

# Deep Learning

## CSC-Elective

Instructor : Dr. Muhammad Ismail Mangrio

Slides prepared by Dr. M Asif Khan

[ismail@iba-suk.edu.pk](mailto:ismail@iba-suk.edu.pk)

*Week 2*

# Contents

- DL
- Discussed Multi-layer Perceptron
- MLP and XOR
- Training MLP
- Chain rule of derivative
- Vanishing gradient
- Selecting Activation function

# Learning Rules and Representation from Data

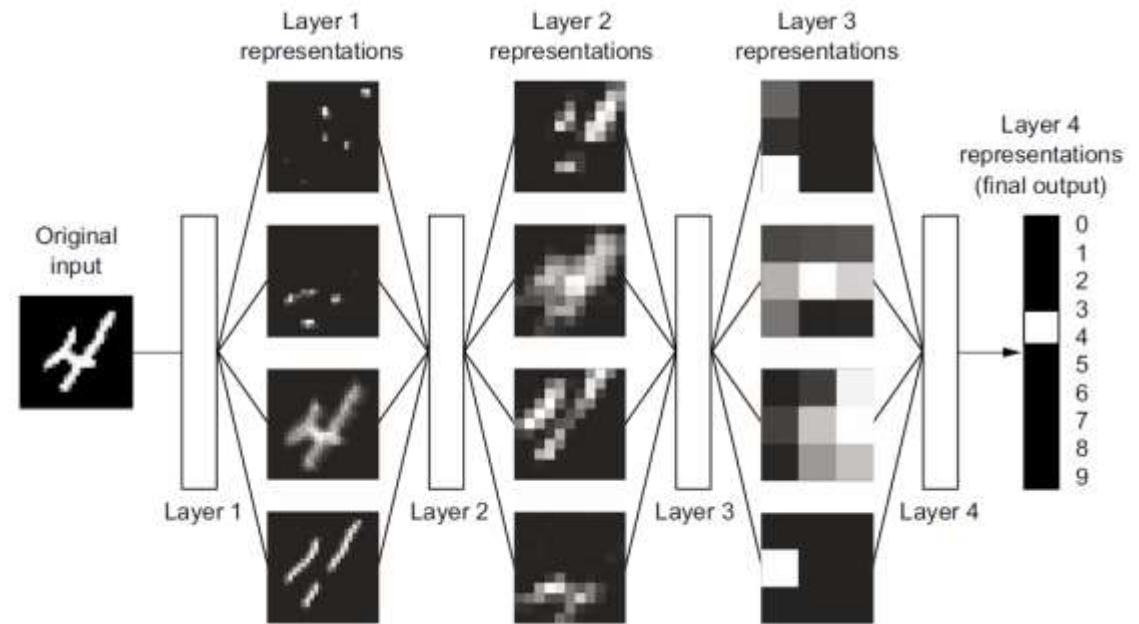
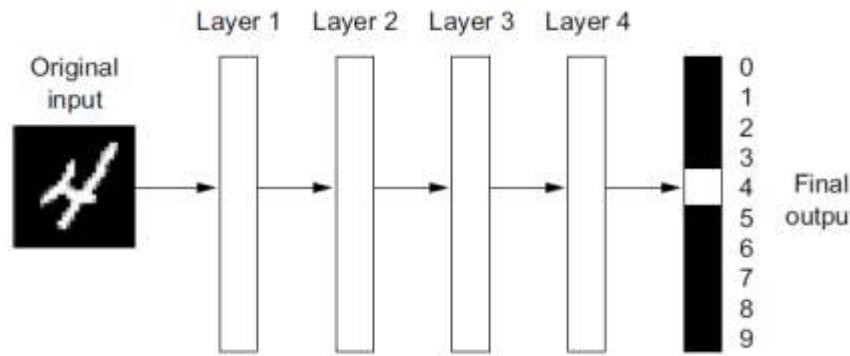
- How about digit classification task? Can you produce classification rules manually?
- This is possible to an extent.
- Rules based on representations of digits such as “number of closed loops” or vertical and horizontal pixel histograms can do a decent job of telling apart handwritten digits.
- But finding such useful representations by hand is hard work, and, as you can imagine, the resulting rule-based system is very difficult to maintain.
- To address such tasks, we use machine learning or deep learning

7 → 7 5 → 5

8 → 8 3 → 3

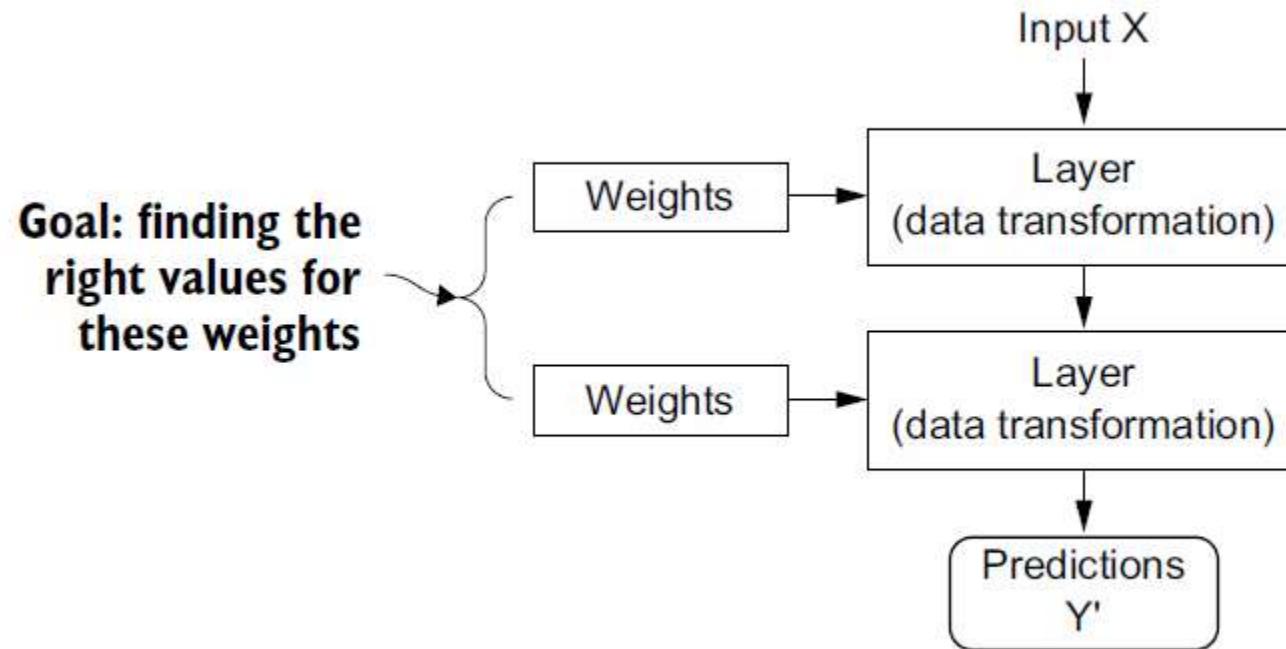
2 → 2 4 → 4

# The deep in Deep Learning



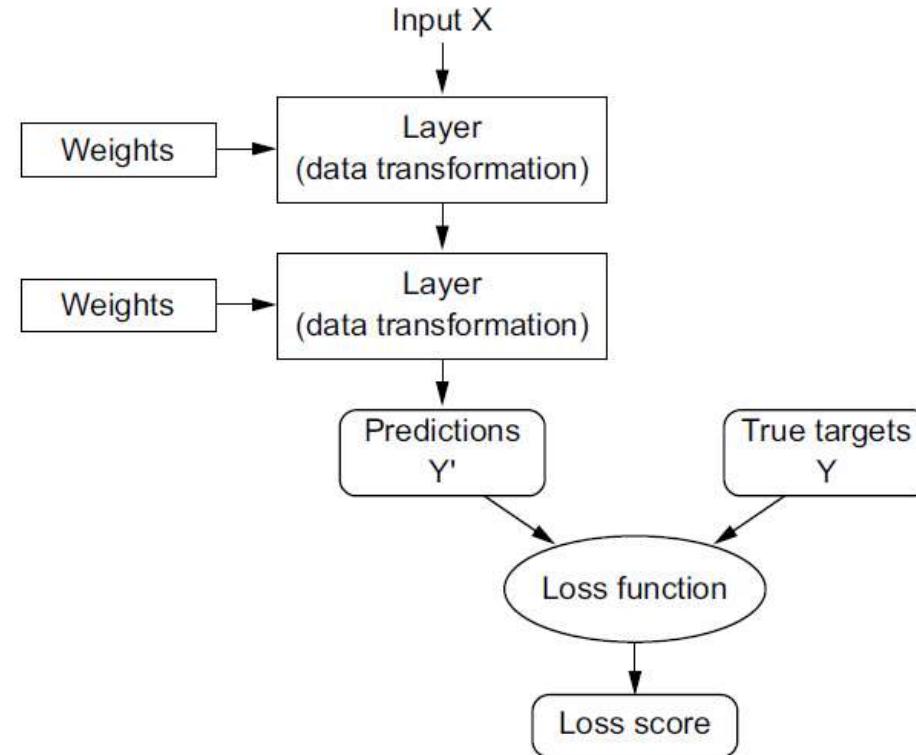
- In the right figure, the network transforms the digit image into representations that are increasingly different from the original image and increasingly informative about the final result.
- Think of a deep network as a multistage information distillation process, where information goes through successive filters and comes out increasingly purified

# Understanding how deep learning works, in three figures



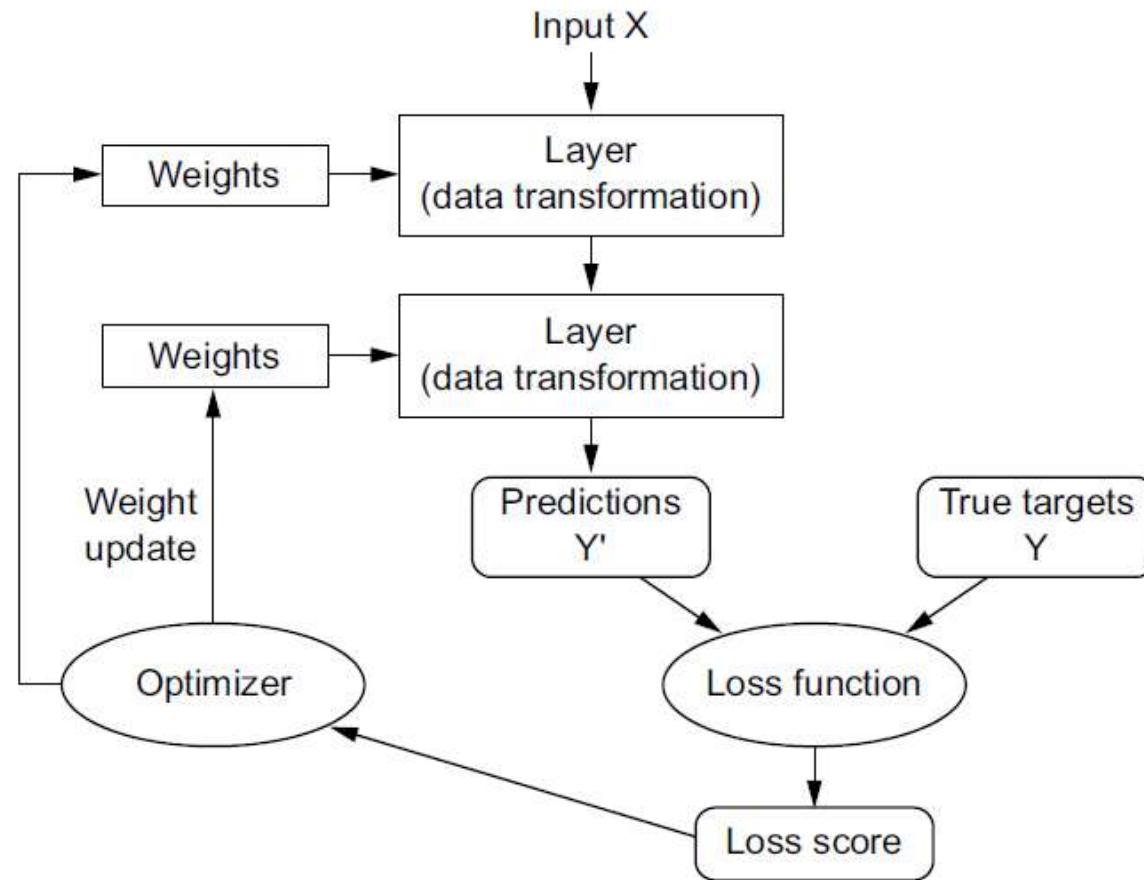
**Figure 1.7** A neural network is parameterized by its weights.

# Understanding how deep learning works, in three figures



**Figure 1.8** A loss function measures the quality of the network's output.

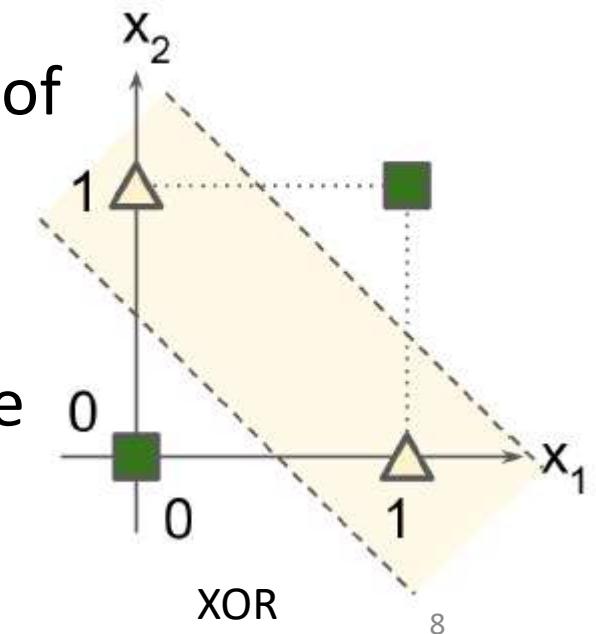
# Understanding how deep learning works, in three figures



**Figure 1.9** The loss score is used as a feedback signal to adjust the weights.

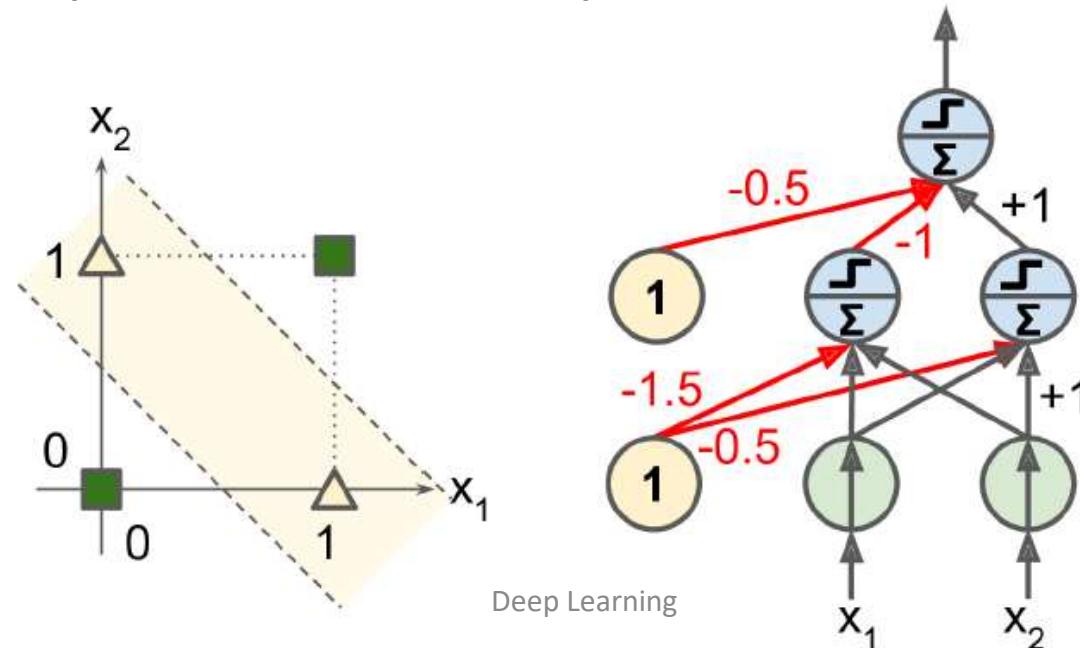
# Artificial Neural Network (Multi Layer Perceptron)

- Contrary to Logistic Regression classifiers, Perceptron **do not output a class probability**; rather, they just make predictions based on a hard threshold.
- This is one of the good reasons to **prefer Logistic Regression over Perceptrons**.
- Marvin Minsky and Seymour Papert highlighted a number of **serious weaknesses of Perceptrons**, in particular the fact that they are **incapable of solving some trivial problems** (e.g., the Exclusive OR (XOR) classification problem; see the left side of Figure 10-6).



