Deep Learning
CSC-Elective

Instructor : Dr. Muhammad Ismail Mangrio

Slides prepared by Dr. M Asif Khan

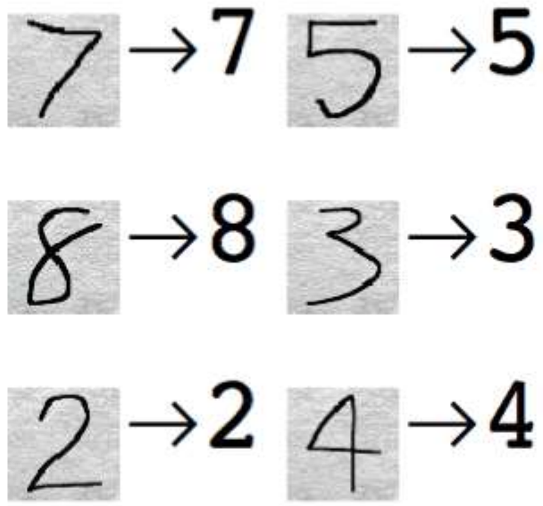
ismail@iba-suk.edu.pk

Week 2-3

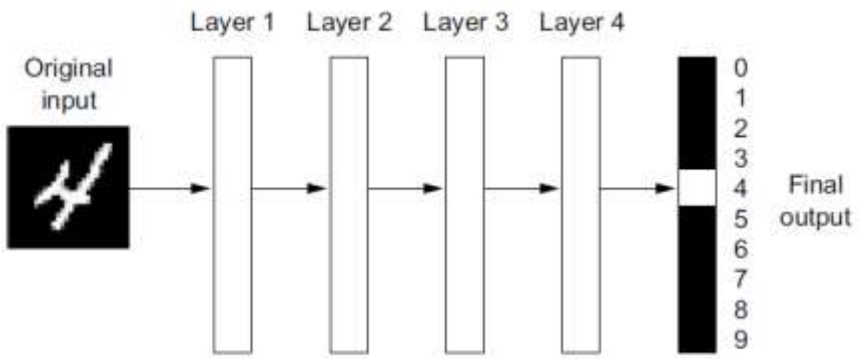
Contents

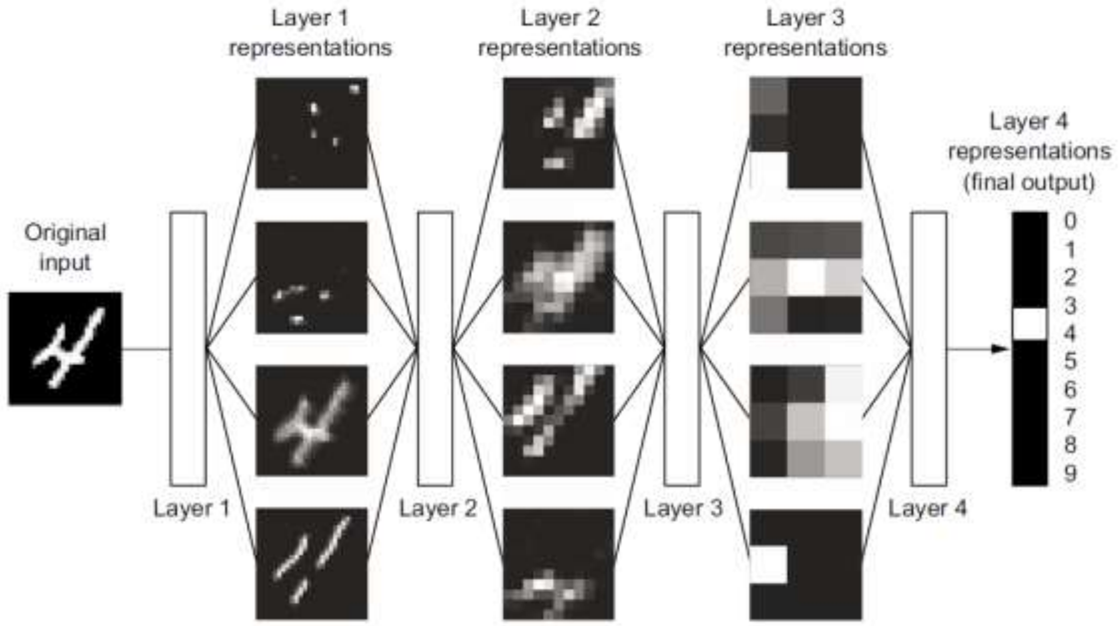
Learning Rules and Representation from Data

- 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



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

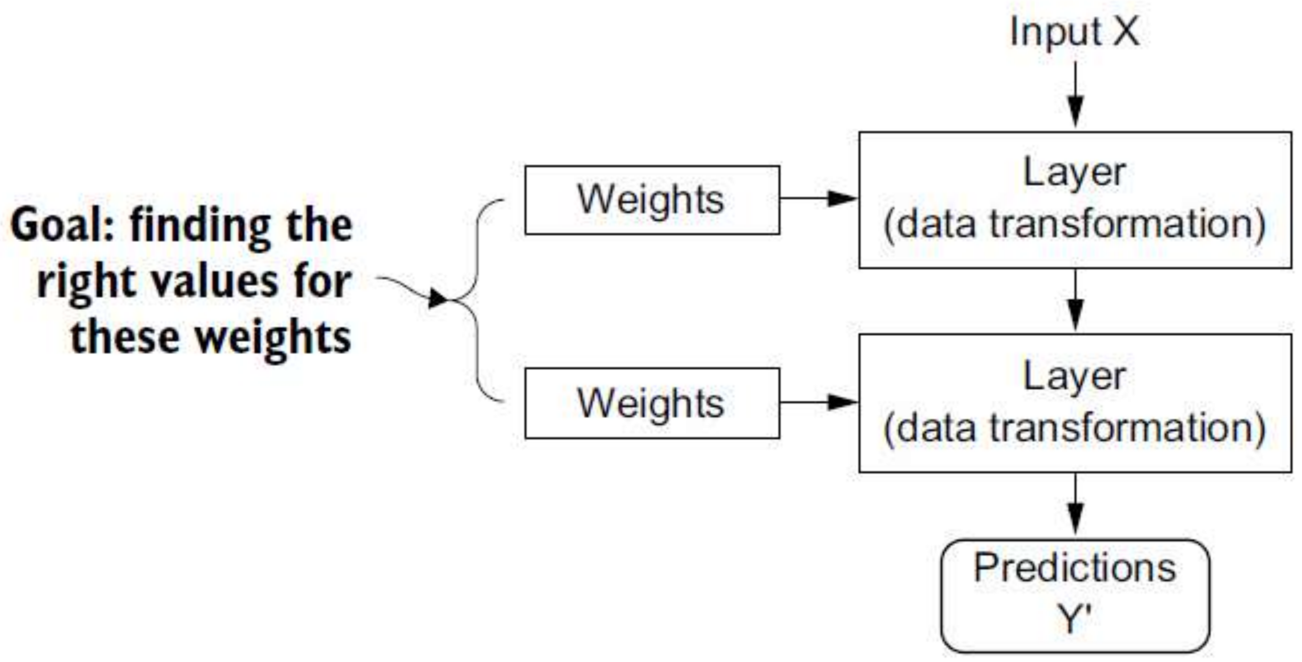

Input X
Weights
Layer (data transformation)
Weights
Layer (data transformation)
Predictions Y'
Goal: finding the right values for these weights

Figure 1.7 A neural network is parameterized by its weights.

Understanding how deep learning works, in three figures

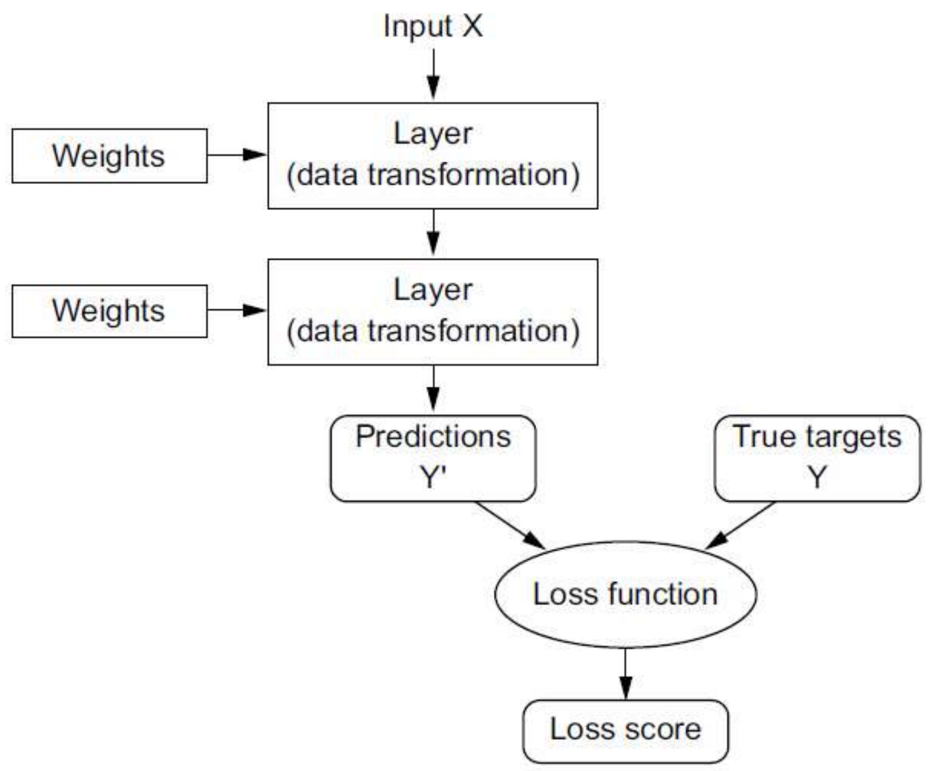


Figure 1.8 A loss function measures the quality of the network's output.

Understanding how deep learning works, in three figures

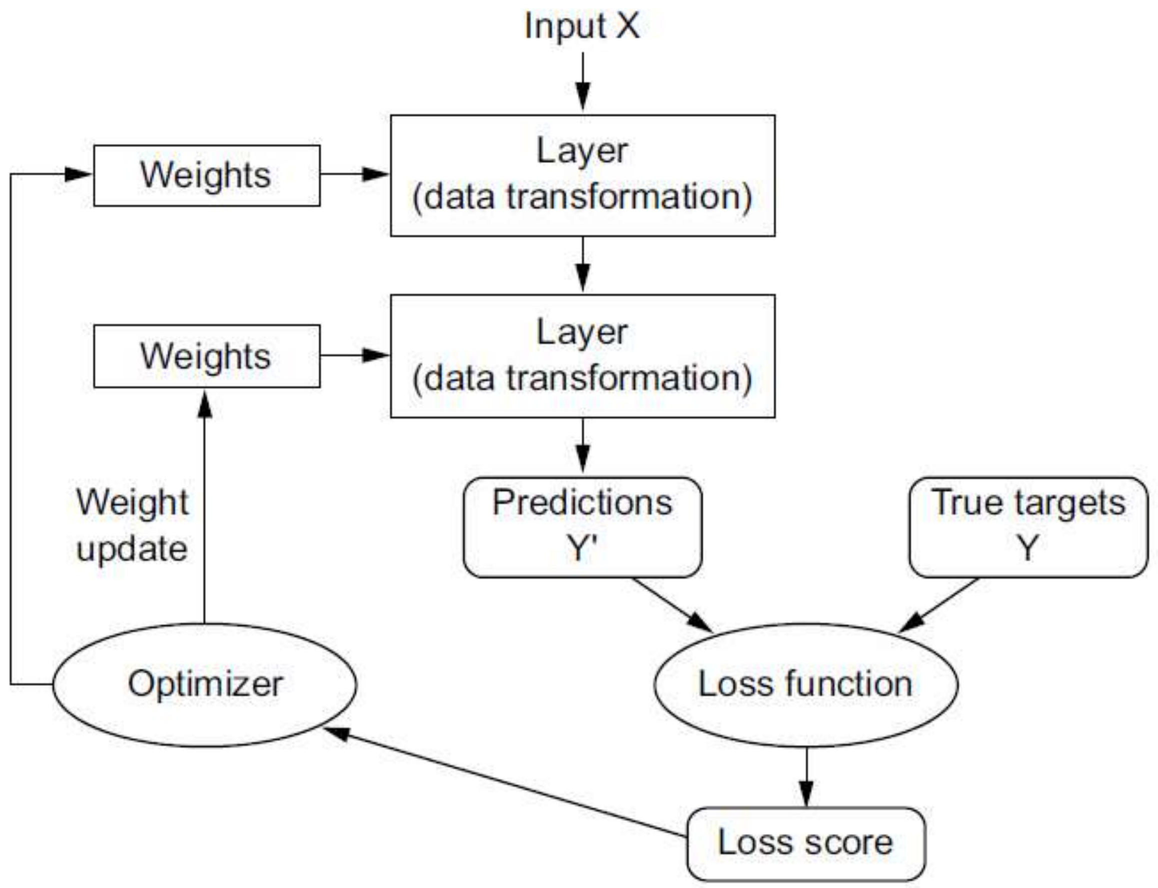
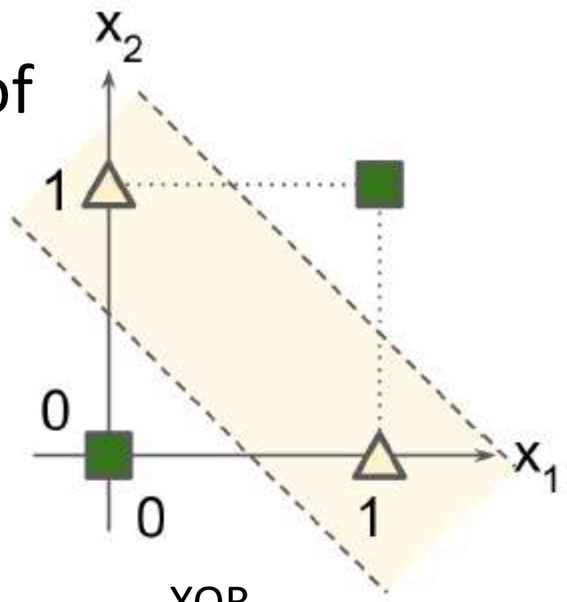


Figure 1.9 The loss score is used as a feedback signal to adjust the weights.

Artificial Neural Network (Multi Layer Perceptron)

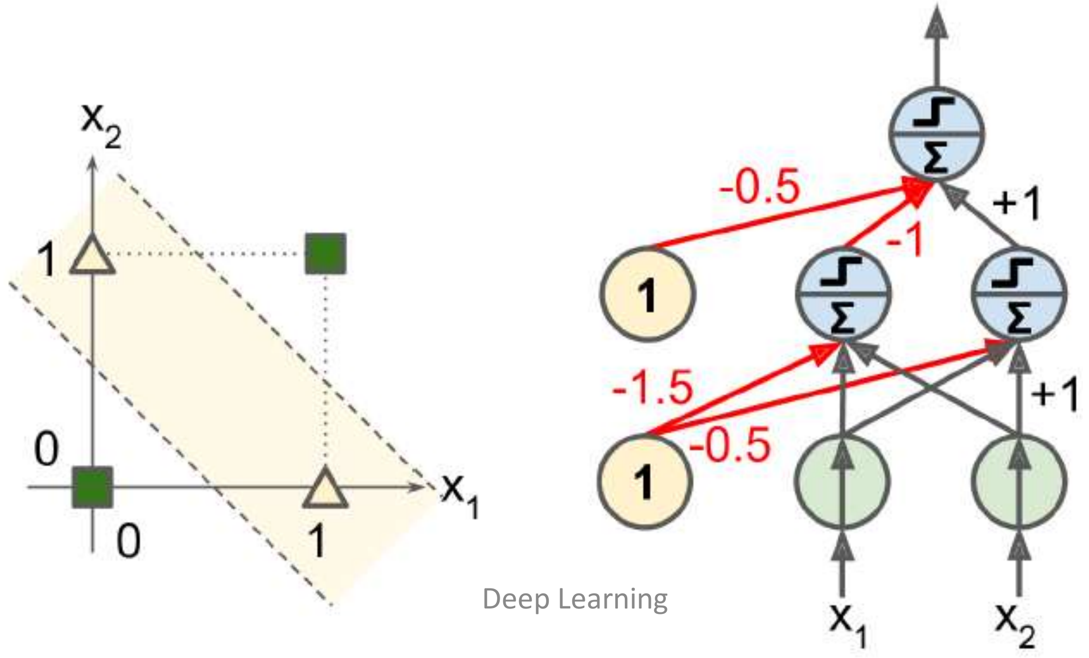
- Contrary to Logistic Regression classifiers, Perceptron **do not output a class probability**; rather, they just make predictions based on a hard threshold.
- This is one of the good reasons to **prefer Logistic Regression over Perceptrons**.
- Marvin Minsky and Seymour Papert highlighted a number 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).



XOR

Artificial Neural Network (Multi Layer Perceptron)

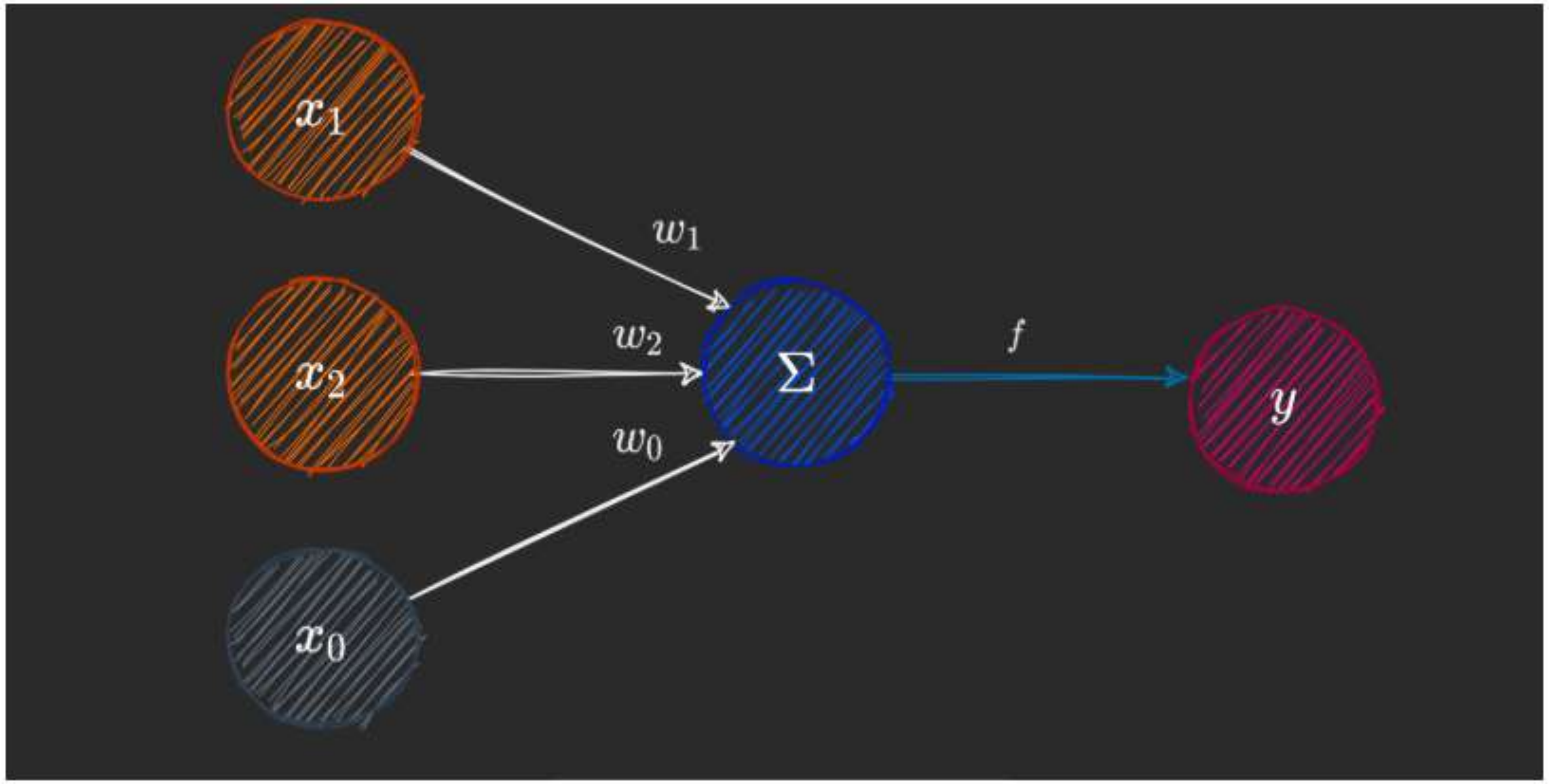
- However, it turns out that some of the limitations of Perceptrons can be eliminated by **stacking multiple Perceptrons**.
- The resulting ANN is called a **Multi-Layer Perceptron** (MLP).
- In particular, an **MLP can solve the XOR problem**, as you can verify by computing the output of the MLP represented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



