

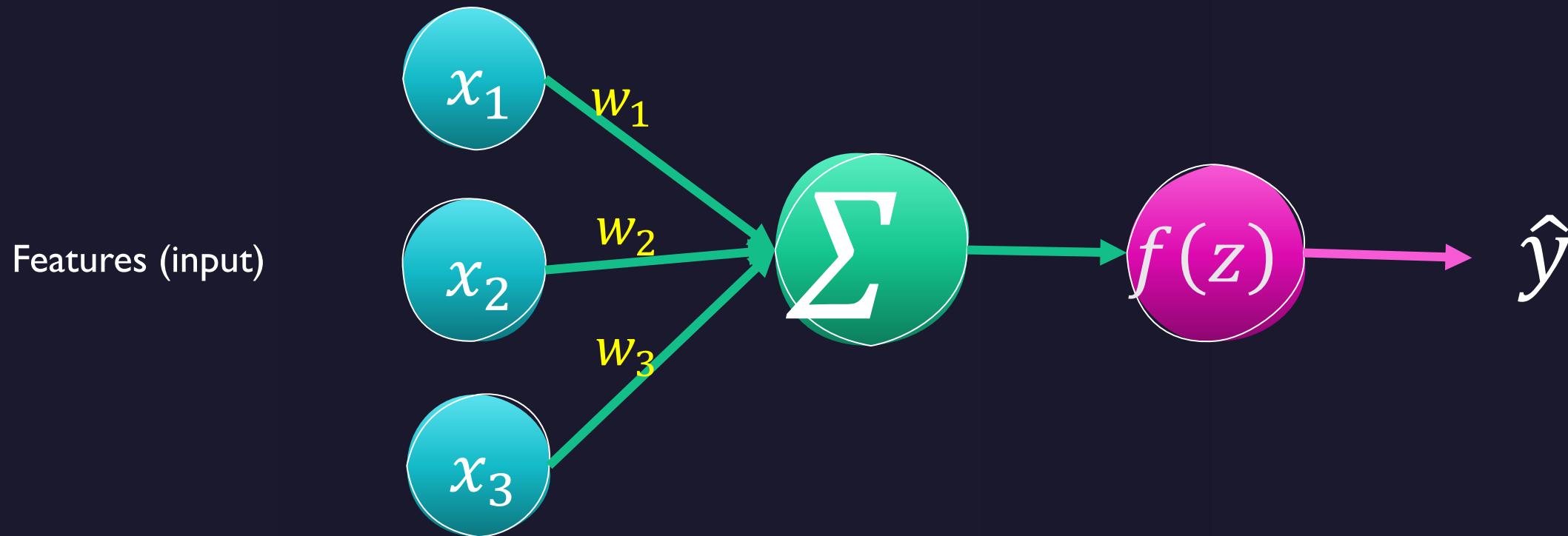


Artificial *Neural* *Networks (ANN)*

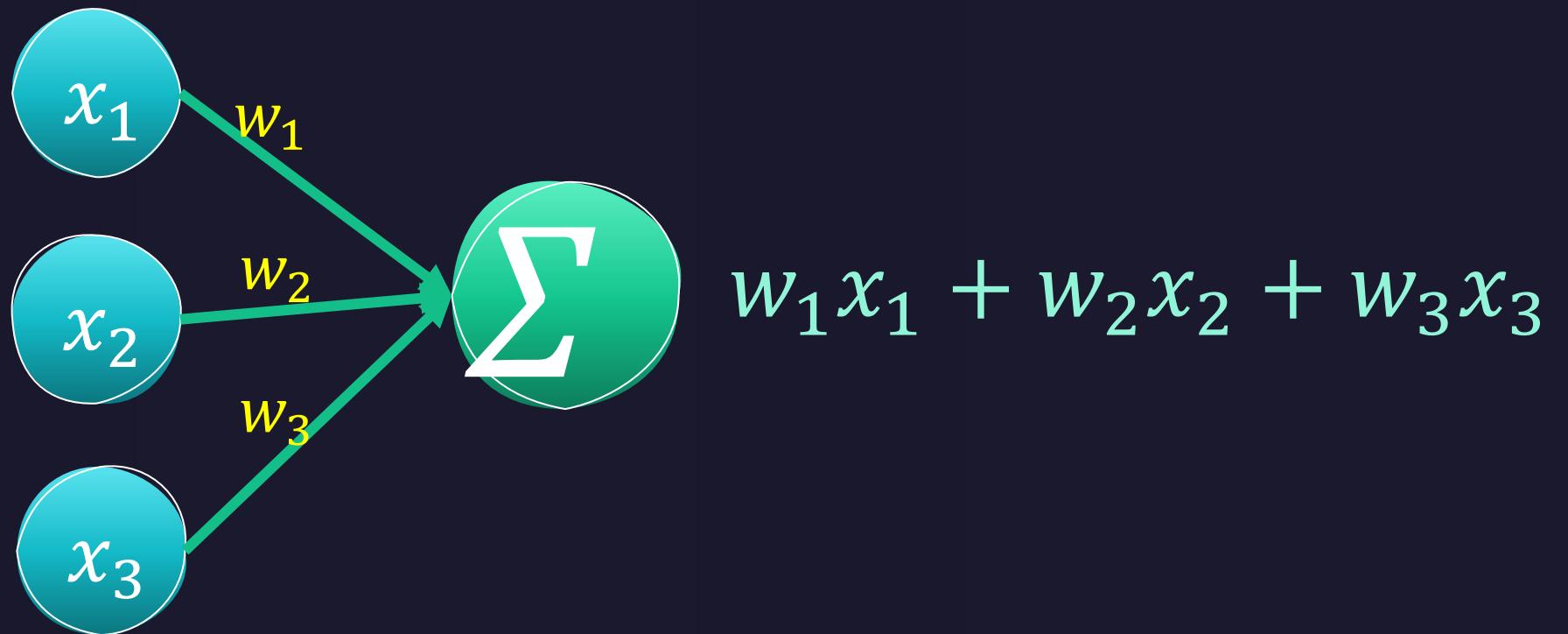
Hossam Ahmed

Mario MAMDOUN

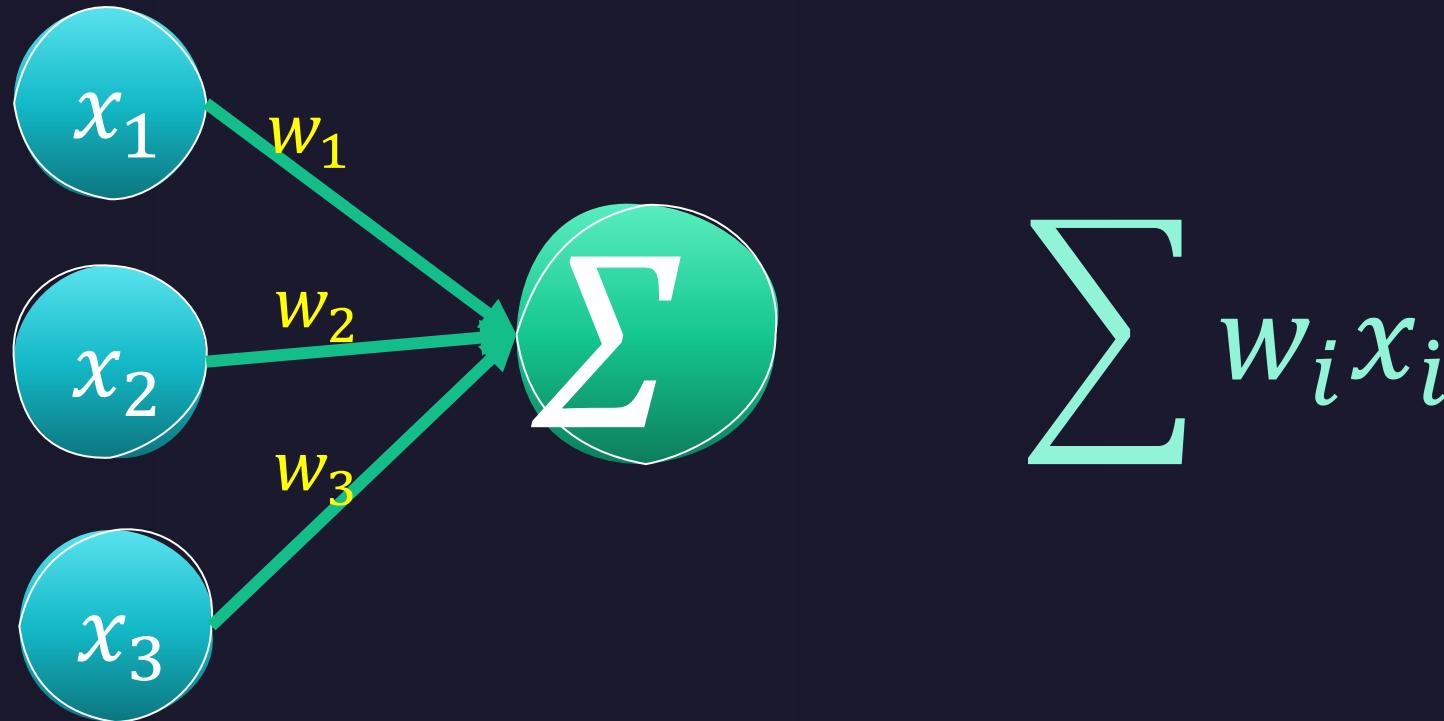
Perceptron



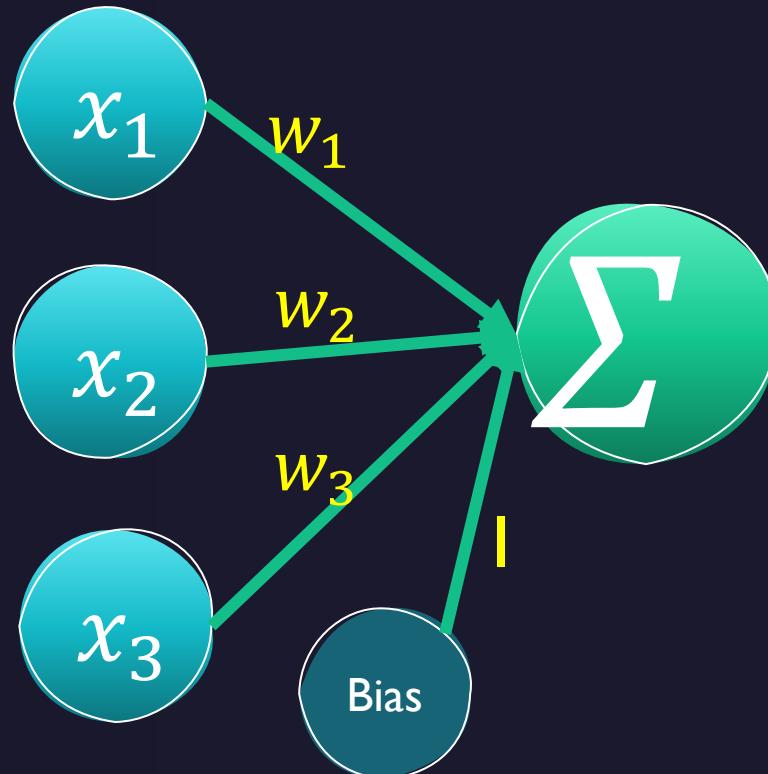