# Artificial Neural Network (Multi Layer Perceptron)

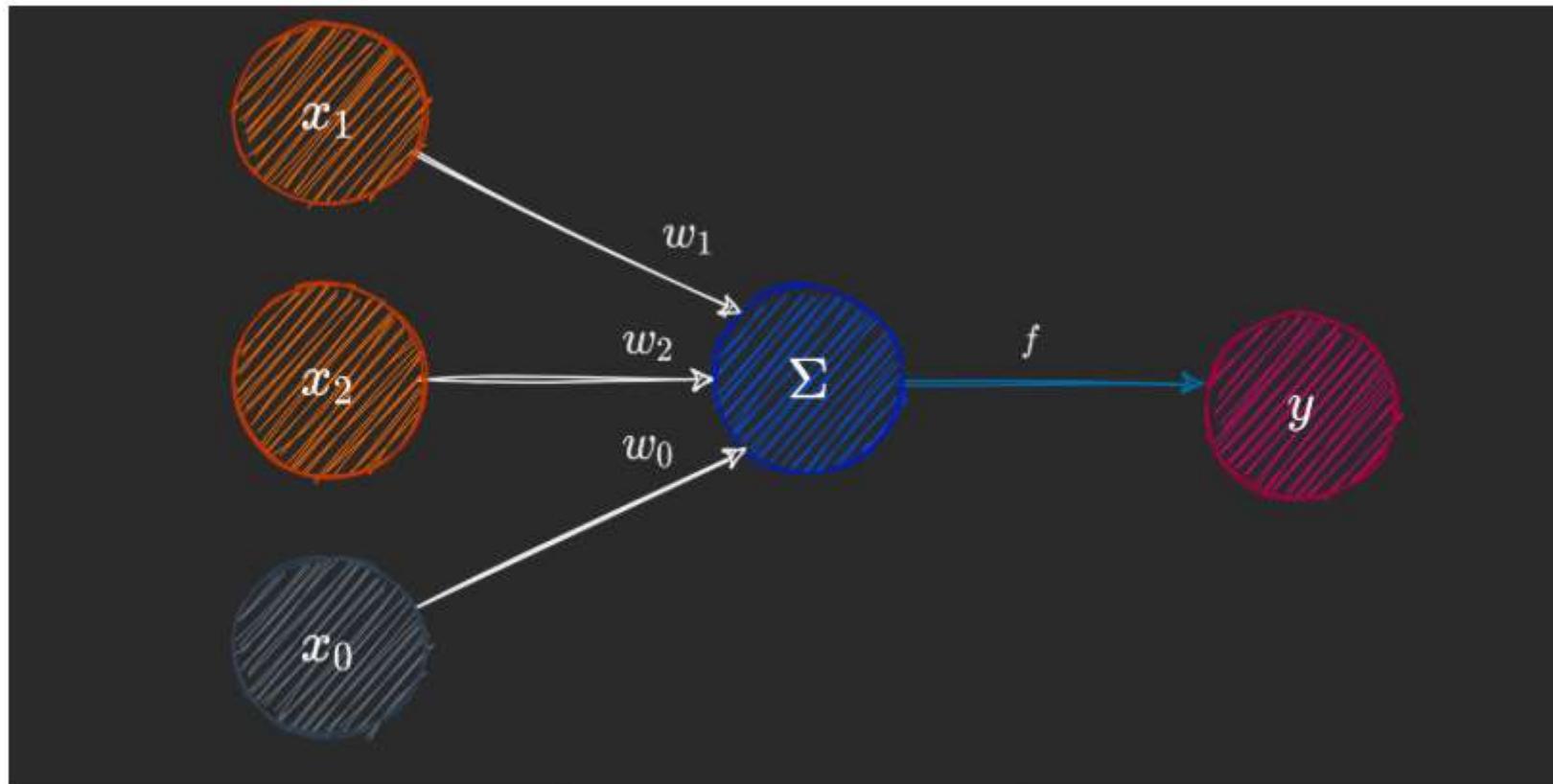
- However, it turns out that some of the limitations of Perceptrons can be eliminated by **stacking multiple Perceptrons**.
- The resulting ANN is called a **Multi-Layer Perceptron (MLP)**.
- In particular, an **MLP can solve the XOR problem**, as you can verify by computing the output of the MLP represented on the right of Figure 10-6,



Assignment#01 Solve XOR using  
MLP on paper with hand written  
Solution.

# MLP and XOR Problem

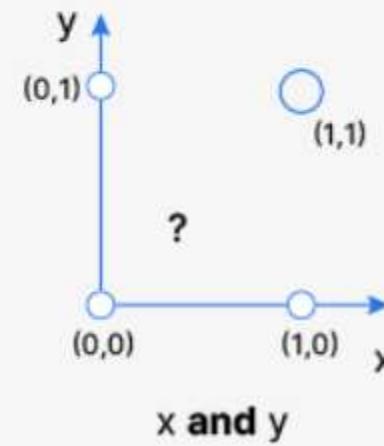
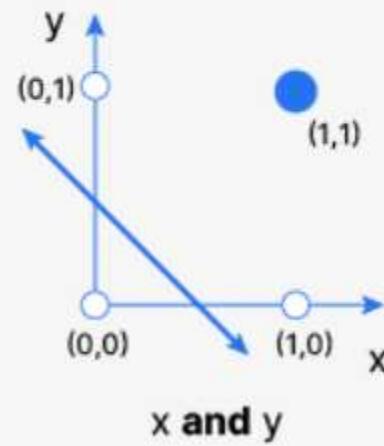
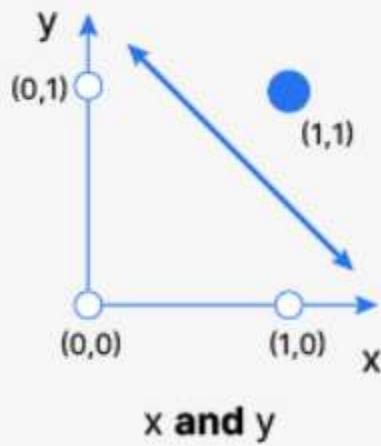
# Understanding Neural Networks



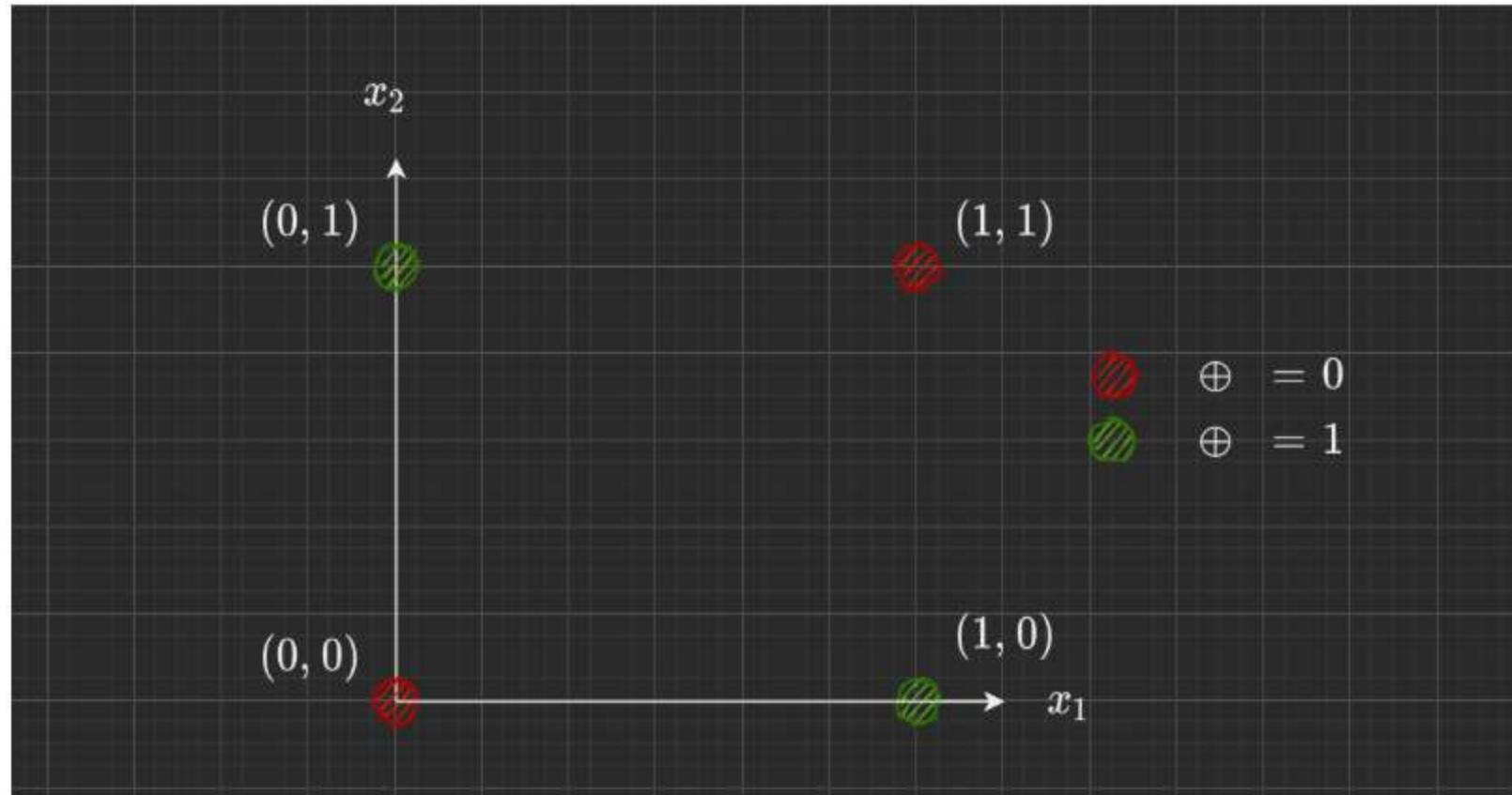
A representation of a single-layer perceptron with 2 input nodes – Image by Author using [draw.io](#)

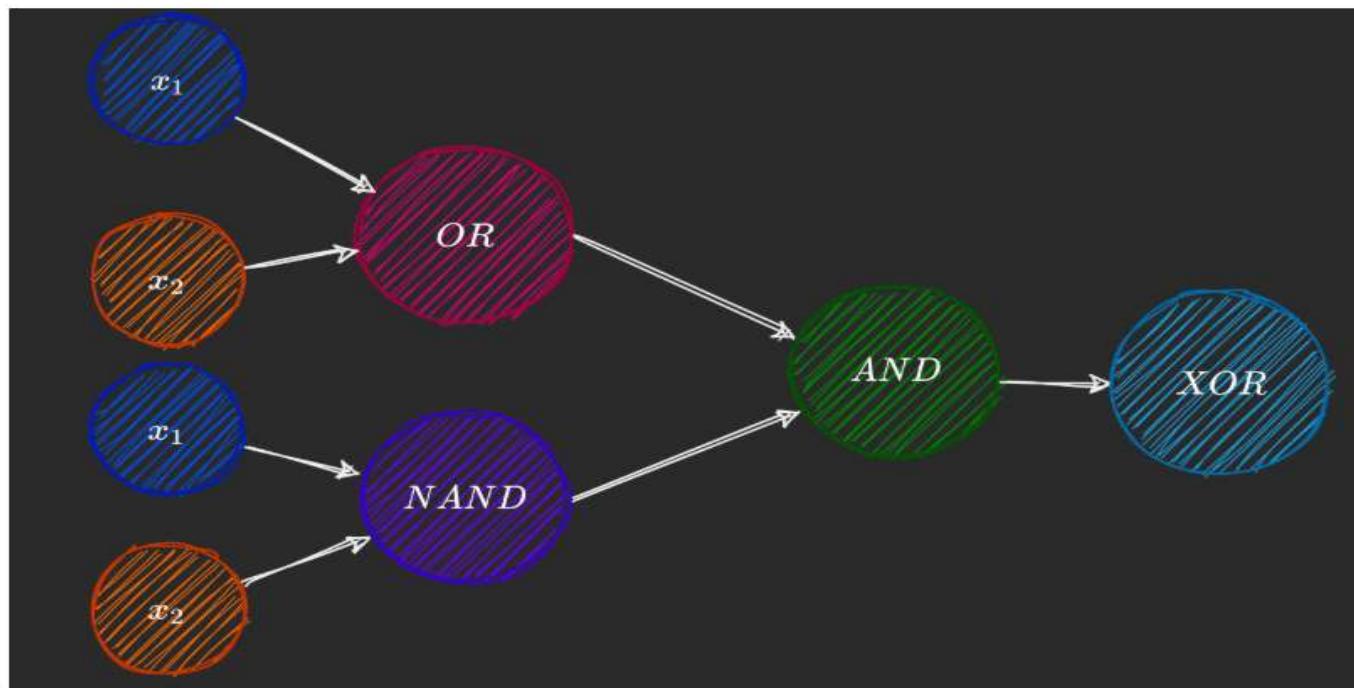
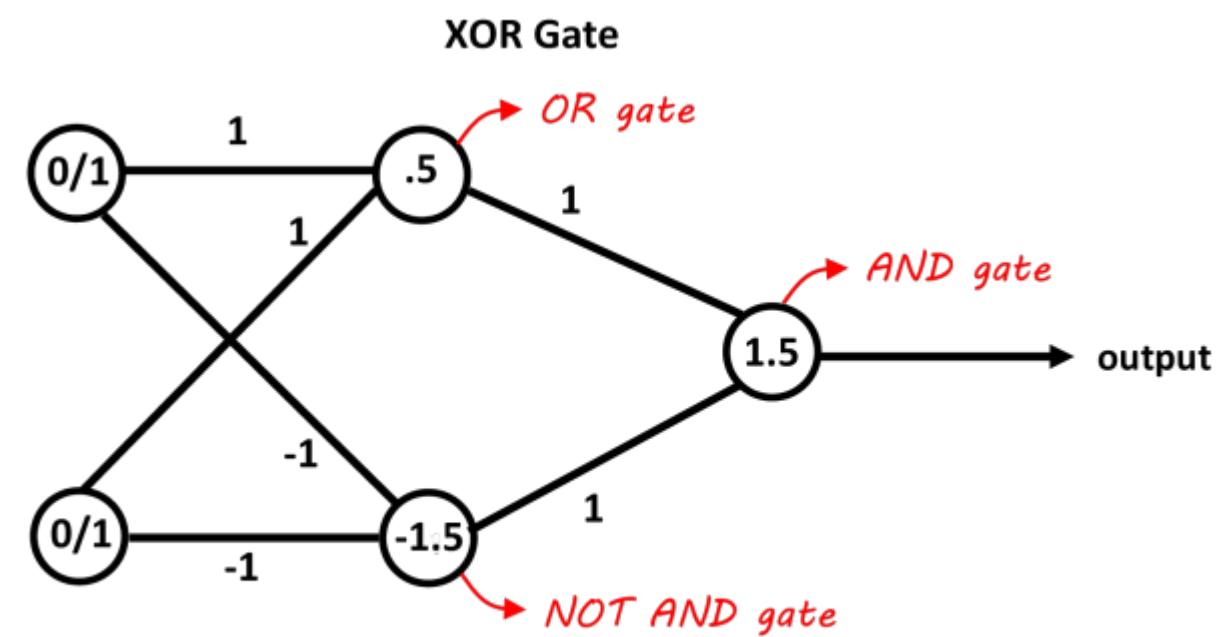
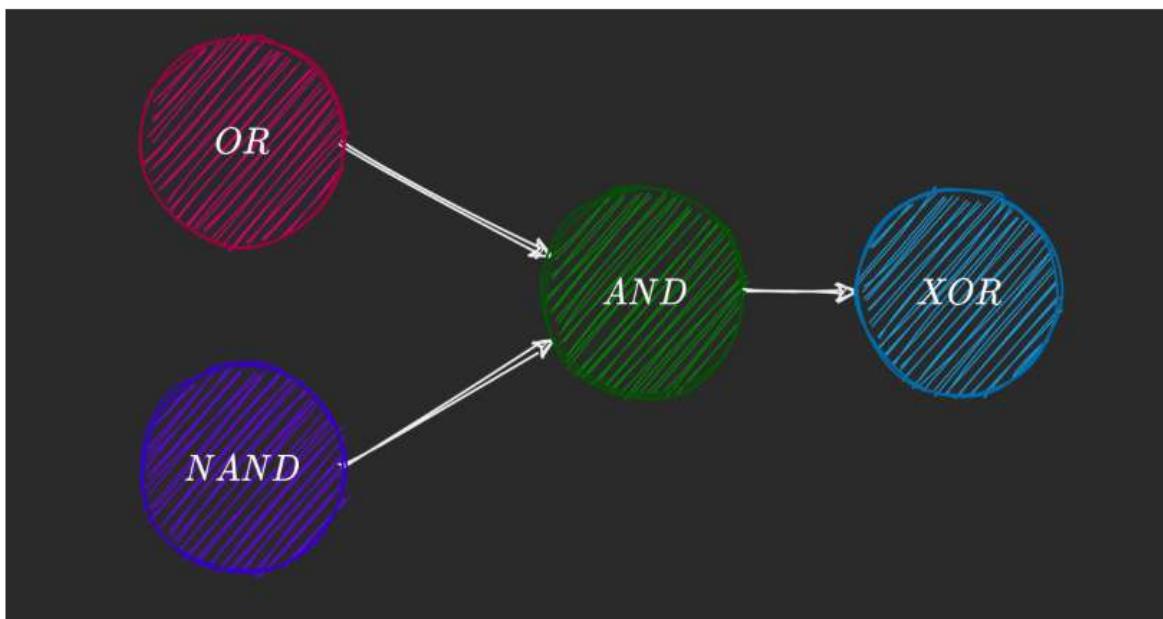
# Explaining the XOR Problem