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

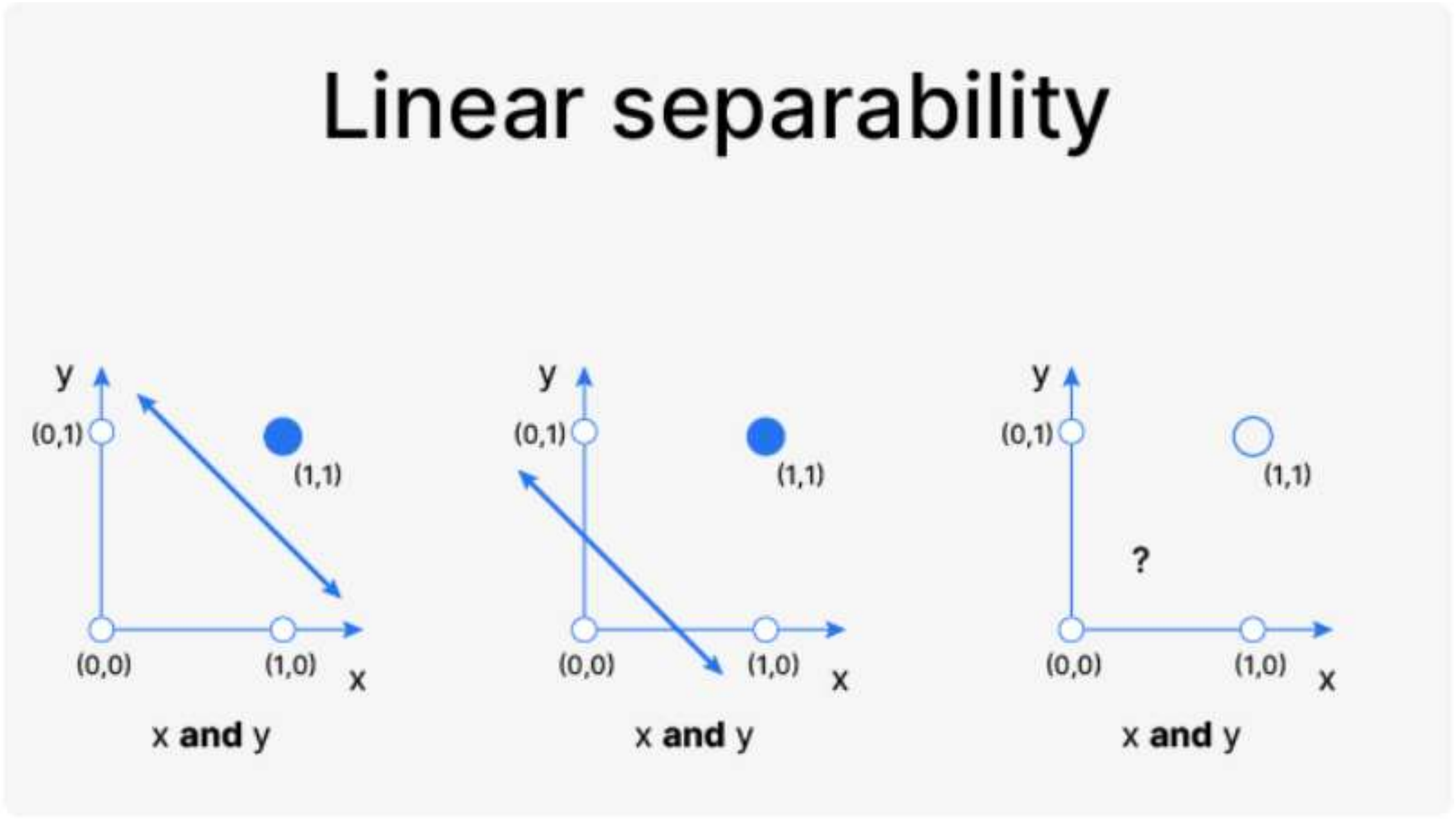
MLP and XOR Problem

Understanding Neural Networks



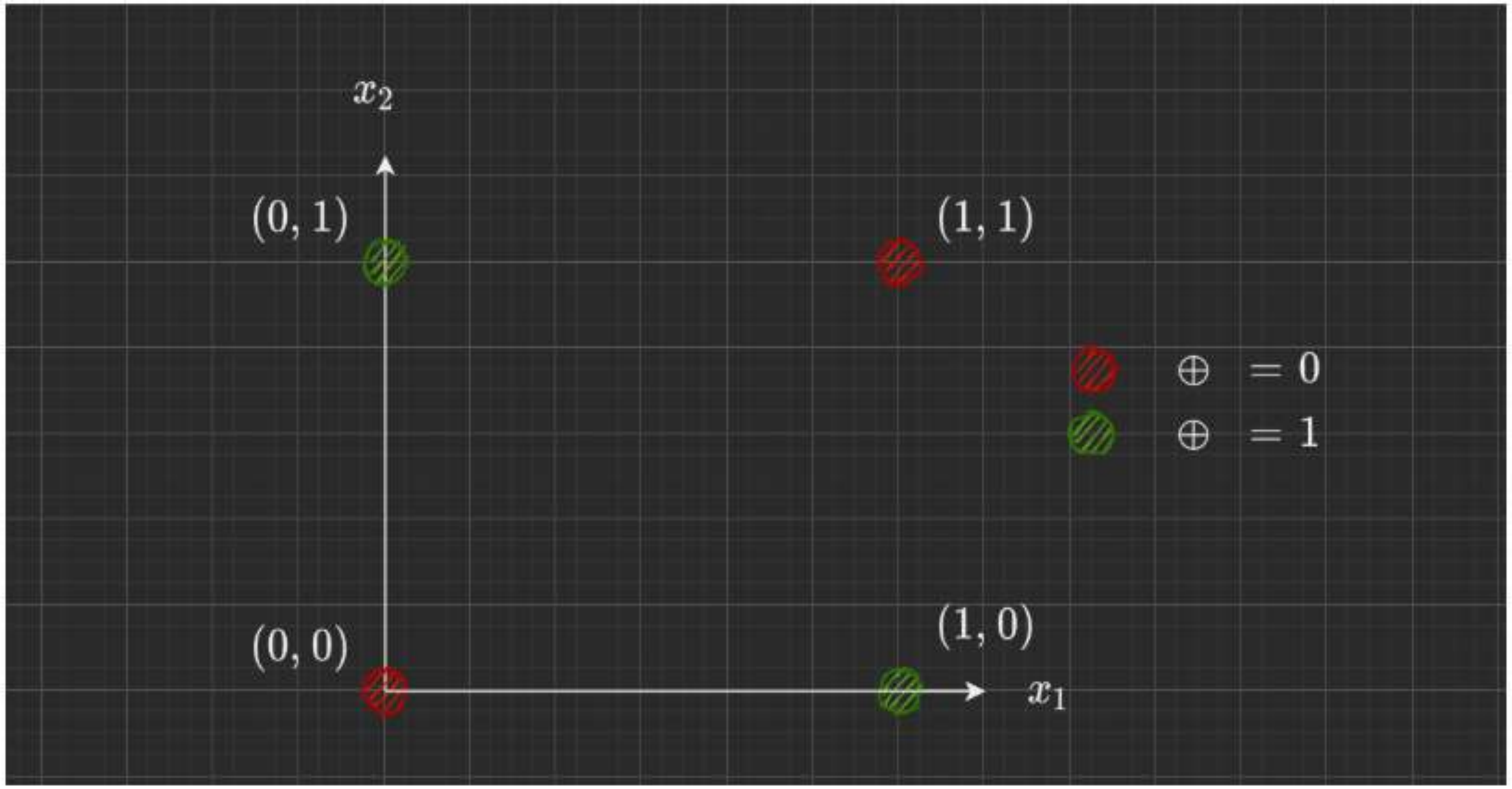
A representation of a single-layer perceptron with 2 input nodes – Image by Author using draw.io

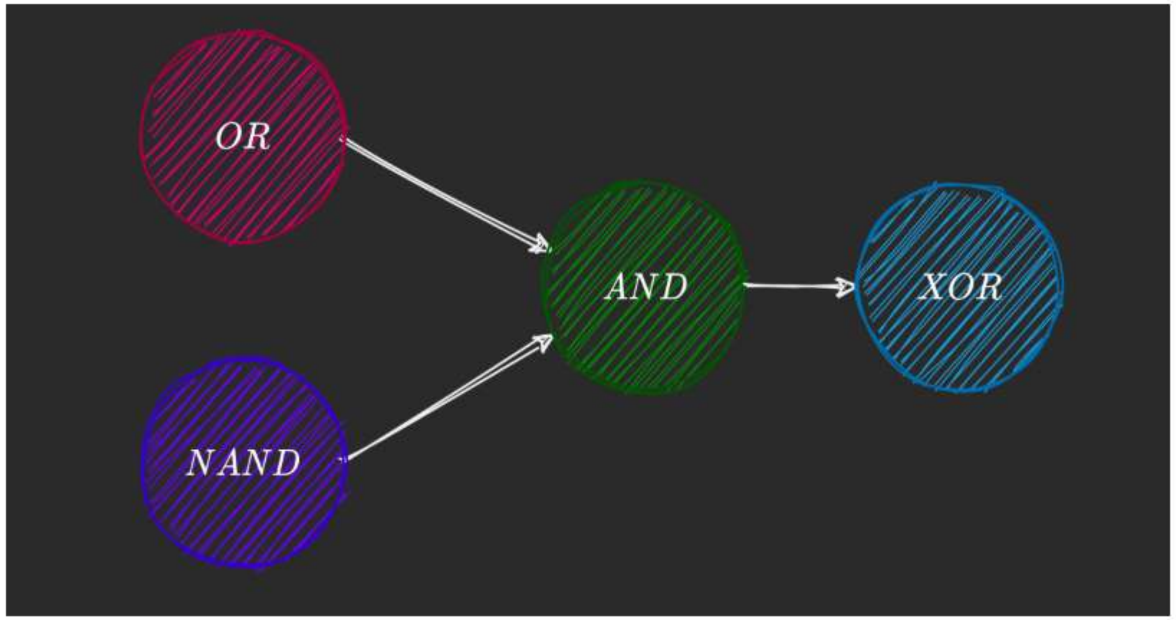
Explaining the XOR Problem



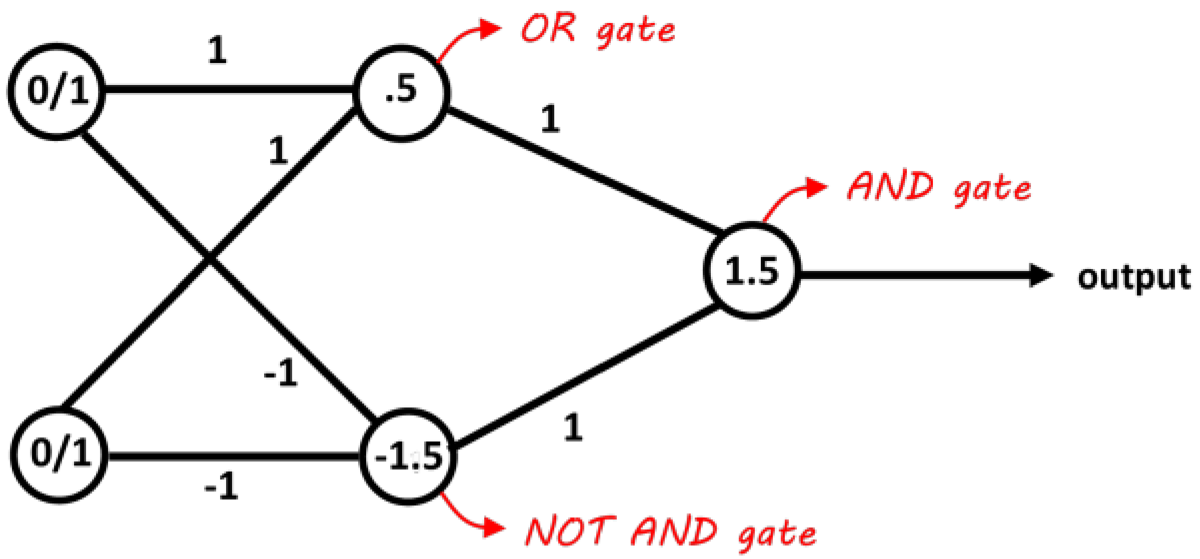
SUKKUR IBA UNIVERSITY
COMPUTER SCIENCE

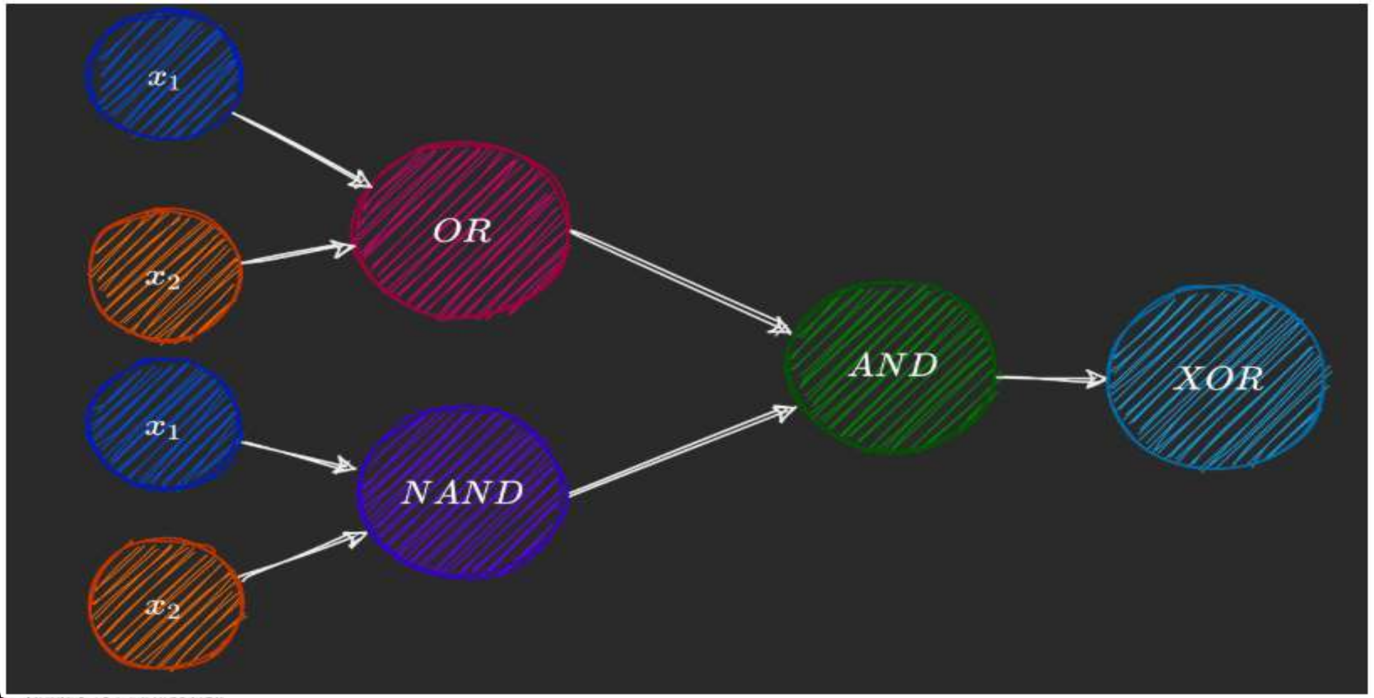
The 2D XOR problem



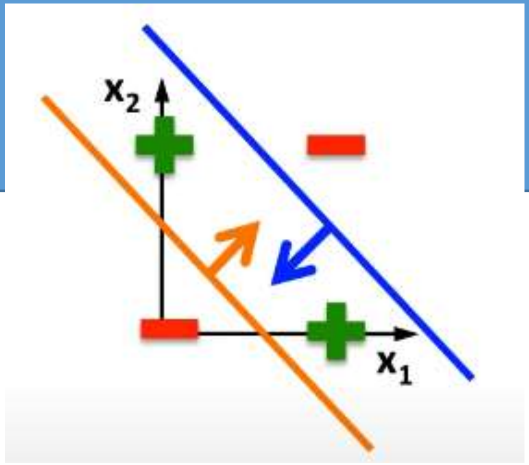

OR
NAND
AND
XOR

XOR Gate

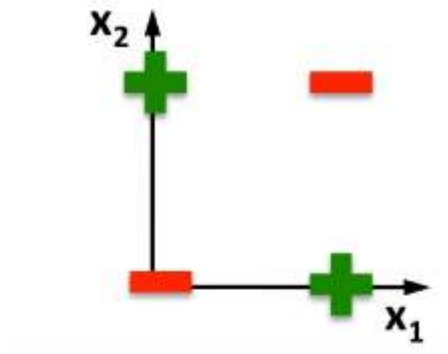

0/1
0/1
.5
-1.5
1.5
1
1
-1
-1
1
1
OR gate
AND gate
NOT AND gate
output

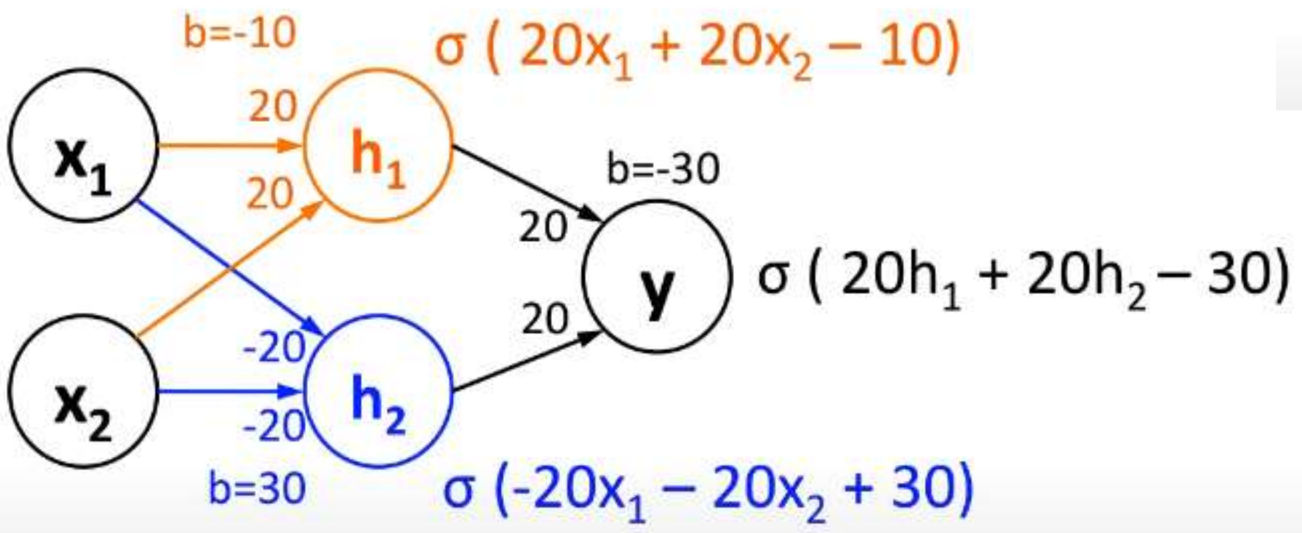

x_1
x_2
x_1
x_2
OR
NAND
AND
XOR

Example: Solving XOR with a hidden layer



Linear classifiers cannot solve this





b=-10
$\sigma(20x_1 + 20x_2 - 10)$
20
20
x_1
h_1
b=-30
20
y
$\sigma(20h_1 + 20h_2 - 30)$
20
x_2
-20
-20
h_2
b=30
$\sigma(-20x_1 - 20x_2 + 30)$

$\sigma(20*0 + 20*0 - 10) \approx 0$	$\sigma(-20*0 - 20*0 + 30) \approx 1$	$\sigma(20*0 + 20*1 - 30) \approx 0$
$\sigma(20*1 + 20*1 - 10) \approx 1$	$\sigma(-20*1 - 20*1 + 30) \approx 0$	$\sigma(20*1 + 20*0 - 30) \approx 0$
$\sigma(20*0 + 20*1 - 10) \approx 1$	$\sigma(-20*0 - 20*1 + 30) \approx 1$	$\sigma(20*1 + 20*1 - 30) \approx 1$
$\sigma(20*1 + 20*0 - 10) \approx 1$	$\sigma(-20*1 - 20*0 + 30) \approx 1$	$\sigma(20*1 + 20*1 - 30) \approx 1$

