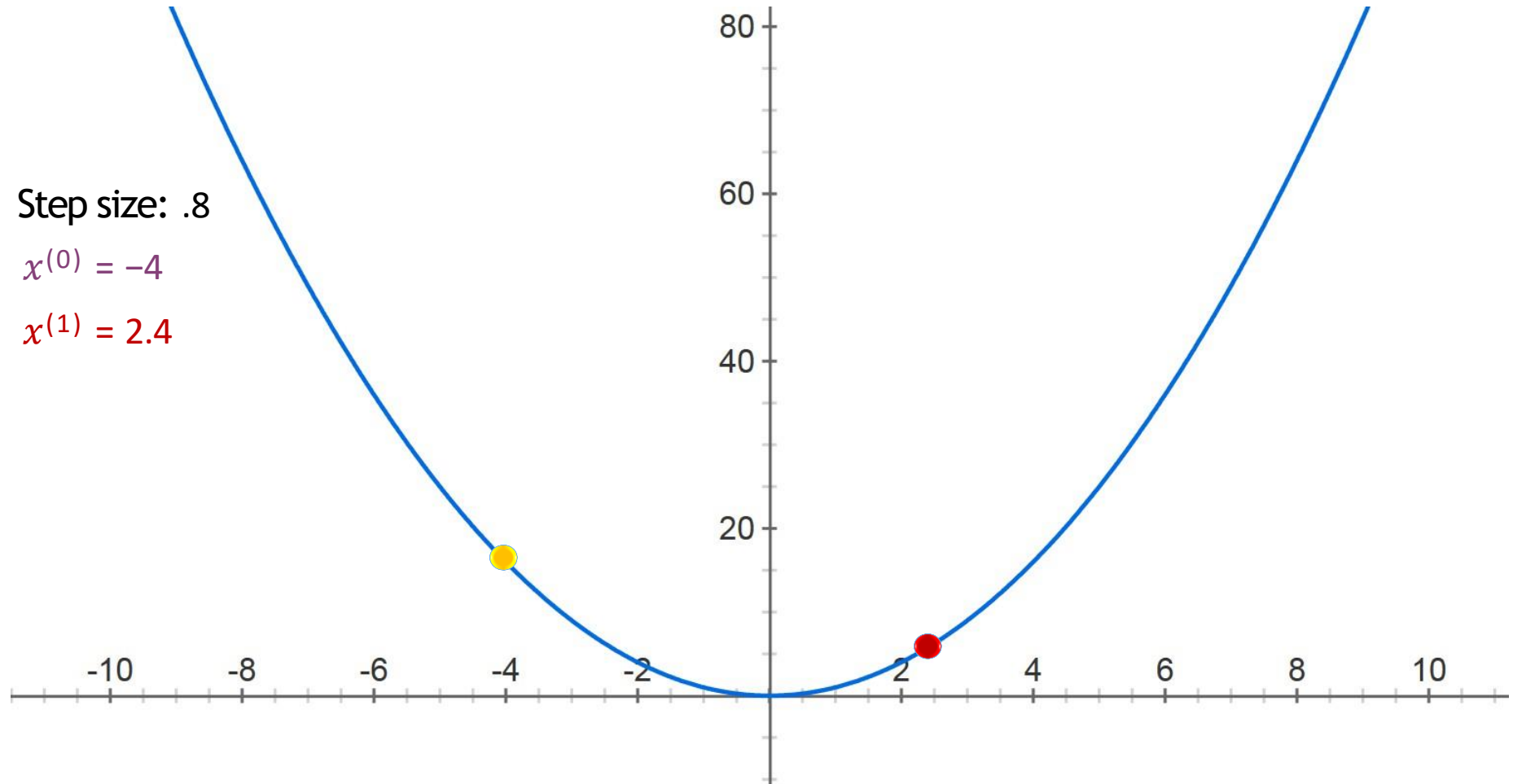
# Gradient Descent

$$f(x) = x^2$$

Step size: .8

$$x^{(0)} = -4$$

$$x^{(1)} = 2.4$$



# Gradient Descent

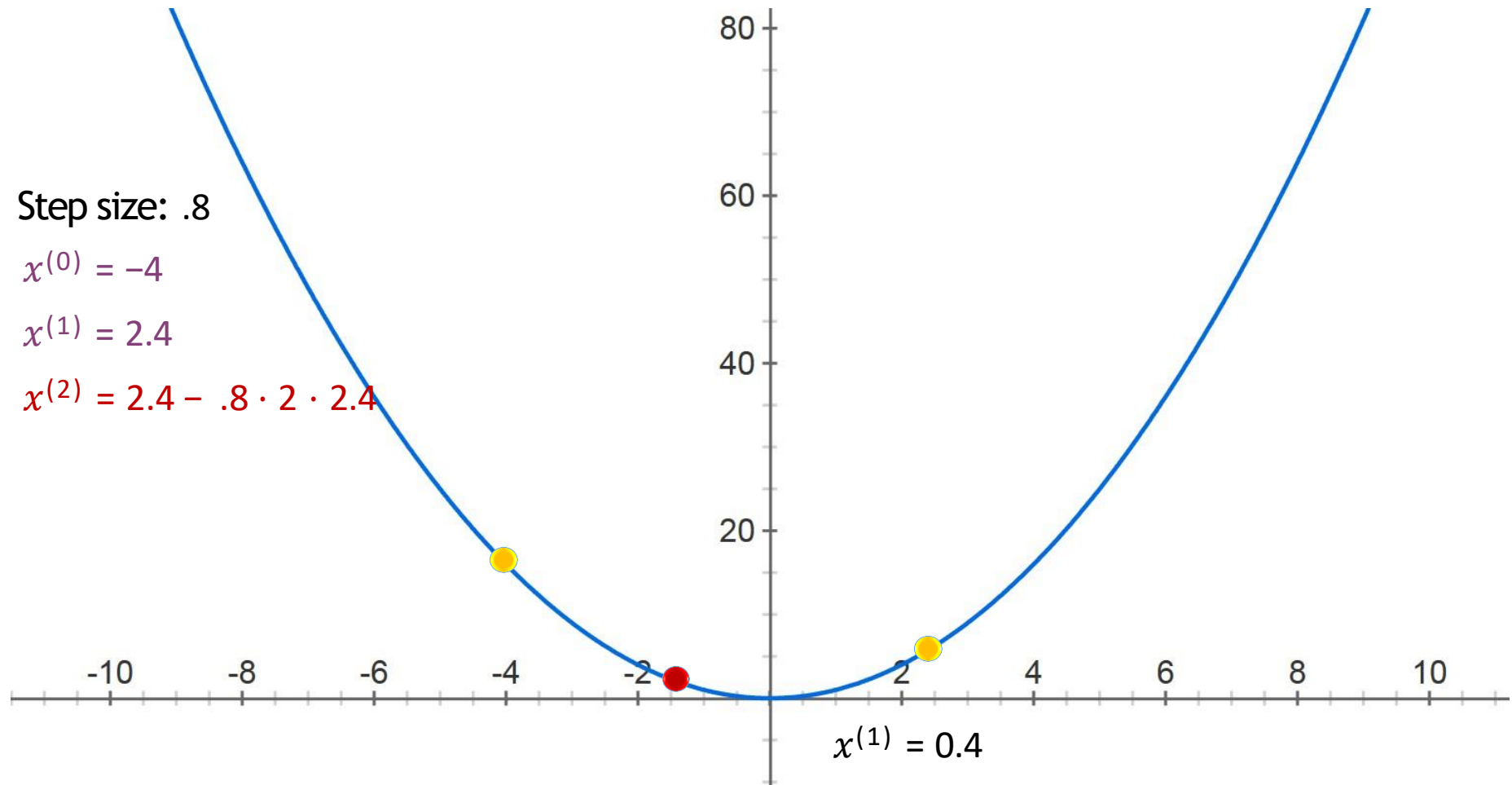
$$f(x) = x^2$$

Step size: .8

$$x^{(0)} = -4$$

$$x^{(1)} = 2.4$$

$$x^{(2)} = 2.4 - .8 \cdot 2 \cdot 2.4$$





# Gradient Descent

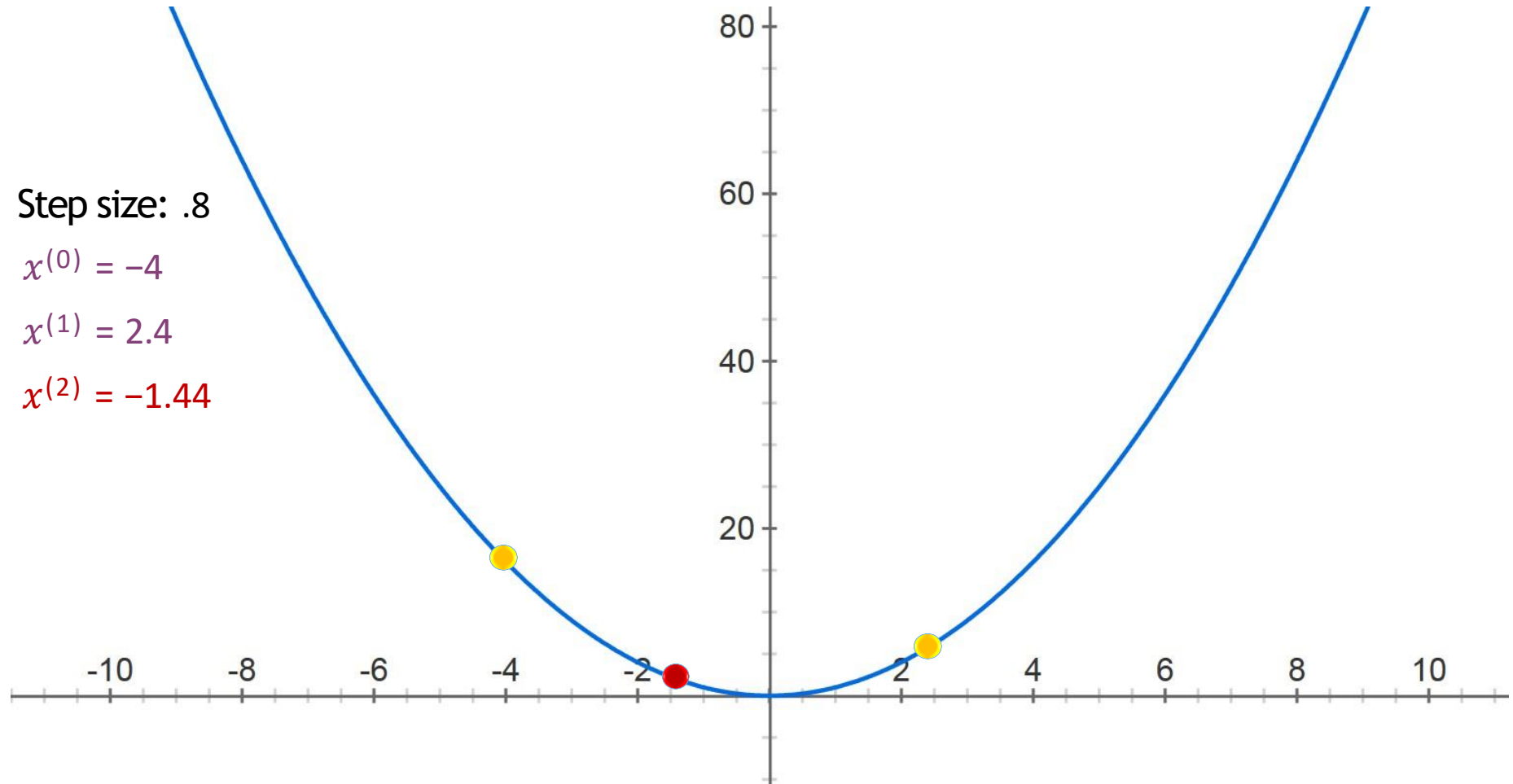
$$f(x) = x^2$$

Step size: .8

$$x^{(0)} = -4$$

$$x^{(1)} = 2.4$$

$$x^{(2)} = -1.44$$



# Gradient Descent

$$f(x) = x^2$$

Step size: .8

$$x^{(0)} = -4$$

$$x^{(1)} = 2.4$$

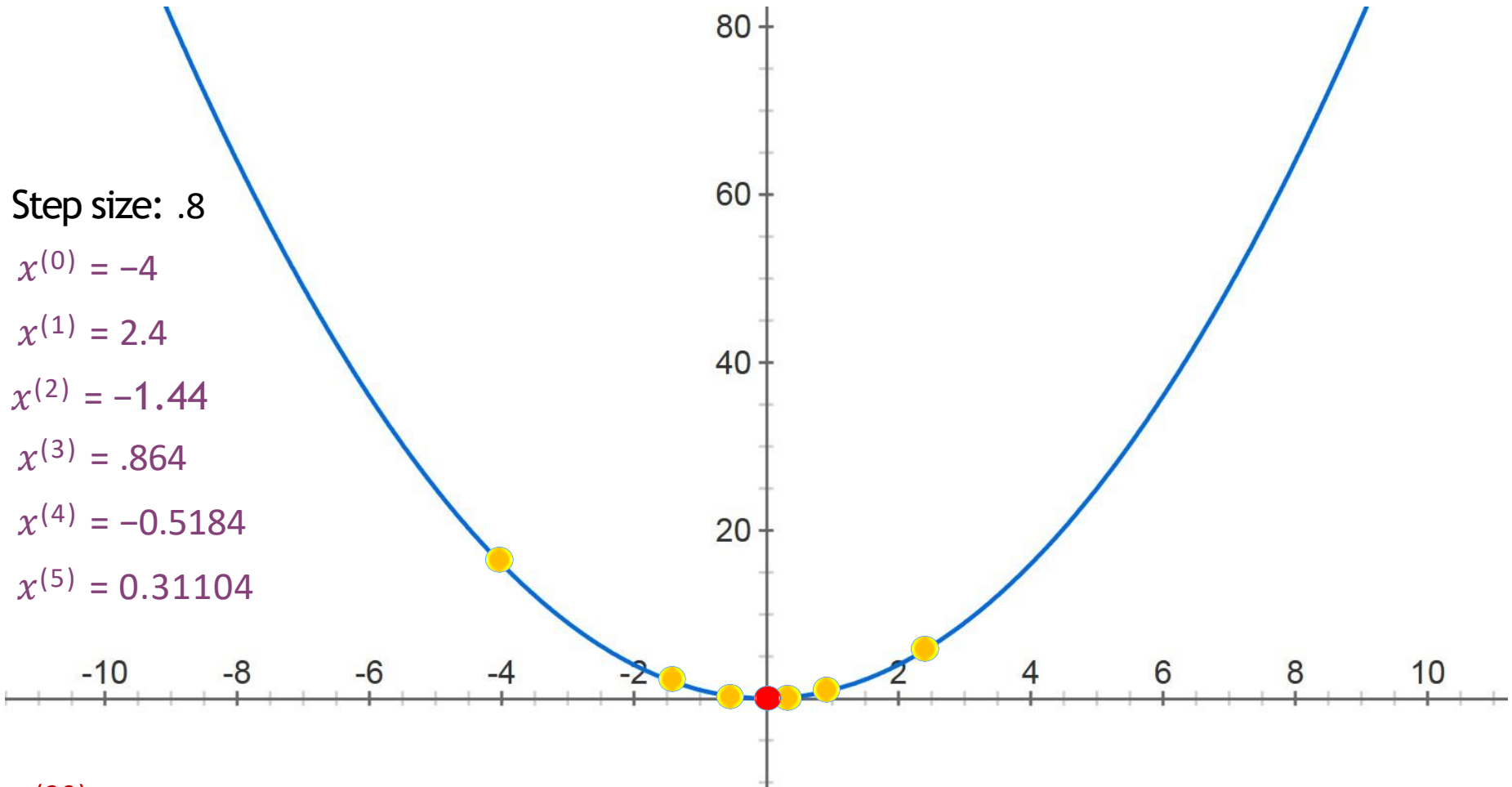
$$x^{(2)} = -1.44$$

$$x^{(3)} = .864$$

$$x^{(4)} = -0.5184$$

$$x^{(5)} = 0.31104$$

$$x^{(30)} = -8.84296e - 07$$



# Gradient Descent: Convex Functions

For convex functions, local optima are always global optima (this follows from the definition of convexity)

If gradient descent converges to a critical point, then the result is a global minimizer

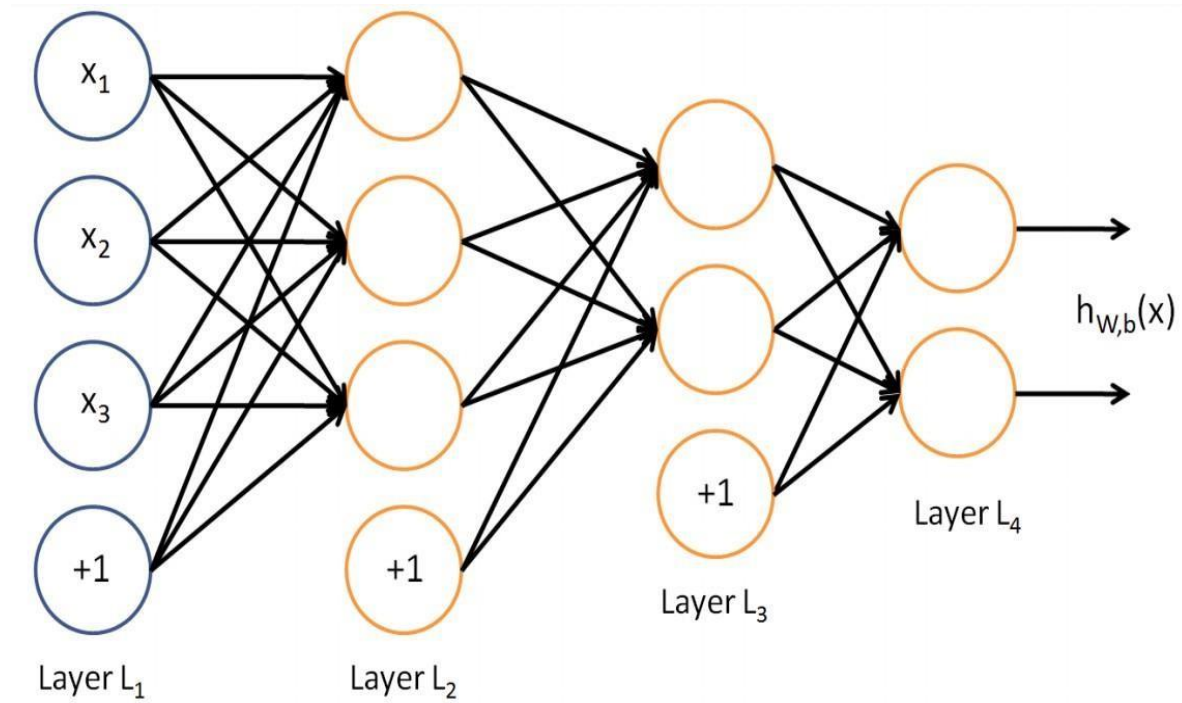
Not all convex functions are differentiable, can we still apply gradient descent?

## Diminishing Step Size Rules

- ❑ A fixed step size may not result in convergence for non-differentiable functions
- ❑ Instead, can use a diminishing step size:
  - ❑ Required property: step size must decrease as number of iterations increase but not too quickly that the algorithm fails to make progress
- ❑ Common diminishing step size rules:
  - ❑  $\gamma_t = \frac{a}{b+t}$  for some  $a > 0, b \geq 0$
  - ❑  $\gamma_t = \frac{a}{\sqrt{t}}$  for some  $a > 0$

# Gradient Descent

A feedforward neural network is a composition of multiple functions, organized as layers



What do we need to implement gradient descent? Compute gradient of loss function w.r.t. each weight in the network. How to do this?

# Gradient Descent

**Back-propagation**, a procedure which combines gradient computation using chain rule and parameter updation using Gradient Descent, thus fully describing the neural network training algorithm.

# Backpropagation

- Backpropagation is an algorithm that trains neural networks by effectively adjusting their weights and biases to minimize the error between their predictions and the desired outputs.