## Linear separability

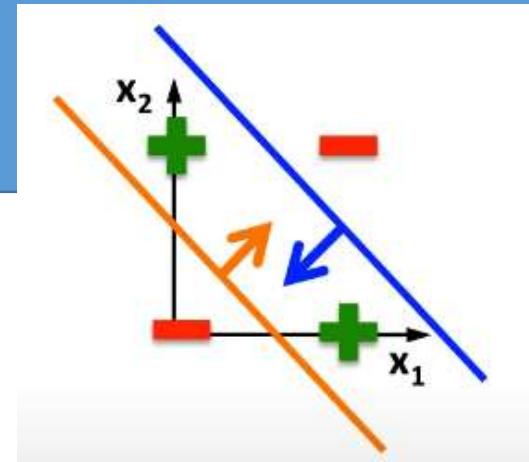


# The 2D XOR problem

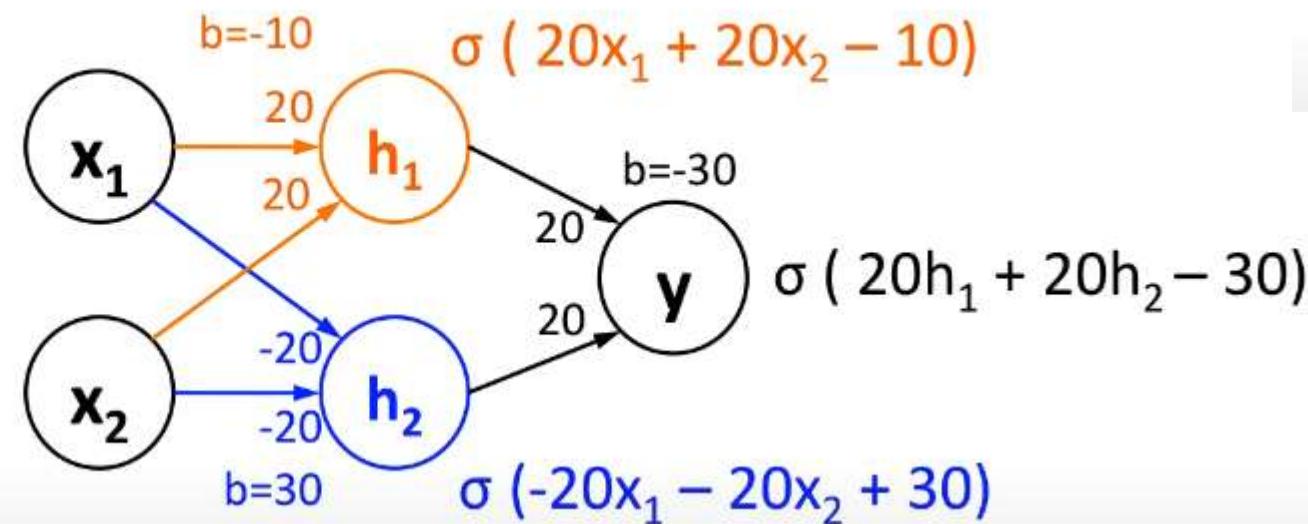
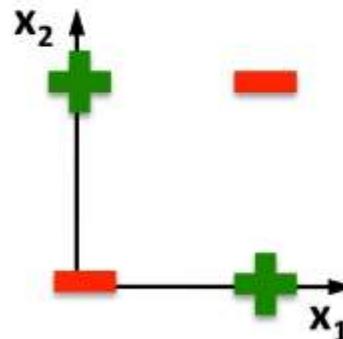




# Example: Solving XOR with a hidden layer



Linear classifiers  
cannot solve this



$$\sigma(20*0 + 20*0 - 10) \approx 0$$

$$\sigma(-20*0 - 20*0 + 30) \approx 1$$

$$\sigma(20*0 + 20*1 - 30) \approx 0$$

$$\sigma(20*1 + 20*1 - 10) \approx 1$$

$$\sigma(-20*1 - 20*1 + 30) \approx 0$$

$$\sigma(20*1 + 20*0 - 30) \approx 0$$

$$\sigma(20*0 + 20*1 - 10) \approx 1$$

$$\sigma(-20*0 - 20*1 + 30) \approx 1$$

$$\sigma(20*1 + 20*1 - 30) \approx 1$$

$$\sigma(20*1 + 20*0 - 10) \approx 1$$

$$\sigma(-20*1 - 20*0 + 30) \approx 1$$

$$\sigma(20*1 + 20*1 - 30) \approx 1$$

```

import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Define the XOR input and output data
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])

# Build the neural network model
model = Sequential()
model.add(Dense(2, input_dim=2, activation='relu')) # Hidden layer with 2 neurons
model.add(Dense(1, activation='sigmoid')) # Output layer with 1 neuron

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Train the model
model.fit(X, y, epochs=10000, verbose=0)

# Evaluate the model
accuracy = model.evaluate(X, y)
print(f"Accuracy: {accuracy * 100:.2f}%")

# Make predictions
predictions = model.predict(X)
predictions = np.round(predictions).astype(int)

print("Predictions:")
for i in range(len(X)):
    print(f"Input: {X[i]} => Predicted Output: {predictions[i]}, Actual Output: {y[i]}")

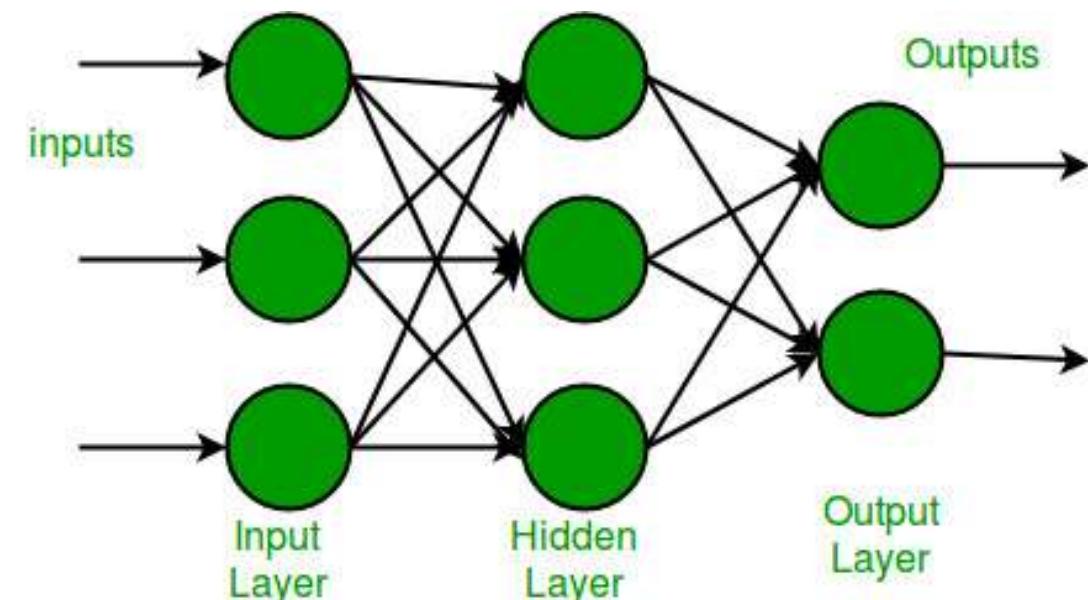
```

# Multi-Layer Perceptron (MLP)

- MLP consists of fully connected dense layers that transform input data from one dimension to another.
- It is called **multi-layer** because
- it contains an input layer, one or more hidden layers and an output layer.
- The purpose of an MLP is to model complex relationships between inputs and outputs

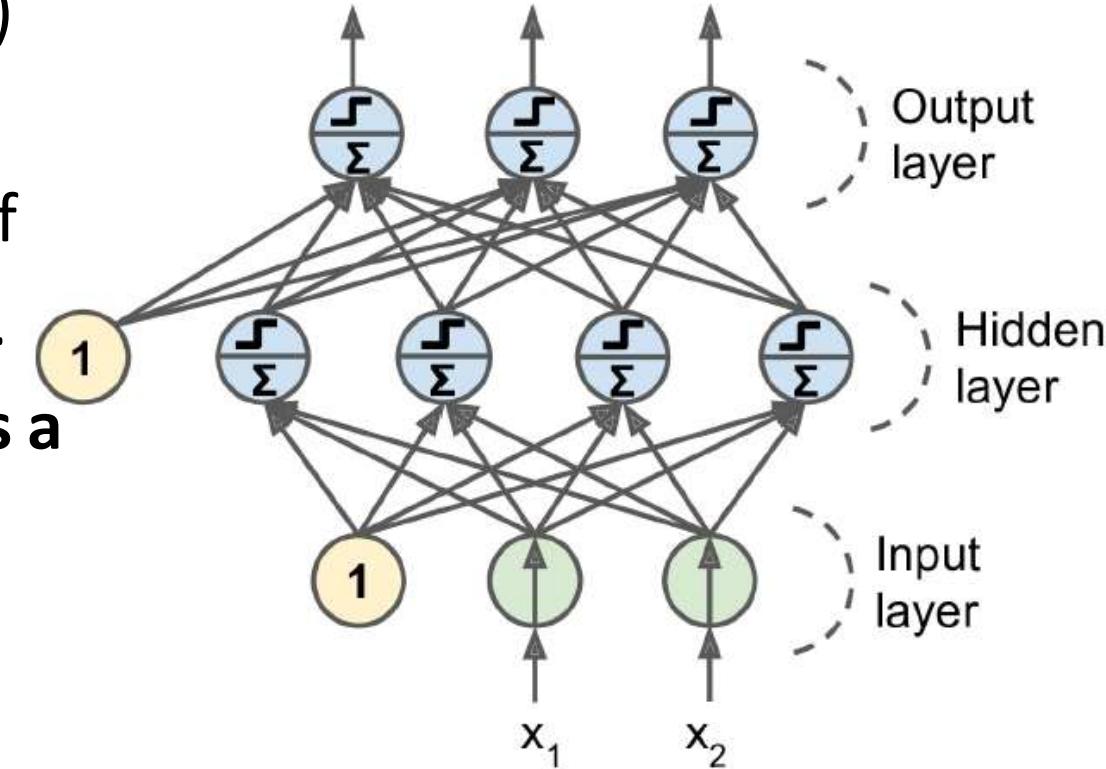
# Multi-Layer Perceptron (MLP)

- **Input Layer:** Each neuron or node in this layer corresponds to an input feature. For instance, if you have three input features the input layer will have three neurons.
- **Hidden Layers:** MLP can have any number of hidden layers with each layer containing any number of nodes. These layers process the information received from the input layer.
- **Output Layer:** The output layer generates the final prediction or result. If there are multiple outputs, the output layer will have a corresponding number of neurons

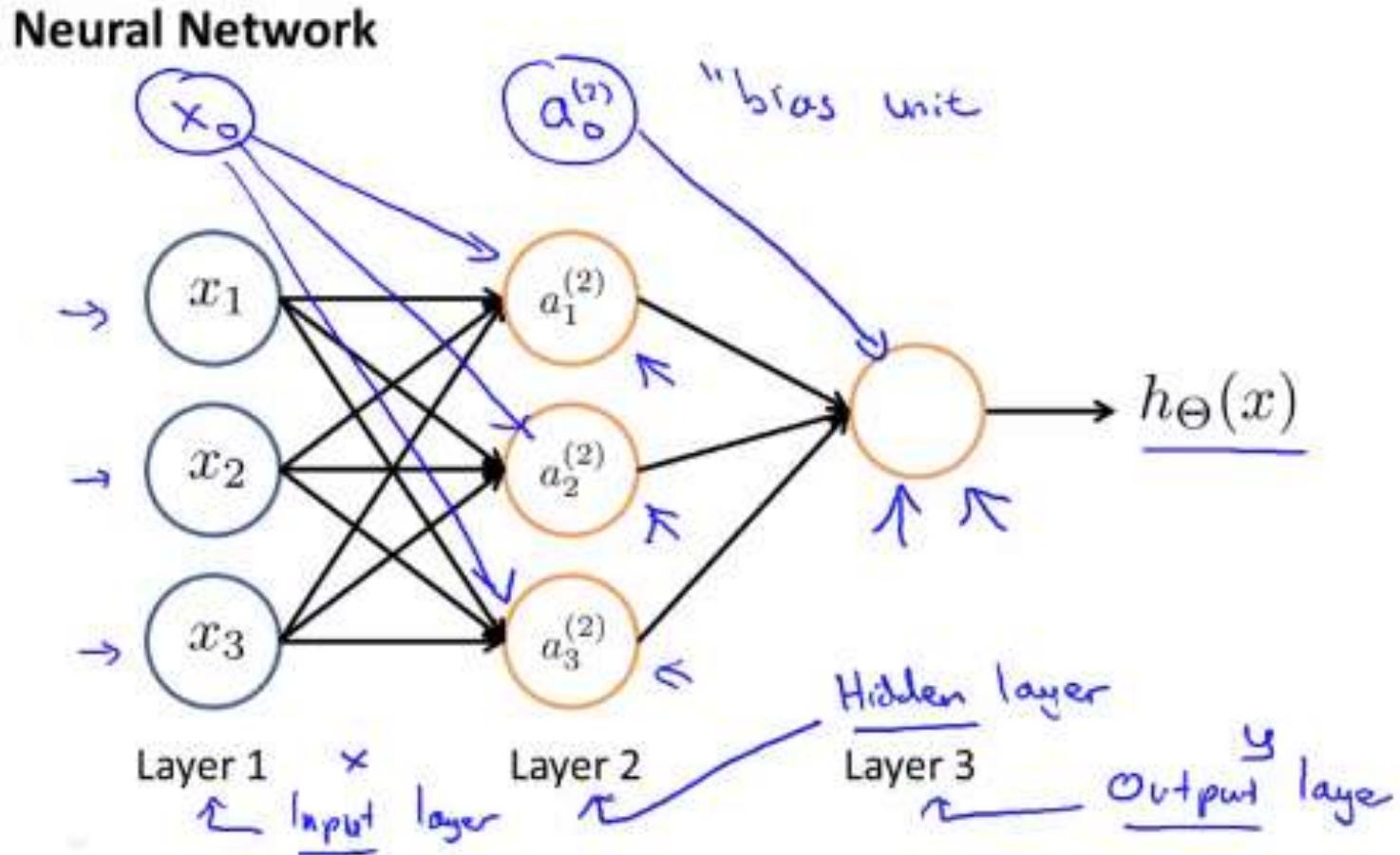


# Artificial Neural Network (MLP)

- An MLP is composed of **one** (passthrough) **input layer**, one or more layers of LTUs, called **hidden layers**, and one final layer of LTUs called the **output layer** (Figure 10-7).
- Every layer except the input layer **includes a bias neuron** and is **fully connected** to the next layer.
- When an ANN has two or more hidden layers, it is called a **Deep Neural Network (DNN)**.