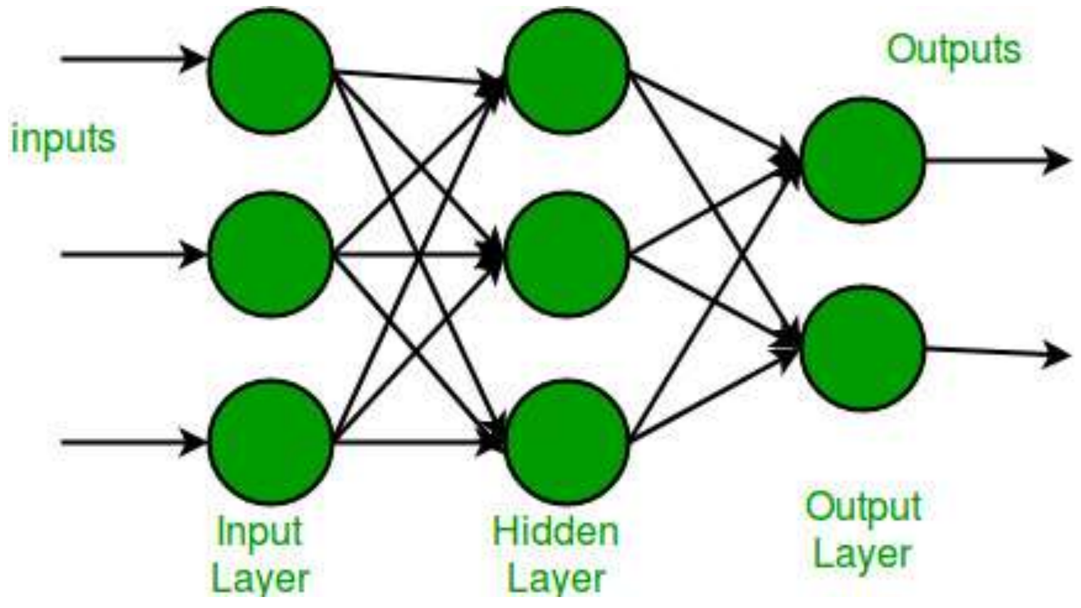
```
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# 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

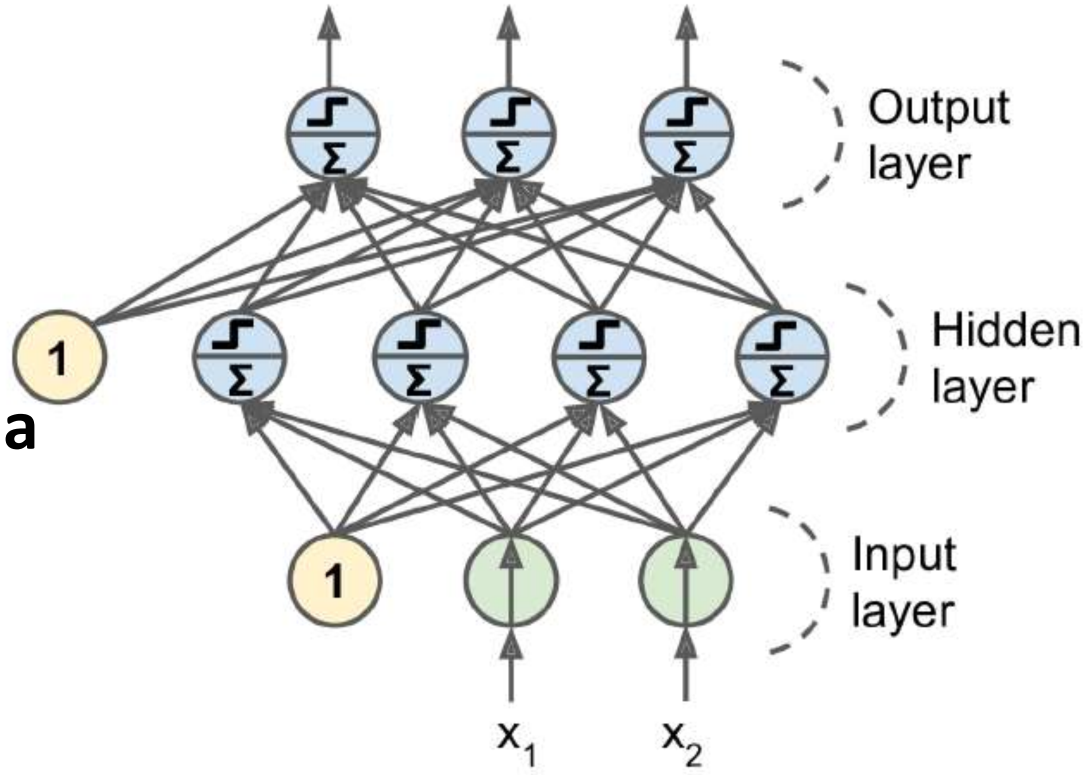
Multi-Layer Perceptron (MLP)

- **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

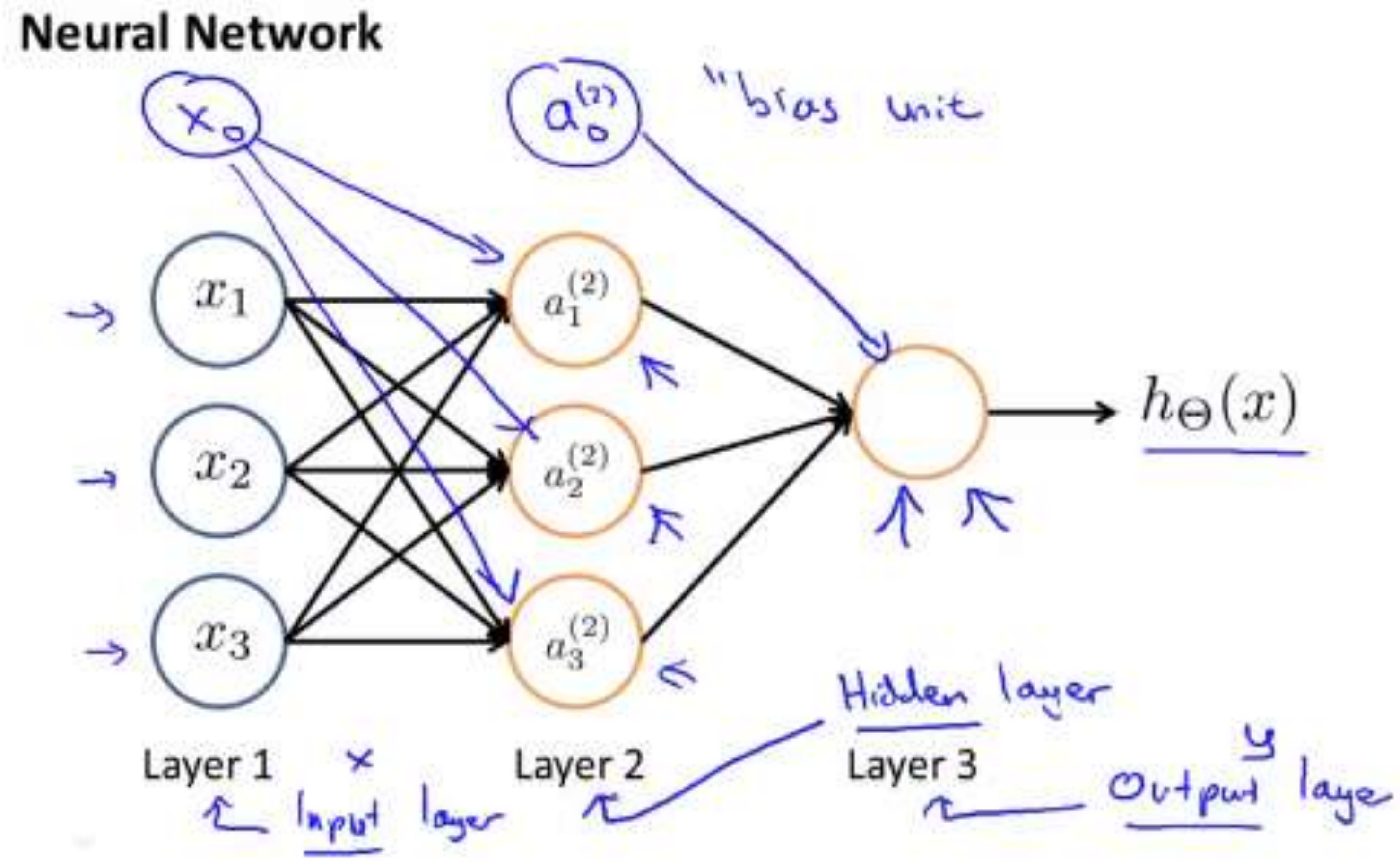


Artificial Neural Network (MLP)

- 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

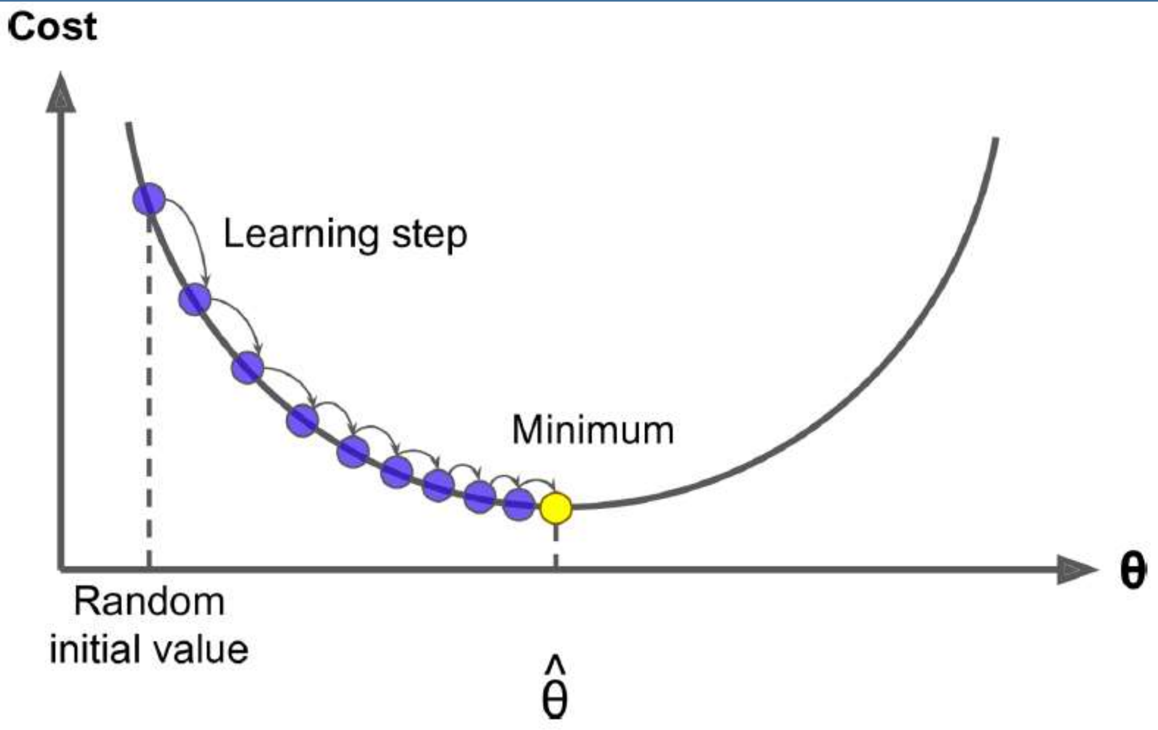
Backward propagation

Important concepts (Gradient Descent)

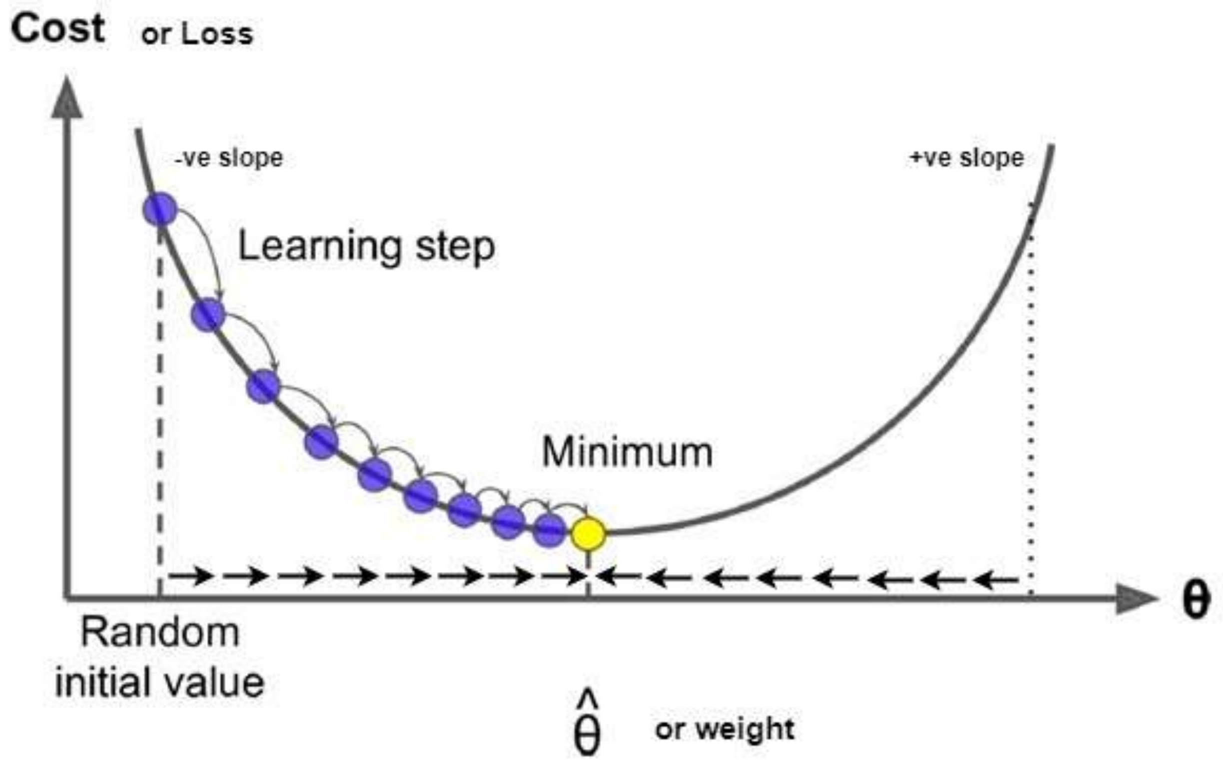
- Gradient Descent is a **very generic optimization algorithm**.
- The general idea of Gradient Descent is to tweak parameters iteratively to **minimize a cost function**.
- Suppose you are lost in the mountains in a dense fog; you **can only feel the slope** of the ground below your feet. A good strategy 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.

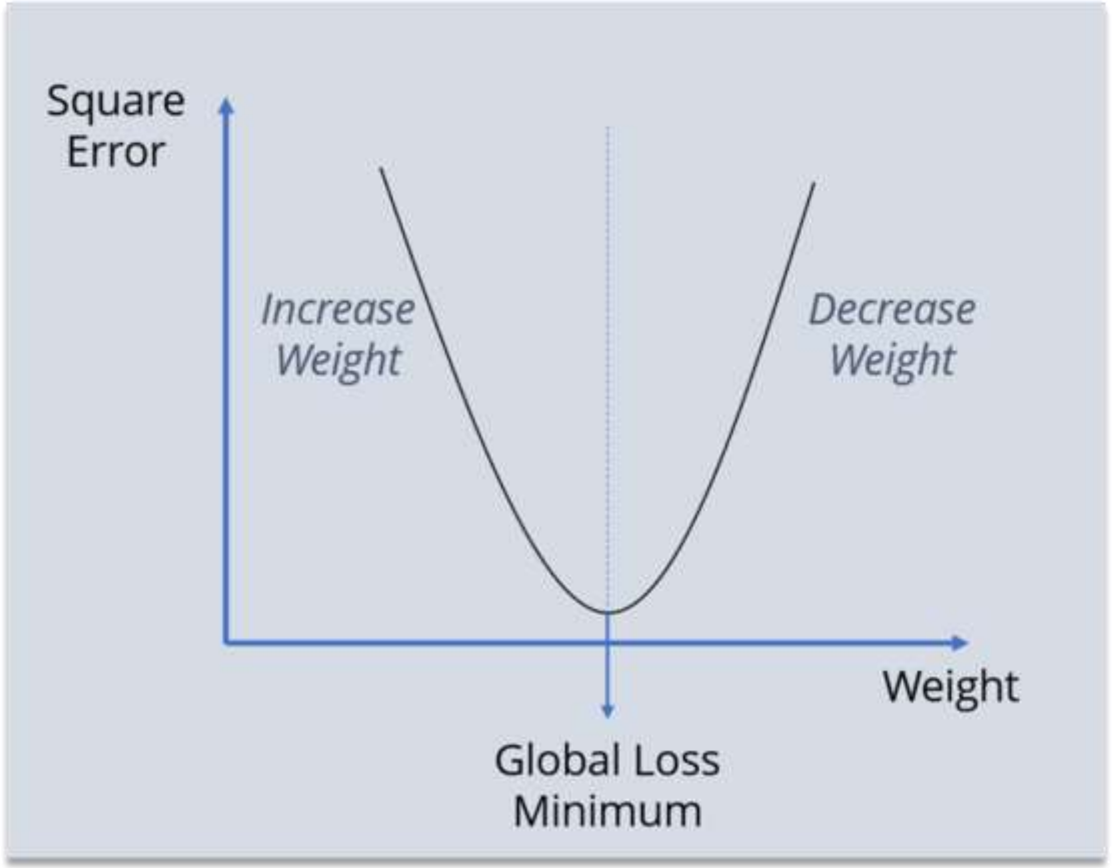
Important concepts (Gradient Descent)

- It measures **local gradient** of **error function** with regards to **parameter vector θ (weight)**, and it goes in direction of **descending gradient**.
- Once **gradient is zero**, you have **reached a minimum!**
- You **start by filling θ 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

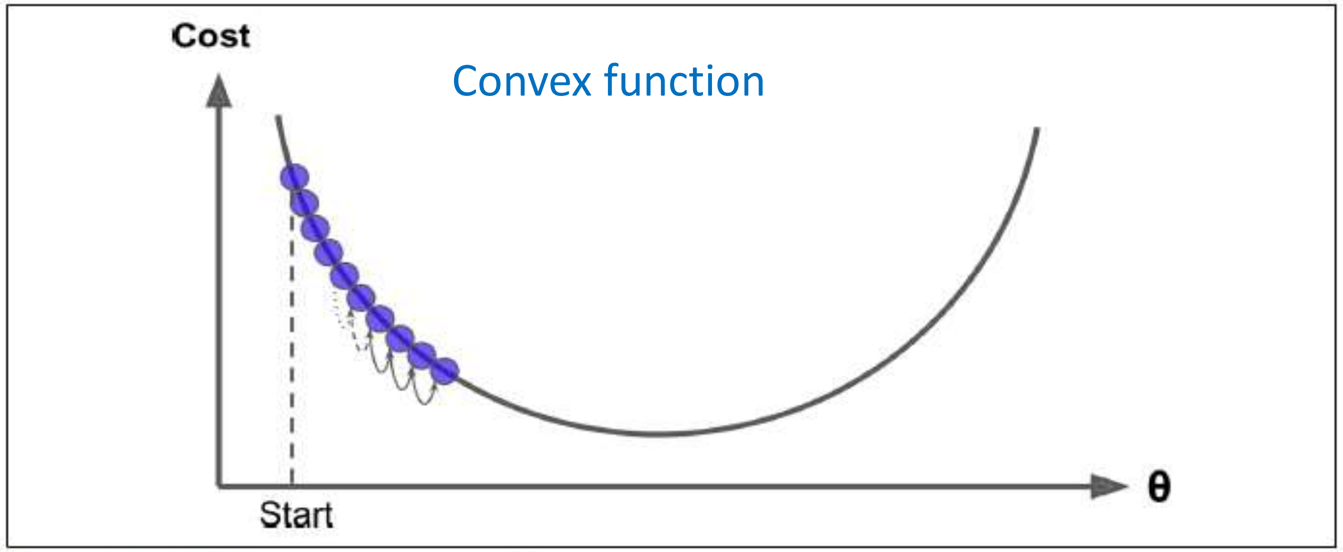


Figure 4-4. Learning rate too small

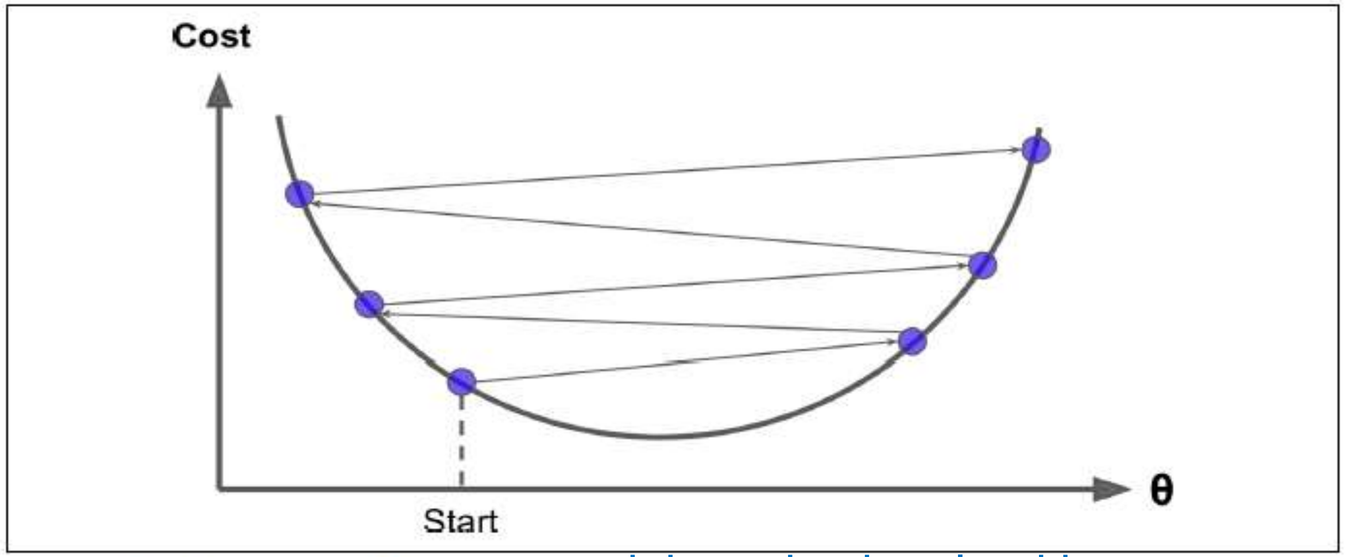


Figure 4-5. Learning rate too large

might make the algorithm diverge

COMPUTER SCIENCE

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

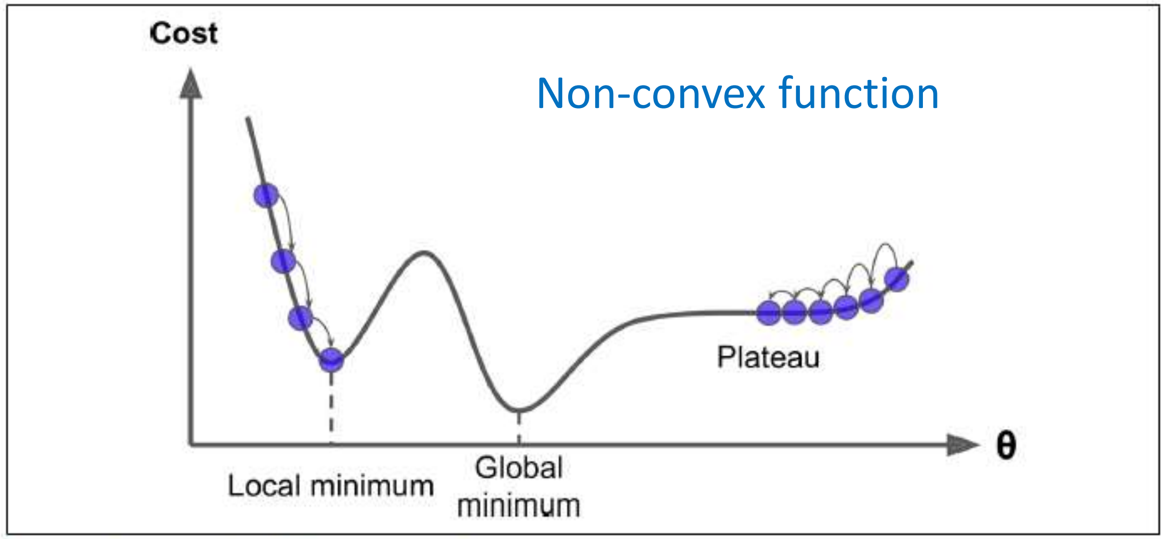
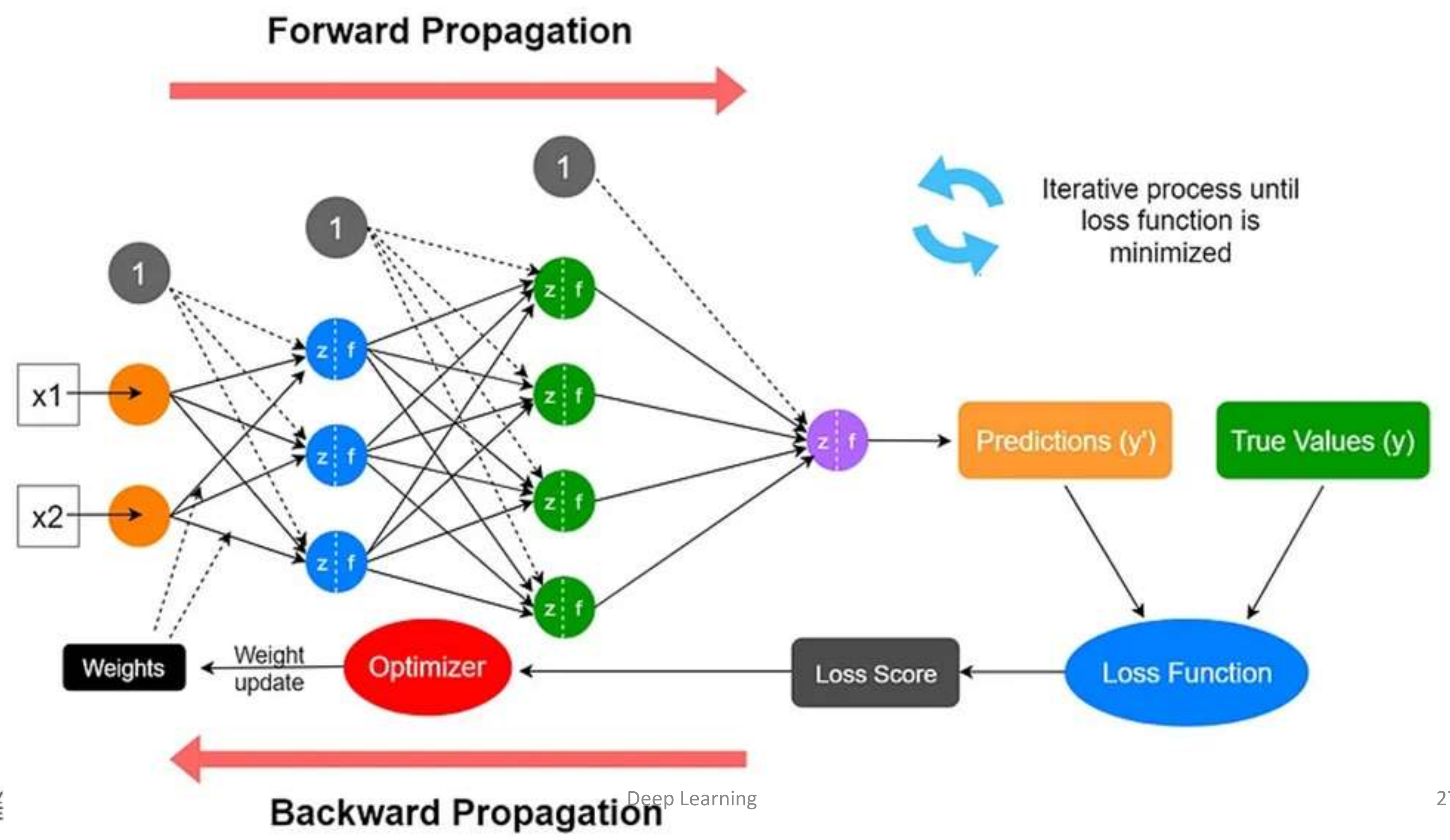


Figure 4-6. Gradient Descent pitfalls

ANN Learning Process



Back propagation and weights updating

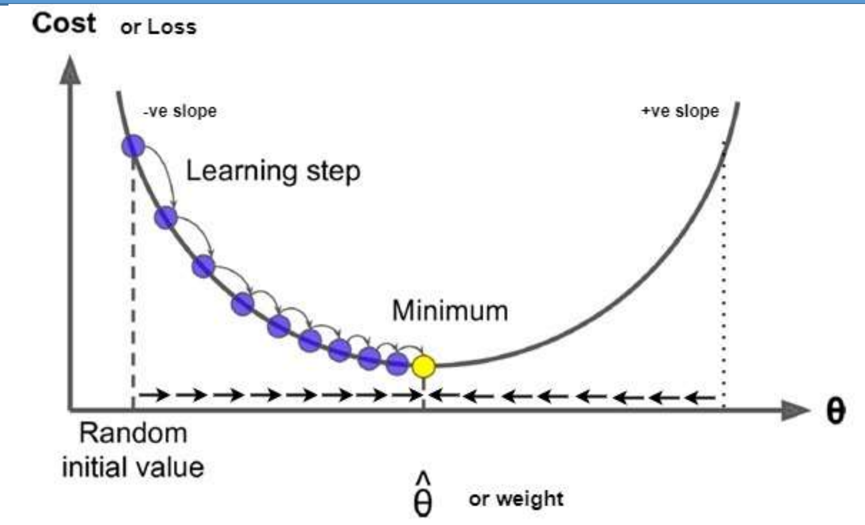
- Weight update formula

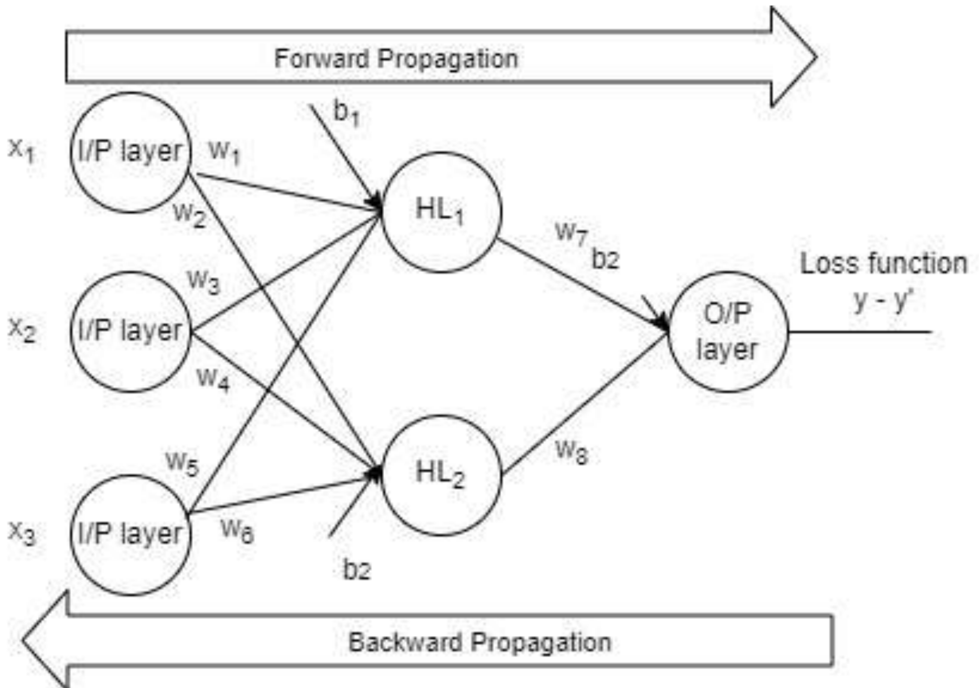
$$W_{new} = W_{old} - \eta \frac{\partial L}{\partial W_{old}}$$

  - Here **η 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 η 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:

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

$$W_{new} >> W_{old}$$

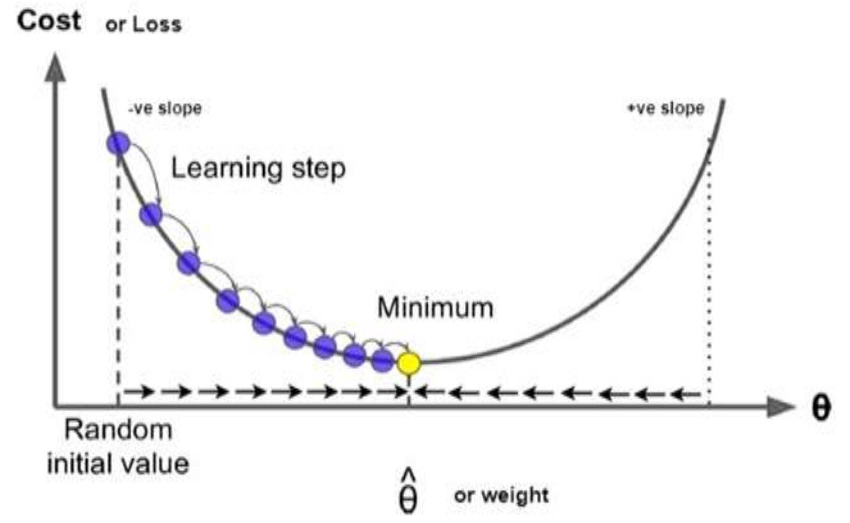
  - On-contrary gradient/slope from the **right side curve is +ve**:

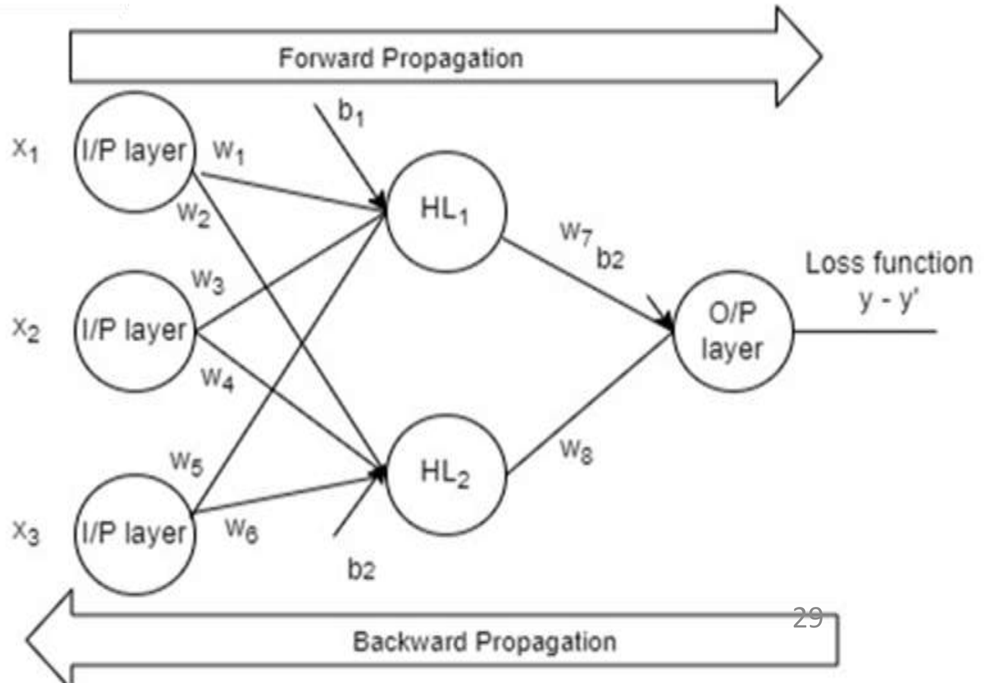
$$W_{new} = W_{old} - \eta \{+ve\ slope\}$$

$$W_{new} << W_{old}$$

  - 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. :

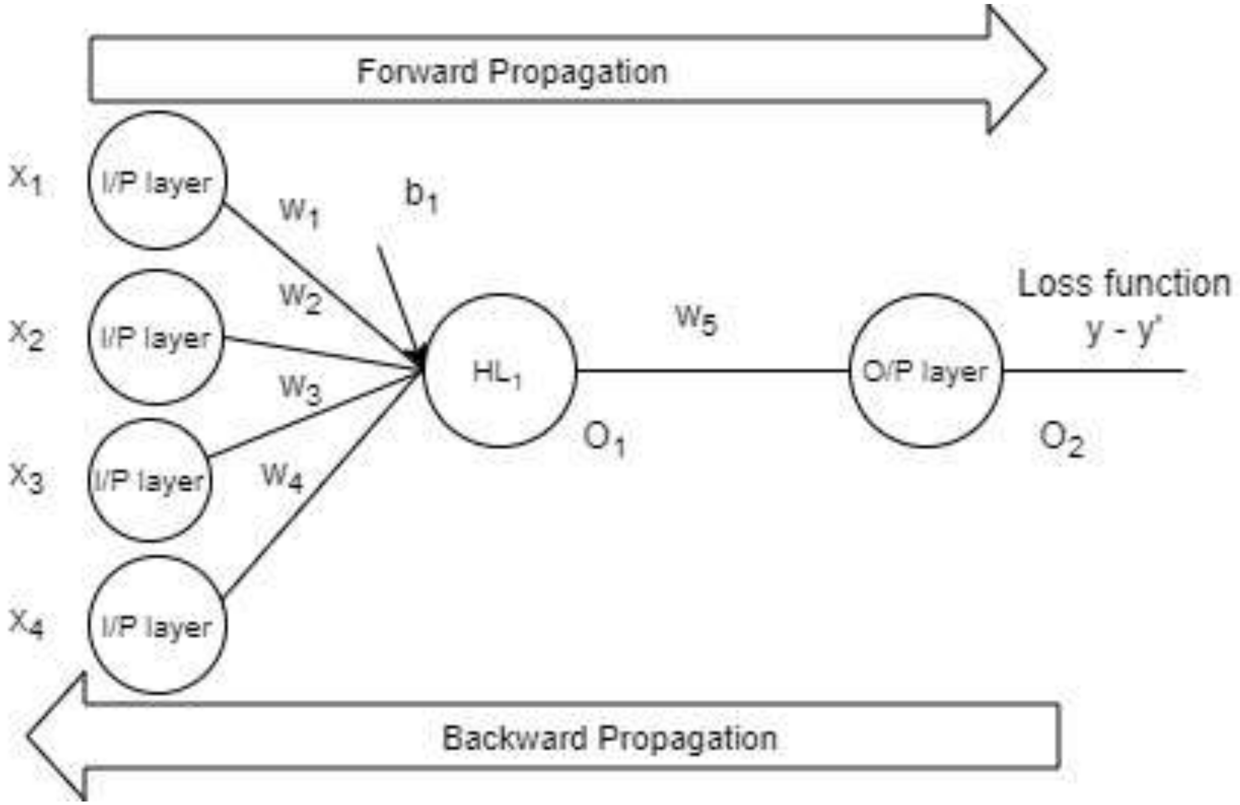
$$W_{new} = W_{old} - \eta \frac{\partial L}{\partial W_{old}}$$

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

- Partial derivative of **L** w.r.t to $\mathbf{W_{5old}}$ 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



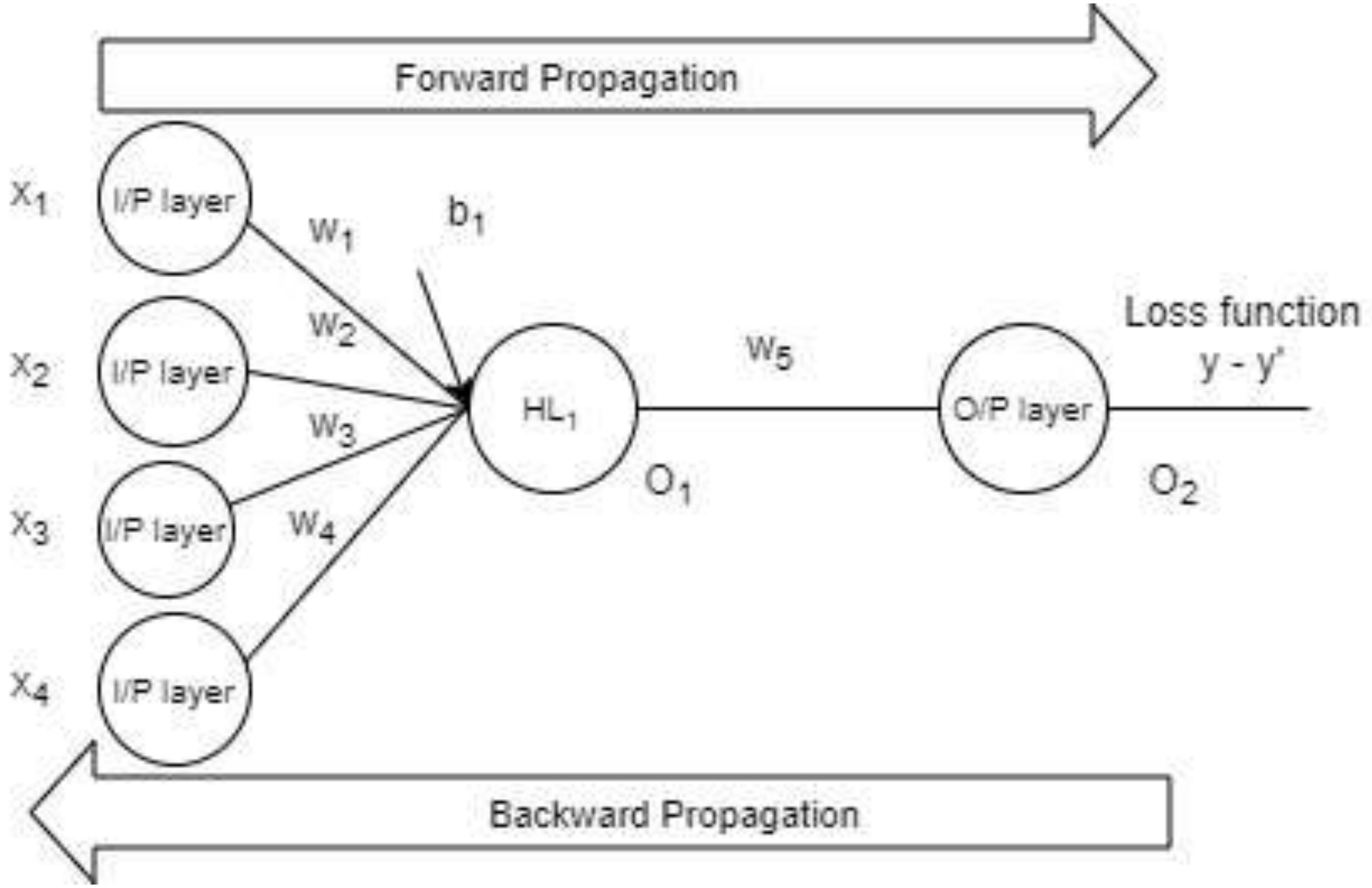
Chain rule of derivatives

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

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

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

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

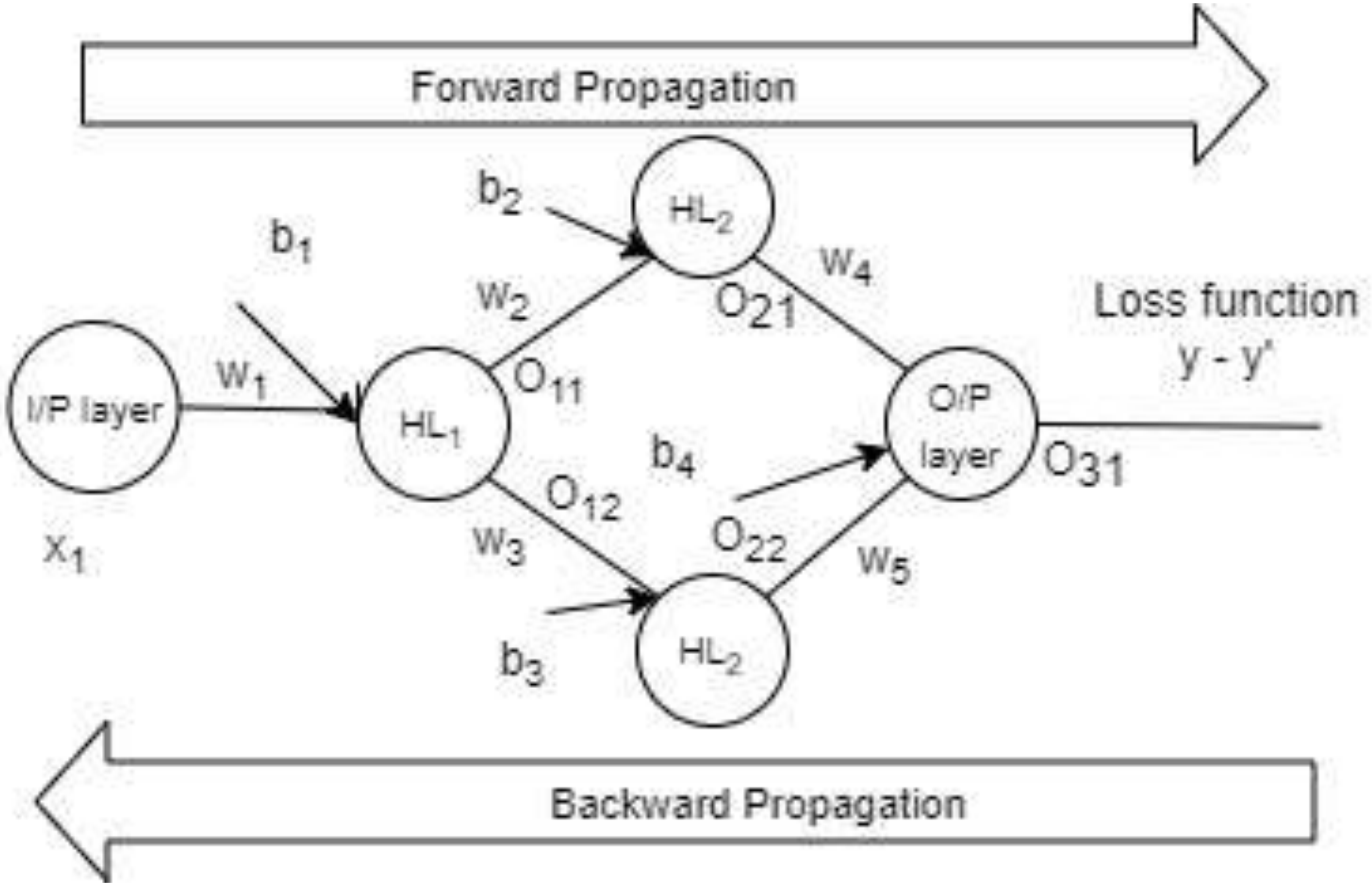


What about partial derivative of L w.r.t W_4?

Chain rule of derivatives

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

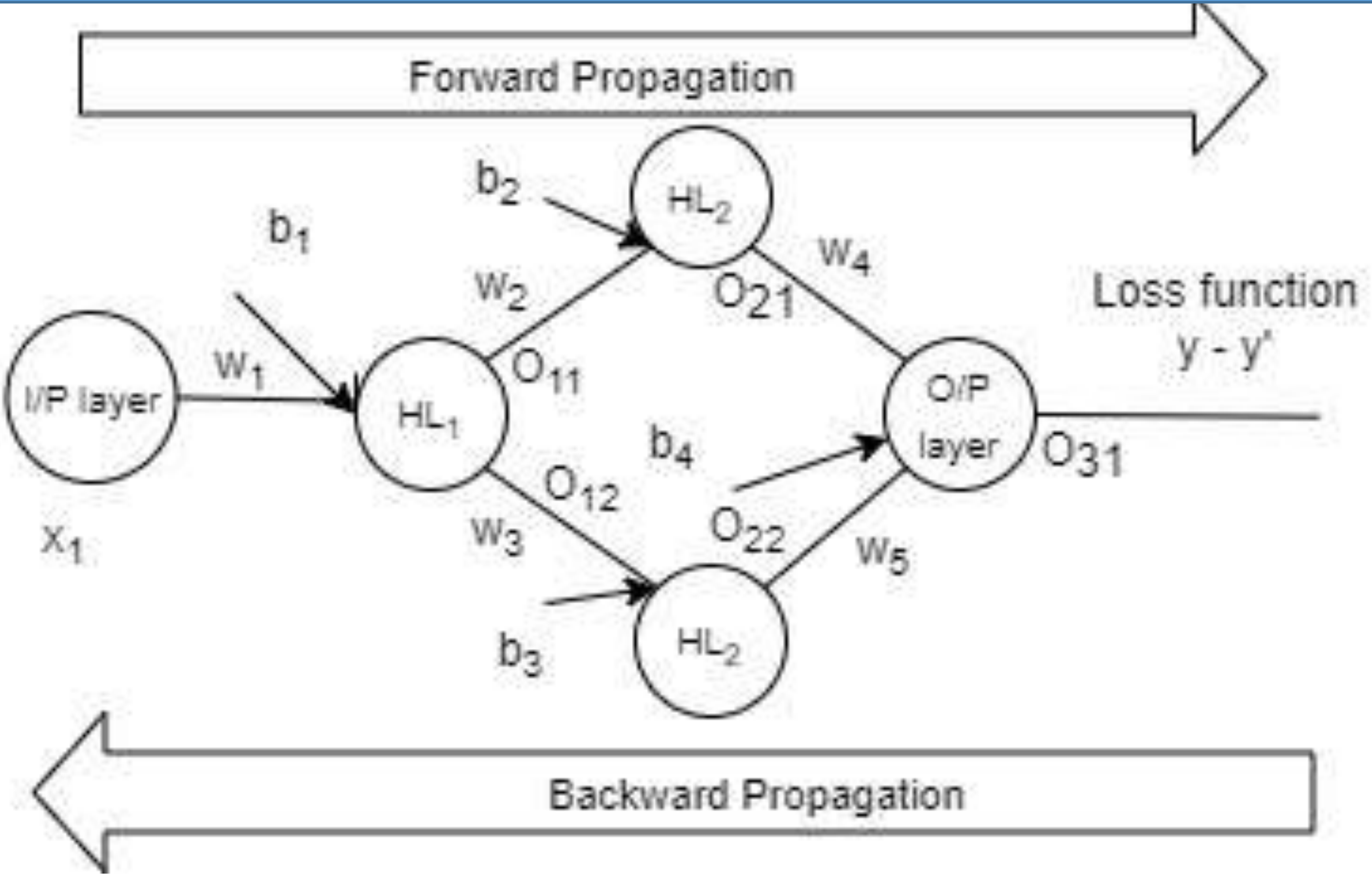
What about partial derivative of L w.r.t to W_1?



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:

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

Activation Functions and When to Use Them?

- When our brain is fed with a lot of information simultaneously, it tries hard to understand and classify the information into *"useful"* and *"not-so-useful"* information.
- We need a similar mechanism for classifying incoming information as *"useful"* or *"less-useful"* in case of Neural Networks.

- This is important in the way a network learns because not all the information is equally useful. Some of it is just noise.
  - This is where activation functions come into picture. The activation functions help the network use the important information and suppress the irrelevant data points.

- The brain receives the stimulus from the outside world, does the processing on the input, and then generates the output.
  - As the task gets complicated, multiple neurons form a complex network, passing information among themselves.

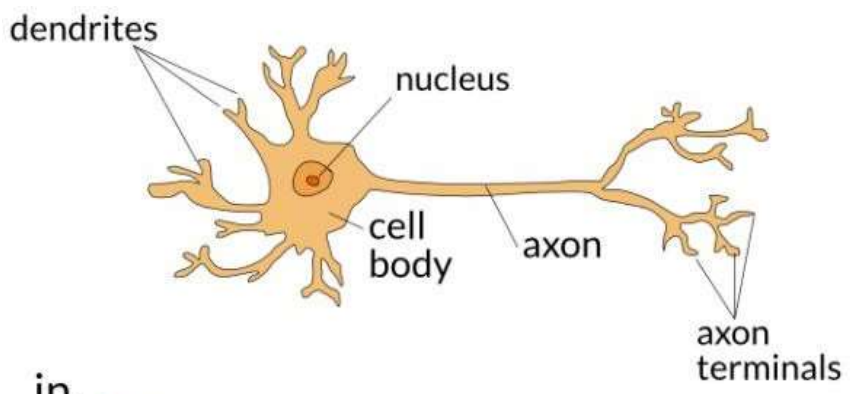
Overview of Activation Function in neural networks

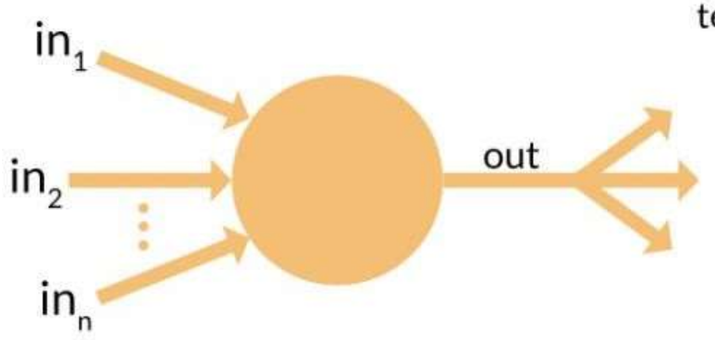
Network you see below is a neural network made of interconnected neurons. Each neuron is characterized by its weight, bias and activation function in deep learning.

$$Y = \text{Activation}(\Sigma(weight * input) + bias)$$

Finally, the output from the activation function moves to the next hidden layer and the same process is repeated. This forward movement of information is known as the ***forward propagation***.

What if the output generated is far away from the actual value? Using the output from the forward propagation, error is calculated. Based on this error value, the weights and biases of the neurons are updated. This process is known as ***back-propagation***.





Why do we need Non-linear activation function?

Imagine a neural network without the activation functions. In that case, every neuron will only be performing a linear transformation on the inputs using the weights and biases. Although linear transformations make the neural network simpler, but this network would be **less powerful** and will **not be able to learn the complex patterns** from the data.

Thus we use a non linear transformation to the inputs of the neuron and this non-linearity in the network is introduced by an activation function.

1. **Data Processing:** Neural networks process data layer by layer. Each layer performs a weighted sum of its inputs and adds a bias.

2. **Linear Limitation:** If all layers used linear activation functions (y = mx + b), stacking these layers would just create another linear function. No matter how many layers you add, the output would still be a straight line.

3. **Introducing Non-linearity:** Non-linear activation functions are introduced after the linear step in each layer. These functions transform the linear data into a non-linear form.

4. Learning Complex Patterns: Because of this non-linear transformation, the network can learn complex patterns in the data that wouldn't be possible with just linear functions. Imagine stacking multiple curved shapes instead of straight lines.

5. Beyond Linear Separation: This allows the network to move beyond simply separating data linearly, like logistic regression. It can learn intricate relationships between features in the data.

6. Foundation for Complex Tasks: By enabling the network to represent complex features, non-linear activation functions become the building blocks for neural networks to tackle tasks like image recognition, natural language processing.

Binary Step Function

- Threshold based classifier i.e. whether or not the neuron should be activated based on the value from the linear transformation.
  - if the input to the activation function is greater than a threshold, then the neuron is activated, else it is deactivated, i.e. its output is not considered for the next hidden layer.

- Can be used as an activation Function while creating a binary classifier. As you can imagine, this function will not be useful when there are multiple classes in the target variable. One of the limitations of binary step function.

- Gradient of the step function is zero which causes a hindrance in the back propagation process. That is, if you calculate the derivative of f(x) with respect to x, it comes out to be 0.
  - Thus, weights and biases don't update.

Selection of activation function for backpropagation

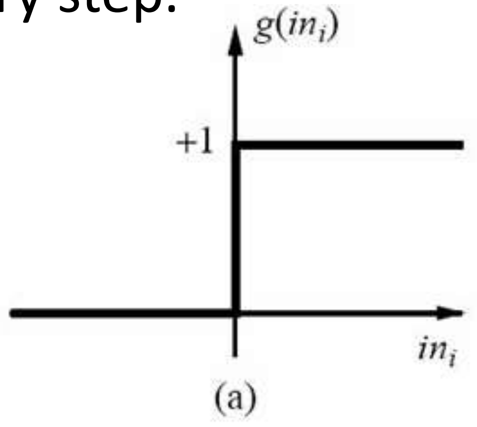
- For each training instance the back propagation algorithm first makes a **prediction (forward pass), measures the error**,
- then **goes through each layer in reverse** to measure the error contribution from each connection (reverse pass), and finally **slightly tweaks the connection weights to reduce 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**,

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

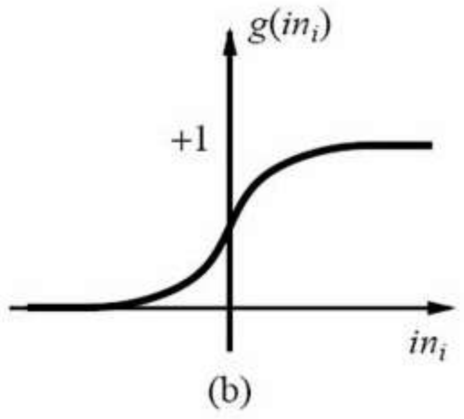
Selection of activation function for backpropagation

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

- 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



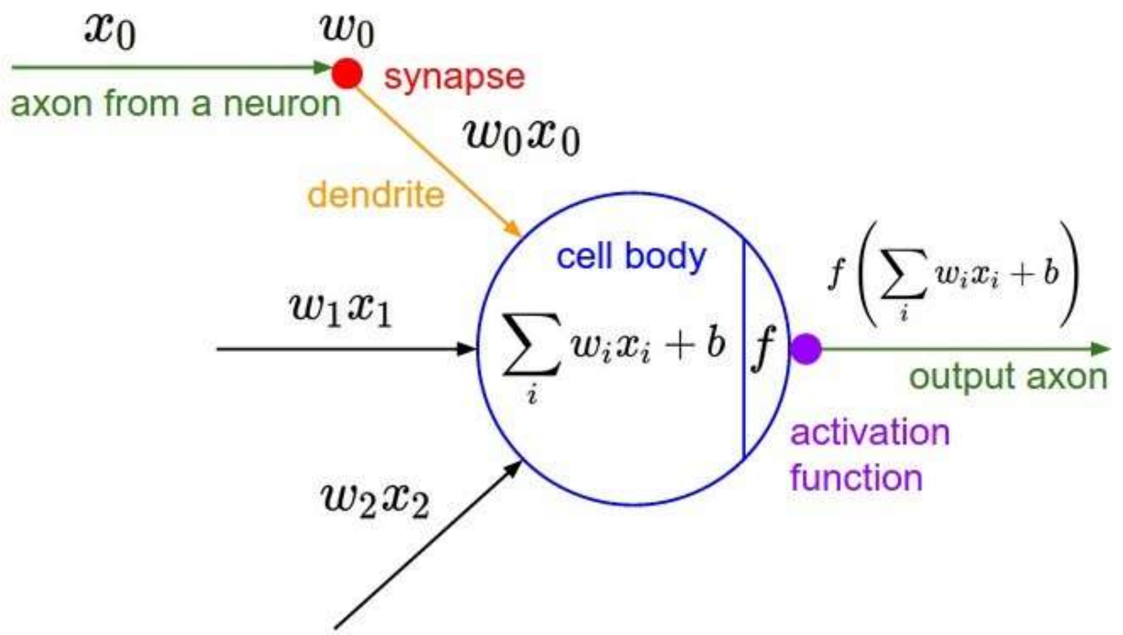
sigmoid function

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

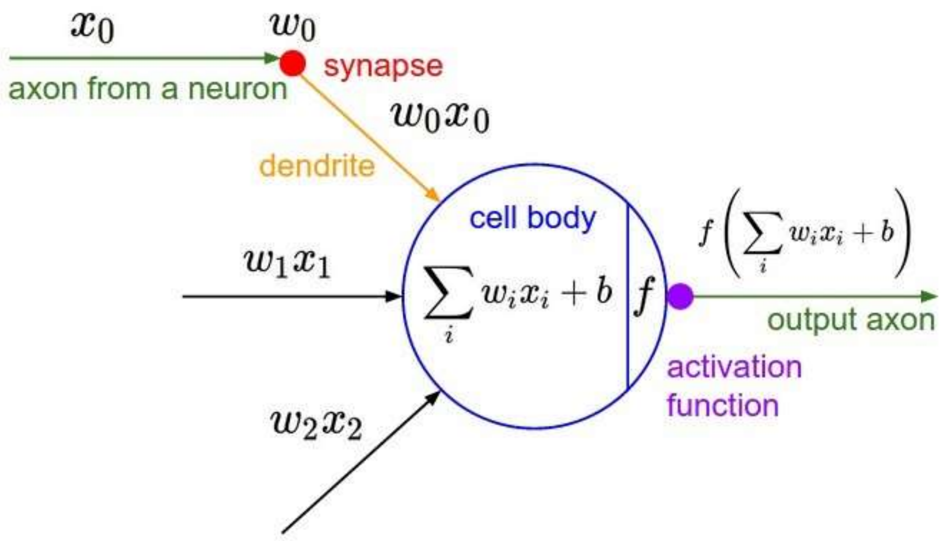
Activation functions

- 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.

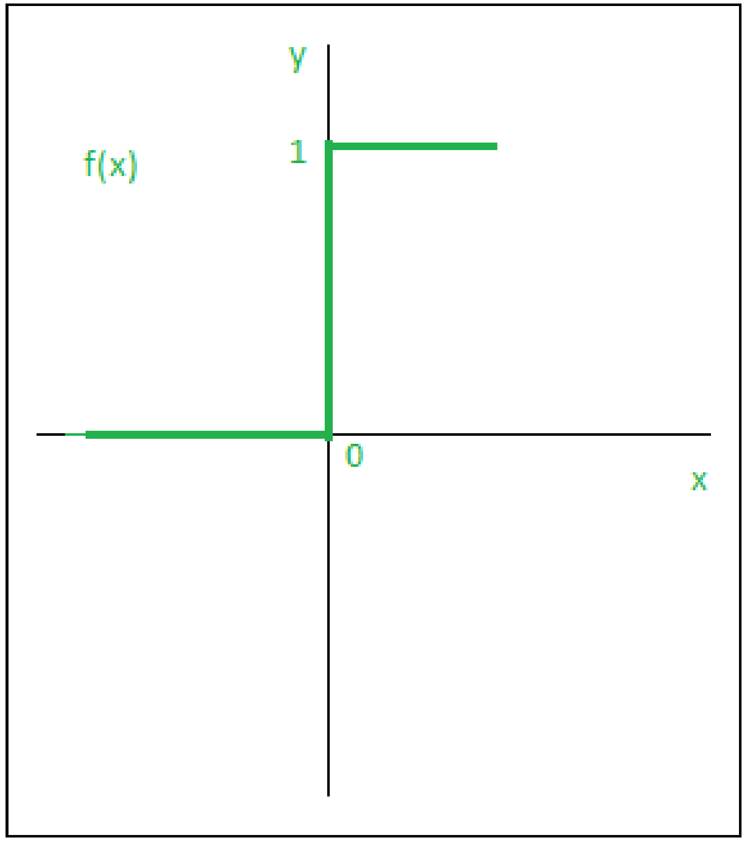


Activation functions (Linear)

- 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: $\boldsymbol{f(x) = x,\ \forall x}$
  - 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)

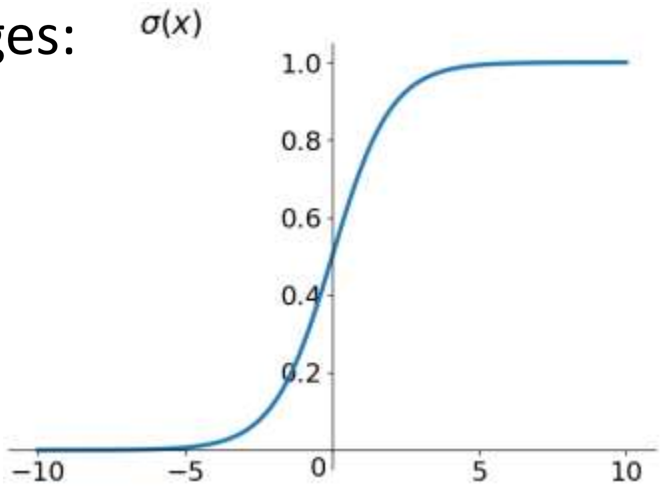
- **Step Function** is one of the simplest kind of activation functions.
- In this, we consider a threshold value and if the value of net input say y is greater than the threshold then the neuron is activated.
- $\boldsymbol{f(x) = 1, if\ x \geq 0}$
- $\boldsymbol{f(x) = 0, if\ x < 0}$

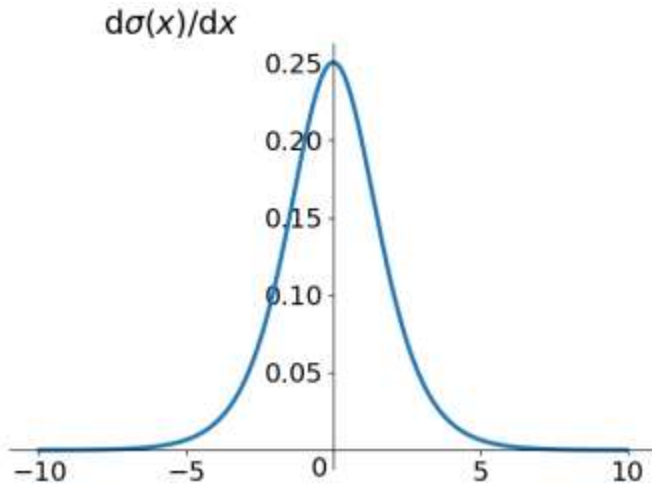


Activation functions (Sigmoid)

- In the sigmoid function, we can see that its output is in the **open interval (0,1)**.
- The function output **is not cantered 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**

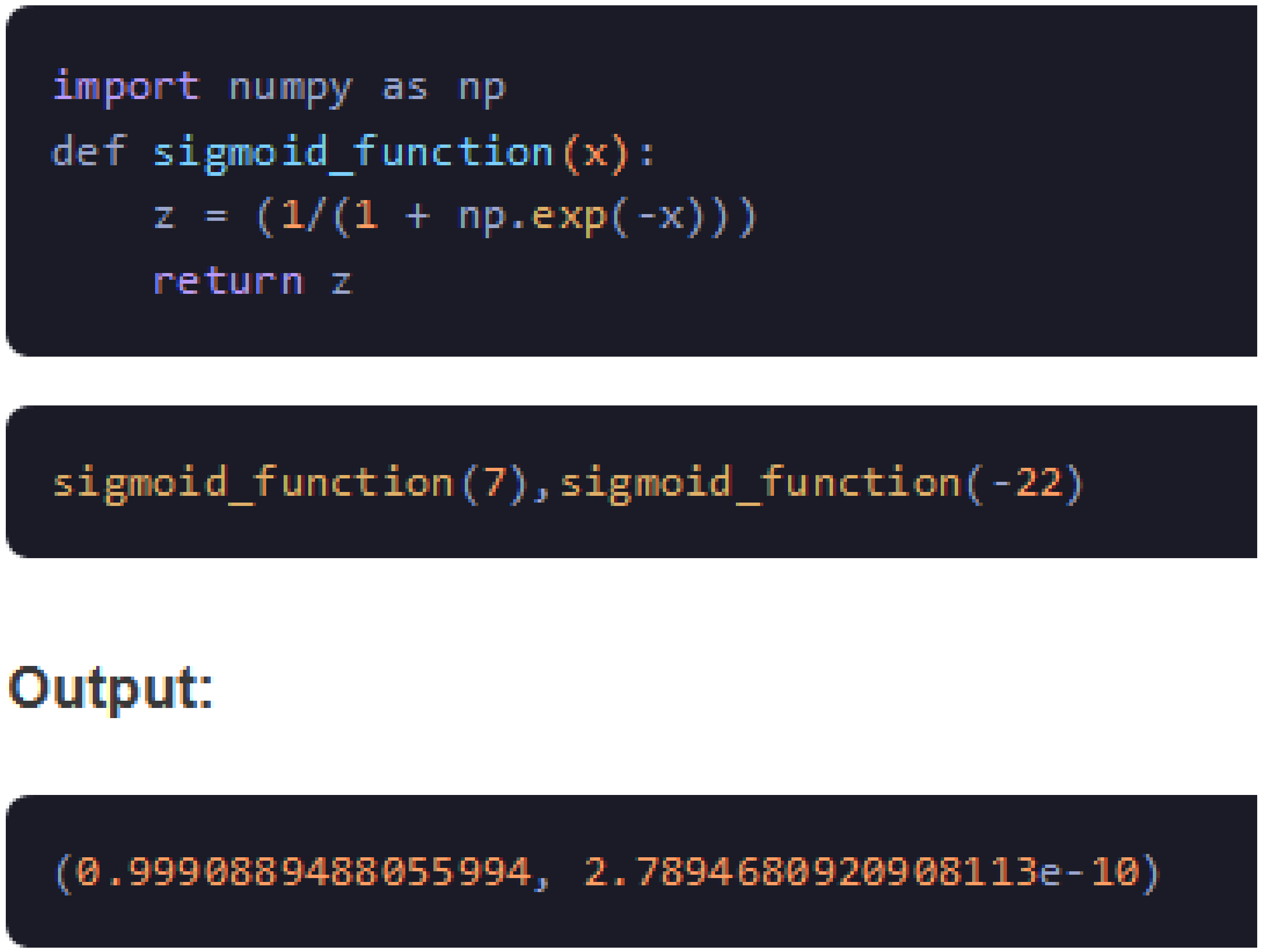
$$\sigma(x) = \frac{1}{1+e^{-x}}$$





Activation functions (Sigmoid)

- Unlike the binary step and linear functions, sigmoid is a **non-linear** function.
- When we have multiple neurons having sigmoid function as their activation function, the output is non linear as well.