- It works by propagating the error back through the network, layer by layer, using the chain rule of calculus to calculate how much each weight and bias contributed to the error.

# Steps of Backpropagation

## 1. Forward Pass:

1. Input data is fed into the input layer of the network.
2. Each neuron in the network performs calculations based on its inputs and activation function, passing its output to the neurons in the next layer.
3. This process continues until the output layer produces a prediction.

## 2. Error Calculation:

1. The predicted output is compared to the actual target value, and the difference (error) is calculated using a loss function.

## 3. Backward Pass:

1. The error signal is propagated backward through the network, layer by layer, using the chain rule of differentiation.
2. At each layer, the algorithm calculates the partial derivative of the error with respect to each weight and bias.



# Steps of Backpropagation

## 4. Parameter Update:

1. The calculated partial derivatives are used to update the weights and biases of the network, moving them in a direction that reduces the error.
2. The amount of change is determined by the learning rate, which controls how quickly the network learns.

## 5. Iteration:

1. Steps 1-4 are repeated for a large number of training examples, adjusting the weights and biases after each example.
2. Over time, the network learns to make better predictions by minimizing the error on the training data.

# Backpropagation

Consider a simple feed forward neural network (or multilayer perceptron)

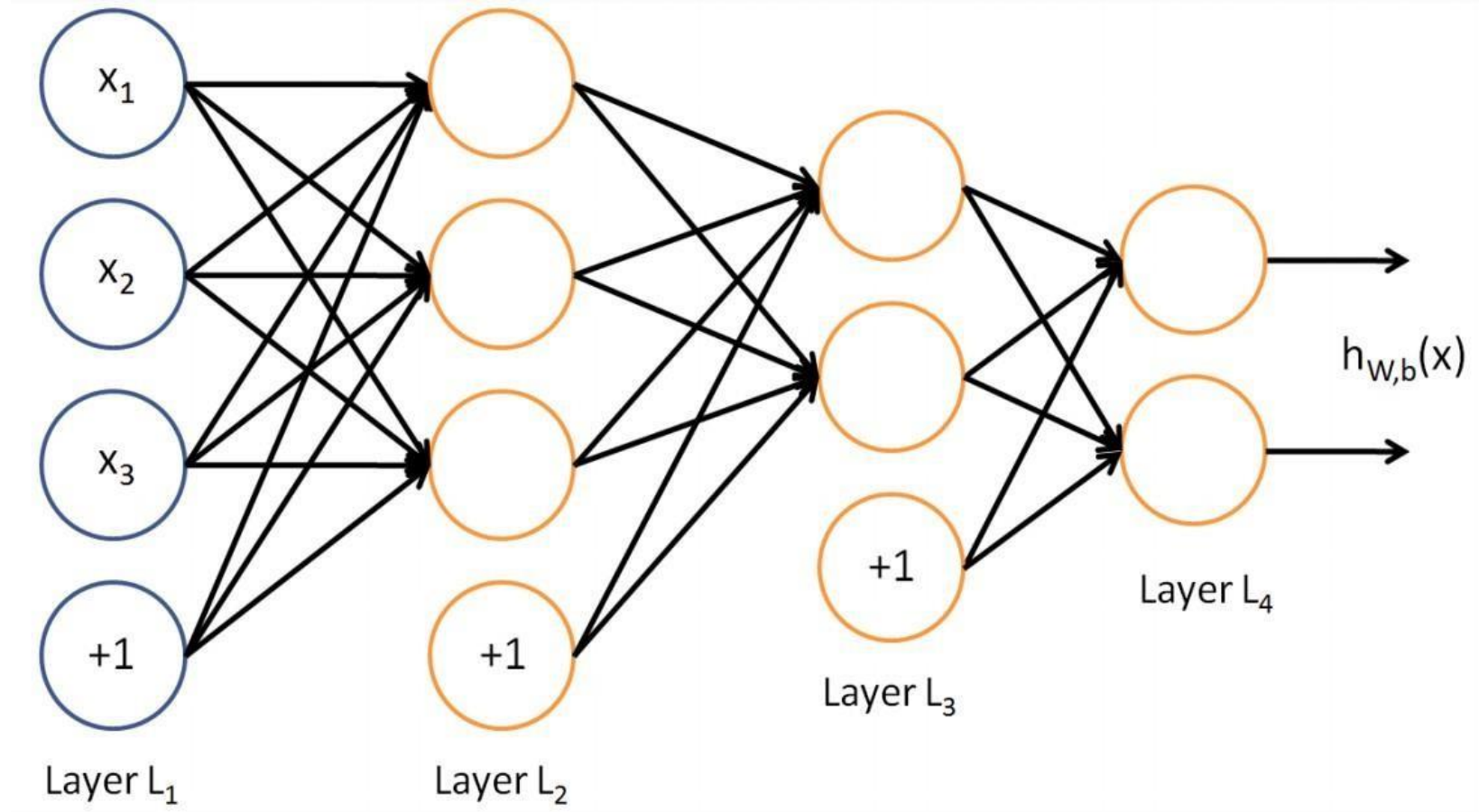


Figure Credit: Stanford UFLDL Tutorial

# Backpropagation

- A fixed training set  $\{x^{(i)}, y^{(i)}\}_{i=1}^M$  of  $M$  training samples
- Parameters  $= \theta \{W, b\}$ , weights and biases