# Artificial Neural Network (MLP)



# ANN Learning Process

1. Input layer
2. Weights
3. Hidden layer
4. Activation function
5. Loss function ( $y-y'$ )

Forward propagation

6. Optimizers (e.g. gradient descent) to reduce the cost function
7. Updating of weights to approach global minima

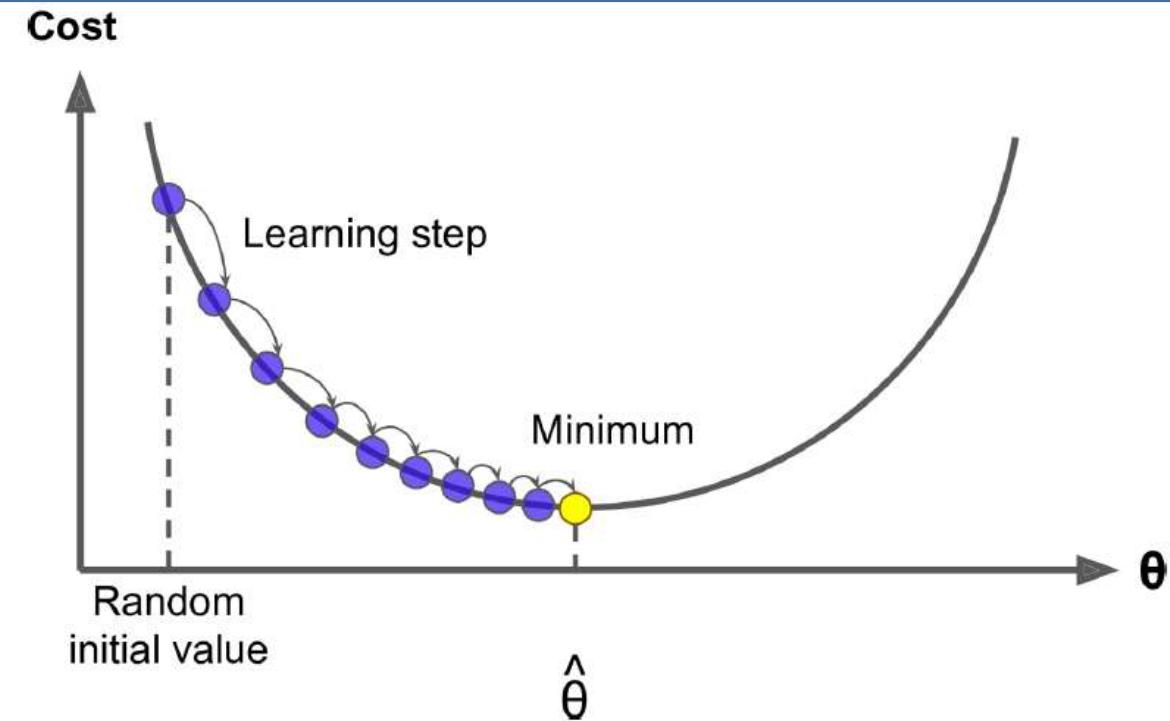
Backward propagation

# Important concepts (Gradient Descent)

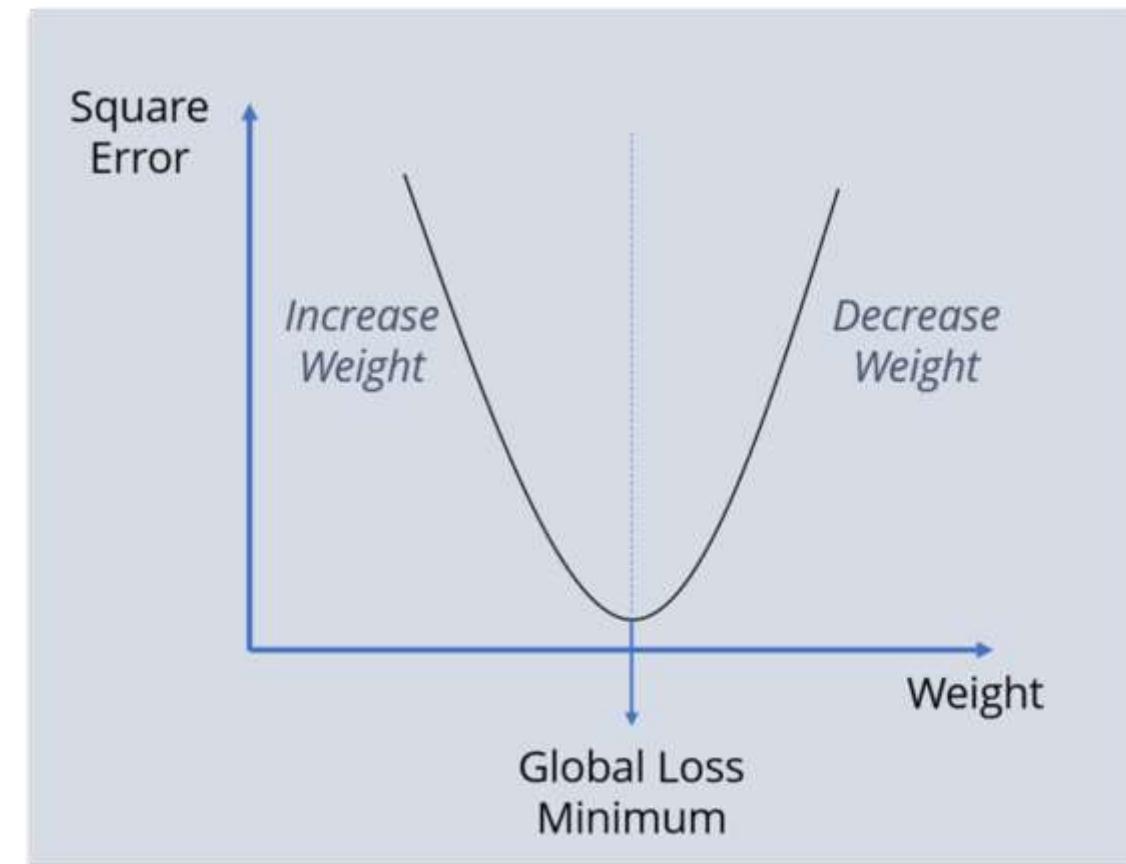
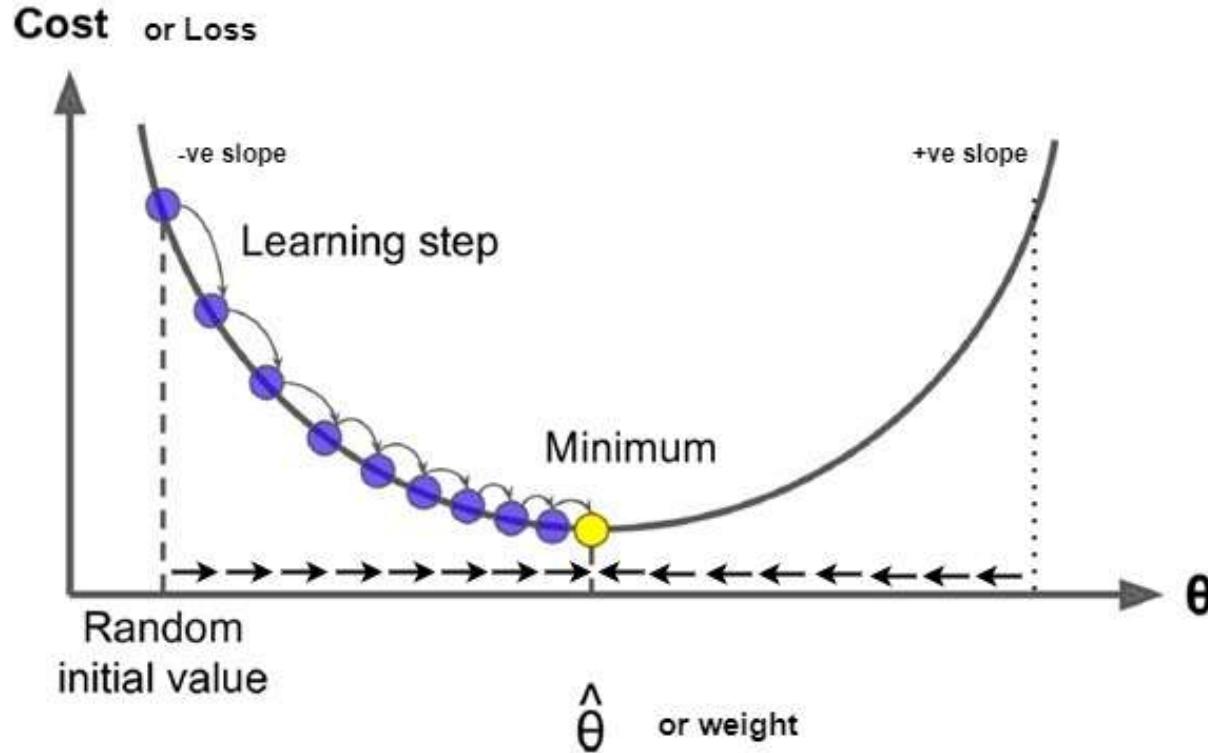
- Gradient Descent is a **very generic optimization algorithm**.
- The general idea of Gradient Descent is to tweak parameters iteratively to **minimize a cost function**.
- Suppose you are lost in the mountains in a dense fog; you **can only feel the slope** of the ground below your feet. A good strategy to get to the bottom of the valley quickly is to go downhill in the **direction of the steepest slope**. This is what the Gradient descent does.
- Gradient descent is one of the basic type **optimizer** used in ANN for its training.

# Important concepts (Gradient Descent)

- It measures **local gradient** of **error function** with regards to **parameter vector  $\theta$  (weight)**, and it goes in direction of **descending gradient**.
- Once **gradient is zero**, you have **reached a minimum!**
- You **start by filling  $\theta$  with random values** (this is called random initialization), and then you **improve it gradually**, taking one baby step at a time, **each step attempting to decrease the cost function** (e.g., the MSE), until the algorithm **converges to a minimum** (see Figure 4-3 in book).



# Important concepts (Gradient Descent)



# Important concepts (Gradient Descent)

- Learning rate (a hyperparameter) controls the step size in GD.
  - If the learning rate is too small,
    - algorithm may go through many iterations to converge, which will take a long time.
- Convex Case (Linear Regression)
  - Cost curve is a U-shape (convex).
  - Gradient Descent → always finds the single global minimum.
  - No false traps

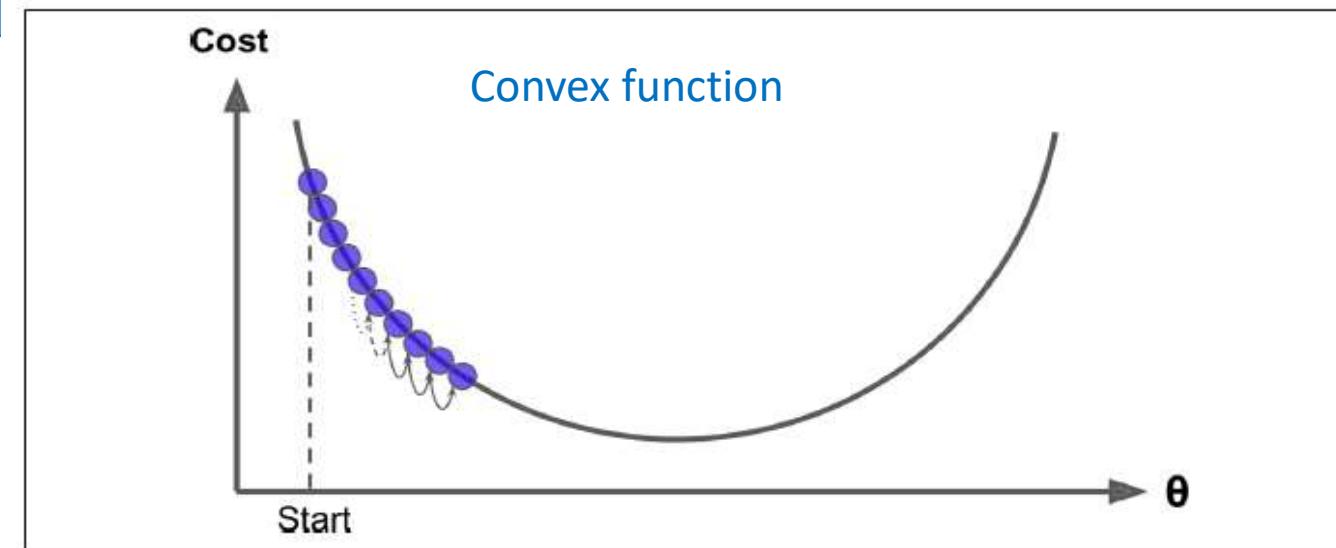


Figure 4-4. Learning rate too small

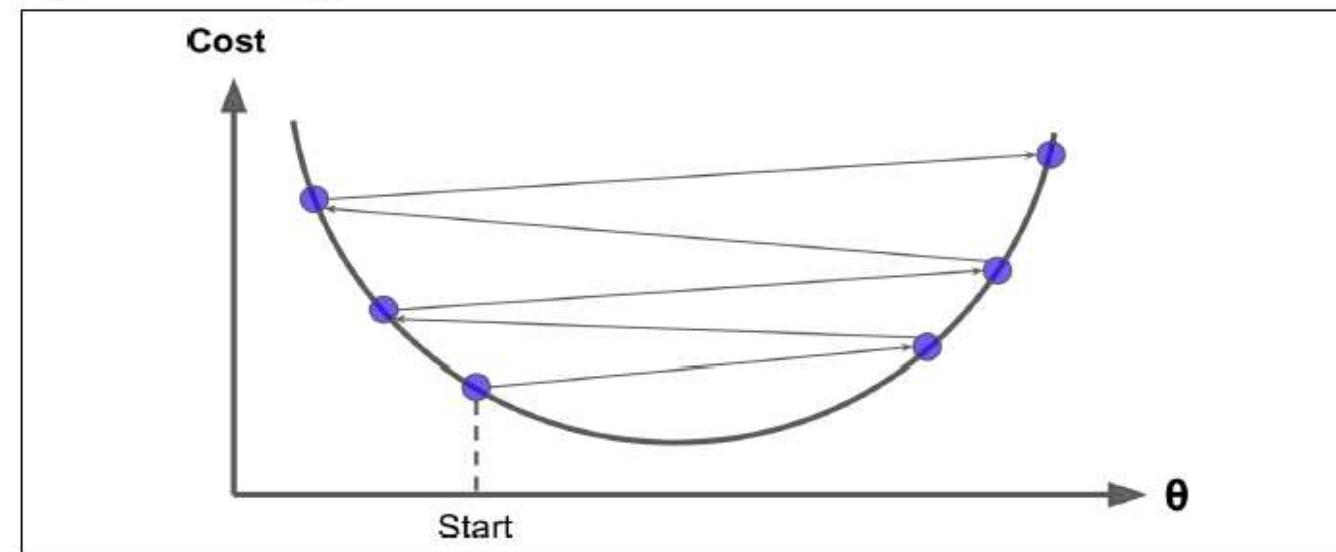


Figure 4-5. Learning rate too large

might make the algorithm diverge

# Important concepts (Gradient Descent)

- In ANNs (with nonlinear activations), the cost surface is usually non-convex (many hills and valleys). That's why optimization is harder.