Perceptron



Perceptron



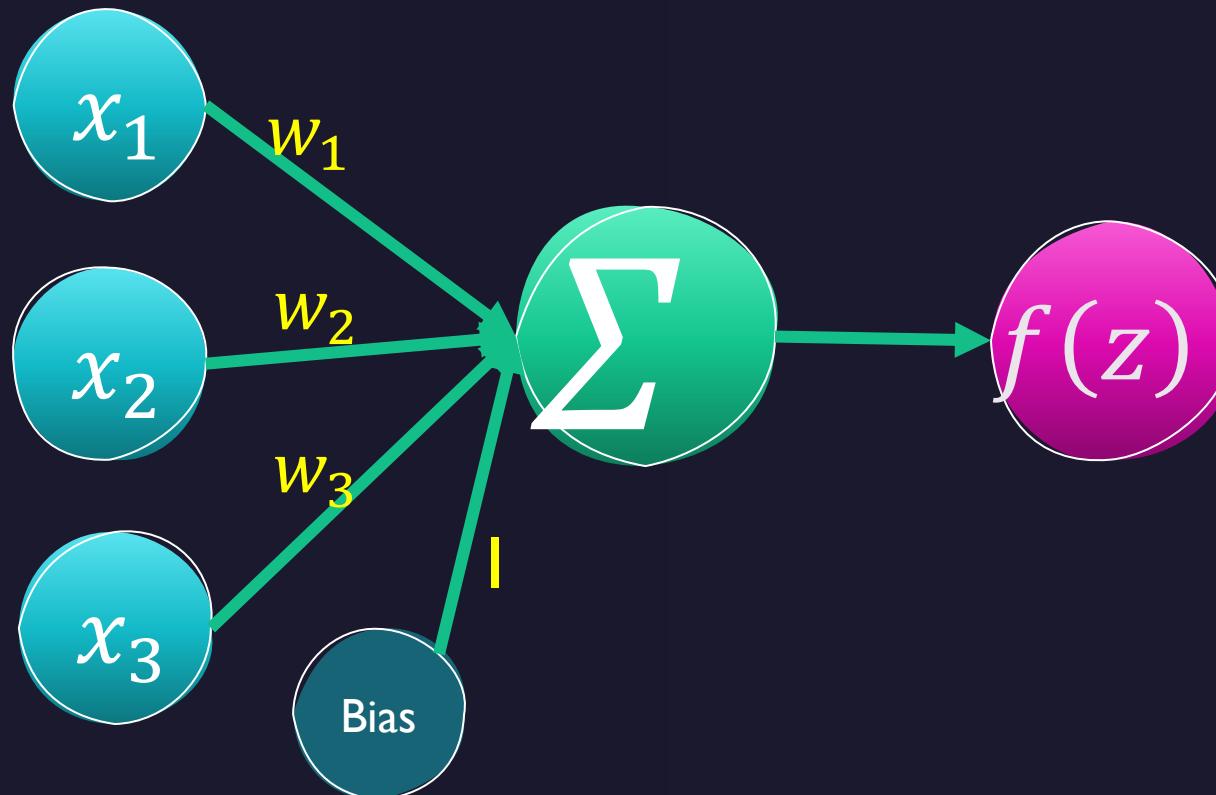
Perceptron



$$\text{Bias} + \sum w_i x_i$$

- Why using **Bias** ?
- allows the network to learn more complex functions and improve its ability to generalize to new data.
- Considering some factors that is not in the input and may cause error.
- It's as why we used bias in linear regression it adds more moving over y axis that provide better options for fitting the data
- Until now this image **represent a linear regression** a linearity .