# Backpropagation

- A fixed training set  $\{x^{(i)}, y^{(i)}\}_{i=1}^M$  of  $M$  training samples
- Parameters  $\theta = \{W, b\}$ , weights and biases
- Mean square cost function for a single example:

$$L(\theta; x, y) = \frac{1}{2} \|h_{\theta}(x) - y\|^2$$

- Overall cost function is given by:

$$\begin{aligned} L(\theta) &= \frac{1}{M} \sum_{i=1}^M L(\theta; x^{(i)}, y^{(i)}) \\ &= \frac{1}{2M} \sum_{i=1}^M \|h_{\theta}(x^{(i)}) - y^{(i)}\|^2 \end{aligned}$$

# Backpropagation: Notations

- We have  $n_l$  layers in the network,  $l = 1, 2, \dots, n_l$

# Backpropagation: Notations

- We have  $n_l$  layers in the network,  $l = 1, 2, \dots, n_l$
- We denote activation of node  $i$  at layer  $l$  as  $a_i^{(l)}$

# Backpropagation: Notations

- We have  $n_l$  layers in the network,  $l = 1, 2, \dots, n_l$
- We denote activation of node  $i$  at layer  $l$  as  $a_i^{(l)}$
- We denote weight connecting node  $i$  in layer  $l$  and node  $j$  in layer  $l + 1$  as  $w_{ij}^{(l)}$ . The weight matrix between layer  $l$  and layer  $l + 1$  is denoted as  $W^{(l)}$

# Backpropagation: Notations

We have  $n_l$  layers in the network,  $l = 1, 2, \dots, n_l$

We denote activation of node  $i$  at layer  $l$  as  $a_i^{(l)}$

We denote weight connecting node  $i$  in layer  $l$  and node  $j$  in layer  $l + 1$  as  $w_{ij}^{(l)}$ . The

weight matrix between layer  $l$  and layer  $l + 1$  is denoted as  $W^{(l)}$

- For a 3-layer network shown earlier, compact vectorized form of a **forward pass** to compute neural network's output is shown below:

$$z^{(2)} = W^{(1)}x + b^{(1)}$$

$$a^{(2)} = f(z^{(2)})$$

$$z^{(3)} = W^{(2)}a^{(2)} + b^{(2)}$$



# Backpropagation: Notations

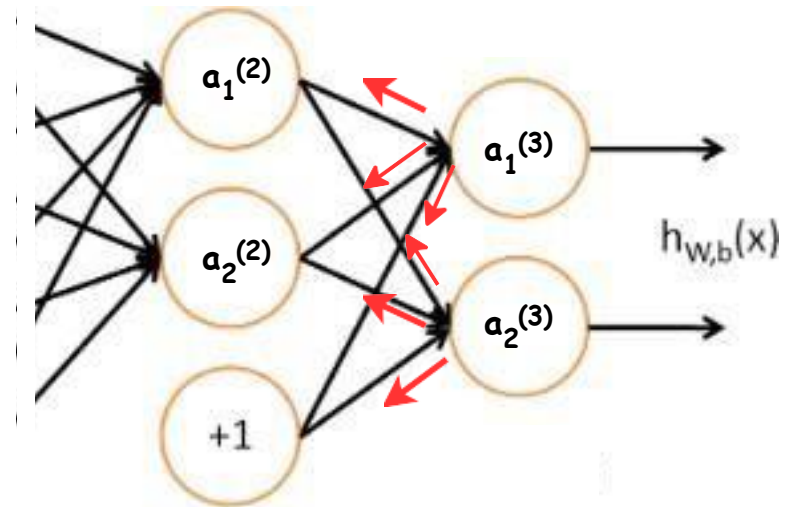
- We have  $n_l$  layers in the network,  $l = 1, 2, \dots, n_l$
- We denote activation of node  $i$  at layer  $l$  as  $a_i^{(l)}$
- We denote weight connecting node  $i$  in layer  $l$  and node  $j$  in layer  $l + 1$  as  $w_{ij}^{(l)}$ . The weight matrix between layer  $l$  and layer  $l + 1$  is denoted as  $W^{(l)}$
- For a 3-layer network shown earlier, compact vectorized form of a **forward pass** to compute neural network's output is shown below:

$$\begin{aligned}z^{(2)} &= W^{(1)}x + b^{(1)} \\a^{(2)} &= f(z^{(2)}) \\z^{(3)} &= W^{(2)}a^{(2)} + b^{(2)} \\h(x) &= a^{(3)} = f(z^{(3)})\end{aligned}$$

- Function  $f$  can denote any activation function such as sigmoid, ReLU, identity, etc.

# Backpropagation: Backward Pass

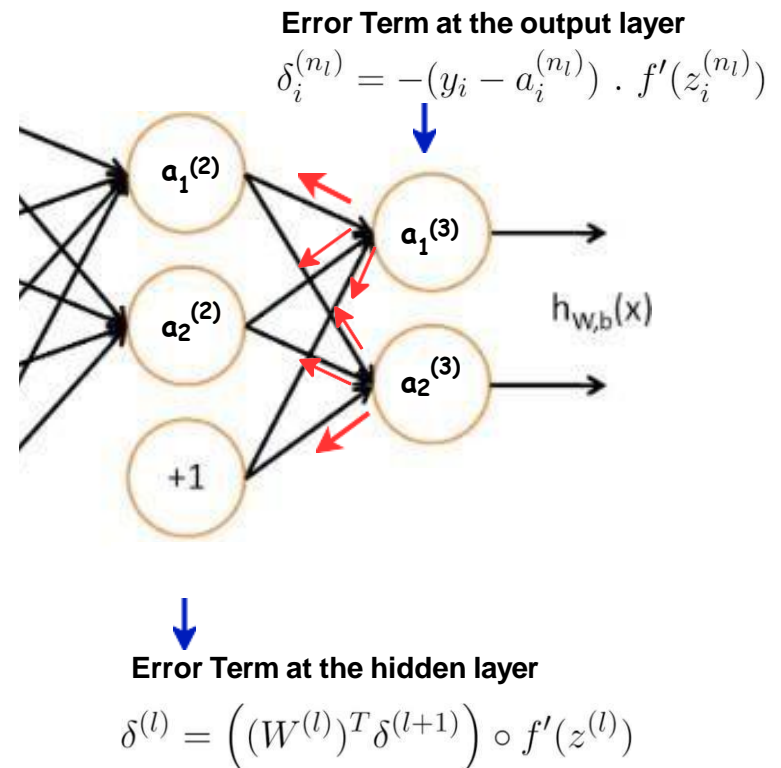
- During the forward pass, we successively compute each layer's outputs from left to right.
- During backward pass, we aim to compute derivatives of each parameter starting from the right most layer to the left most one i.e., layer  $n, n-1, \dots, 1$ .
- Once the derivatives are computed, we use Gradient Descent to update the parameters.



# Backpropagation: Backward Pass

- For each node, we define an **error term**  $\delta_i^{(l)}$  to denote how much the node was responsible for the loss computed
- If  $l = n_l$  i.e., **last layer**, error term computation is straightforward, since we directly take derivative of loss function (MSE, in this case, between output and target values)

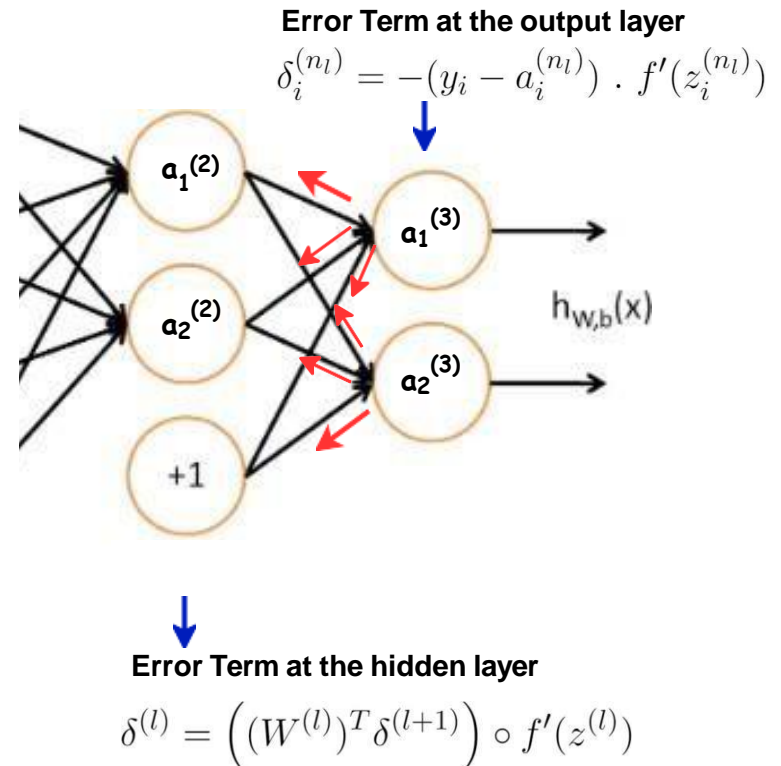
$$\delta_i^{(n_l)} = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)})$$



# Backpropagation: Backward Pass

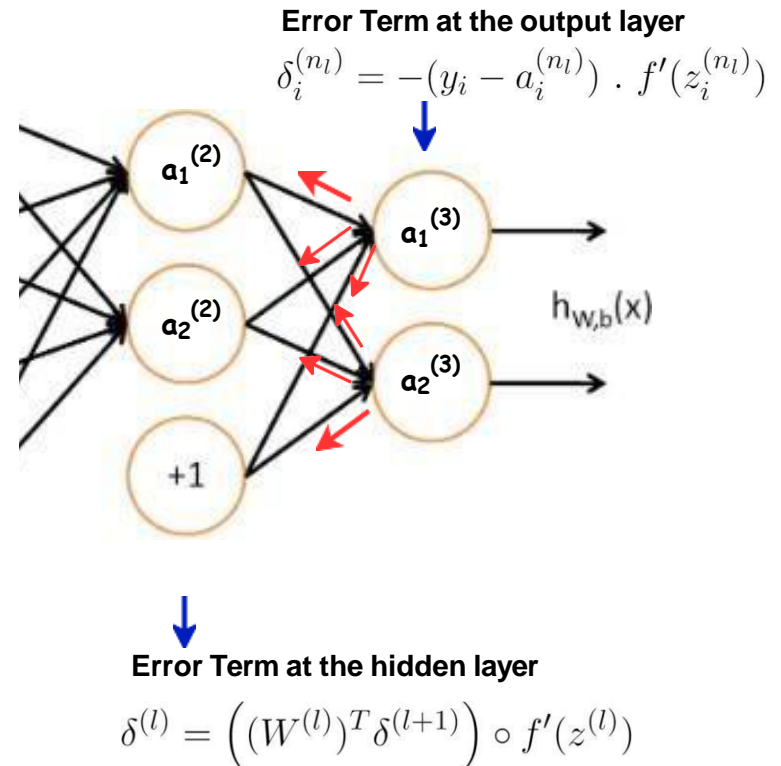
- To compute error term for **hidden layers**,  $l = n_l - 1, n_l - 2, \dots$ , we rely on error terms from subsequent layers
- In particular, we compute error term as sum of error terms in next layer, weighted by weights along connections to next layer:

$$\delta_i^{(l)} = \left( \sum_{j=1}^{n_{l+1}} W_{ij}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$$



# Backpropagation: Backward Pass

- Note that  $f'$  denotes derivative of activation function
- For a linear neuron  $f(x) = x$ , derivative is 1
- For a sigmoid neuron  
 $f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$ , derivative turns out to be  $\sigma(x)(1 - \sigma(x))$



# Backpropagation: Algorithm

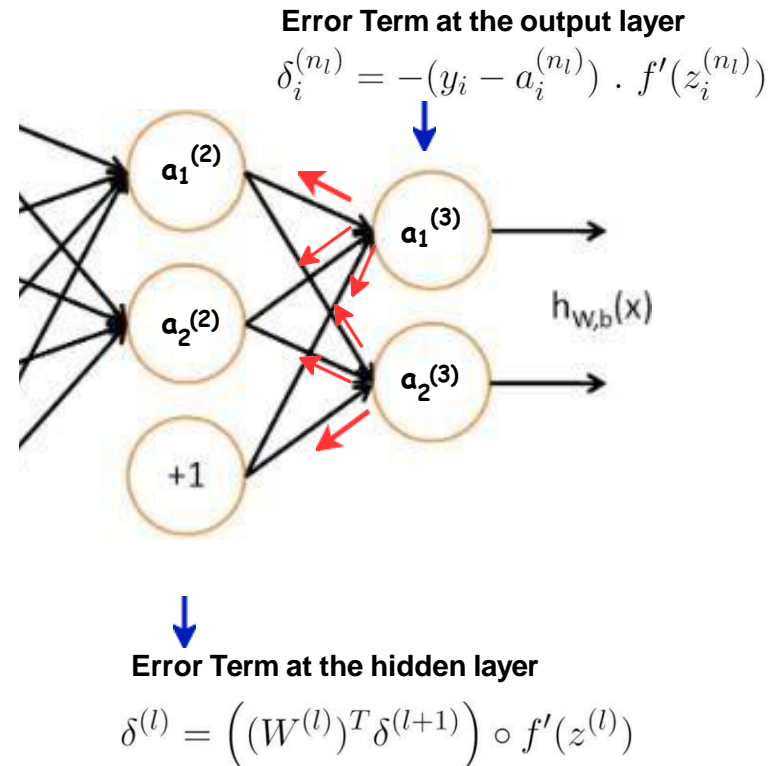
- Perform a feedforward pass, computing the activations for layers 1, 2, ...  $n_l$ .