```python
import numpy as np
def sigmoid_function(x):
    z = (1/(1 + np.exp(-x)))
    return z
```

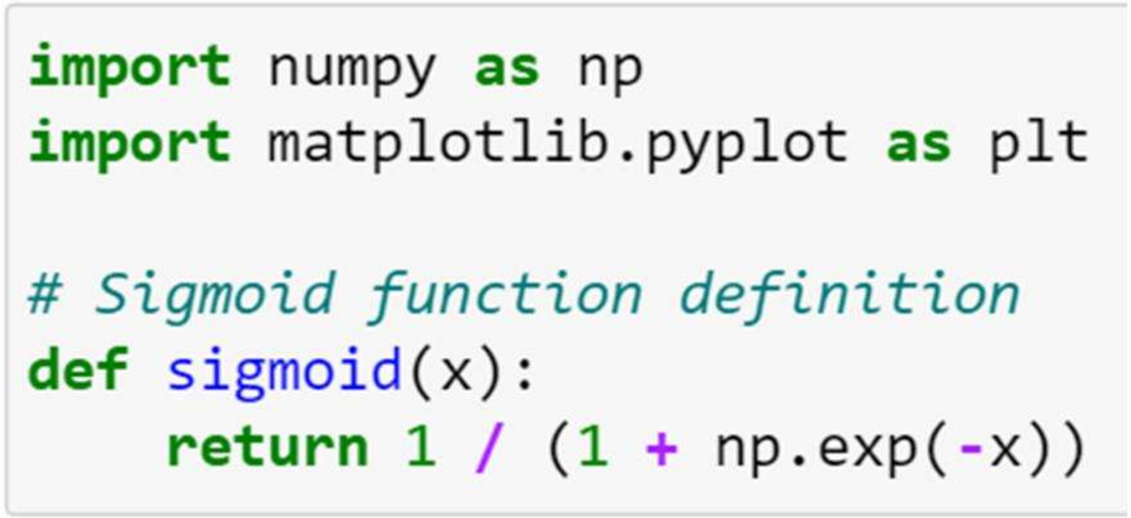
```python
sigmoid_function(7),sigmoid_function(-22)
```

Output:

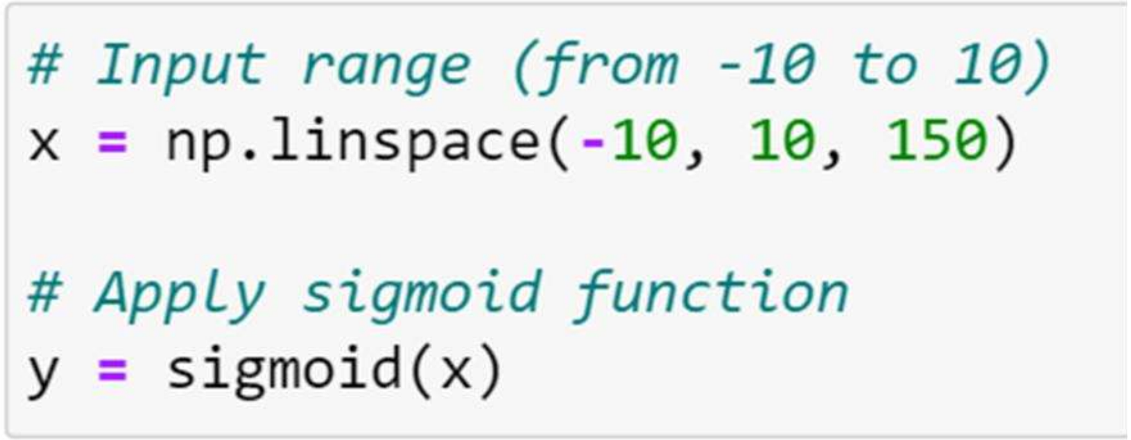
```
(0.9990889488055994, 2.7894680920908113e-10)
```

Activation functions (Sigmoid)

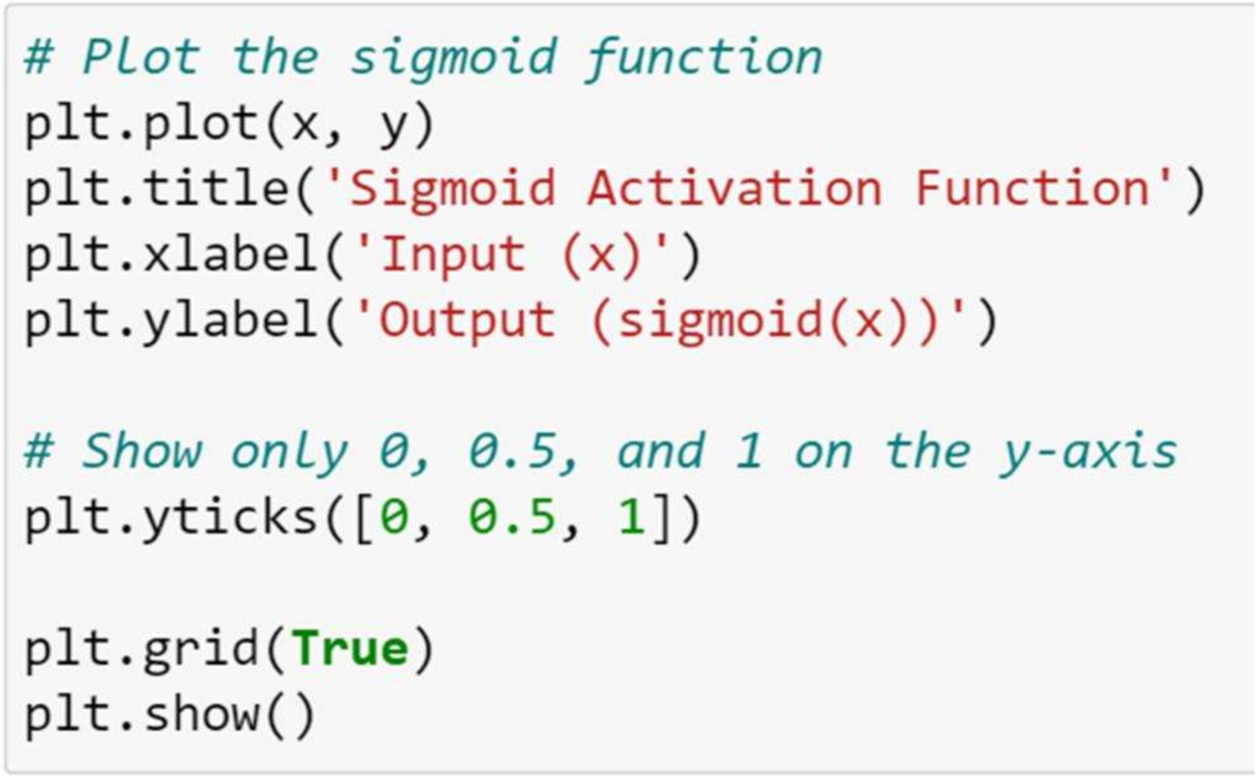
Additionally, as you can see in the graph above, this is a smooth S-shaped function and is continuously differentiable. The derivative of this function comes out to be (sigmoid(x)*(1-sigmoid(x)). Let’s look at the plot of it’s gradient.