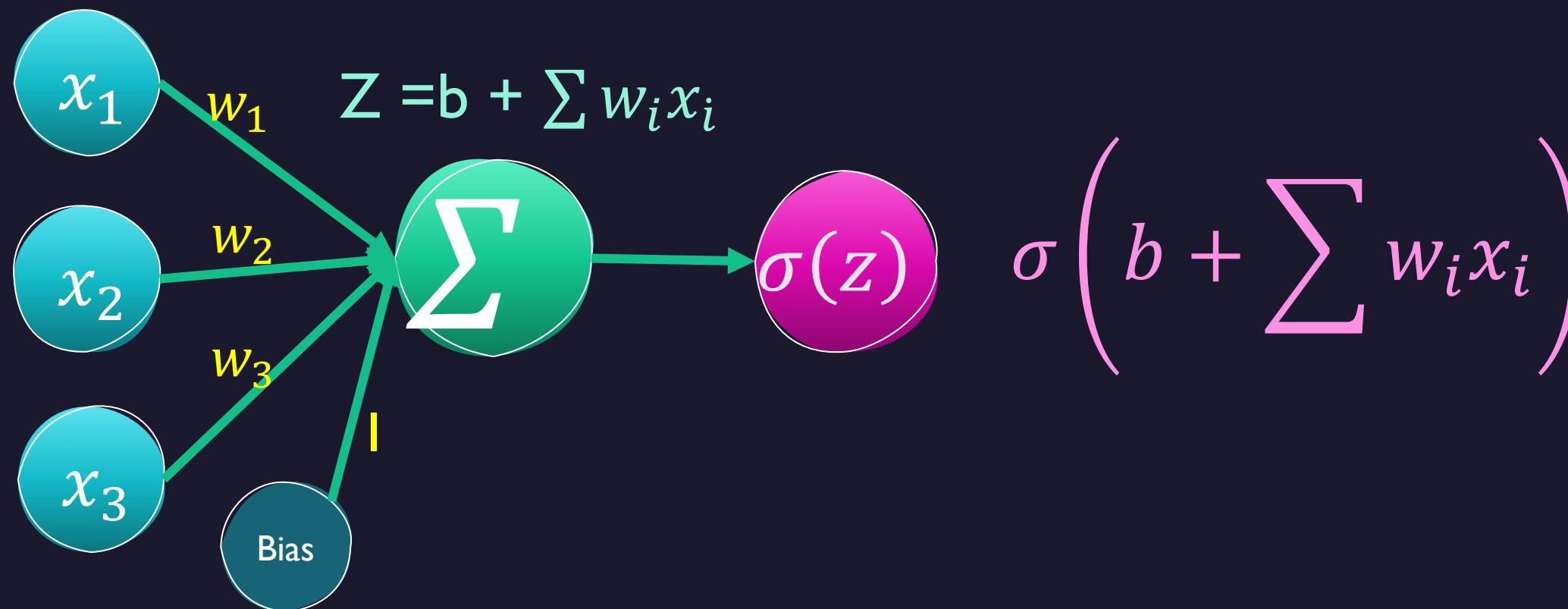
Perceptron



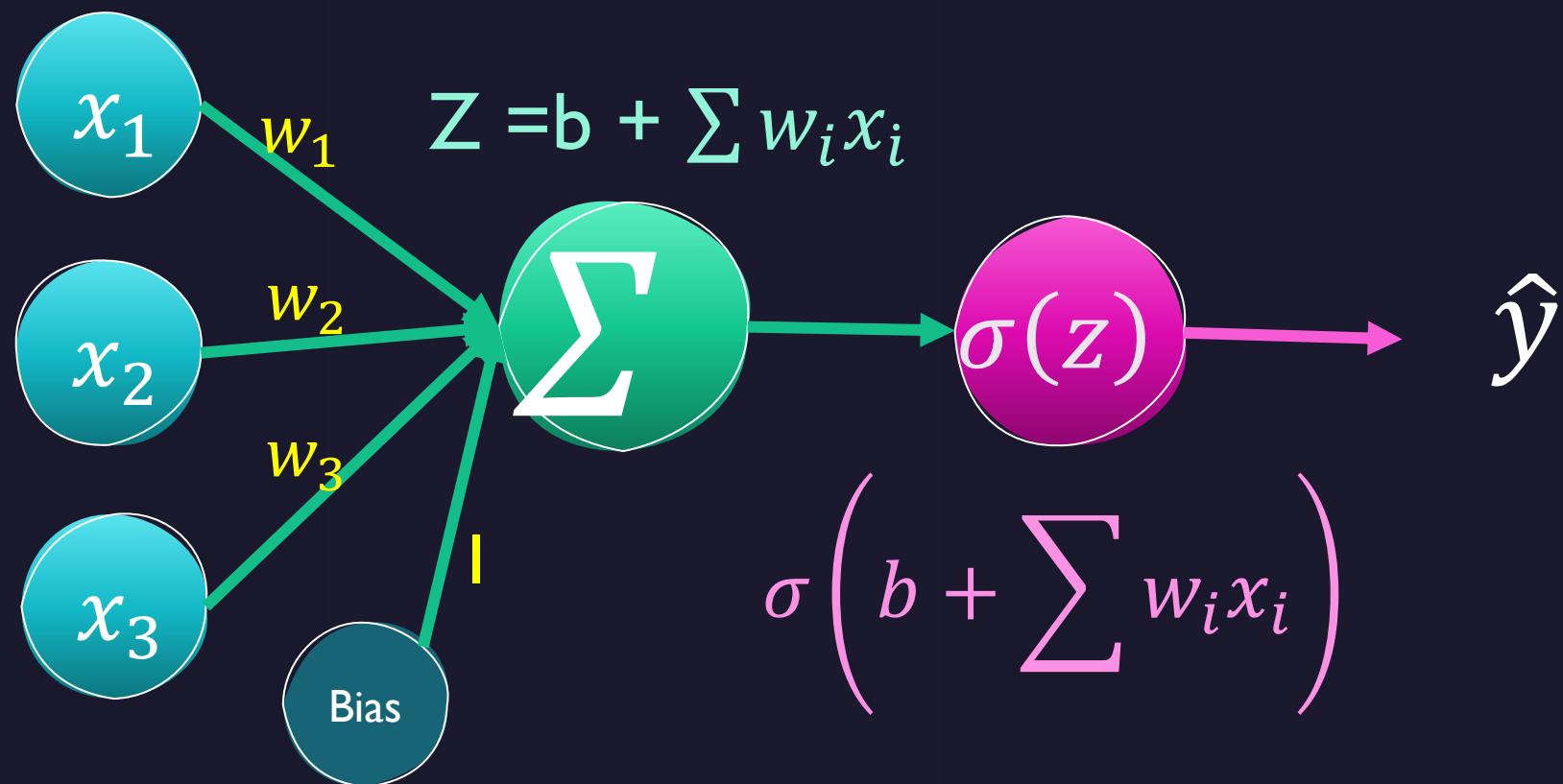
- We need to **get rid of the linearity !**
- **Linear Models** not useful in **Classification** unless it mapped to probabilities using a function like **Logistic function**
- So, we choose a logistic function to be **F(Z)**
- This is called **Activation function**