- For each output unit  $i$  in layer  $n_l$  set,

$$\delta^{(n_l)} = -(y_i - a_i^{(n_l)}) \odot f'(z_i^{(n_l)})$$

- For  $l = n_l - 1, n_l - 2, n_l - 3, \dots, 2$
  - For each node in layer  $l$  set,

$$\delta^{(l)} = \left( (W^{(l+1)})^T \delta^{(l+1)} \right) \odot f'(z^{(l)})$$

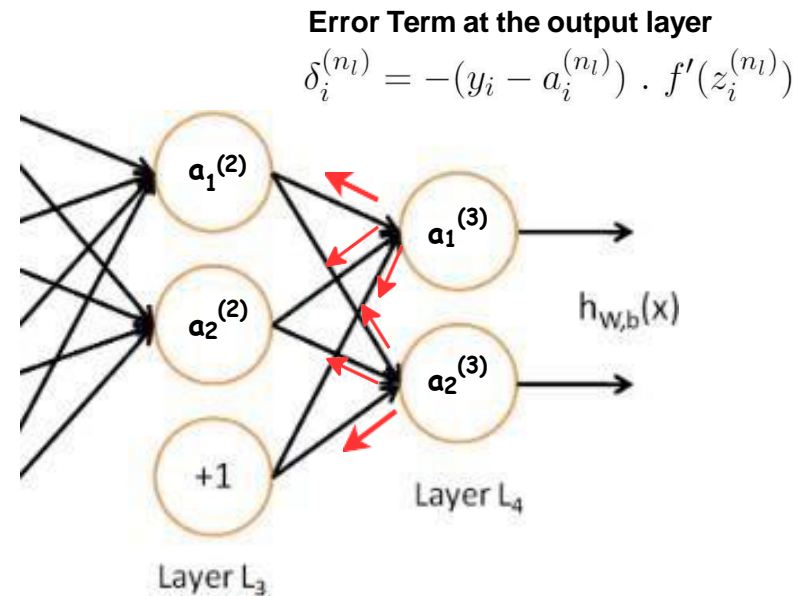


# Backpropagation: Algorithm

- Compute the desired partial derivatives, as:

$$\nabla_{W^{(l)}} L(W, b; x, y) = \delta^{(l+1)} (a^{(l)})^T$$

$$\nabla_{b^{(l)}} L(W, b; x, y) = \delta^{(l+1)}$$



**Error Term at the hidden layer**

$$\delta^{(l)} = \left( (W^{(l)})^T \delta^{(l+1)} \right) \circ f'(z^{(l)})$$

# Gradient Descent using Backpropagation

- ① Set  $\Delta W^{(l)} := 0, \Delta b^{(l)} = 0$  (matrix/vector of zeros) for all  $l$ .
- ② For  $i = 1$  to  $M$ 
  - ① Use backpropagation to compute  $\nabla_{\theta^{(l)}} L(\theta; x, y)$ .
  - ② Set  $\Delta \theta^{(l)} := \Delta \theta^{(l)} + \nabla_{\theta^{(l)}} L(\theta; x, y)$
- ③ Update the parameters:

$$W^{(l)} = W^{(l)} - \eta \left[ \frac{1}{M} \Delta W^{(l)} \right]$$
$$b^{(l)} = b^{(l)} - \eta \left[ \frac{1}{M} \Delta b^{(l)} \right]$$

- ④ Repeat for all points until convergence.



# 1.1 Math Behind Machine Learning: Statistics, Probability, Conditional Probabilities, Posterior Probability, Distributions, Samples Versus Population, Resampling Methods, Selection Bias, Likelihood

## Math behind Machine Learning

### Statistics Definition:

- Statistics involves collecting, analyzing, interpreting, and presenting data.
- It's a foundation for understanding patterns and making data-driven decisions.

### Key Concepts:

- Descriptive Statistics: Summarizes data (mean, median, mode, variance).
- Inferential Statistics: Draws conclusions from data samples about a population.

### Applications in Machine Learning:

- Feature selection and scaling, Model performance evaluation.

### Important Techniques:

- Sampling: Ensuring data is representative of the population.
- Hypothesis Testing: Testing assumptions about data.
- Statistical Measures: Correlation, standard deviation, probability distributions.

### Why It Matters?

- Helps identify data trends, anomalies, and patterns essential for model training and validation.

Probability The measure of the likelihood of an event occurring, expressed as a value between 0 and 1.

- ❑ Probability- Probability is the ratio of favorable outcomes to the total number of outcomes; Odds is the ratio of favorable outcomes to unfavorable outcomes.
- ❑ Probability Vs Odds-
- ❑ Conditional Probabilities-the probability of a given event based on the existing presence of another event occurring
  - ❑  $P(E | F)$  where:
    - ❑ E is the event for which we're interested in a probability.
    - ❑ F is the event that has already occurred.

# Probability

Bayes's Theorem provides a way to calculate the probability of an event A, given that another event B has occurred.

## Conditional Probabilities-

The formula is:

**BAYES'S THEOREM**

$$P(A | B) = \frac{P(B | A) P(A)}{P(B)}$$

$P(A|B)$ : The probability of A happening, given that B has occurred.

$P(B|A)$ : The probability of B happening, given that A has occurred.

$P(A)$ : The prior probability of A happening on its own.

$P(B)$ : The prior probability of B happening on its own.

# Probability

❓ **Posterior Probability-** It reflects how likely the event is, given what we now know.

❓ conditional probability we assign after the evidence is considered.

❓ Posterior probability distribution is defined as the probability distribution of an unknown quantity conditional on the evidence collected from an experiment treated as a random variable

Bayes's Theorem directly helps calculate the posterior probability.

# Distributions

## ? Continuous -

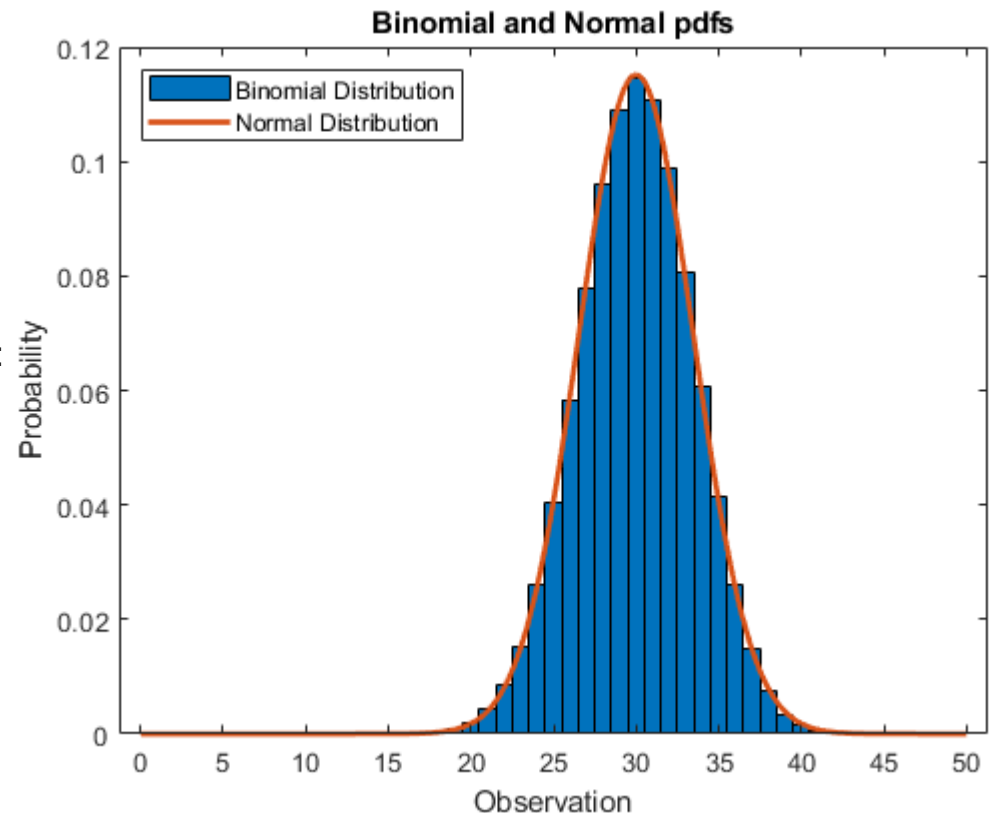
- ? Defined by mean and std deviation
- ? normal distribution

## ? Discrete- binomial distribution

Probability (y-axis):  
The likelihood of  
each observation.

The graph compares:

- Binomial Distribution (blue bars): Shows the probabilities of discrete outcomes.
- Normal Distribution (orange line): A smooth, continuous curve that often approximates the binomial distribution when the sample size is large.



Observation (x-axis): The outcomes being analyzed.

## Samples Versus Population

- ❑ Population - All of the units we'd like to study or model in our experiment.
- ❑ Sample- Subset of the population of data that hopefully represents the accurate distribution of the data without introducing sampling bias

# Resampling Methods

## ❓ Bootstrapping-

- ❓ Random samples drawn from another sample to generate a new sample that has a balance between the number of samples per class.
- ❓ Used to develop a model against a dataset with highly unbalanced classes.

What is Bootstrapping?

- It's a statistical method where random samples are repeatedly drawn from an existing sample (with replacement; after selecting a data point, it is put back into the "pool," so it remains available for future selections.) to create a new dataset.

- The aim is to ensure a balance between the number of samples per class.

Why is it Used?

- This technique is especially helpful when dealing with imbalanced datasets, where certain classes have significantly more data than others.

- Bootstrapping helps to generate balanced data, making models less biased towards the dominant class.

Application in Machine Learning

- Used to train robust models on datasets with uneven class distributions.

- Ensures that minority classes contribute equally during training, which improves model accuracy and fairness.

# Resampling Methods

Cross-validation is a technique for evaluating how well a model generalizes to unseen data. The process involves:

1. Splitting the training dataset into multiple partitions (commonly referred to as splits).
2. Alternating these splits between training and testing groups.
3. Iteratively training on one group of splits while testing on the others, ensuring all possible combinations are covered.

This approach reduces the risk of overfitting and provides a reliable estimate of model performance. Although there's no fixed rule for the number of splits, research suggests that using 10 splits is effective. Additionally, a portion of the held-out data is often used as a validation set during training for further tuning.

## ❓ Cross-validation-

- ❓ Method used to estimate how well a model generalizes on a training dataset
- ❓ Split the training dataset into N number of splits and then separate the splits into training and test groups.
- ❓ Train on the training group of splits and then test the model on the test group of splits.
- ❓ Rotate the splits between the two groups many times until we've exhausted all the variations.
- ❓ No hard number for the number of splits to use but researchers have found 10 splits to work well in practice.
- ❓ Common approach - Use a separate portion of the held- out data used as a validation dataset during training.