```python
import numpy as np
import matplotlib.pyplot as plt

# Sigmoid function definition
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```



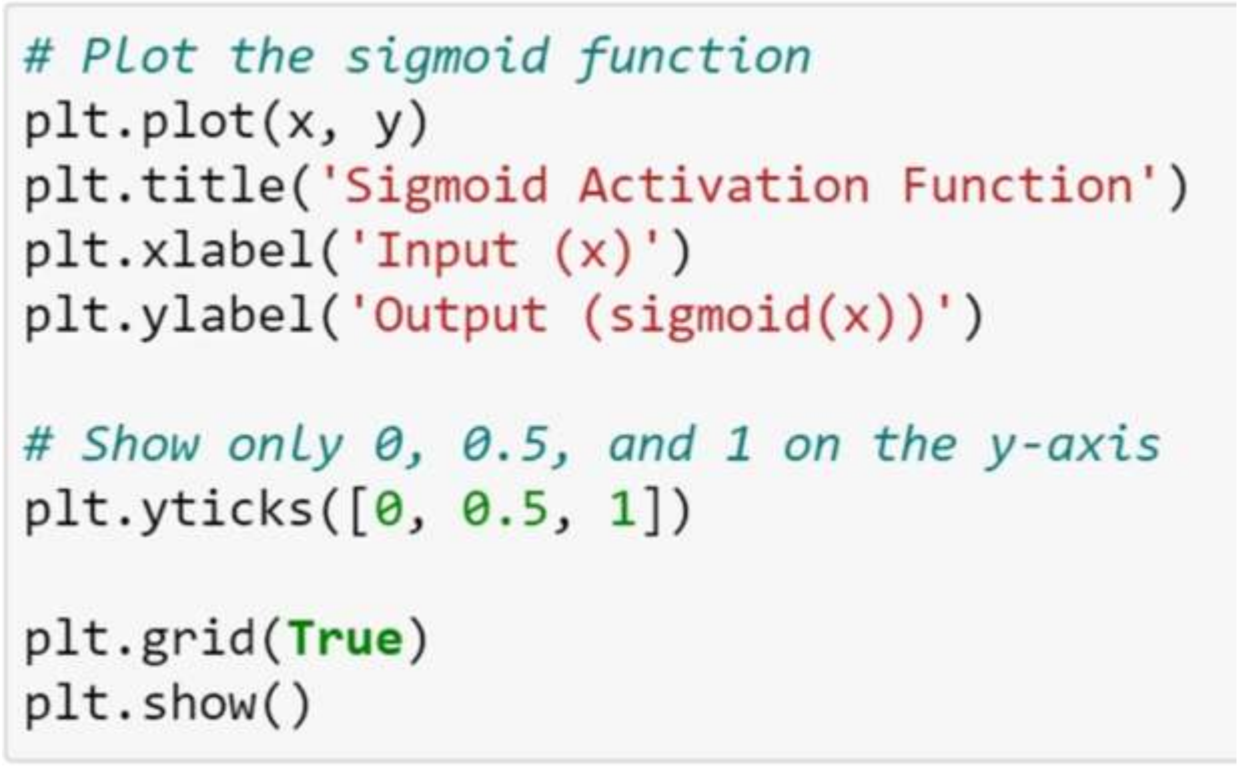
```python
# Input range (from -10 to 10)
x = np.linspace(-10, 10, 150)

# Apply sigmoid function
y = sigmoid(x)
```