- Non-Convex Case (ANN with nonlinear activations)**
  - Cost surface has many local minima & saddle points.
  - GD may get stuck or wander.
  - That's why we use tricks like optimizers etc

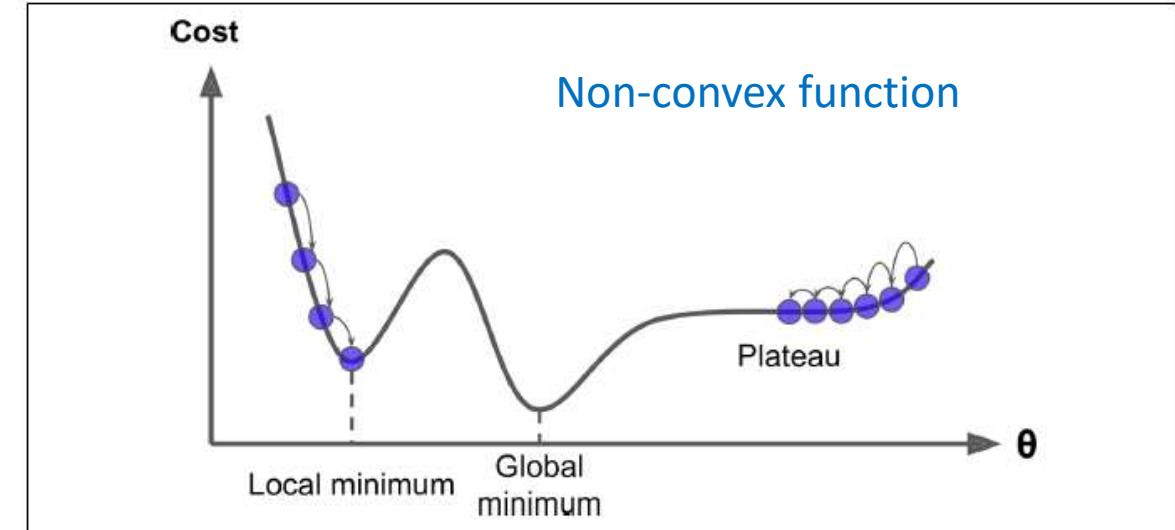
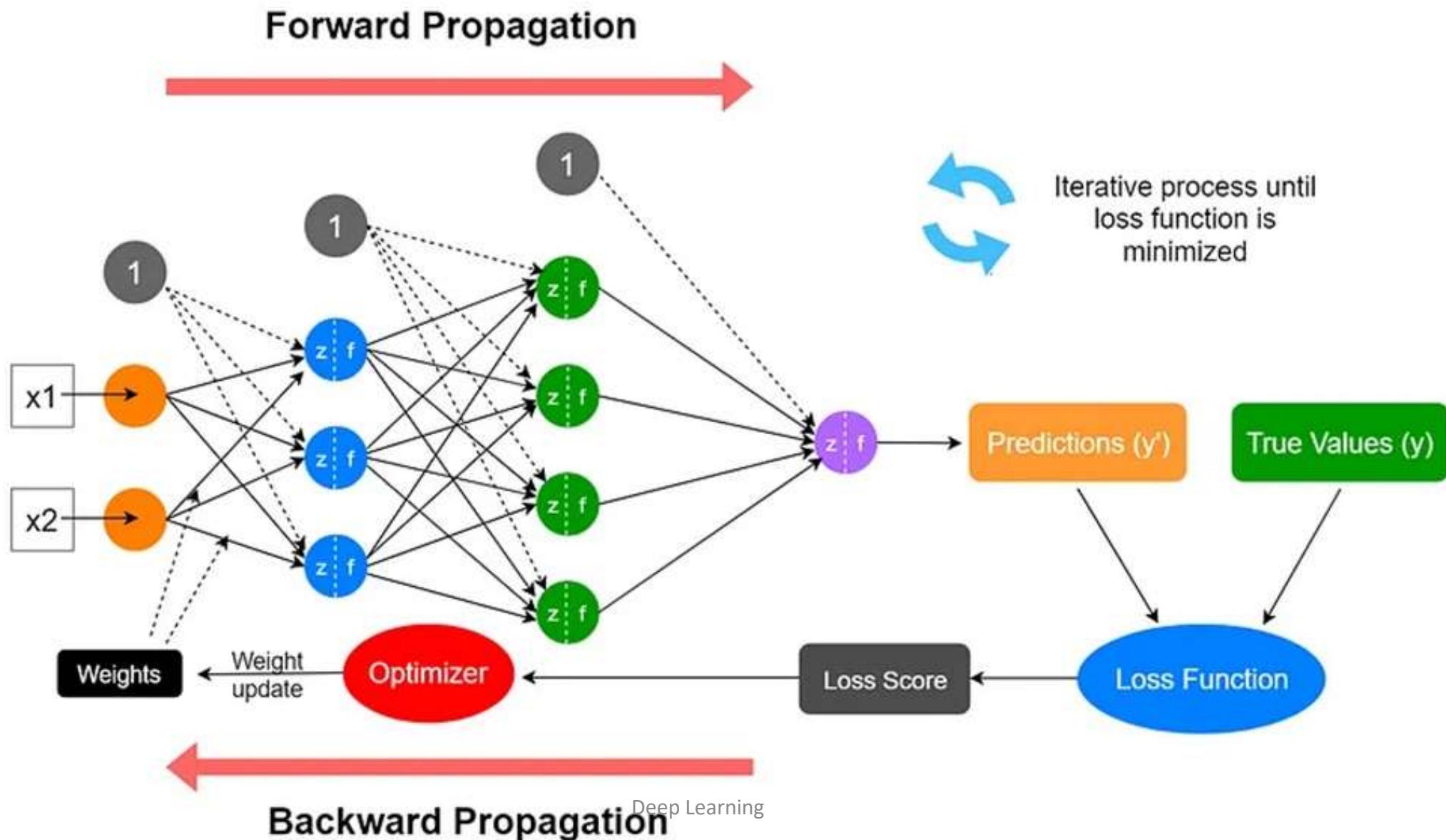


Figure 4-6. Gradient Descent pitfalls

# ANN Learning Process



# Back propagation and weights updating

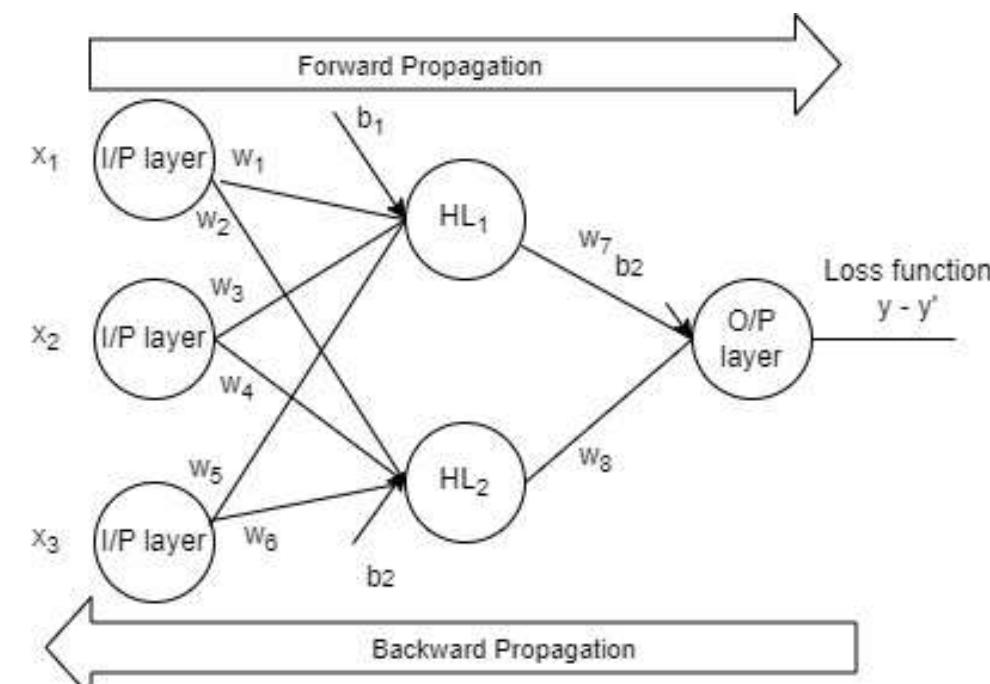
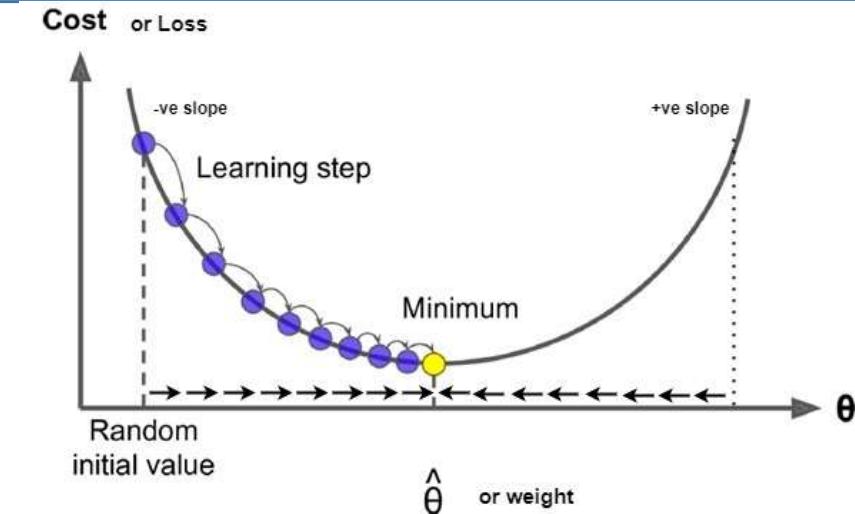
- Weight update formula

$$W_{new} = W_{old} - \eta \frac{\partial L}{\partial W_{old}}$$

- Here  $\eta$  is **learning rate**
- The **partial derivate of loss w.r.t to  $W_{old}$**  is the **slope of tangent line towards minima**
- For  $W_8$  to update its weight the formula would be:

$$W_{8new} = W_{8old} - \eta \frac{\partial L}{\partial W_{8old}}$$

- The learning rate  $\eta$  is **kept small** to avoid big jumps, it is usually kept **0.001**.



# Back propagation and weights updation

- Weight update formula

$$W_{new} = W_{old} - \eta \frac{\partial L}{\partial W_{old}}$$

- From **left side** of the curve the gradient/slope is **-ve**. Therefore:

$$W_{new} = W_{old} - \eta \{-ve\ slope\}$$

$$W_{new} \gg W_{old}$$

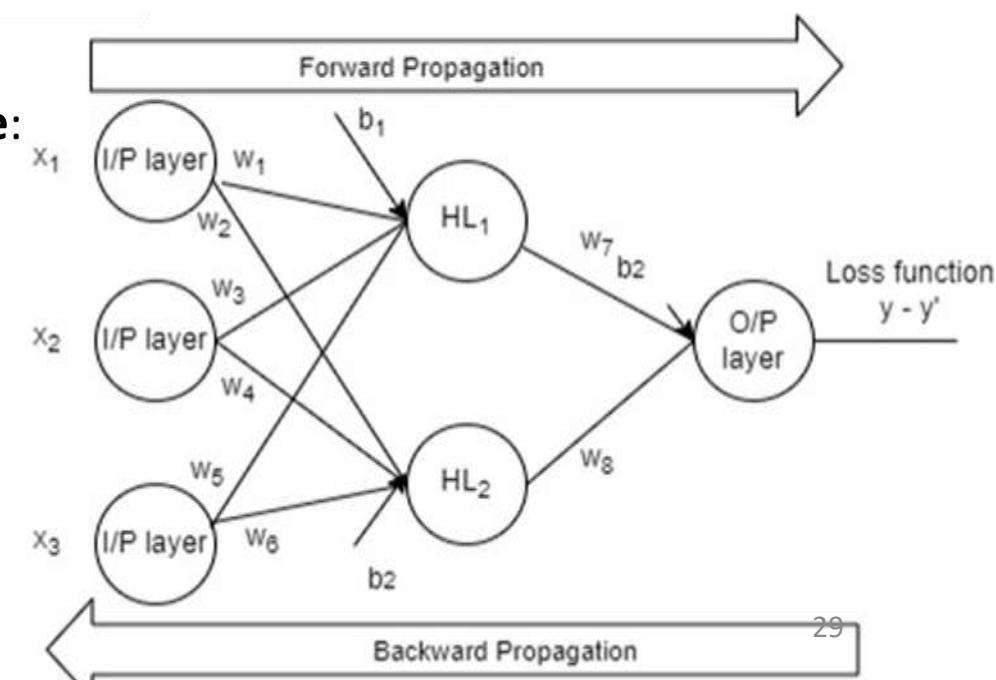
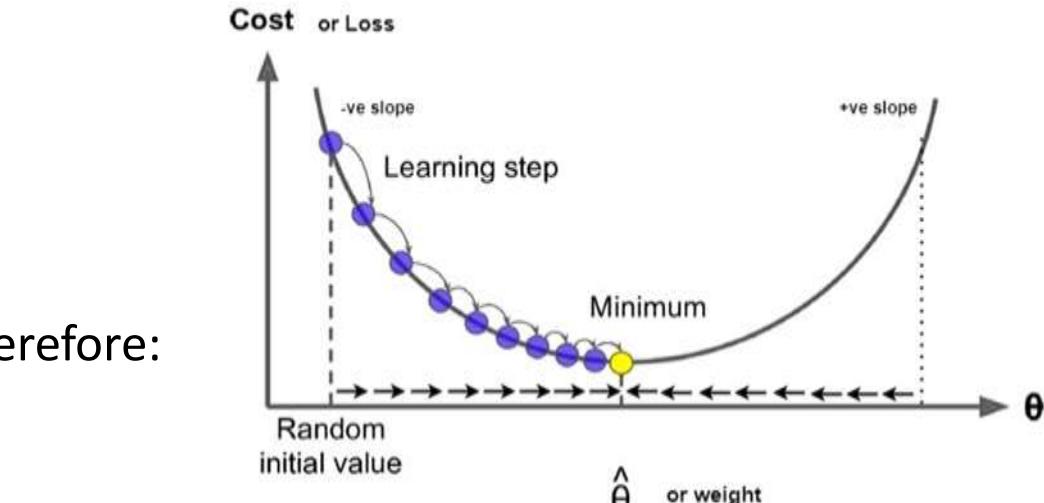
- On-contrary gradient/slope from the **right side curve** is **+ve**:

$$W_{new} = W_{old} - \eta \{+ve\ slope\}$$

$$W_{new} \ll W_{old}$$

- When updated W reaches **global minima**:

$$W_{new} = W_{old}$$



# Chain rule of derivatives

- The chain rule allows us to **differentiate a vast array of functions** that are expressed as **compositions of simpler functions**.
- Without the chain rule, finding derivatives of such functions would be **extremely difficult**, if not impossible. :

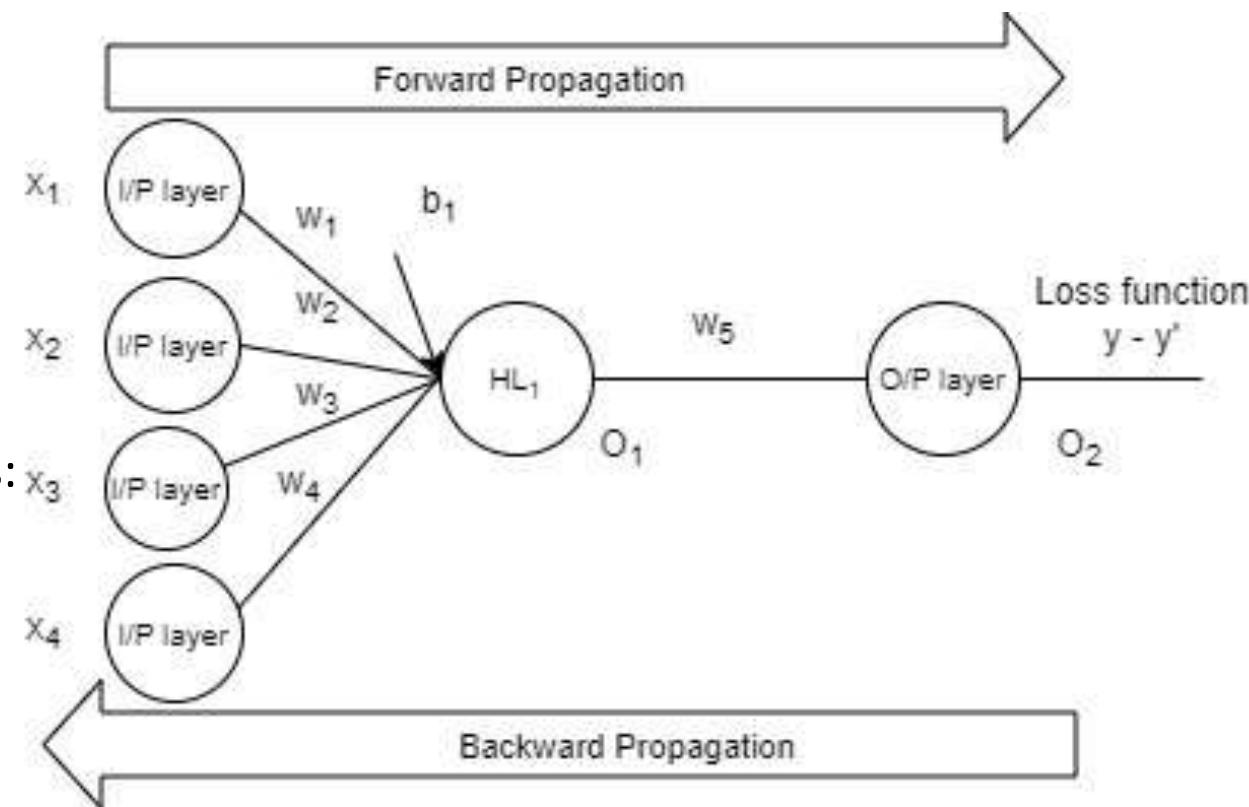
$$W_{new} = W_{old} - \eta \frac{\partial L}{\partial W_{old}}$$

$$W_{5new} = W_{5old} - \eta \boxed{\frac{\partial L}{\partial W_{5old}}}$$

- Partial derivative of  $L$  w.r.t to  $W_{5old}$  can be re-written as:

$$\frac{\partial L}{\partial W_{5old}} = \frac{\partial L}{\partial O_2} \times \frac{\partial O_2}{\partial W_{5old}}$$