# Resampling Methods

Method	Definition	Limitations
<b>Holdout</b>	The original data is divided into two sets, training and test set. The model is induced using the training set and then its performance is evaluated using the test set. The accuracy of the induced model on the test set is used to estimate the accuracy of the classifier	<ol style="list-style-type: none"> <li>1. Because the data was divided into two sets fewer label examples are available for training. As a result, the induced model may not be as good</li> <li>2. It may be highly dependent on the composition of the training and test sets.</li> <li>3. The two sets are no longer independent because they are subsets of a larger set.</li> </ol>
<b>Random Subsampling</b>	Repeats the holdout method several times to improve estimation of a classifier's performance. Overall accuracy, $acc_{sub}$ is based on the accuracy for each run, $acc_i$ . $acc_{sub} = \sum_{i=1}^k acc_i / k$	<ol style="list-style-type: none"> <li>1. Still encounters some of the holdout problems.</li> <li>2. Some records might be used for training more often than others.</li> </ol>
<b>Cross-Validation</b>	<p>Partitions the data into k disjoint subset. It then uses k-1 subsets for the training model and one set for the test set. It does this k number of times switching the test set each time thereby each record is used the same number of times for training and exactly one time for testing. The total error is found by summing the errors of all k runs.</p> <p>Leave-one-out k=n:-basically all the data except for one is used for the training set and the one is used for the test set. This is good because it uses as much data as possible for training</p>	<ol style="list-style-type: none"> <li>1. Training algorithm has to be rerun from scratch k times.</li> <li>2. When the data set size is small splitting it compromises it integrity.</li> </ol>
<b>Bootstrap</b>	The training records are sampled with replacement. "A common variation used is the .632 bootstrap which computes the overall accuracy by combining the accuracies of each bootstrap sample with the accuracy computed from a training set that contains all the labeled examples in the original data" <sup>10</sup>	<ol style="list-style-type: none"> <li>1. Bootstrap can yield poor results in certain situations</li> <li>2. Though the bootstrap easily accommodates some violations of traditional statistical assumptions (e.g., non-normality), it is susceptible to others (e.g., non-independence)<sup>2</sup>.</li> </ol>

# Selection Bias

- ❓ Sampling method that does not have proper randomization and skews the sample in a way such that the sample is not representative of the population we'd like to model.
- ❓ Possibility of introducing bias into our models that will lower our model's accuracy on data from the larger population

Selection bias arises when certain groups or subsets of the population are overrepresented or underrepresented during sampling. This flawed representation distorts the conclusions drawn from data analysis, as the sample no longer mirrors the true population. In machine learning, such bias can cause models to misinterpret patterns, reduce accuracy, and fail to generalize effectively to unseen data.

# Likelihood

- ❓ Likeliness that an event will occur yet do not specifically reference its numeric probability.
- ❓ About an event that has a reasonable probability of happening but still might not.
- ❓ Informally, likelihood is also used as a synonym for probability.

Real-World Example: The likelihood of it raining tomorrow can be discussed based on weather patterns, even if we don't know the exact probability.

Comparison with Probability: While probability focuses on the chance of an event occurring, likelihood often emphasizes the plausibility of an event based on observed data.

Context in Statistics or Machine Learning: In statistics, likelihood measures how well a model explains observed data. In Maximum Likelihood Estimation (MLE), parameters are determined to maximize the likelihood of observed outcomes.

# Evaluating Models

## Confusion matrix

	Positive prediction Label was positive		Negative prediction Label was positive	
	P' (Predicted)		N' (Predicted)	
P (Actual)	True Positive		False Negative	
N (Actual)	False Positive	Positive prediction Label was negative	True Negative	Negative prediction Label was negative

# Evaluating Models

- ❑ False positive - “type I error”
- ❑ False negative - “type II error”

# Evaluating Models

## ❓ Sensitivity versus specificity-

- ❓ two different measures of a binary classification model. -

### ❓ Sensitivity-

- ❓ The true positive rate measures how often we classify an input record as the positive class and its correct classification.
- ❓ This also is called sensitivity ,or recall;.
- ❓ Sensitivity quantifies how well the model avoids false negatives.
- ❓  $\text{Sensitivity} = \text{TP} / (\text{TP} + \text{FN})$

# Evaluating Models

## ☐ Sensitivity versus specificity-

☐ **Specificity**-Specificity quantifies how well the model avoids false positives.

☐  $\text{Specificity} = \text{TN} / (\text{TN} + \text{FP})$

# Evaluating Models

## ❑ Accuracy-

- ❑ Accuracy is the degree of closeness of measurements of a quantity to that quantity's true value.
- ❑  $\text{Accuracy} = (\text{TP} + \text{TN}) / (\text{TP} + \text{FP} + \text{FN} + \text{TN})$
- ❑ Accuracy can be misleading in the quality of the model when the class imbalance is high.



# Evaluating Models

## ❓ Precision

- ❓ The degree to which repeated measurements under the same conditions give us the same results in the context of science and statistics.
- ❓ Positive prediction value.
- ❓  $\text{Precision} = \text{TP} / (\text{TP} + \text{FP})$
- ❓ A measurement can be accurate yet not precise, not accurate but still precise, neither accurate nor precise, or both accurate and precise.
- ❓ We consider a measurement to be valid if it is both accurate and precise.

# Evaluating Models

## ❑ Recall-

- ❑ Same as sensitivity and is also known as the true positive rate or the hit rate.

## ❑ F1 score-

- ❑ In binary classification we consider the F1 score (or F- score, F-measure) to be a measure of a model's accuracy.
- ❑ Harmonic mean of both the precision and recall measures (described previously) into a single score:
- ❑ 
$$F1 = 2TP / (2TP + FP + FN)$$

# Gradient Descent and Variants



## 2.3 Gradient-Based Learning(extended)



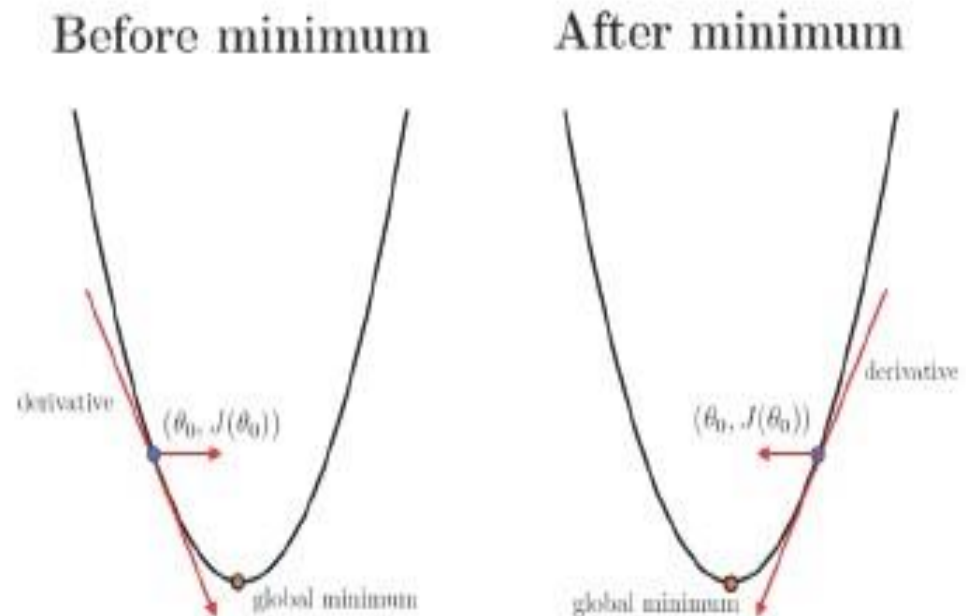
## 2.3 Gradient-Based Learning (Extended)

### Review: Gradient Descent (GD)

- Optimization algorithm used to find minima of a given differentiable function
- At each step, parameters ( $\theta$ ) are pushed in negative direction of gradient of a cost function ( $J(\theta; x, y)$ , in figure alongside) w.r.t parameters

$$\theta_{new} = \theta_{old} - \alpha \Delta \theta_{old}$$

where  $\alpha$  is learning rate



Since the derivative is negative, if we subtract the derivative from  $\theta_0$ , it will increase and go closer the minimum.

Since the derivative is positive, if we subtract the derivative from  $\theta_0$ , it will decrease and go closer the minimum.

# Gradient Descent Algorithm

---

**Require:** Learning rate  $\alpha$ , initial parameters  $\theta_t$ , training dataset  $\mathcal{D}_{tr}$

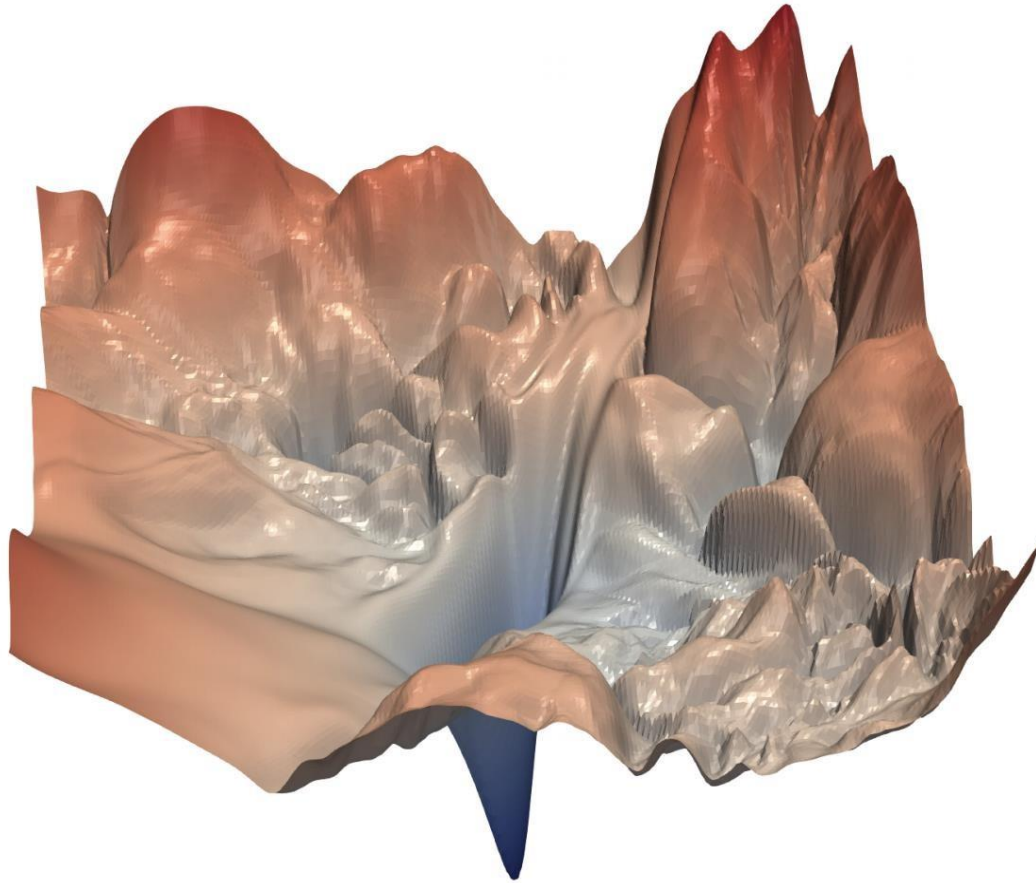
---

```
1: while stopping criterion not met do
2:   Initialize parameter updates  $\Delta\theta_t = 0$ 
3:   for each  $(x^{(i)}, y^{(i)})$  in  $\mathcal{D}_{tr}$  do
4:     Compute gradient using backpropagation  $\nabla_{\theta_t} \mathcal{L}(\theta_t; x^{(i)}, y^{(i)})$ 
5:     Aggregate gradient  $\Delta\theta_t = \Delta\theta_t + \nabla_{\theta_t} \mathcal{L}$ 
6:   end for
7:   Apply update  $\theta_{t+1} = \theta_t - \alpha \frac{1}{|\mathcal{D}_{tr}|} \Delta\theta_t$ 
8: end while
```