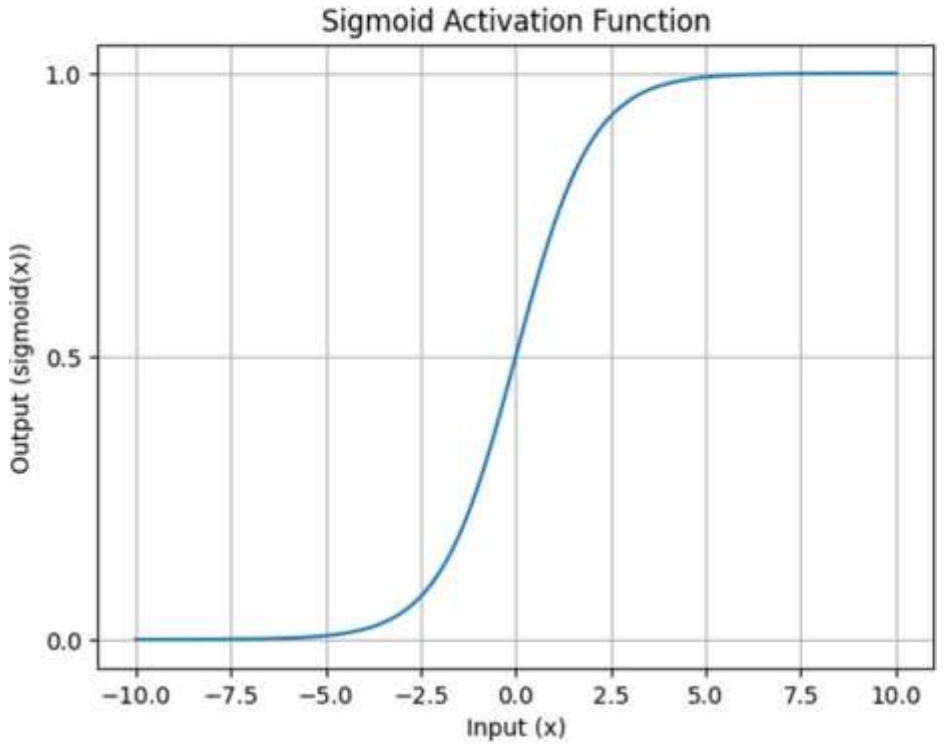
```python
# Plot the sigmoid function
plt.plot(x, y)
plt.title('Sigmoid Activation Function')
plt.xlabel('Input (x)')
plt.ylabel('Output (sigmoid(x))')

# Show only 0, 0.5, and 1 on the y-axis
plt.yticks([0, 0.5, 1])

plt.grid(True)
plt.show()
```

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

Activation functions (Sigmoid)



```
# Plot the sigmoid function
plt.plot(x, y)
plt.title('Sigmoid Activation Function')
plt.xlabel('Input (x)')
plt.ylabel('Output (sigmoid(x))')

# Show only 0, 0.5, and 1 on the y-axis
plt.yticks([0, 0.5, 1])

plt.grid(True)
plt.show()
```



Sigmoid Function | Vanishing gradient problem

- The vanishing gradient problem can occur when training deep NNs, especially with many layers and activation functions like sigmoid and tanh
- In a deep NN, we adjust the weights during **back propagation**.
- This adjustment process uses something called **gradients**.
- When the network has many layers, the gradients can become **very small**.
- This makes it difficult for the earlier layers of the network (**close to input**) to learn properly.
- This problem happens when gradients get too small, making it hard for deep networks to learn effectively, especially in the early layers.

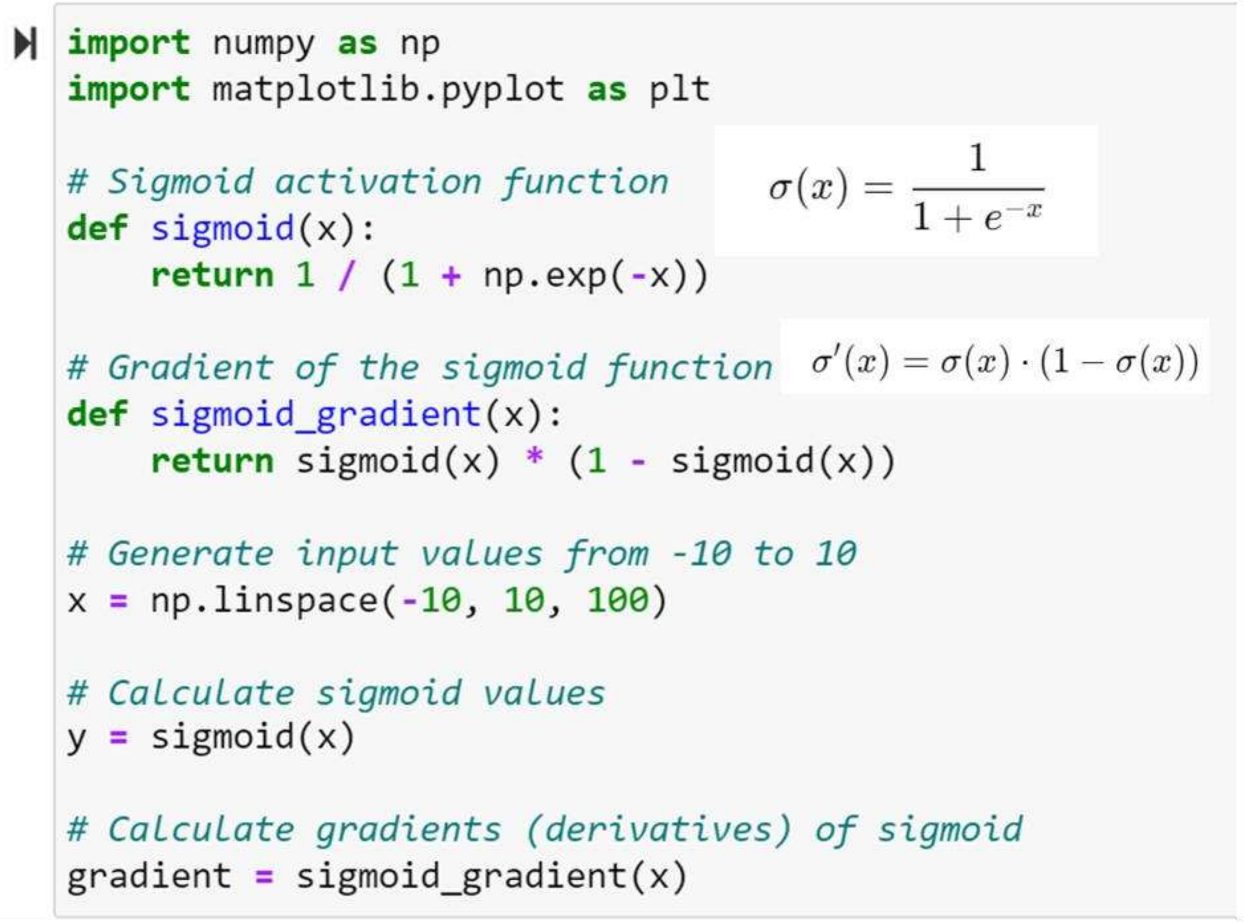
Sigmoid Function | Vanishing gradient problem

- Imagine you're trying to tell a friend something important through a long chain of people.
- You shout the message to the first person (the **final layer** in a neural network), but by the time the message reaches your friend (the early layers), it's so quiet (tiny gradient) that they can’t hear it clearly.
- The layers close to the input (beginning of the network) don't learn much because they don’t get clear feedback.
- If early layers don’t learn well, the network overall doesn't perform as well.
- Solution to this problem is to use the better activation functions like ReLU (Rectified Linear Unit).
    - It helps prevent gradients from becoming too small.

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**

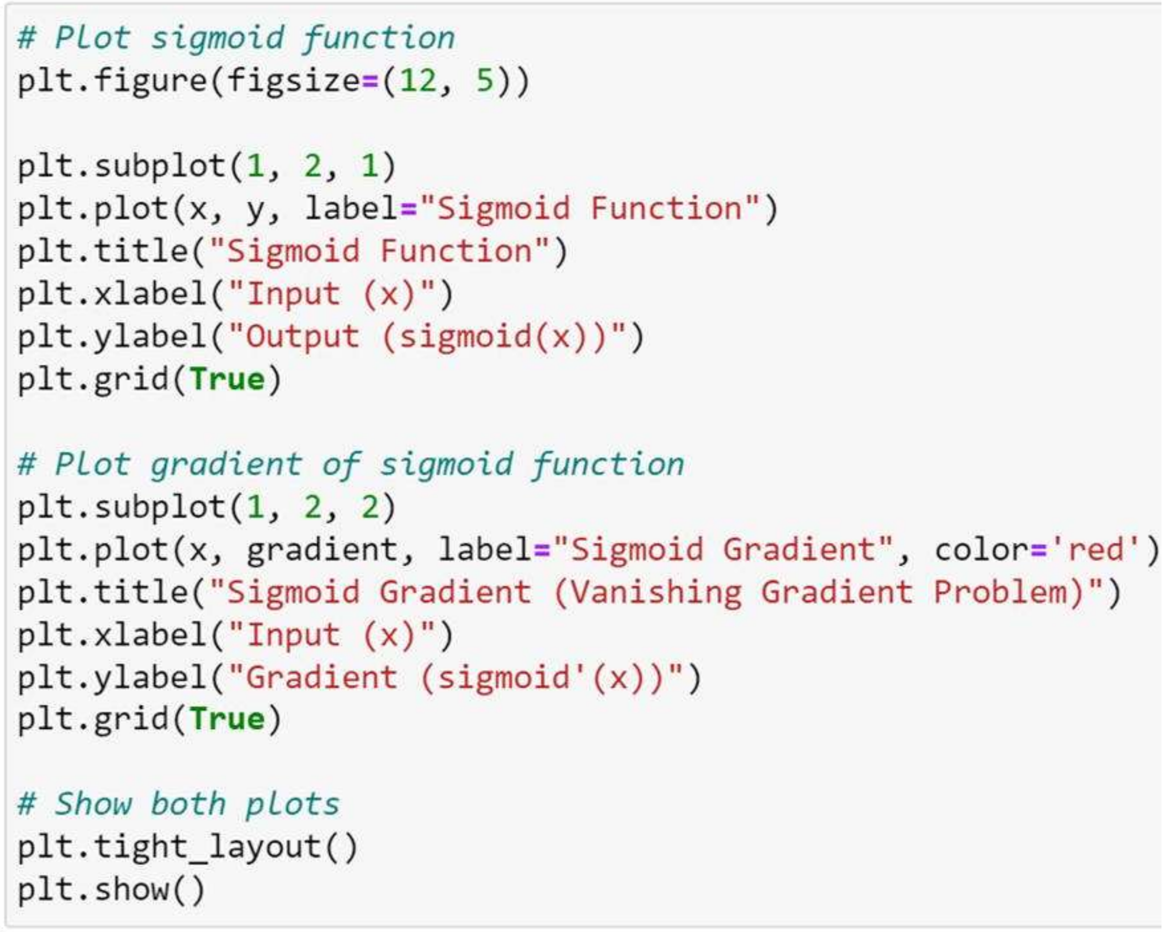
Demonstrating Vanishing Gradient Problem



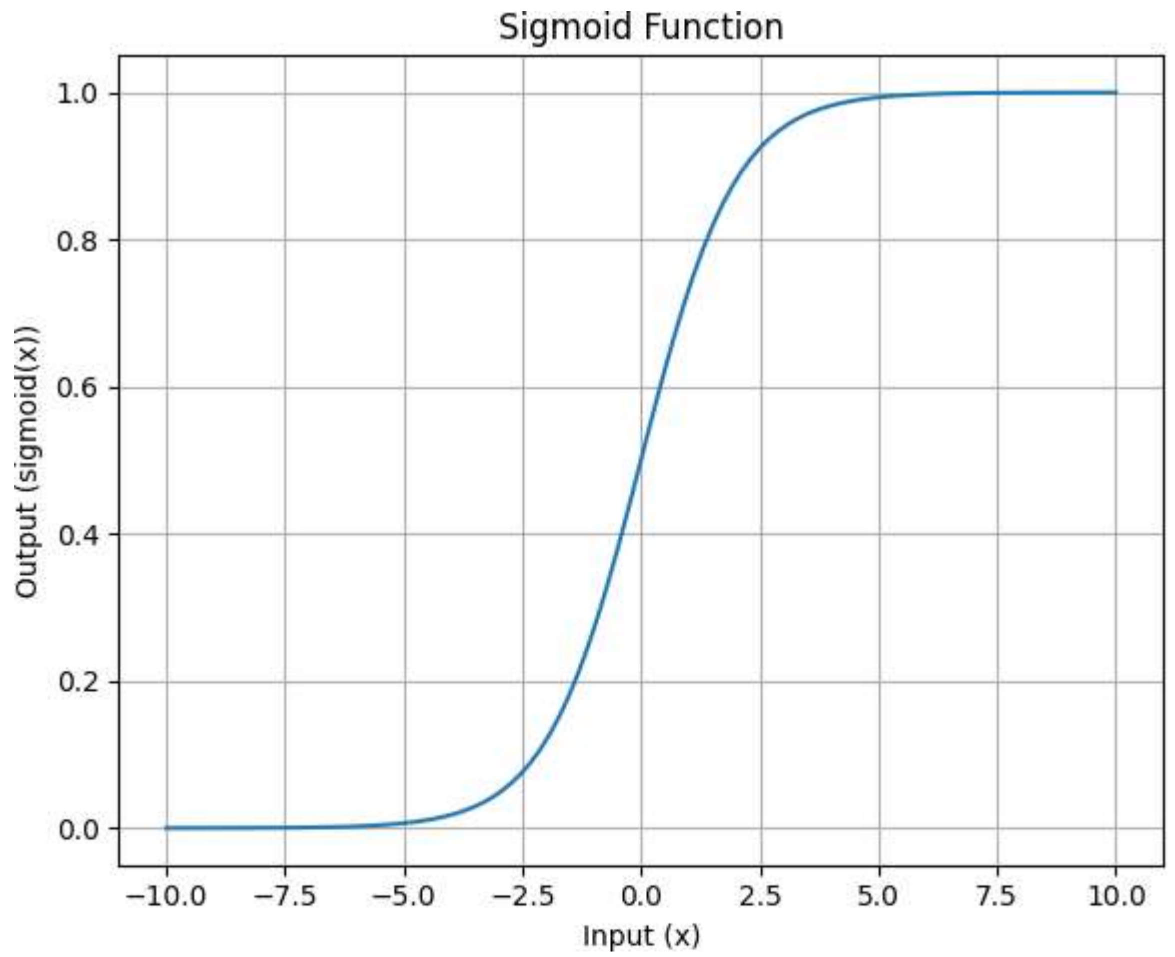
```python
import numpy as np
import matplotlib.pyplot as plt

# Sigmoid activation function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Gradient of the sigmoid function
def sigmoid_gradient(x):
    return sigmoid(x) * (1 - sigmoid(x))

# Generate input values from -10 to 10
x = np.linspace(-10, 10, 100)

# Calculate sigmoid values
y = sigmoid(x)

# Calculate gradients (derivatives) of sigmoid
gradient = sigmoid_gradient(x)
```

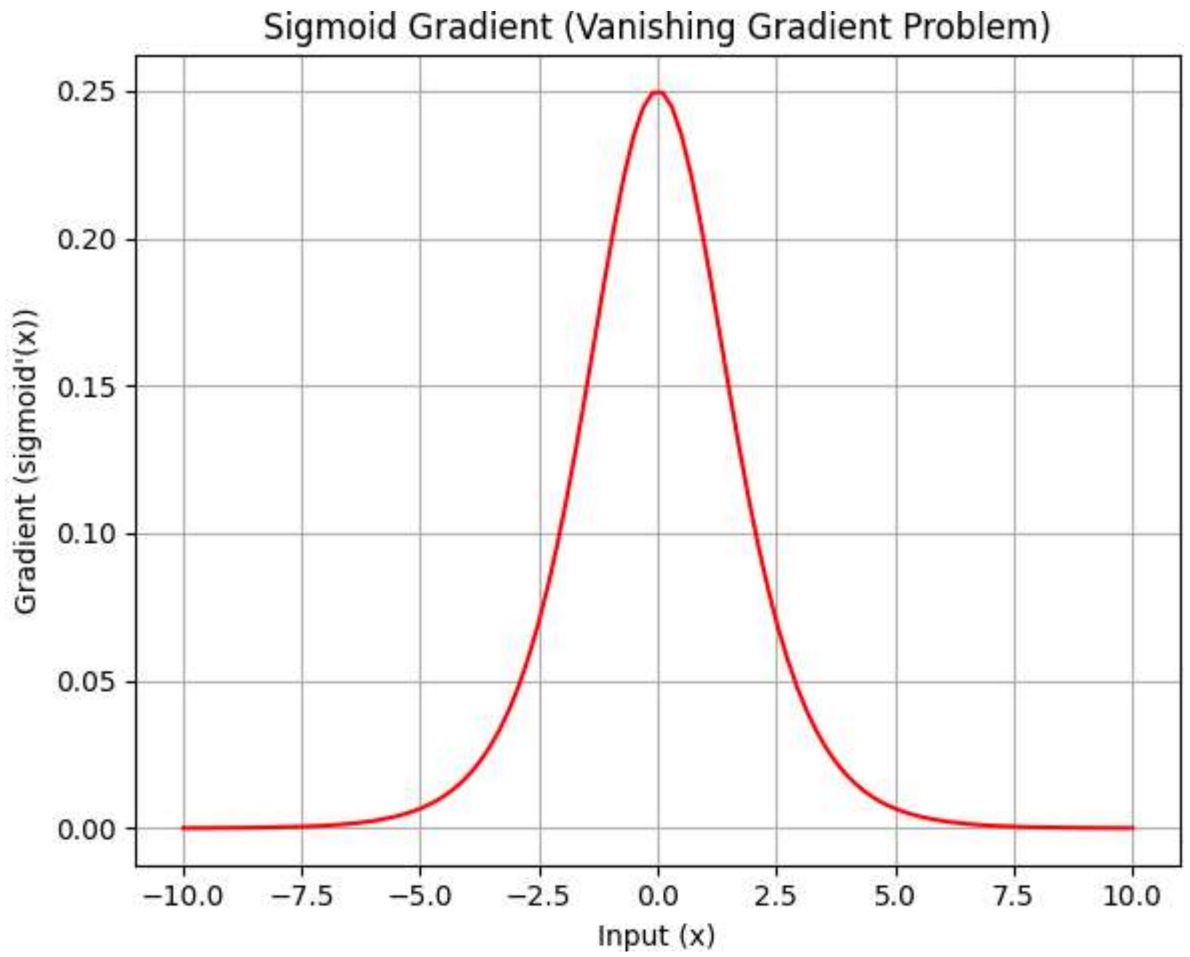
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$\sigma'(x) = \sigma(x) \cdot (1 - \sigma(x))$$



```python
# Plot sigmoid function
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(x, y, label="Sigmoid Function")
plt.title("Sigmoid Function")
plt.xlabel("Input (x)")
plt.ylabel("Output (sigmoid(x))")
plt.grid(True)

# Plot gradient of sigmoid function
plt.subplot(1, 2, 2)
plt.plot(x, gradient, label="Sigmoid Gradient", color='red')
plt.title("Sigmoid Gradient (Vanishing Gradient Problem)")
plt.xlabel("Input (x)")
plt.ylabel("Gradient (sigmoid'(x))")
plt.grid(True)

# Show both plots
plt.tight_layout()
plt.show()
```


Sigmoid Function
Output (sigmoid(x))
Input (x)
1.0
0.8
0.6
0.4
0.2
0.0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0


Sigmoid Gradient (Vanishing Gradient Problem)
Gradient (sigmoid'(x))
Input (x)
0.25
0.20
0.15
0.10
0.05
0.00
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

Vanishing gradient problem

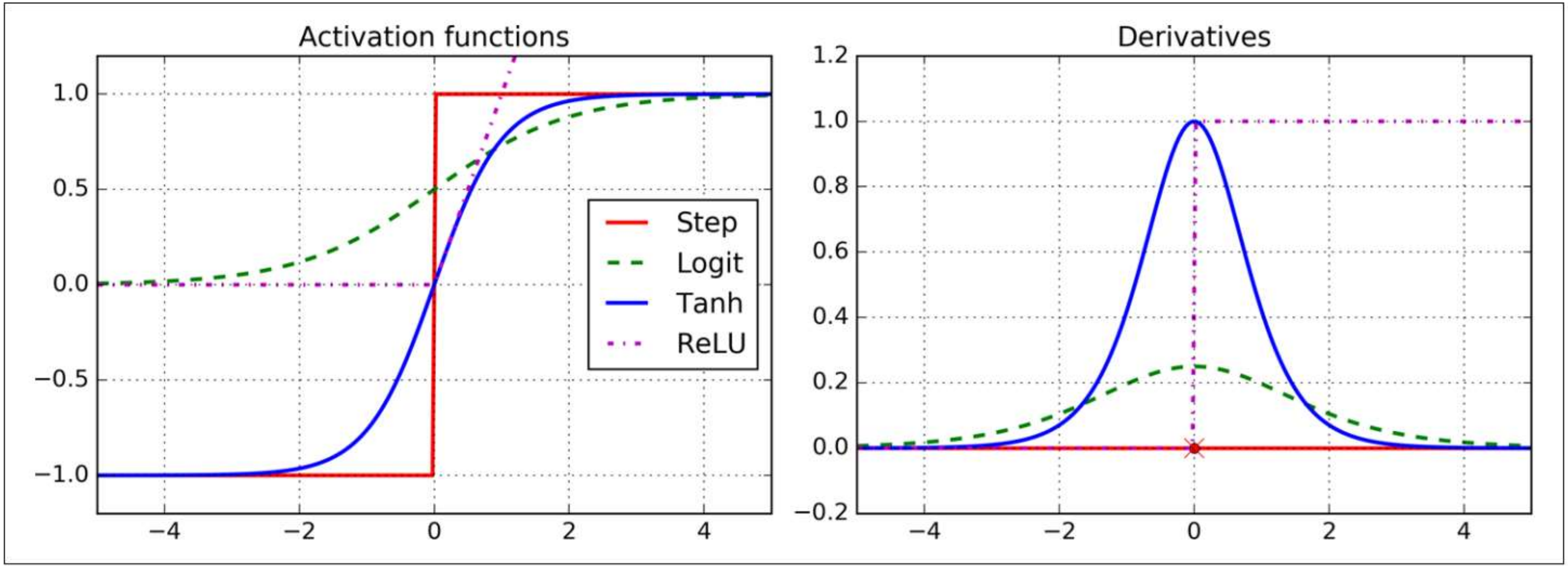


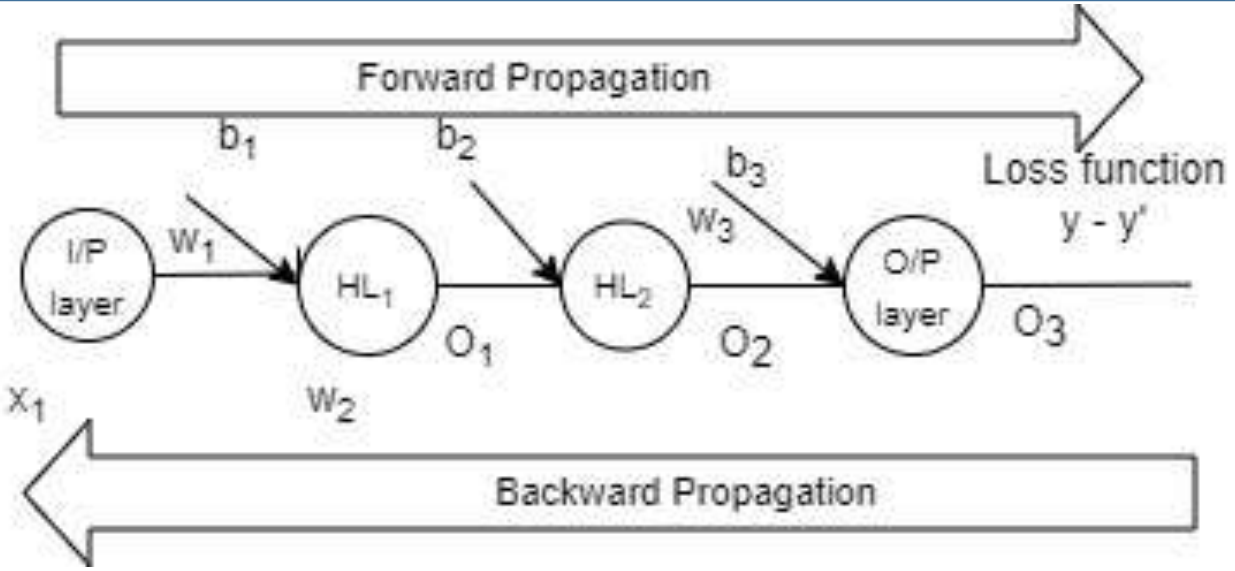
Figure 10-8. Activation functions and their derivatives

Vanishing gradient problem

- Here the **commonly used value of η 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 (η**).
- 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**.