- And this is called Chain rule of derivatives



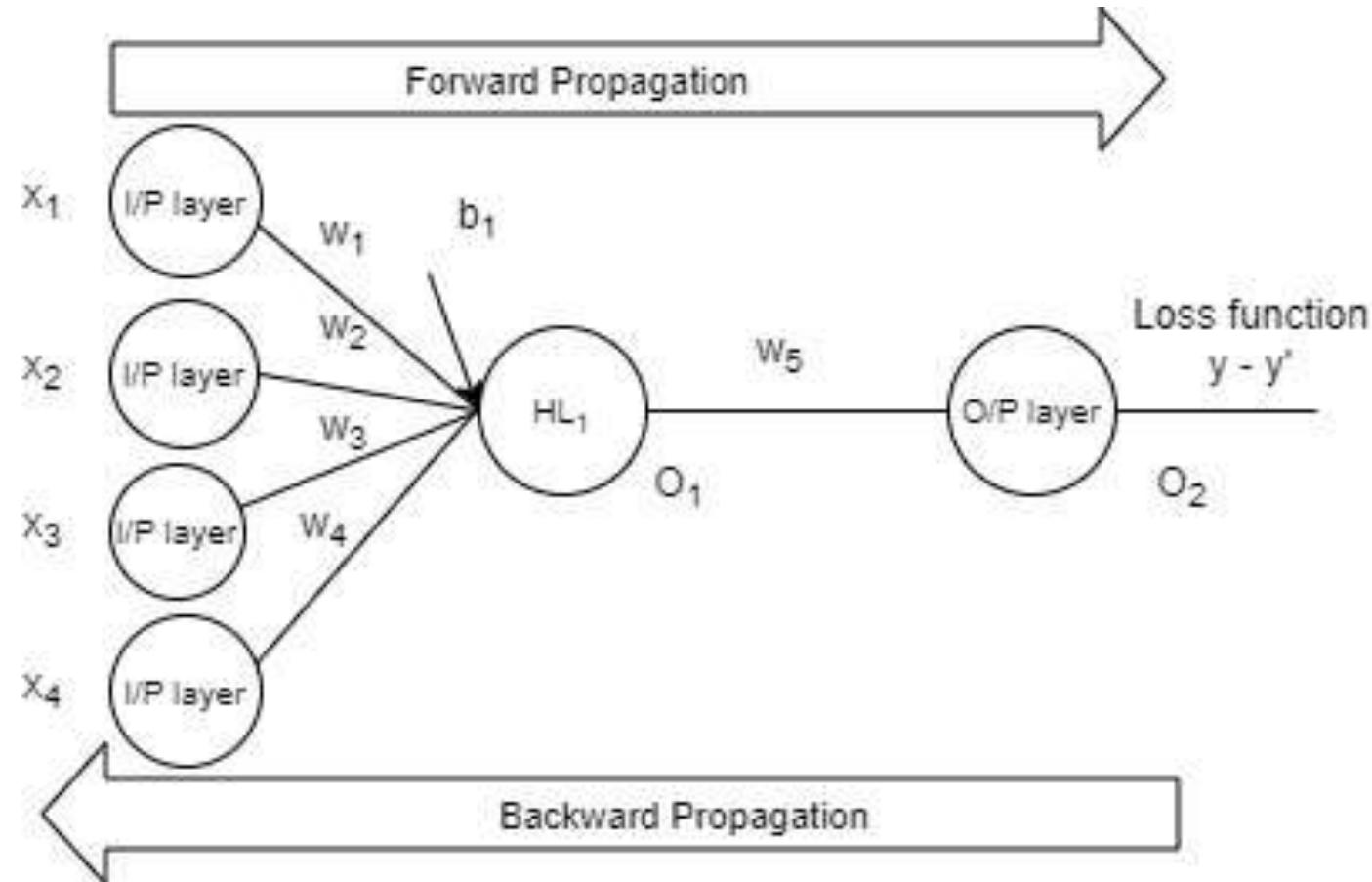
# Chain rule of derivatives

$$W_{new} = W_{old} - \eta \frac{\partial L}{\partial W_{old}}$$

$$W_{5new} = W_{5old} - \eta \boxed{\frac{\partial L}{\partial W_{5old}}}$$

$$\frac{\partial L}{\partial W_{5old}} = \frac{\partial L}{\partial O_2} \times \frac{\partial O_2}{\partial W_{5old}}$$

$$\frac{\partial L}{\partial W_{1old}} = \frac{\partial L}{\partial O_2} \times \frac{\partial O_2}{\partial O_q} \times \frac{\partial O_1}{\partial W_{1old}}$$

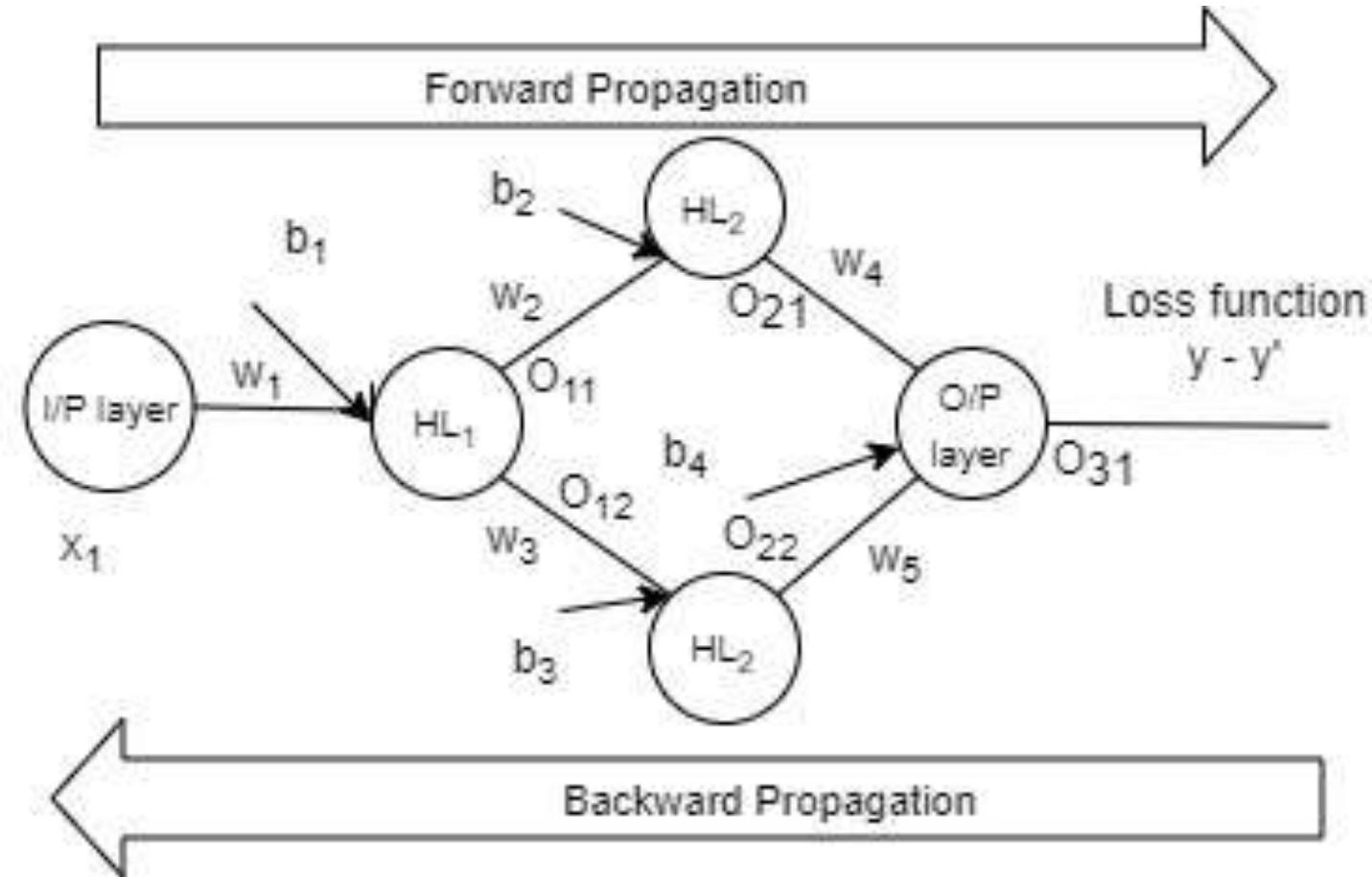


What about partial derivative of L w.r.t W<sub>4</sub>?

# Chain rule of derivatives

$$W_{new} = W_{old} - \eta \frac{\partial L}{\partial W_{old}}$$

What about partial derivative of L w.r.t to  $W_1$ ?



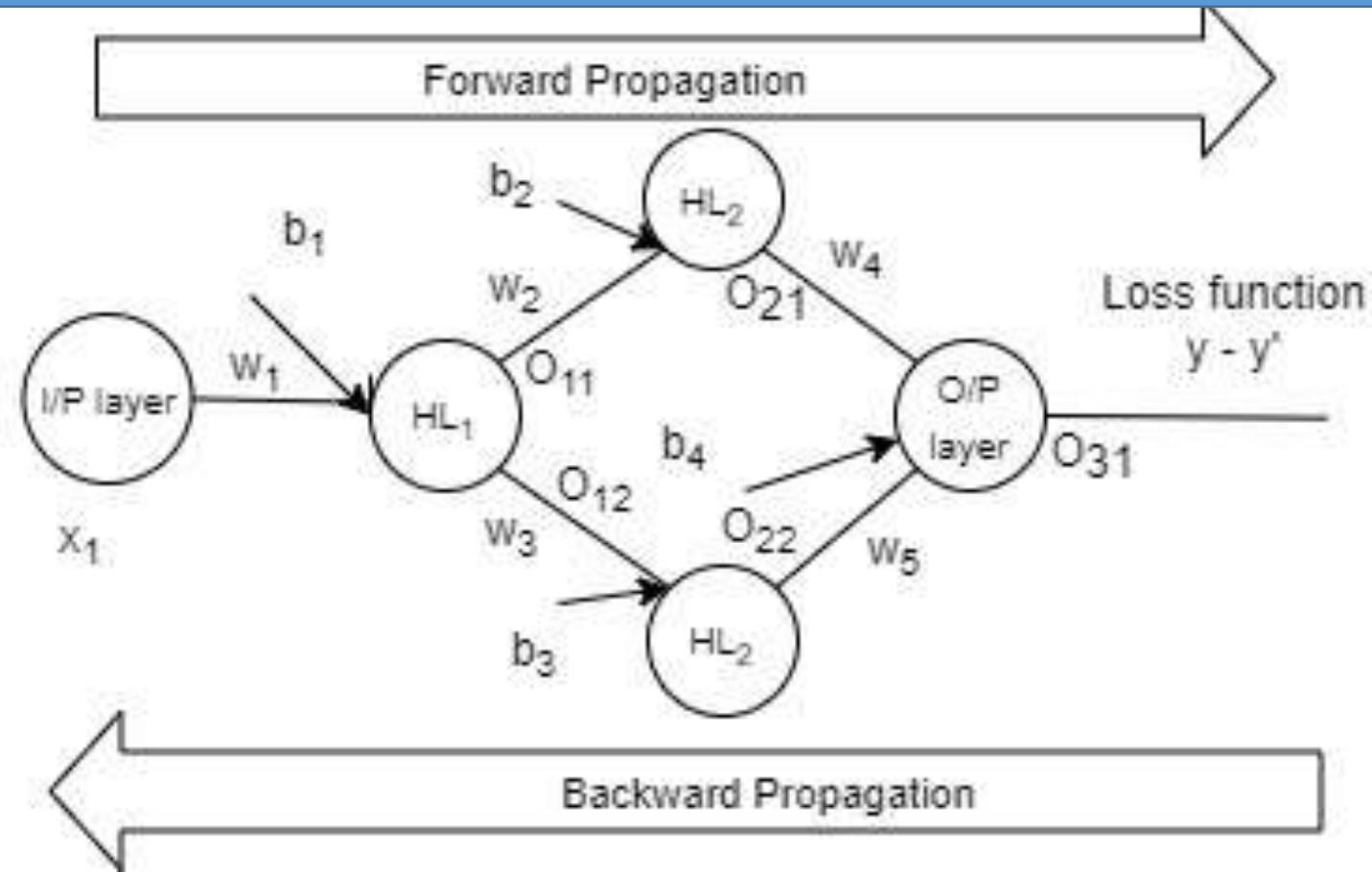
# Chain rule of derivatives

$$W_{new} = W_{old} - \eta \frac{\partial L}{\partial W_{old}}$$

$$W_{1new} = W_{1old} - \eta \frac{\partial L}{\partial W_{1old}}$$

Partial derivative of L w.r.t to  $W_1$  is:

$$\frac{\partial L}{\partial W_{1old}} = \left[ \frac{\partial L}{\partial O_{31}} \times \frac{\partial O_{31}}{\partial O_{11}} \times \frac{\partial O_{11}}{\partial W_{1old}} \right] + \left[ \frac{\partial L}{\partial O_{31}} \times \frac{\partial O_{31}}{\partial O_{22}} \times \frac{\partial O_{22}}{\partial W_{1old}} \right]$$



# Selection of activation function for backpropagation

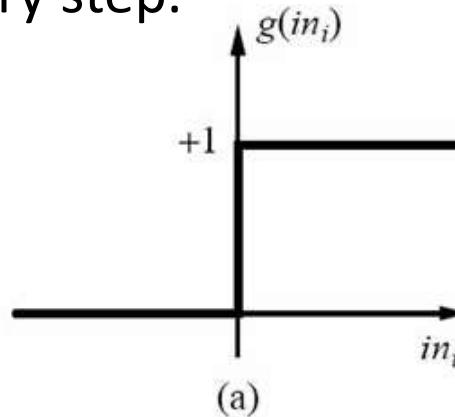
- For each training instance the back propagation algorithm first makes a **prediction (forward pass)**, **measures the error**,
- then **goes through each layer in reverse** to measure the error contribution from each connection (reverse pass), and finally **slightly tweaks the connection weights to reduce the error** (Gradient Descent step).
- In order for this algorithm to work properly,
- **the authors made a key change to the MLP's architecture: they replaced the step function with the logistic function**,

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

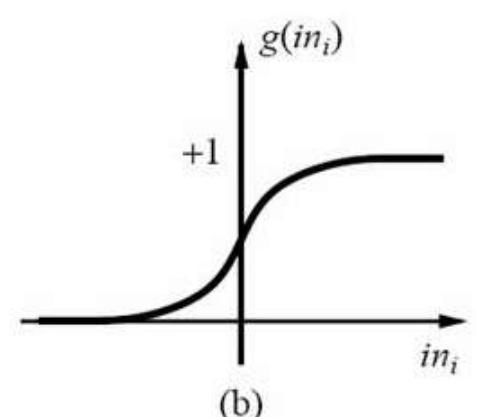
# Selection of activation function for backpropagation

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

- This was essential because the step function **contains only flat segments**, so there is **no gradient to work** with (Gradient Descent cannot move on a flat surface),
- while the logistic function has a **well-defined non-zero derivative every where**, allowing Gradient Descent to make some progress at every step.