---

# Error Surface of Neural Networks

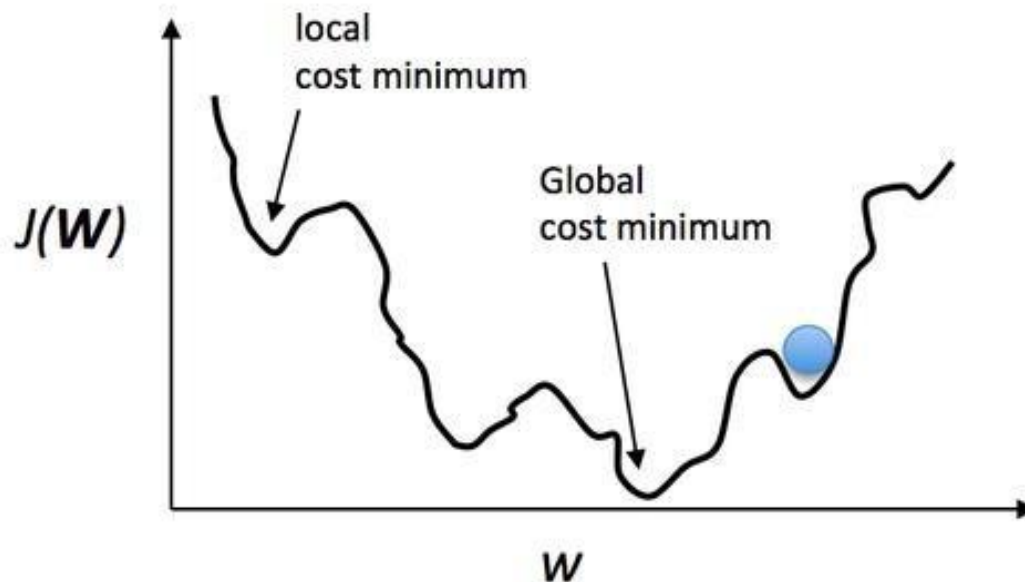
- Visualization of error surface of a neural network (ResNet-56)



Credit: Li et al, Visualizing the Loss Landscape of Neural Nets, NeurIPS 2018

# Local Minima

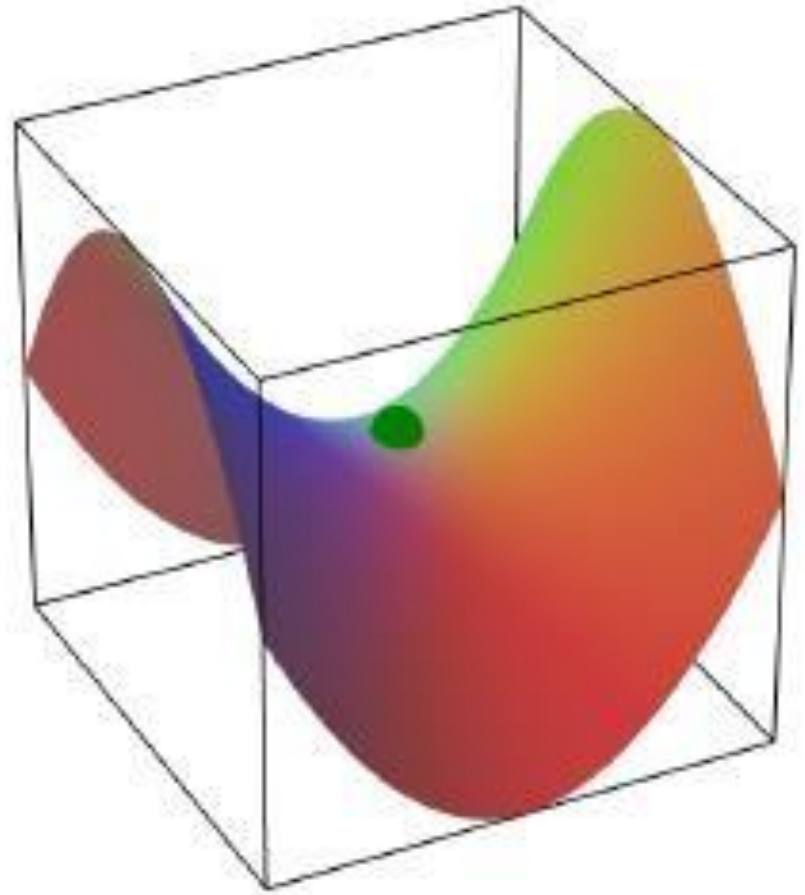
- ❓ Unlike convex objective functions that have a global minimum, non-convex functions as in deep neural networks have multiple local minima<sup>1</sup>



<sup>1</sup>Choromanska et al, The Loss Surface of Multilayer Nets, AISTATS 2015

# Saddle Points

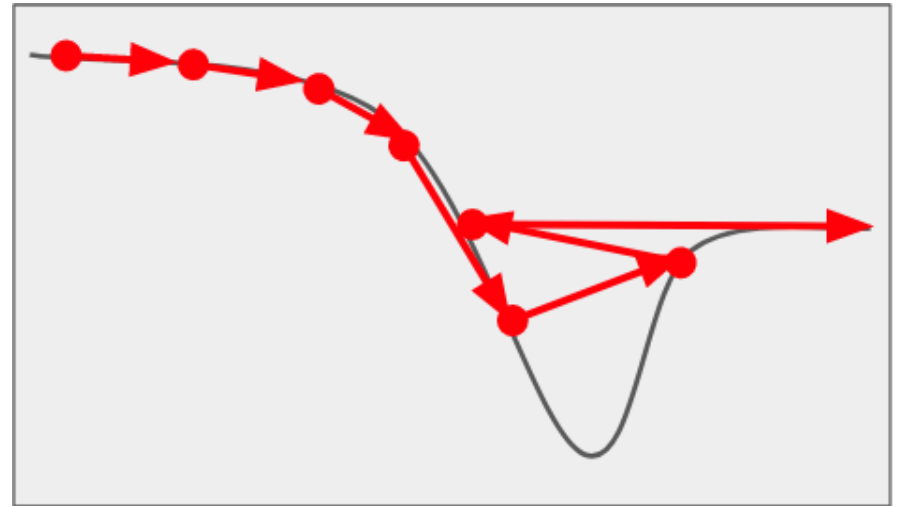
- ❑ Local maximum along one cross-section of cost function and local minimum along another
- ❑ Though it isn't a local minimum, gradient is zero (or almost close to zero) =) gives impression of convergence
- ❑ Though local minima are prevalent in lower dimensional spaces, saddle points become more common in higher-dimensional spaces





# Plateaus and Flat Regions

- ❓ They constitute portions of error surface where gradient is highly non-spherical
- ❓ Gradient descent spends a long time traversing in these regions as the updates are small
- ❓ How about increasing the learning rate?
- ❓ Though traversal becomes faster in plateaus, there is a risk of divergence



Momentum helps the optimization process by accelerating the descent along relevant directions and dampening oscillations in irrelevant directions. The idea is inspired by the concept of momentum in physics: when an object is moving, it tends to keep moving in the same direction unless acted upon by an external force.

## Momentum-based G D

- Intuition: With increasing confidence, increase step size; and with decreasing confidence, decrease step size
- Weight update given by:

Momentum  
Term

$$v_t = \gamma v_{t-1} + \alpha \nabla_{\theta_t} \mathcal{L}(\theta_t; x^{(i)}, y^{(i)})$$

$$\theta_{t+1} = \theta_t - v_t$$

# Momentum-based GD: Algorithm

---

**Require:** Learning rate  $\alpha$ , momentum parameter  $\gamma$ , initial parameters  $\theta_t$ , training dataset  $\mathcal{D}_{tr}$

---

```
1: Initialize  $v_{t-1} = 0$ 
2: while stopping criterion not met do
3:   Initialize weight updates  $\Delta\theta_t = 0$ 
4:   for each  $(x^{(i)}, y^{(i)})$  in  $\mathcal{D}_{tr}$  do
5:     Compute gradient using backpropagation  $\nabla_{\theta_t} \mathcal{L}(\theta_t; x^{(i)}, y^{(i)})$ 
6:     Aggregate weight updates:  $\Delta\theta_t = \Delta\theta_t + \nabla_{\theta_t} \mathcal{L}$ 
7:   end for
8:   Update velocity:  $v_t = \gamma v_{t-1} + \alpha \Delta\theta_t$ 
9:   Apply update:  $\theta_{t+1} = \theta_t - v_t$ 
10: end while
```

---

# Nesterov Accelerated Momentum

Nesterov, however, anticipates where the current parameters ( $\theta_t$ ) will move based on the momentum, and evaluates the gradient at that future point. This helps in making better-informed updates to the parameters.

- [?] Based on Nesterov's Accelerated Gradient Descent published

- in 1983; re-introduced by

- [?] Sutskever in ICML'13

- [?] Key idea: Look before you leap

- [?] Assess how gradient changes after taking a step of momentum  $v_t$  and use this to get better

es 
$$v_t = \gamma v_{t-1} + \alpha \nabla_{\bar{\theta}_t} \mathcal{L}(\theta_t - \gamma v_{t-1}; x^{(i)}, y^{(i)})$$

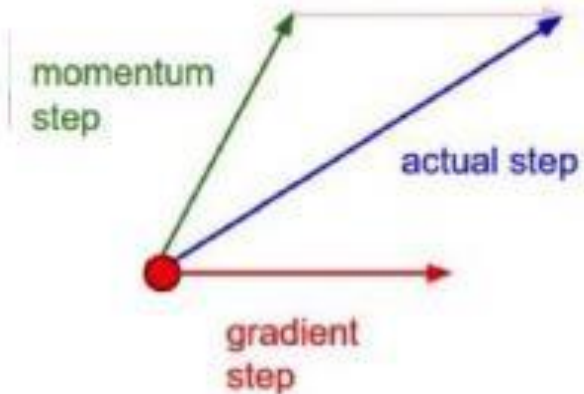
- [?]

$$\theta_{t+1} = \theta_t - v_t$$

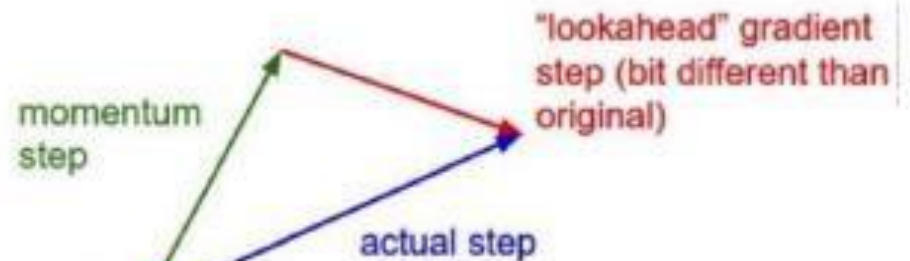
- [?] Empirically found to give good performance

# Nesterov Accelerated Momentum: Visualization

Momentum update



Nesterov momentum update



# Nesterov Accelerated Momentum: Algorithm

---

**Require:** Learning rate  $\alpha$ , momentum parameter  $\gamma$ , initial parameters  $\theta_t$ , training dataset  $\mathcal{D}_{tr}$

---

```
1: Initialize  $v_{t-1} = 0$ 
2: while stopping criterion not met do
3:   Initialize gradients  $\Delta\theta_t = 0$ 
4:   Get look-ahead parameters  $\tilde{\theta}_t = \theta_t - \gamma v_{t-1}$ 
5:   for each  $(x^{(i)}, y^{(i)})$  in  $\mathcal{D}_{tr}$  do
6:     Compute gradient using look-ahead parameters  $\nabla_{\tilde{\theta}_t} \mathcal{L}(\tilde{\theta}_t; x^{(i)}, y^{(i)})$ 
7:     Aggregate gradient  $\Delta\theta_t = \Delta\theta_t + \nabla_{\tilde{\theta}_t} \mathcal{L}$ 
8:   end for
9:   Update velocity  $v_t = \gamma v_{t-1} + \alpha \Delta\theta_t$ 
10:  Apply update  $\theta_{t+1} = \theta_t - v_t$ 
11: end while
```

---

# Batch GD: Pros and Cons

Batch GD is a type of gradient descent where the optimizer processes the entire dataset for every single parameter update. This is why it's referred to as "Batch" GD, as it works on the complete batch (full dataset) at once.

- ❓ For every parameter update, G D parses the entire dataset, hence called Batch G D

- ❓ Advantages of Batch G D

- ❓ Conditions of convergence well-understood
- ❓ Many acceleration techniques (e.g. conjugate gradient) operate in batch G D setting

- ❓ Disadvantages of Batch G D

- ❓ Computationally slow
  - ❓ E.g. ImageNet (<http://www.image-net.org>), a commonly used dataset in vision, has ~14:2million samples; an iteration over it is going to be very slow
- Scalability Issues: Due to its high computational cost, Batch GD isn't ideal for extremely large datasets or environments with limited computational resources.