Perceptron

- We created a **Classification Perceptron** equivalent to **Logistic Regression**

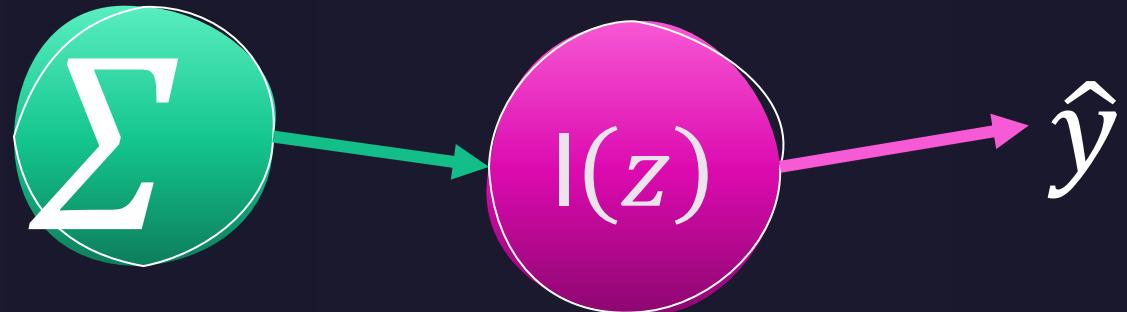


Perceptron



Why Nonlinear Activation Function?