step function



sigmoid function

# Vanishing gradient problem

- The back propagation algorithm works by going from the output layer to the input layer, propagating the error gradient on the way.
- Once the algorithm has computed the gradient of the cost function with regards to each parameter in the network, it uses these gradients to update each parameter with a Gradient Descent step.
- Unfortunately, **gradients often gets smaller and smaller as the algorithm progresses down to the lower layers.**
- As a result, the Gradient Descent **update leaves the lower layer connection weights virtually unchanged**, and training never converges to a good solution.
- This is called the **vanishing gradients problem**

# Vanishing gradient problem

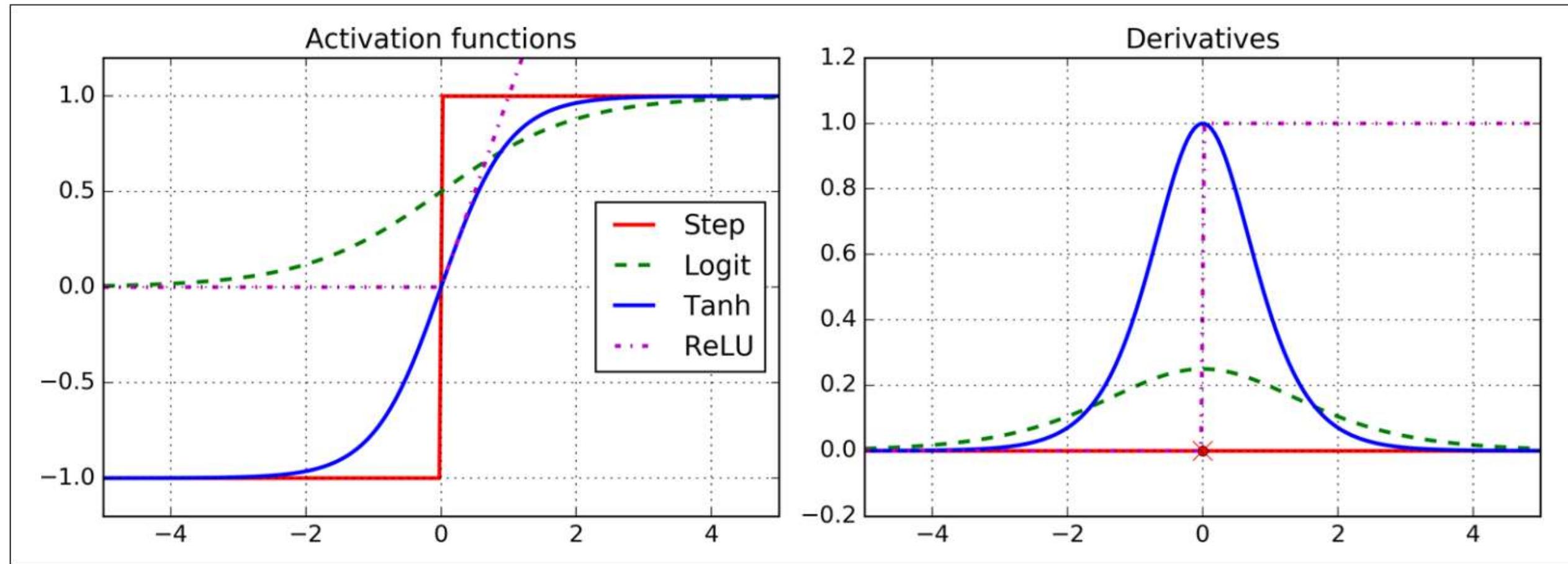


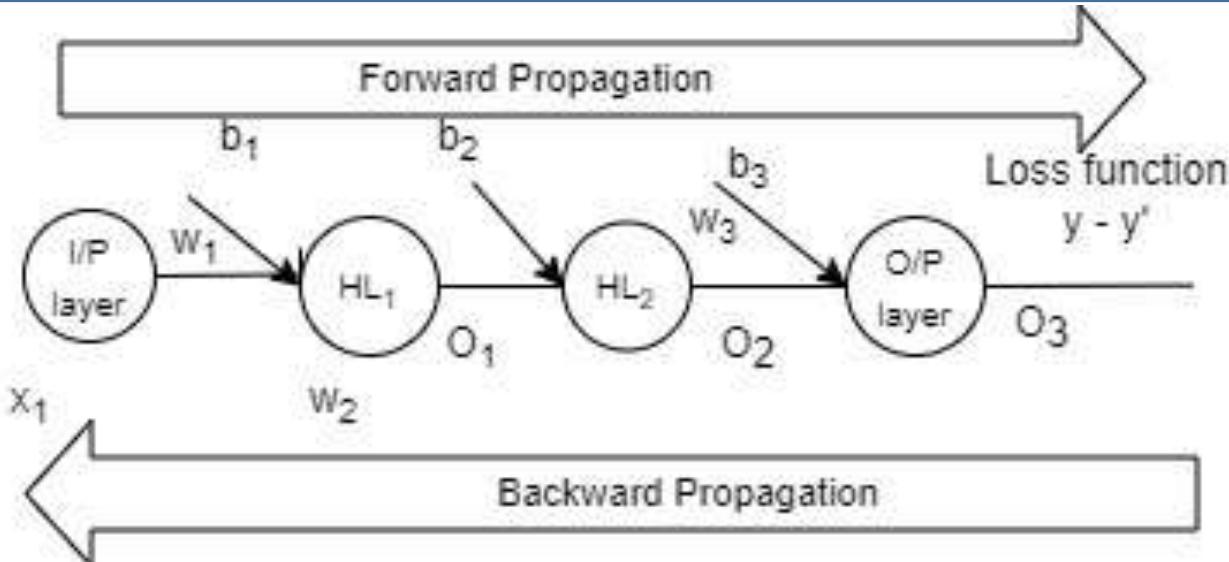
Figure 10-8. Activation functions and their derivatives

# Vanishing gradient problem

- Here the **commonly used value of  $\eta$  is 0.01.**
- As you can see from figure 10-5 the value of **sigmoid is between (0, 1)**, and its **derivative value ranges between (0,0.25).**
- This will **make derivative of L w.r.t  $W_{1old}$**  a very small value **multiplied by 0.01 ( $\eta$  )**.
- This will have **negligible impact** on  $W_{1new}$  and learning becomes very slow or stop.

$$W_{1new} \simeq W_{1old}$$

- This is the **Vanishing gradient problem.**
- A **solution to this is to use other proposed activation functions.**



$$W_{1new} = W_{1old} - \eta \frac{\partial L}{\partial W_{1old}}$$

$$\frac{\partial L}{\partial W_{1old}} = \frac{\partial L}{\partial O_3} \times \frac{\partial O_3}{\partial O_2} \times \frac{\partial O_2}{\partial O_1} \times \frac{\partial O_1}{\partial W_{1old}}$$

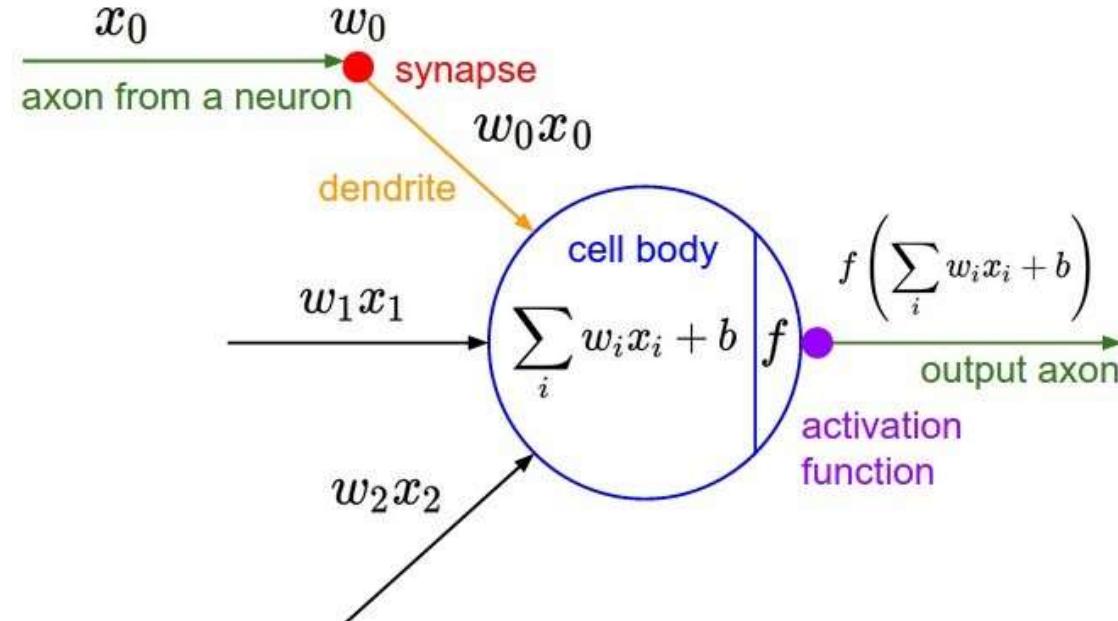
$$\frac{\partial L}{\partial W_{1old}} = \frac{\partial L}{\partial O_3} \times 0.25 \times 0.25 \times 0.25$$

# Activation Functions

- A solution to this is to use other proposed **activation functions**.
- Following are **commonly used activation functions**:
  - Linear function
  - Step function
  - Sigmoid function
  - Tanh function
  - ReLU function
  - Leaky ReLU
  - PReLU (H.W)
  - Swish (H.W)
  - Softmax

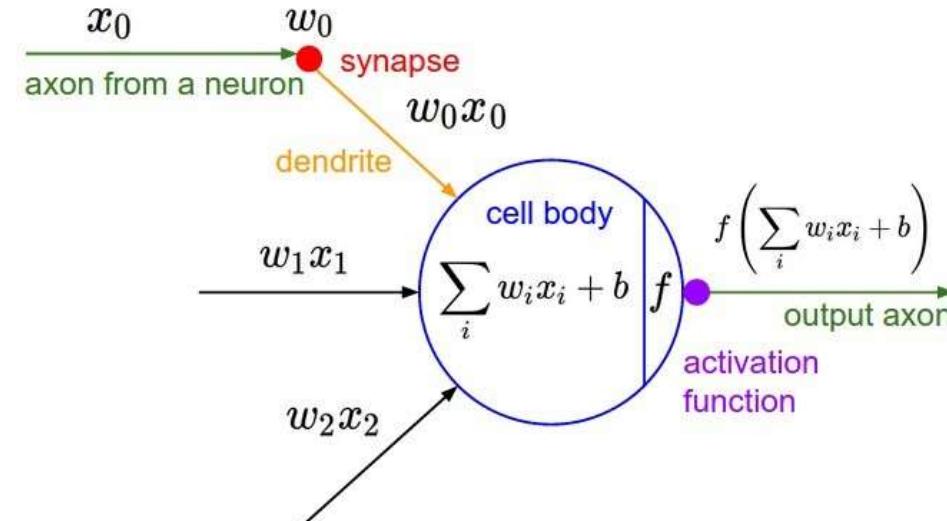
# Activation functions

- Activation functions helps to **determine the output of a neural network**.
- These type of functions are **attached to each neuron** in the network, and **determines whether it should be activated or not**, based on whether each neuron's input is relevant for the model's prediction.
- Activation function also helps to **normalize the output** of each neuron to a range between 1 and 0 or between -1 and 1.



# Activation functions

- The activation function is a **mathematical “gate”** in between the **input feeding the current neuron and its output going to the next layer**.
- It can be as simple as a step function that turns the neuron output on and off, depending on a rule or threshold.
- Neural networks use **non-linear** activation functions, which can help the network **learn complex data, compute and learn** almost any function representing a question, and provide accurate predictions.

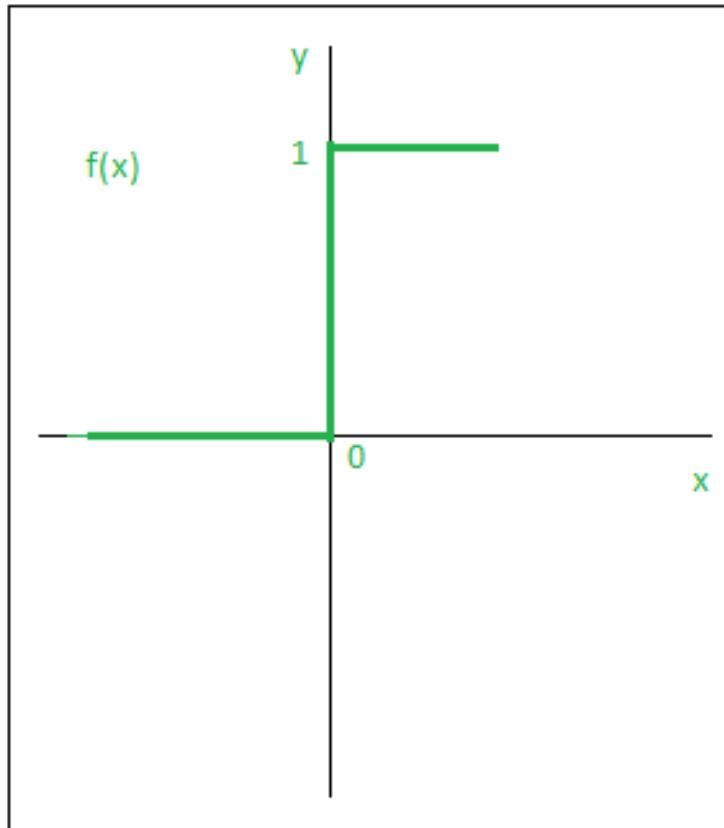


# Activation functions (Linear)

- A **linear activation** function is particularly used in the context of **regression tasks** or in the **final layer** of some neural network architectures.
- Its primary feature is its ability to **output a linear combination of the inputs**.
- The linear activation function can be defined as:  $f(x) = x, \forall x$ 
  - where  $x$  is the input to the activation function
- Characteristics:
  - **Identity:** It is an identity function, meaning that it directly passes the input through to the output.
  - **No Non-Linearity:** Unlike non-linear activation functions such as ReLU, Sigmoid, or Tanh, the linear activation function does not introduce non-linearity into the model. This means that the output is a direct linear transformation of the input.
  - **Range:** The output range of a linear activation function is unbounded, similar to its input range. There is no squashing or normalization of values.