# 1.3 Stochastic Gradient Descent

## Stochastic Gradient Descent

**Stochastic GD (SGD):** Randomly shuffle the training set, and update parameters after gradients are computed for each training example

---

Require: Learning rate  $\alpha$ , initial parameters  $\theta_t$ , training dataset  $\mathcal{D}_{tr}$

---

```
1: while stopping criterion not met do
2:   for each  $(x^{(i)}, y^{(i)})$  in  $\mathcal{D}_{tr}$  do
3:     Compute gradient using backpropagation  $\nabla_{\theta_t} \mathcal{L}(\theta_t; x^{(i)}, y^{(i)})$ 
4:     Gradient  $\Delta\theta_t = \nabla_{\theta_t} \mathcal{L}$ 
5:     Apply update  $\theta_{t+1} = \theta_t - \alpha \Delta\theta_t$ 
6:   end for
7: end while
```

---

Stochastic Gradient Descent (SGD): Unlike Batch Gradient Descent (which processes the entire dataset at once), SGD updates the parameters after computing the gradient for each individual training example. It introduces randomness by shuffling the dataset and picking one sample at a time for gradient computation, which can lead to faster updates but with more noise.



# Mini-batch Stochastic Gradient Descent

**Mini-Batch Stochastic GD:** Update parameters after gradients are computed for a randomly drawn mini-batch of training examples (*default option today, often simply called as SGD*)

---

**Require:** Learning rate  $\alpha$ , initial parameters  $\theta_t$ , mini-batch size  $m$ , training dataset  $\mathcal{D}_{tr}$

---

```
1: while stopping criterion not met do
2:   Initialize gradients  $\Delta\theta_t = 0$ 
3:   Sample  $m$  examples from  $\mathcal{D}_{tr}$  (call it  $\mathcal{D}_{mini}$ )
4:   for each  $(x^{(i)}, y^{(i)})$  in  $\mathcal{D}_{mini}$  do
5:     Compute gradient using backpropagation  $\nabla_{\theta_t} \mathcal{L}(\theta_t; x^{(i)}, y^{(i)})$ 
6:     Aggregate gradient  $\Delta\theta_t = \Delta\theta_t + \nabla_{\theta_t} \mathcal{L}$ 
7:   end for
8:   Apply update  $\theta_{t+1} = \theta_t - \alpha \Delta\theta_t$ 
9: end while
```

---

In Mini-batch SGD, instead of processing the entire dataset at once (like BGD) or a single training example (like SGD), the data is divided into smaller chunks called mini-batches. The algorithm updates the parameters after computing the gradient for one of these randomly selected mini-batches of training examples.

# SGD: Pros and Cons

## Advantages of SGD

- Usually much faster than batch learning; because there is lot of redundancy in batch learning
- Often results in better solutions; SGD's noise can help in escaping local minima (provided neighborhood provides enough gradient information) and saddle points<sup>1</sup>.
- Can be used for tracking changes

## Disadvantages of SGD

- Noise in SGD weight updates -can lead to no convergence!
- Can be controlled using learning rate, but identifying proper learning rate is a problem of its own

<sup>1</sup><http://mitliagkas.github.io/ift6085-2019/ift-6085-bonus-lecture-saddle-points-notes.pdf>

# Training N N s with SGD: Challenges

## ❓ Issues that we might encounter when traversing error surfaces using G D

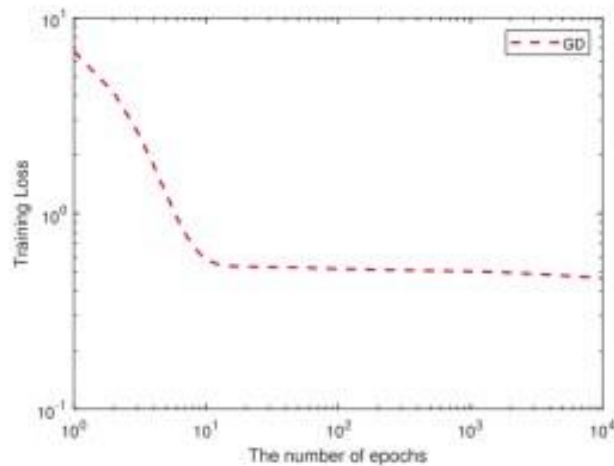
- ❓ Plateaus and flat regions
- ❓ Local minima and saddle points
- ❓ Vanishing and exploding gradients/cliffs

## ❓ Other challenges

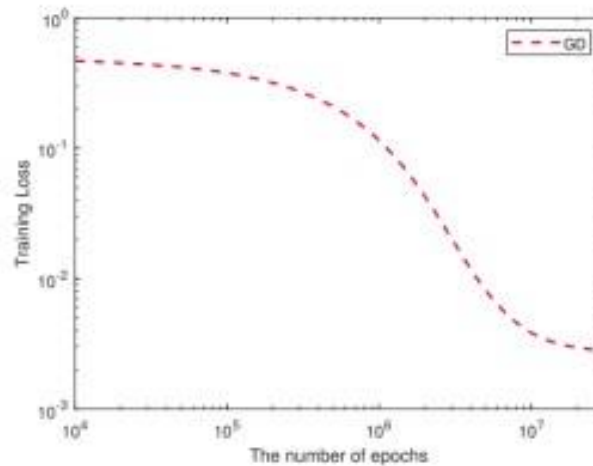
- ❓ Inexact gradients
  - ❓ Poor correspondence between local and global structure
  - ❓ Choosing learning rate and other hyperparameters

# Training N N s with SGD: Challenges

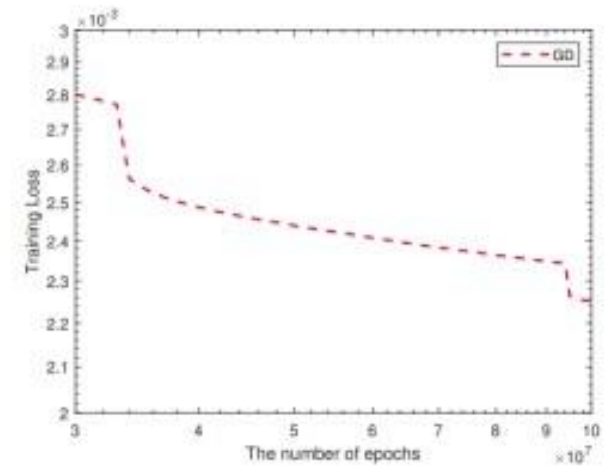
- ? Plateau
  - A plateau is a flat, elevated landform that rises sharply above the surrounding area on at least one



(a)

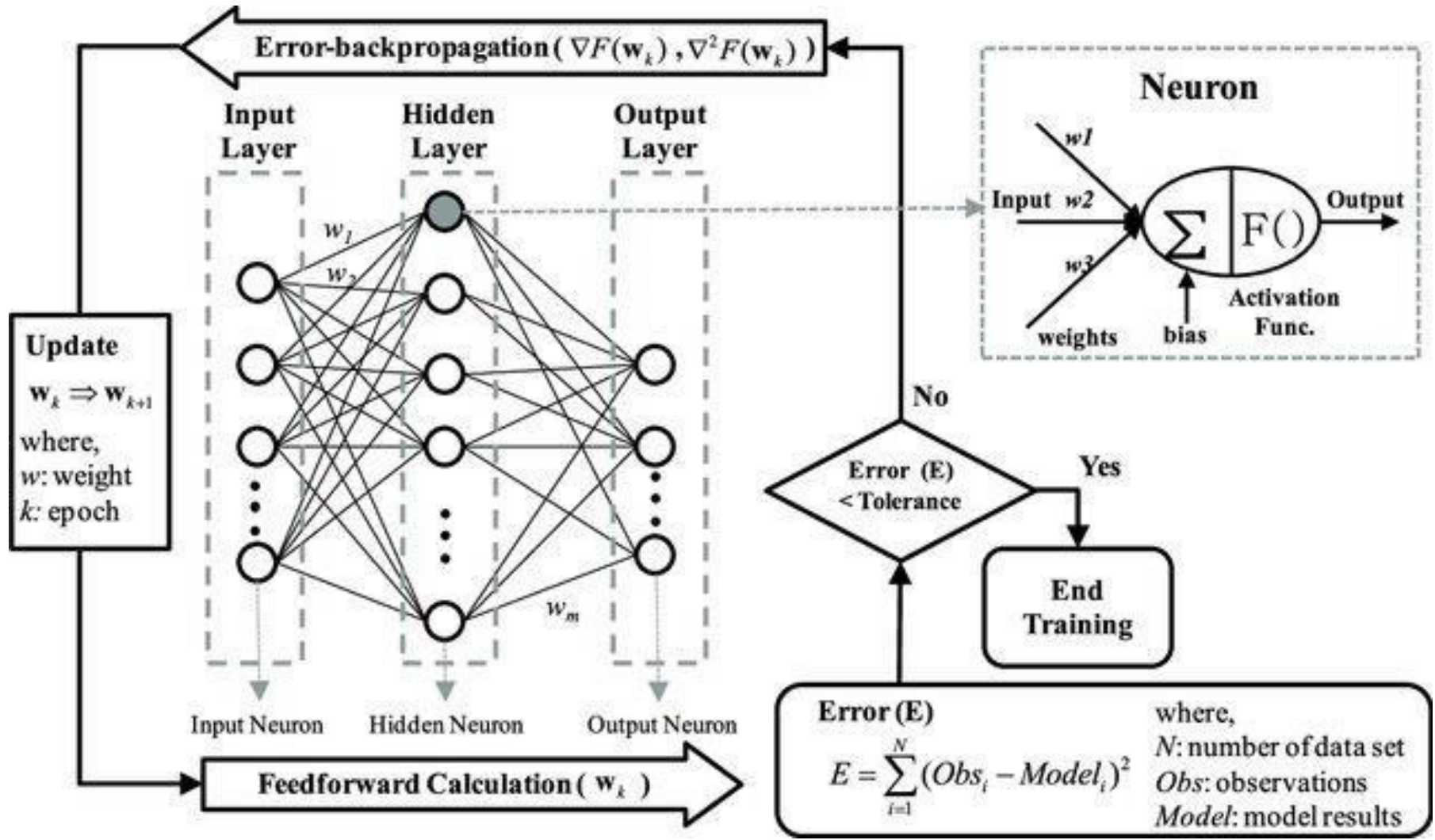


(b)



(c)

# Back propagation



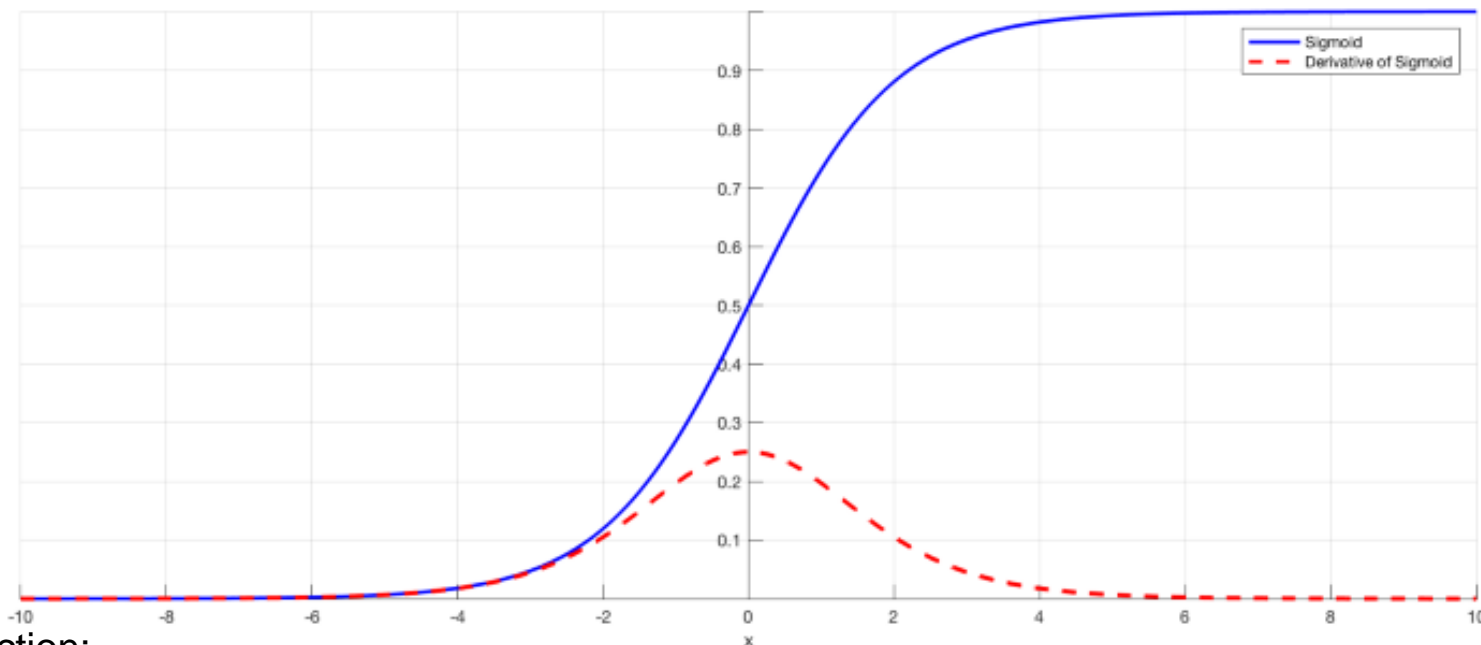
Source: [Google Images](#)

# Vanishing Gradient

What Happens During Training:

During backpropagation, gradients are used to update the weights of the neural network. If the gradient is very small (close to 0), the weight updates become negligible.