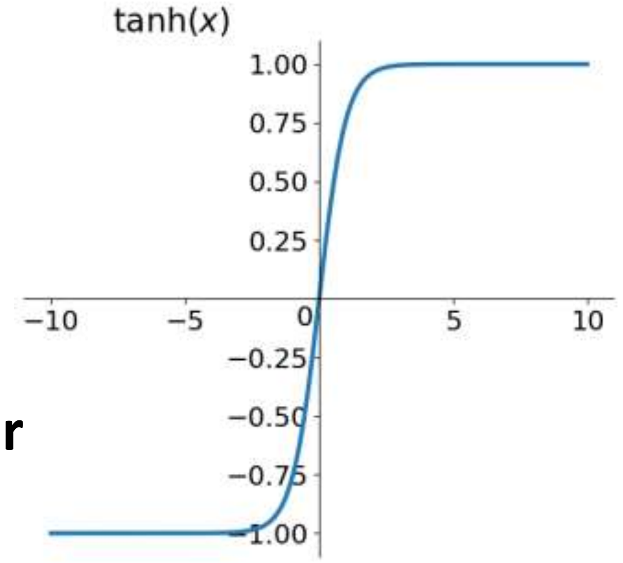
$$W_{1new} = W_{1old} - \eta \frac{\partial L}{\partial W_{1old}}$$

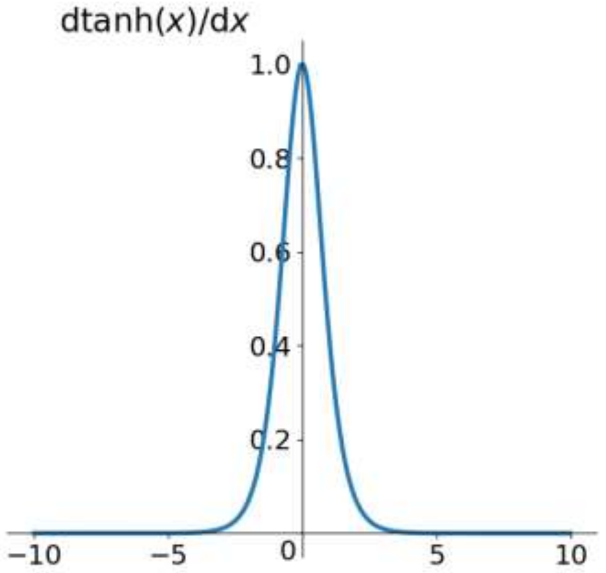
$$\frac{\partial L}{\partial W_{1old}} = \frac{\partial L}{\partial O_3} \times \frac{\partial O_3}{\partial O_2} \times \frac{\partial O_2}{\partial O_1} \times \frac{\partial O_1}{\partial W_{1old}}$$

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

Activation functions (tanh)

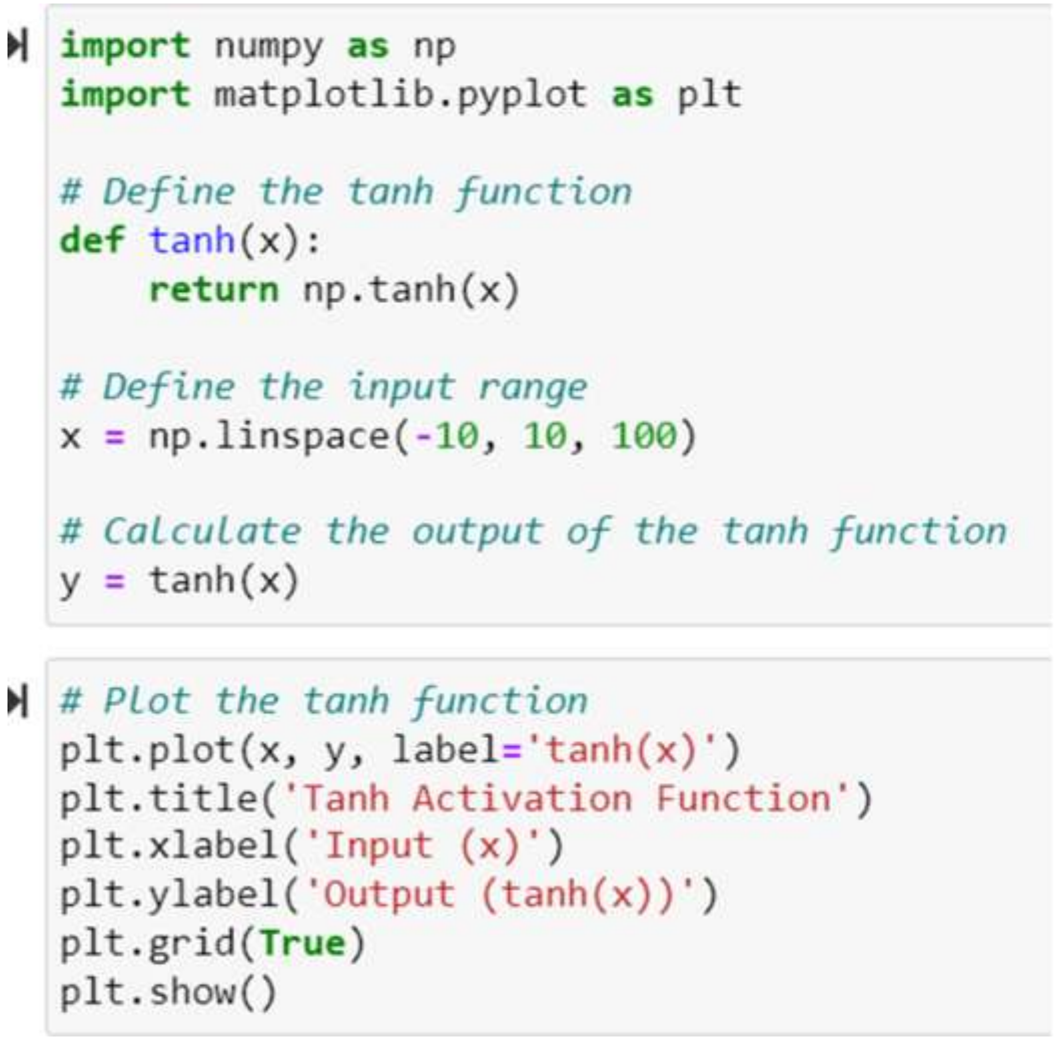
- The output interval **of tanh is (-1, 1),** and the whole function is **0-centric**, which is better than sigmoid.
- In general binary classification problems, the **tanh function is used for the hidden layer** and the **sigmoid function is used for the output layer**.
- This has **solved vanishing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





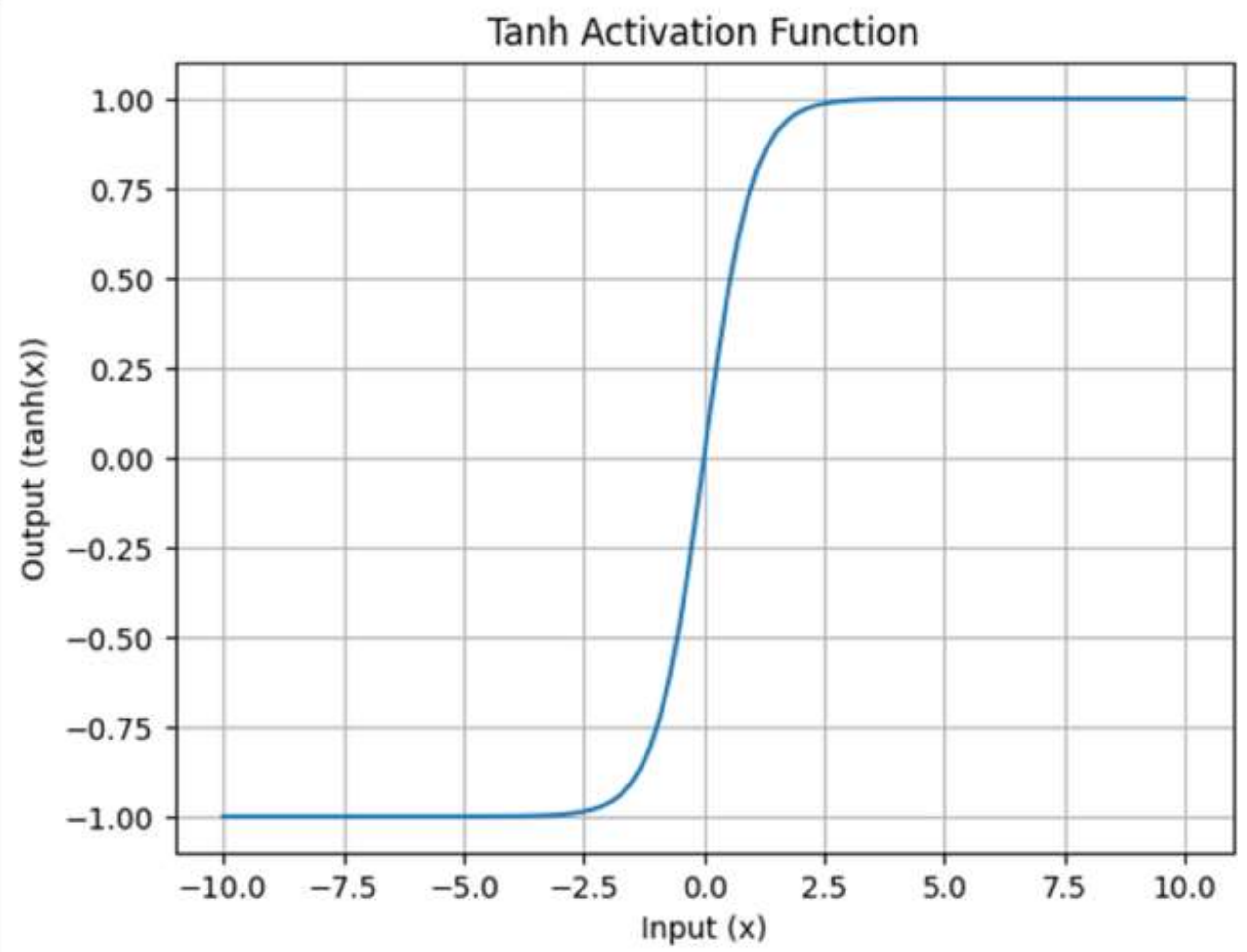
$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

Activation functions (tanh)

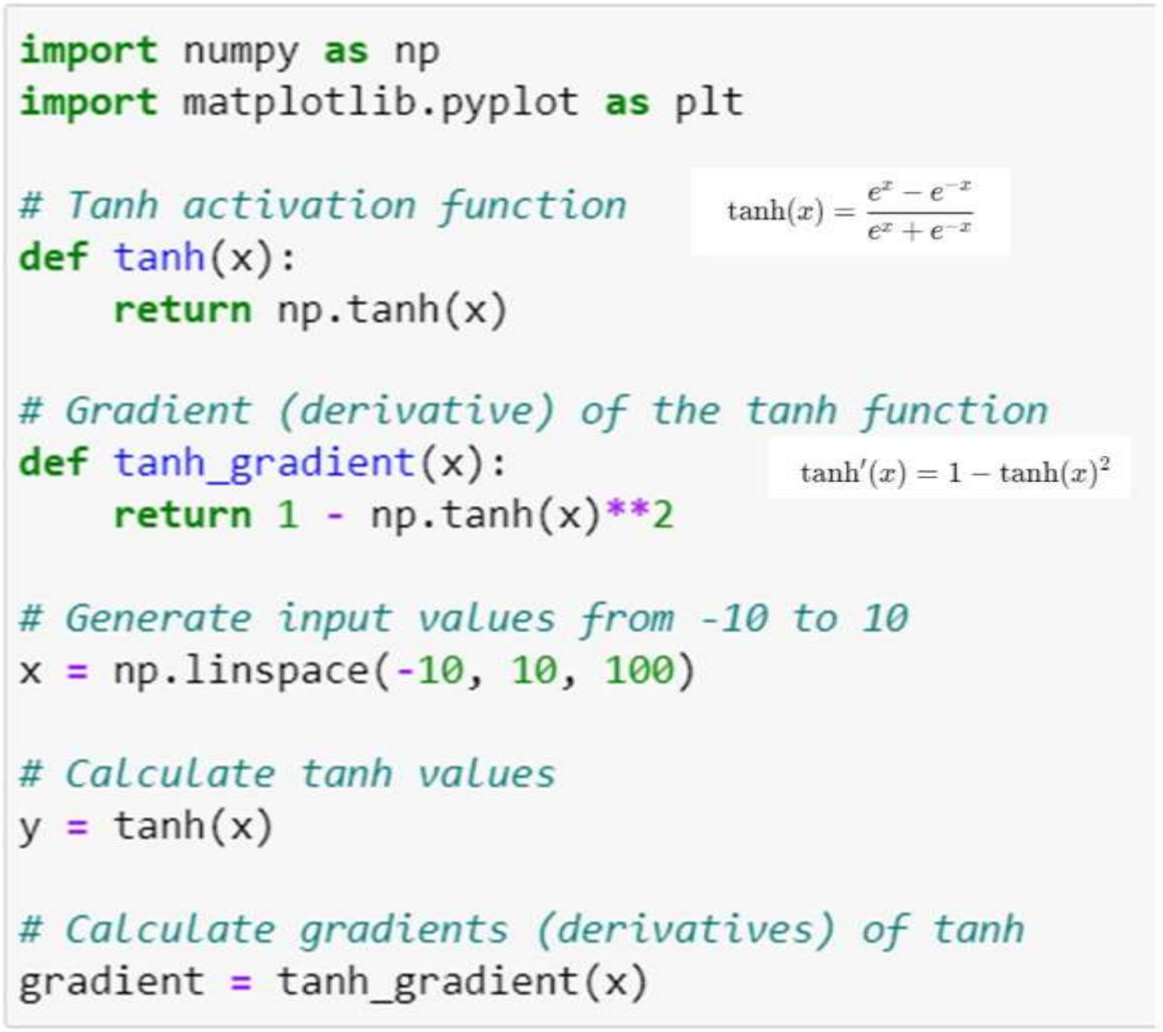


```
import numpy as np
import matplotlib.pyplot as plt

# Define the tanh function
def tanh(x):
    return np.tanh(x)

# Define the input range
x = np.linspace(-10, 10, 100)

# Calculate the output of the tanh function
y = tanh(x)
```

```
# Plot the tanh function
plt.plot(x, y, label='tanh(x)')
plt.title('Tanh Activation Function')
plt.xlabel('Input (x)')
plt.ylabel('Output (tanh(x))')
plt.grid(True)
plt.show()
```



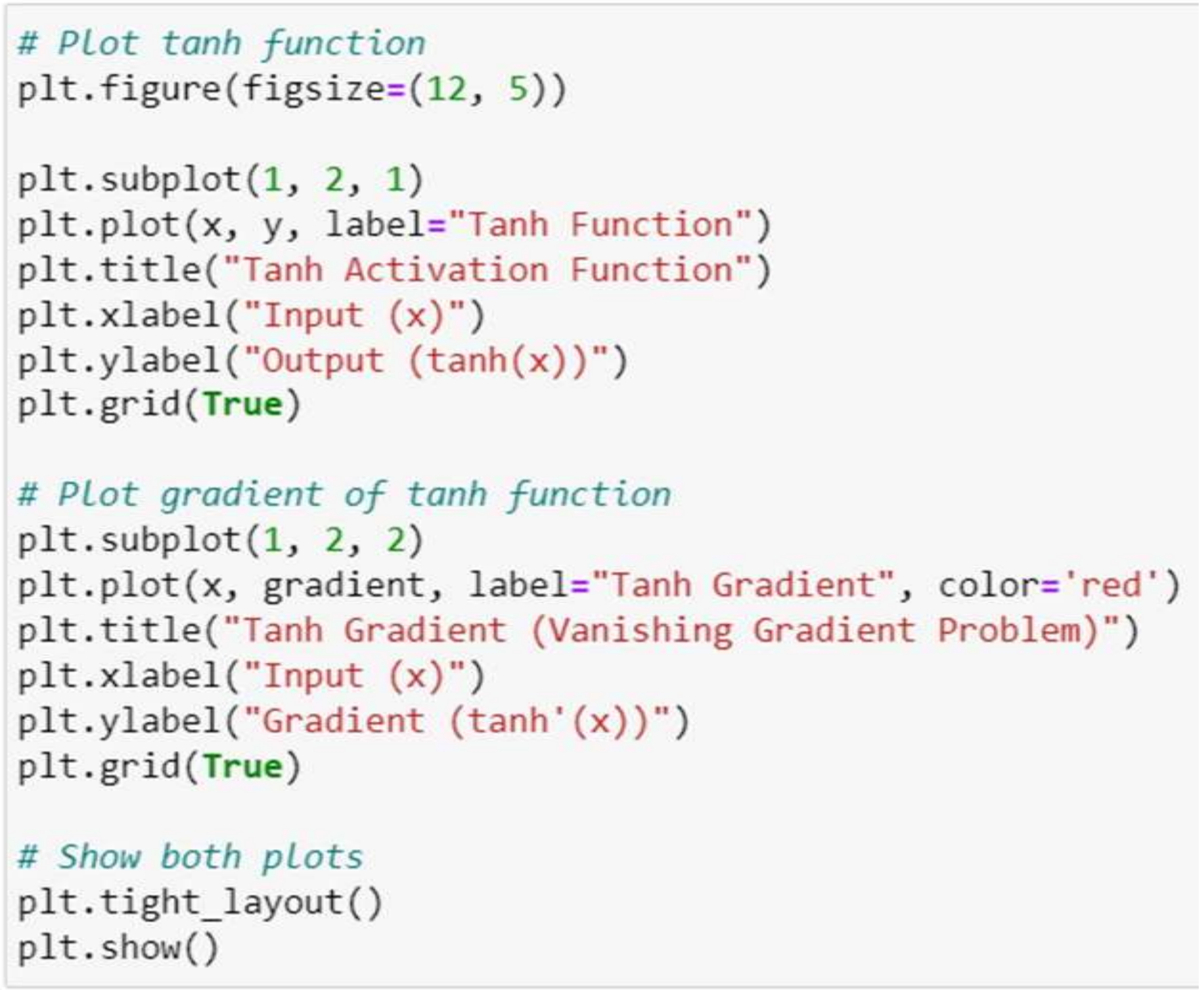
Activation functions (tanh)



```python
import numpy as np
import matplotlib.pyplot as plt

# Tanh activation function
def tanh(x):
    return np.tanh(x)

# Gradient (derivative) of the tanh function
def tanh_gradient(x):
    return 1 - np.tanh(x)**2

# Generate input values from -10 to 10
x = np.linspace(-10, 10, 100)

# Calculate tanh values
y = tanh(x)

# Calculate gradients (derivatives) of tanh
gradient = tanh_gradient(x)
```

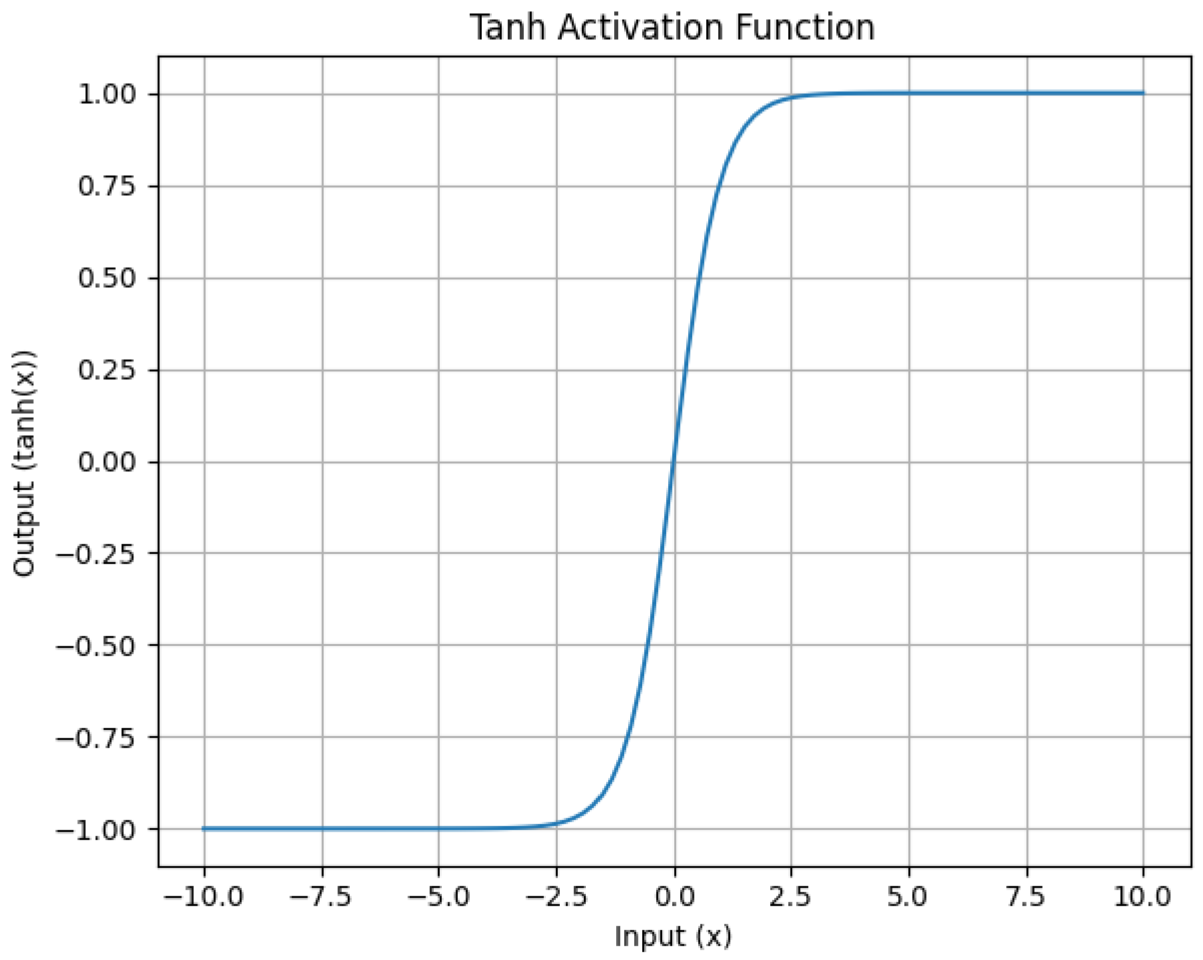
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