$$Z = b + \sum w_i x_i$$



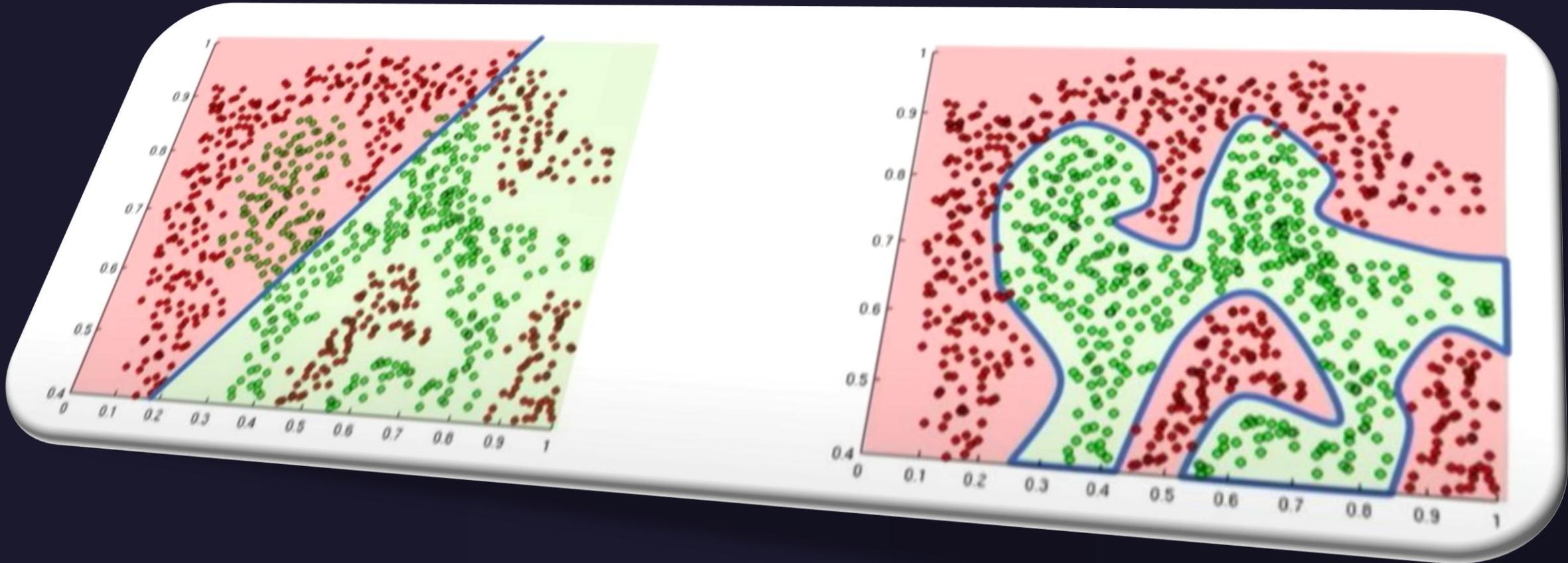
$$x^T w + b$$

$$\text{Line}(b + \sum w_i x_i)$$

$$\text{Line}(\text{Line})$$