In deep networks, this problem compounds across layers, as gradients are multiplied back through many layers. By the time the gradient reaches earlier layers, it becomes so small that those layers effectively stop learning.



Sigmoid Function:

The blue curve represents the sigmoid activation function, which squashes any input into a range between 0 and 1.

For very large positive or negative values of  $x$ , the sigmoid becomes almost flat (near 1 or 0), meaning the slope (or gradient) approaches 0.

Derivative of Sigmoid:

The red dashed curve shows the derivative (or slope) of the sigmoid function.

Notice how the derivative is largest around  $x$  and approaches 0 for large positive or negative  $x$ .

# Exploding Gradient

Axes:

$w$  (x-axis): Represents the parameter (weight).

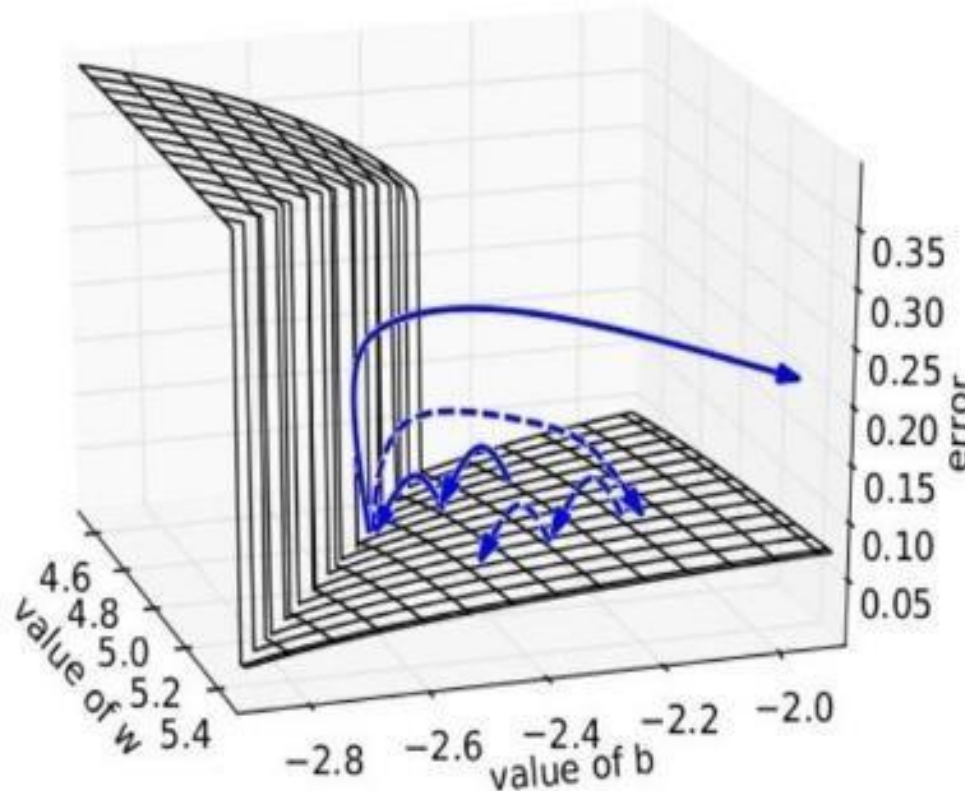
$b$  (y-axis): Represents another parameter (bias).

Error (z-axis): Represents the loss (or cost) function being minimized.

Gradient Descent Path (Blue Arrows):

The arrows start at a point in the parameter space and show how the model updates the weights and biases.

Instead of converging to the minimum (valley), the path diverges rapidly due to excessively large gradient values.



# ***How to know if our model is suffering from the Exploding/Vanishing gradient problem?***

## **Exploding**

Exploding gradients cause instability and divergence.

There is an exponential growth in the model parameters.

The model weights may become NaN during training.

The model experiences avalanche learning.

## **Vanishing**

Vanishing gradients slow down or stop learning entirely. The parameters of the higher layers change significantly whereas the parameters of lower layers would not change much (or not at all).

The model weights may become 0 during training.

The model learns very slowly and perhaps the training stagnates at a very early stage just after a few iterations.



## ? Solutions

### ? **Proper Weight Initialization**

Start the network's weights with carefully chosen values (like Xavier or He initialization) to prevent gradients from vanishing or exploding during training.

### ? **Using Non-saturating Activation Functions**

Replace activation functions like sigmoid or tanh (which can cause vanishing gradients) with ReLU or its variants, which keep gradients more stable.

### ? **Gradient Clipping**

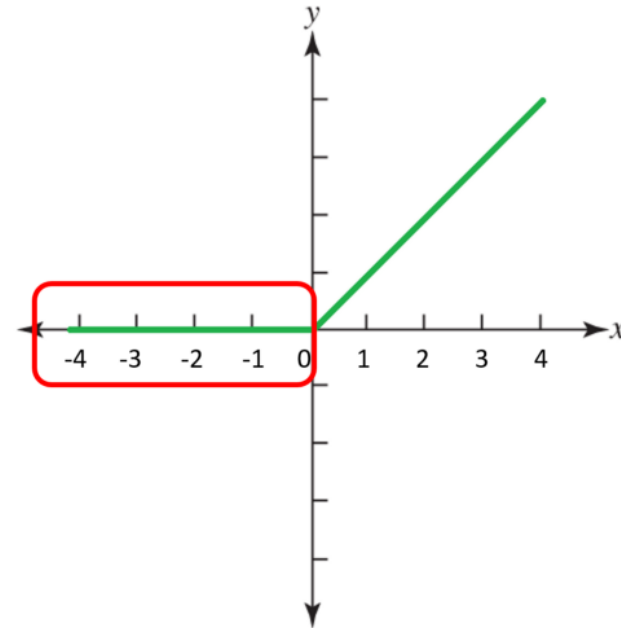
Limit the magnitude of gradients to a maximum threshold to avoid them becoming too large (exploding gradients) and destabilizing the learning process.

# Dying ReLU

The problem occurs when many neurons in the network output only zero during training because their inputs are negative. When this happens, these neurons effectively "die" and stop contributing to the learning process since their gradients are zero (and they don't update anymore).

## What's the Dying ReLU problem?

The dying ReLU problem refers to the scenario when many ReLU neurons only output values of 0. The red outline below shows that this happens when the inputs are in the **negative** range.



During training, if a neuron's weights and biases are set in a way that consistently produces negative inputs, the ReLU function outputs zero. Once stuck at zero, these neurons no longer adjust their weights or biases, as their gradients will always remain zero.

# Dying ReLU- Causes

## ? High learning rate

$$W_{ij}^* = W_{ij} - \alpha \left( \frac{\partial E}{\partial W_{ij}} \right)$$

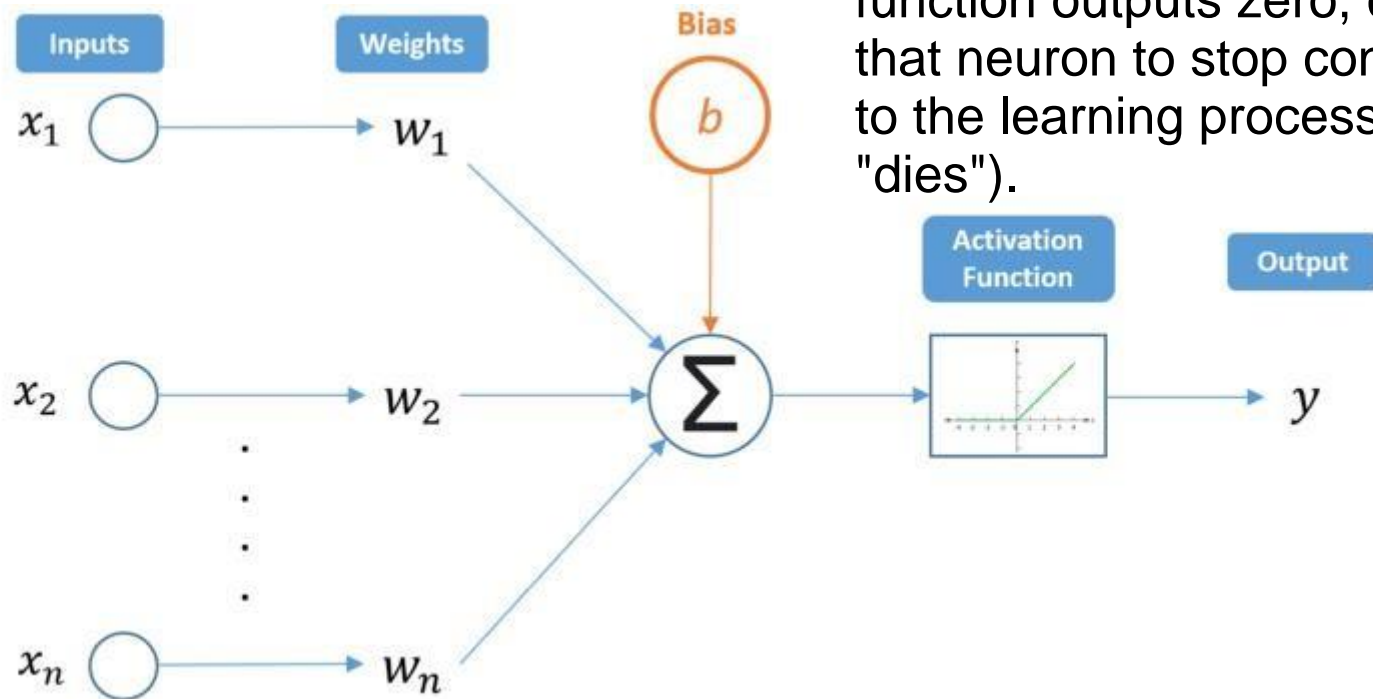
The diagram shows the weight update equation  $W_{ij}^* = W_{ij} - \alpha \left( \frac{\partial E}{\partial W_{ij}} \right)$ . Below the equation, four blue arrows point upwards to specific terms: the first arrow points to  $W_{ij}^*$  (labeled 'New Weight'), the second arrow points to  $W_{ij}$  (labeled 'Old Weight'), the third arrow points to the red  $\alpha$  (labeled 'Learning Rate'), and the fourth arrow points to the derivative term  $\left( \frac{\partial E}{\partial W_{ij}} \right)$  (labeled 'Derivative of Error with respect to Weight').

New Weight      Old Weight      Learning Rate      Derivative of Error with respect to Weight

If the learning rate (alpha) is too high, the weight updates become excessively large. This can lead to weights becoming very negative, causing neurons to consistently receive negative inputs. This forces the ReLU activation function to output zero for these neurons, effectively causing them to "die."

# Dying ReLU- Causes

## ? Large negative bias



If the bias ( $b$ ) is very negative, it can shift the sum to always be negative, regardless of the inputs. As a result, the ReLU function outputs zero, causing that neuron to stop contributing to the learning process (it "dies").

# Solutions to the Dying ReLU problem?

By reducing the learning rate, the weight updates become smaller and more controlled, minimizing the risk of neurons getting stuck.

- [?] Use of a lower learning rate
- [?] Variations of ReLU ----->
- [?] Modification of initialization procedure

Techniques like He initialization are specifically designed for ReLU activations and help maintain a balanced distribution of neuron outputs.

Leaky ReLU: Outputs a small negative value instead of zero when the input is negative, allowing gradients to flow and preventing neurons from dying.  
Parametric ReLU (PReLU): Similar to Leaky ReLU but learns the slope of the negative region during training.  
ELU (Exponential Linear Unit): Smoothly transitions into negative values, which can improve gradient flow.

---

## References

- Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.
- <https://cs231n.github.io/neural-networks-3/sgd>