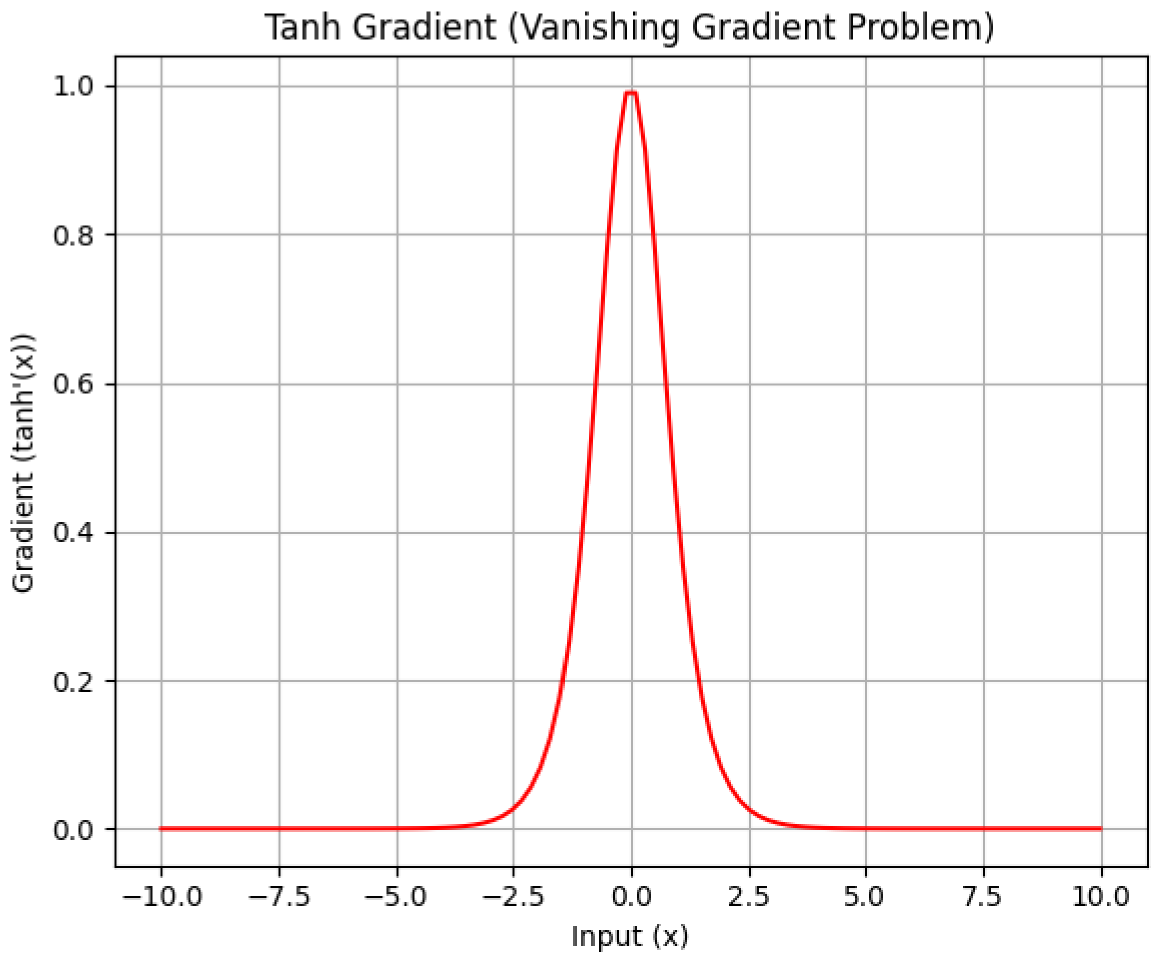
$$\tanh'(x) = 1 - \tanh(x)^2$$



```python
# Plot tanh function
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(x, y, label="Tanh Function")
plt.title("Tanh Activation Function")
plt.xlabel("Input (x)")
plt.ylabel("Output (tanh(x))")
plt.grid(True)

# Plot gradient of tanh function
plt.subplot(1, 2, 2)
plt.plot(x, gradient, label="Tanh Gradient", color='red')
plt.title("Tanh Gradient (Vanishing Gradient Problem)")
plt.xlabel("Input (x)")
plt.ylabel("Gradient (tanh'(x))")
plt.grid(True)

# Show both plots
plt.tight_layout()
plt.show()
```

Activation functions (tanh)

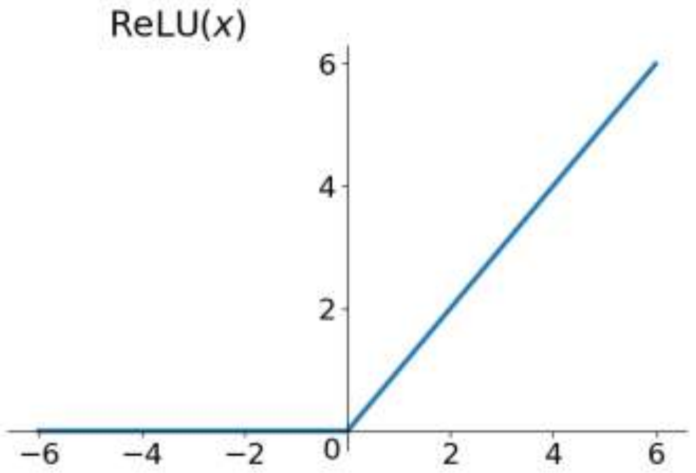


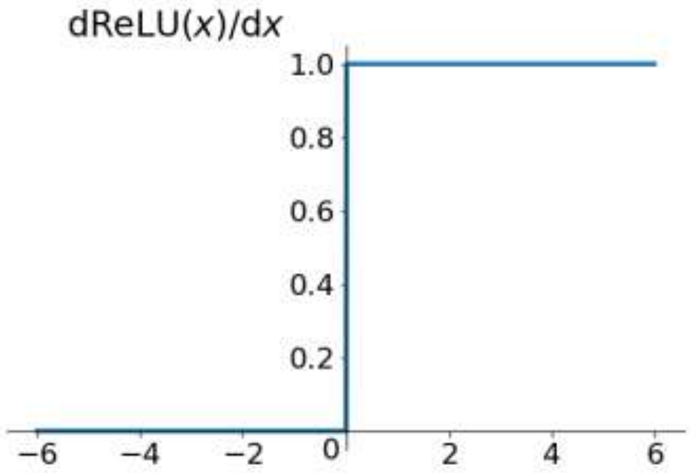


Activation functions (ReLU)

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

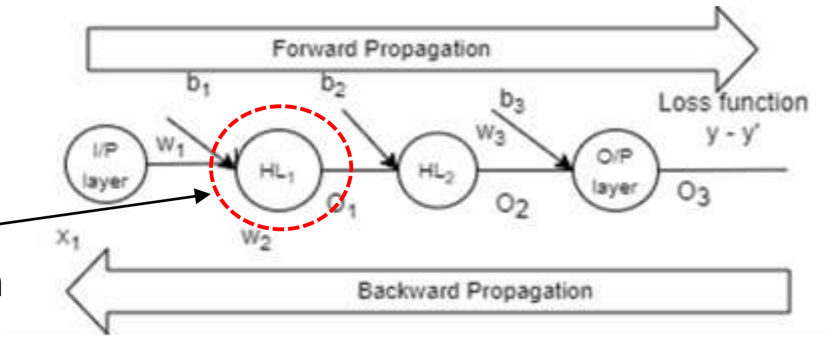
$$\text{ReLU} = \max(0, x)$$





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.

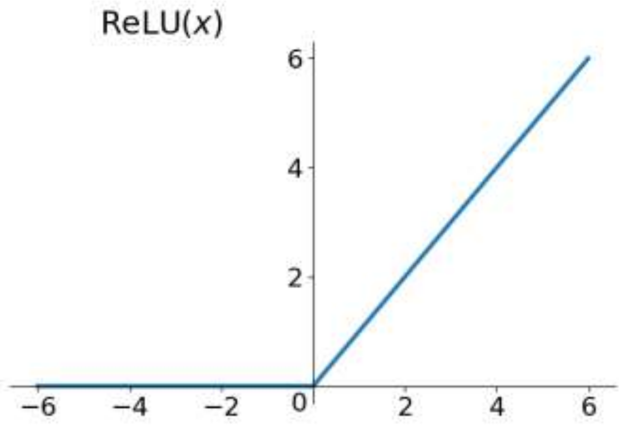


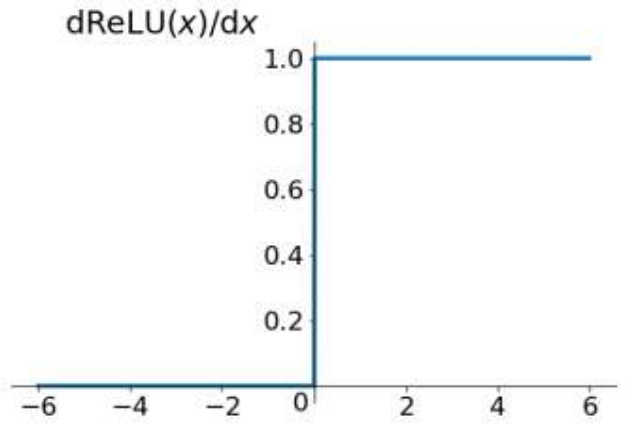
Dead neuron

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

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

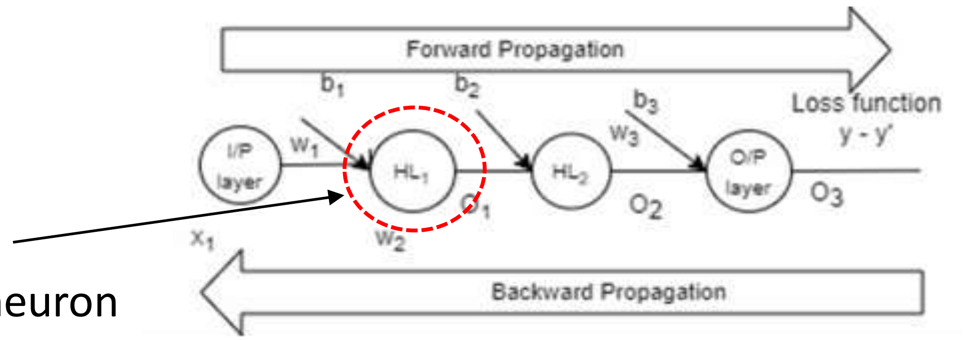
$$\text{ReLU} = \max(0, x)$$





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 ReLU(z) is1**
  - If **z is –ve** then derivative of **ReLU(z) is 0**



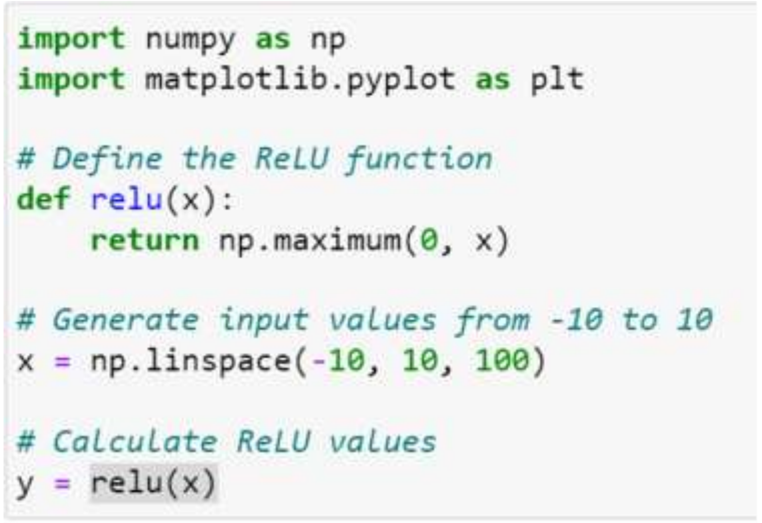
Dead neuron

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

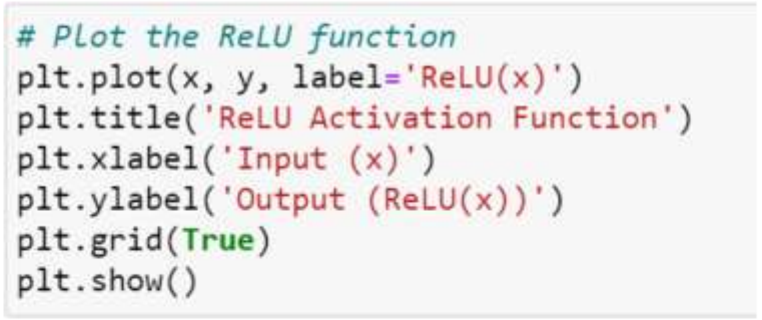
$$\frac{\partial L}{\partial W_{1old}} = \frac{\partial L}{\partial O_3} \times \boxed{\frac{\partial O_3}{\partial O_2}} \times \frac{\partial O_2}{\partial O_1} \times \frac{\partial O_1}{\partial W_{1old}}$$

Activation functions (ReLU)

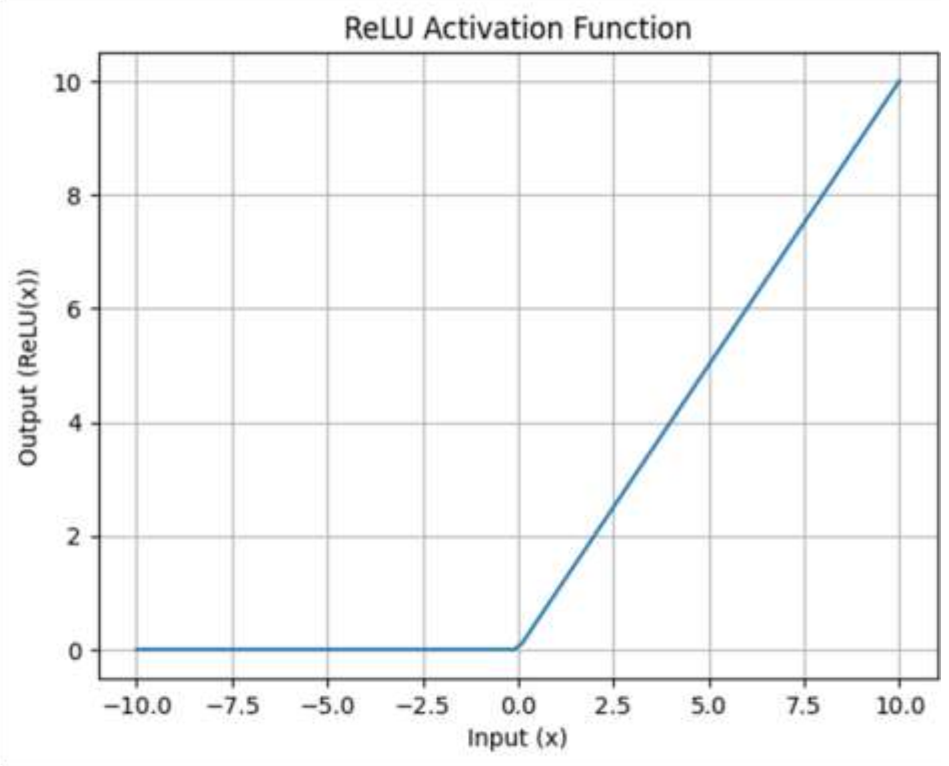
For the negative input values, the result is zero, that means the neuron does not get activated. Since only a certain number of neurons are activated, the ReLU function is far more computationally efficient when compared to the sigmoid and tanh function. Here is the python function for ReLU:



```python
import numpy as np
import matplotlib.pyplot as plt

# Define the ReLU function
def relu(x):
    return np.maximum(0, x)

# Generate input values from -10 to 10
x = np.linspace(-10, 10, 100)

# Calculate ReLU values
y = relu(x)
```



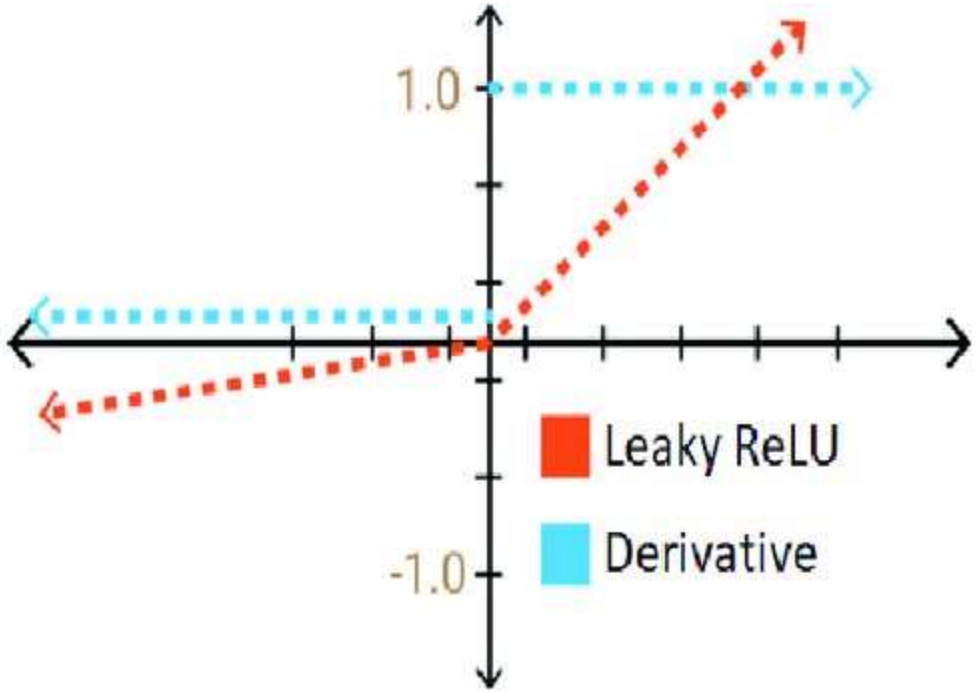
```python
# Plot the ReLU function
plt.plot(x, y, label='ReLU(x)')
plt.title('ReLU Activation Function')
plt.xlabel('Input (x)')
plt.ylabel('Output (ReLU(x))')
plt.grid(True)
plt.show()
```



Activation functions (Leaky ReLU)

- 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.

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

Softmax

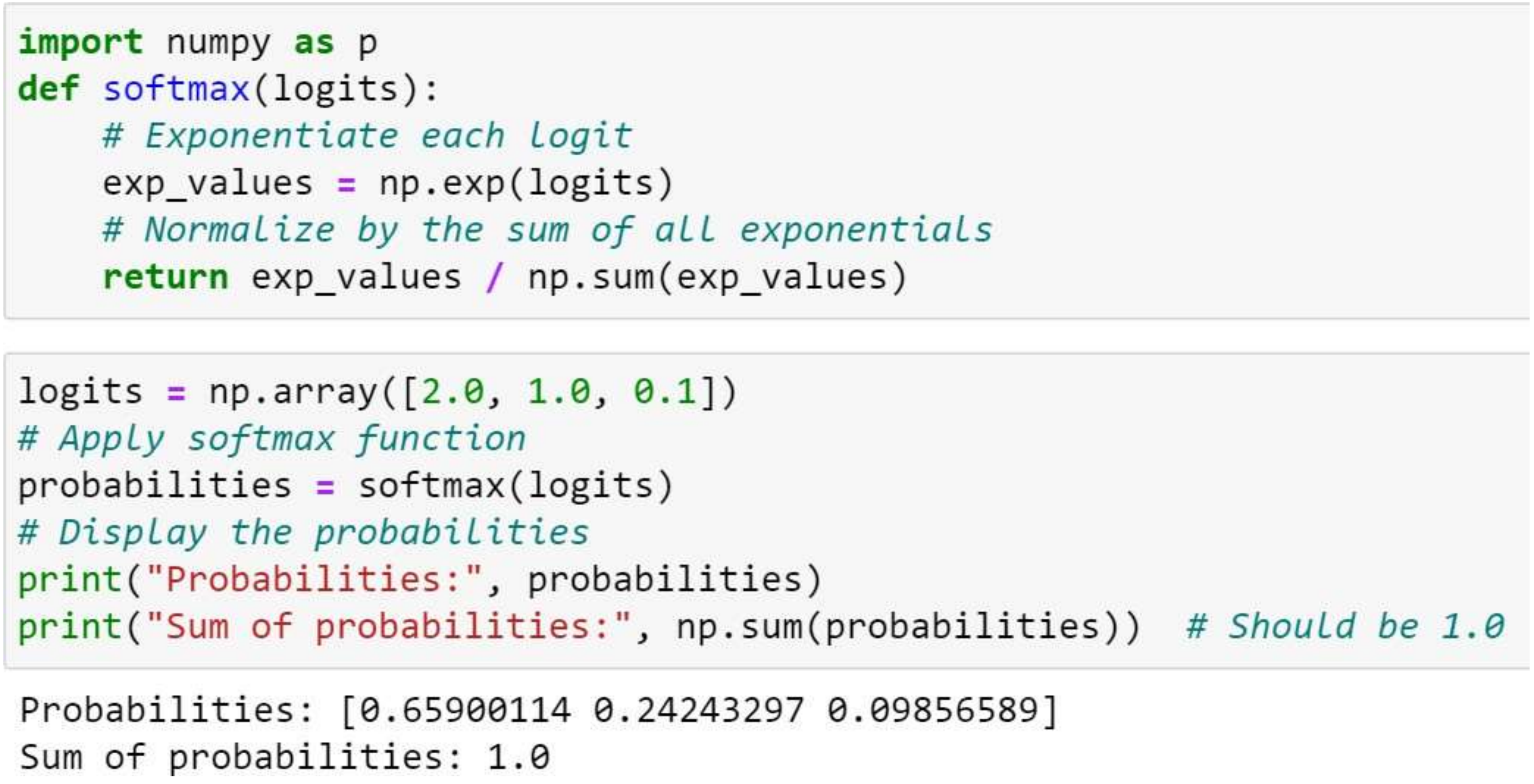
Example:

Imagine you are classifying an image into three categories: **cat, dog, and rabbit**. The neural network produces the following logits (raw scores):

- Cat: 2.0
- Dog: 1.0
- Rabbit: 0.1

The softmax function will convert these scores into probabilities that sum to 1.

Softmax



```python
import numpy as p
def softmax(logits):
    # Exponentiate each logit
    exp_values = np.exp(logits)
    # Normalize by the sum of all exponentials
    return exp_values / np.sum(exp_values)
```

```python
logits = np.array([2.0, 1.0, 0.1])
# Apply softmax function
probabilities = softmax(logits)
# Display the probabilities
print("Probabilities:", probabilities)
print("Sum of probabilities:", np.sum(probabilities))  # Should be 1.0
```

```
Probabilities: [0.65900114 0.24243297 0.09856589]
Sum of probabilities: 1.0
```

Softmax

Explanation:

1. **Raw logits (scores):** `[2.0, 1.0, 0.1]` are the raw scores from a neural network for three classes: Cat, Dog, Rabbit.
2. **Softmax Output (probabilities):** After applying softmax, the output is a probability distribution where the sum of probabilities equals 1.
    - The model predicts **Cat** with a probability of **65.9%**, **Dog** with **24.2%**, and **Rabbit** with **9.9%**.

Why Use Softmax?

- It converts logits into probabilities, which are easier to interpret when dealing with classification tasks.
- The class with the highest probability is typically chosen as the predicted class.

Activation functions (SoftMax)

- Here, K is the number of classes we have.
- The j is jth class
- $S(x_j)$ is the probability of jth class
- The x_j the jth class neuron weighted sum (z = x)

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

Example: Let's say you're working on a classification problem with 3 classes, and the neural network outputs the following raw scores (logits) for a particular input:
z=[2.0,1.0,0.1]. To apply the softmax function, follow these steps:

1. **Exponentiate the Logits**:
   1. Calculate $e^{z1}=e^{2.0} \approx 7.389$
   2. Calculate $e^{z2}=e^{1.0} \approx 2.718$
   3. Calculate $e^{z3}=e^{0.1} \approx 1.105$
2. **Sum of Exponentials**:
   1. Sum=7.389+2.718+1.105 ≈ 11.212
3. **Compute Probabilities**:
   1. $y^{\wedge}_1 = 7.389/11.212 \approx 0.659$
   2. $y^{\wedge}_2 = 2.718/11.212 \approx 0.242$
   3. $y^{\wedge}_3 = 1.105/11.212 \approx 0.099$
4. So, after applying the softmax function, the output probabilities are approximately:
   y^=[0.659, 0.242, 0.099]

Class activity: Draw the architecture of ANN for multi-classification Where you are supposed to deal with 4 classes. Create an ANN with In your drawing show the following:

1. How many Input neuron at input layer?
2. How many Hidden Layers (HL) and how many neurons at each HL?
3. Initial value of learning rate and weights?
4. Which activation functions at HL and output layer?

Choosing the right Activation Function

- Which activation function should be used in which situation. Good or bad – there is no rule of thumb.

- Depending upon the properties of the problem we might be able to make a better choice for easy and quicker convergence of the network.

  - **Sigmoid** functions and their combinations generally work better in the case of classifiers

  - **Sigmoids** and **tanh** functions are sometimes avoided due to the vanishing gradient problem

  - **ReLU** function is a general activation function and is used in most cases these days

  - Always keep in mind that ReLU function should only be used in the hidden layers

  - As a rule of thumb, you can begin with using ReLU function and then move over to other activation functions in case

  - ReLU doesn't provide with optimum results

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