# Activation functions (Step)

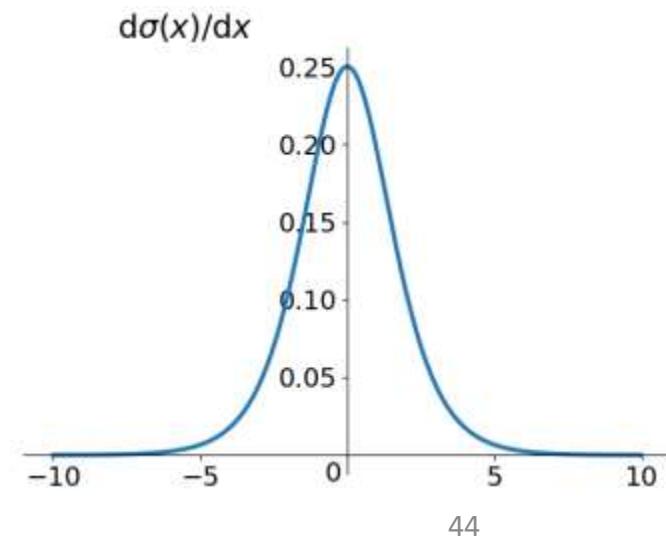
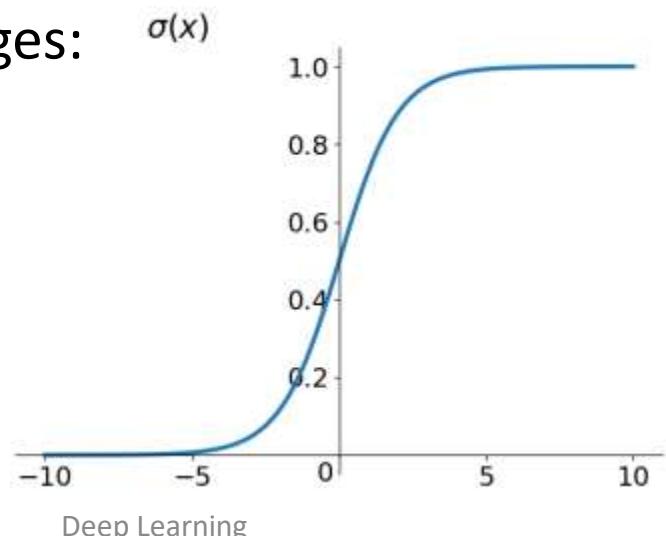
- **Step Function** is one of the simplest kind of activation functions.
- In this, we consider a threshold value and if the value of net input say  $y$  is greater than the threshold then the neuron is activated.
- $f(x) = 1, \text{if } x \geq 0$
- $f(x) = 0, \text{if } x < 0$



# Activation functions (Sigmoid)

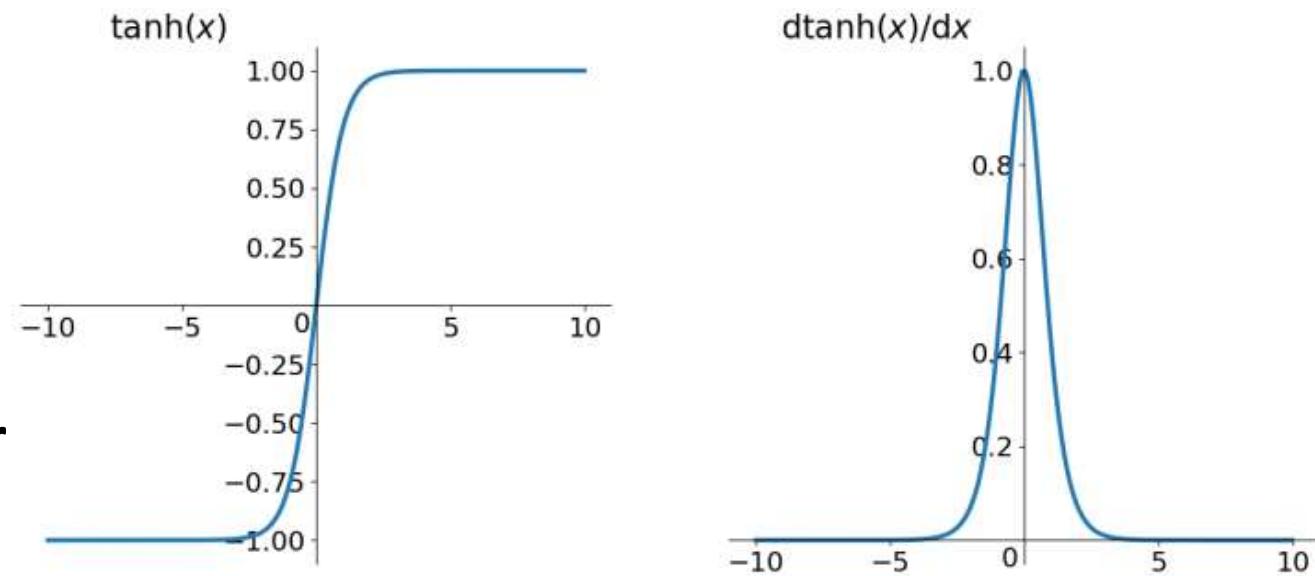
- In the sigmoid function, we can see that its output is in the **open interval (0,1)**.
- The function output **is not centered on 0**, which will reduce the efficiency of weight update.
- The sigmoid function **performs exponential operations**, which is slower for computers.
- Advantages of Sigmoid Function :-
  - **Smooth gradient**, preventing “jumps” in output values.
  - Output values **bound between 0 and 1**, normalizing the output of each neuron.
  - **Clear predictions**, i.e very close to 1 or 0.
- Sigmoid has three major disadvantages:
  - Prone to **gradient vanishing**
  - Function output is **not zero-centered**
  - Relatively **time consuming** due to e

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



# Activation functions (tanh)

- The output interval of tanh is **(-1, 1)**, and the whole function is **0-centric**, which is better than sigmod.
- In general binary classification problems, the **tanh function is used for the hidden layer** and the **sigmod function is used for the output layer**.
- This has **solved vanishing gradient** problem to **some extent**.
- However, with **deep neural network** with more hidden layer the **problem persist**.
- Computational expensive due to e.

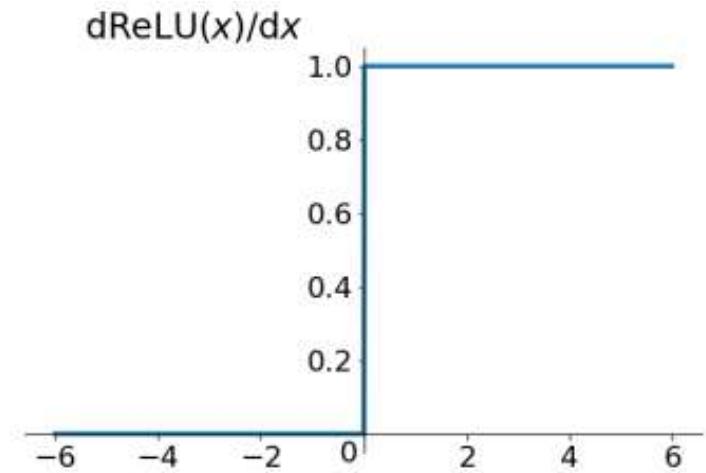
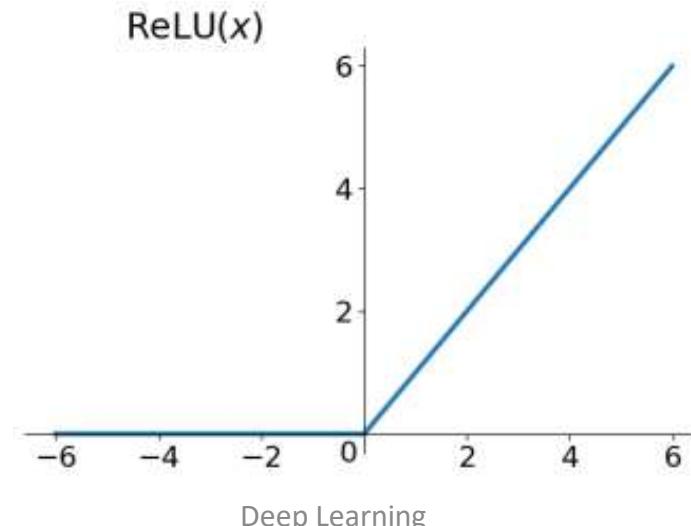


$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

# Activation functions (ReLU)

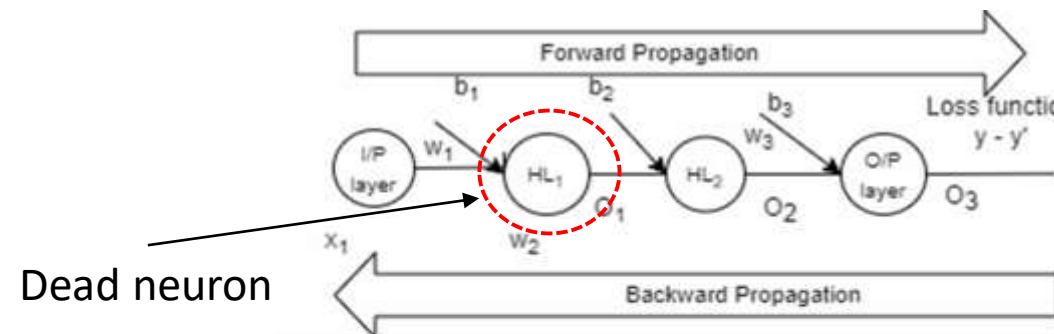
- The ReLU (**Rectified Linear Unit**) function is a function that takes the maximum value.
- Compare to tanh and sigmoid it has following advantages:
  - When the **input is positive, there is no gradient saturation problem.**
  - It is **much faster** than sigmod and tanh. (Sigmod and tanh need to calculate the exponent, which will be slower.)

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



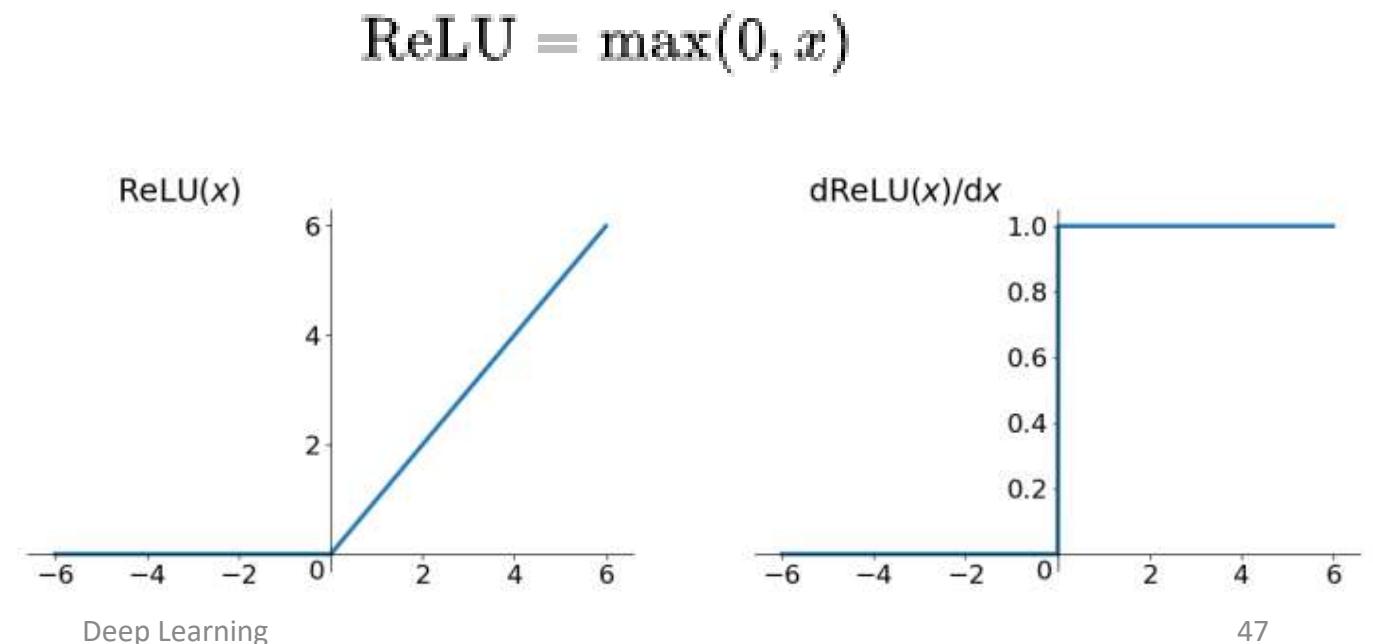
# Activation functions (ReLU)

- Disadvantages
  - When the input is negative, ReLU is **completely inactive**, which means that once a negative number is entered, ReLU will die.
  - But in the backpropagation process, if you enter a negative number, the gradient will be completely zero, which has the same problem as the sigmoid function and tanh function.



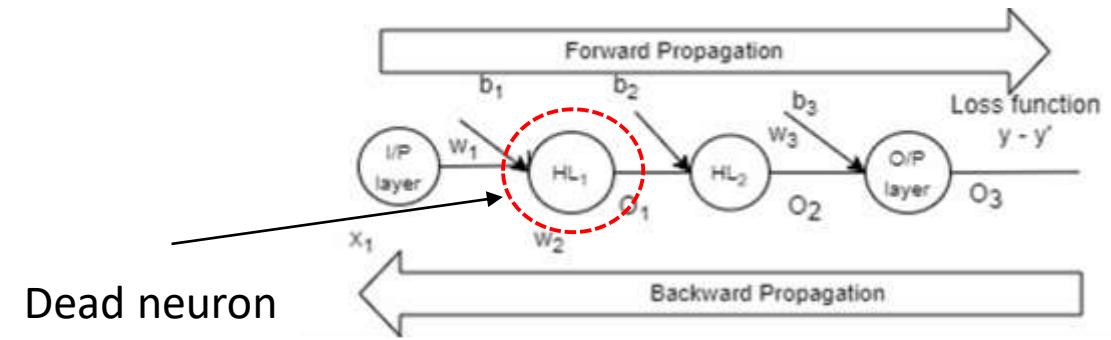
$$W_{new} = W_{old}$$

$$\frac{\partial L}{\partial W_{1old}} = \frac{\partial L}{\partial O_3} \times \boxed{\frac{\partial O_3}{\partial O_2}} \times \frac{\partial O_2}{\partial O_1} \times \frac{\partial O_1}{\partial W_{1old}}$$



# Activation functions (ReLU)

- Disadvantages
  - We find that the output of the ReLU function is **either 0 or a positive number**, which means that the ReLU function is **not a 0-centric function**.
  - If  $z$  is +ve then derivative of  $\text{ReLU}(z)$  is 1
  - If  $z$  is -ve then derivative of  $\text{ReLU}(z)$  is 0



Dead neuron

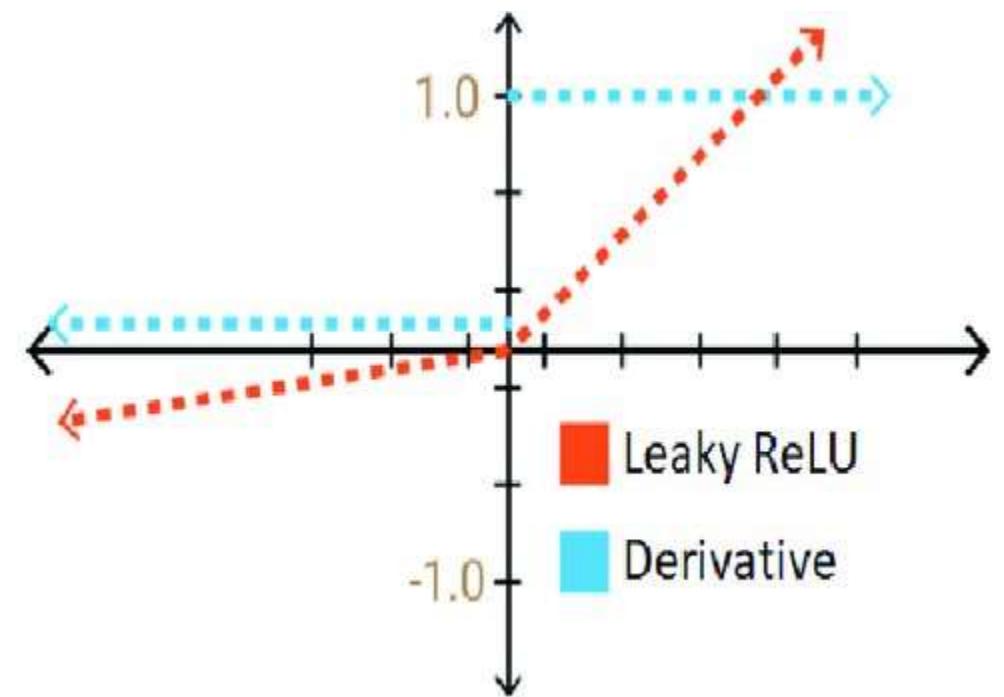
$$W_{new} = W_{old}$$

$$\frac{\partial L}{\partial W_{1old}} = \frac{\partial L}{\partial O_3} \times \boxed{\frac{\partial O_3}{\partial O_2}} \times \frac{\partial O_2}{\partial O_1} \times \frac{\partial O_1}{\partial W_{1old}}$$

# Activation functions (Leaky ReLU)

- In order to solve the Dead ReLU Problem, people proposed to set the first half of ReLU  $0.01x$  instead of 0.
- In theory, Leaky ReLU has all the advantages of ReLU,
- plus there will be no problems with Dead ReLU,
- The disadvantage is it is not zero centric.

$$f(x) = \max(0.01x, x)$$



# Activation functions (SoftMax)

- The **softmax function** is different from other activation functions as it is **placed at the last to normalize the output.**
- We can use **other activation functions in combination with Softmax** to produce the **output in probabilistic form.**
- It is used in **multiclass classification** and generates an **output of probabilities whose sum is 1.**
- The denominator of the Softmax function combines all factors of the original output value, which means that the different probabilities obtained by the Softmax function are related to each other.

$$S(x_j) = \frac{e^{x_j}}{\sum_{k=1}^K e^{x_k}}, j = 1, 2, \dots, K$$

# Activation functions (Code)

- See pynb file from elearning for codes of the various activation functions.
- **H.W**
- Your task is to increase the number of inputs and generate plots of these activation functions in the same file.
- Submit the solution to elearning.
- Deadline 1 week after todays class.

# Summary

- Biological Perceptron
- Discussed basic concepts of ANN
- Single Layer Perception
- Training and error for SLP