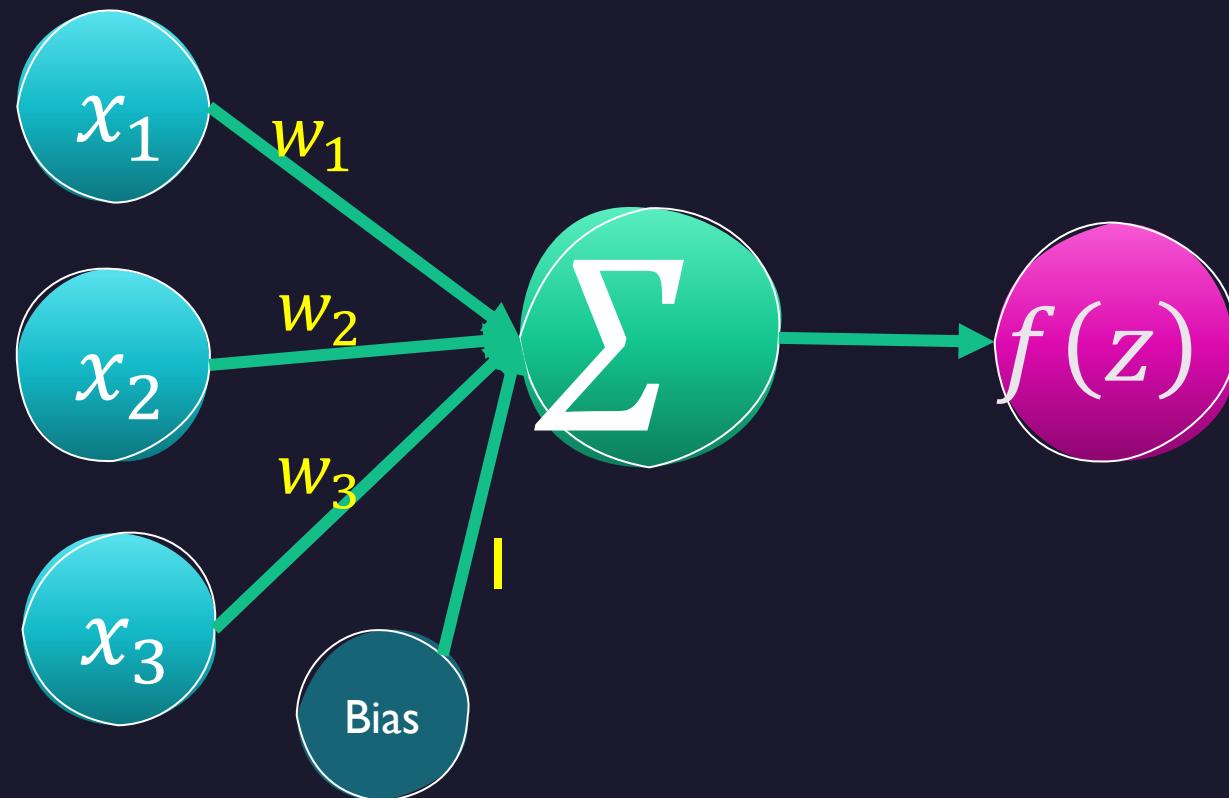
Another Line in different dimensions

Why Nonlinear Activation Function?



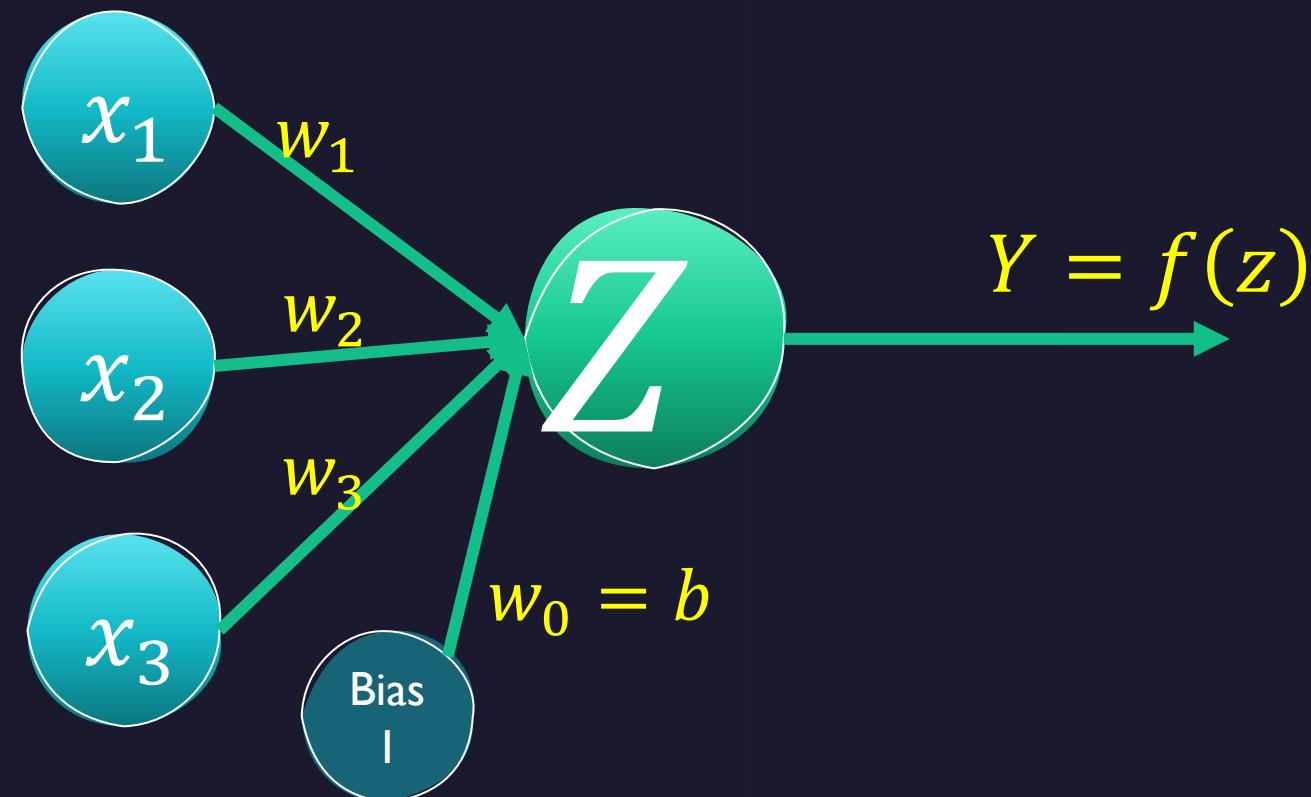
Perceptron

building block



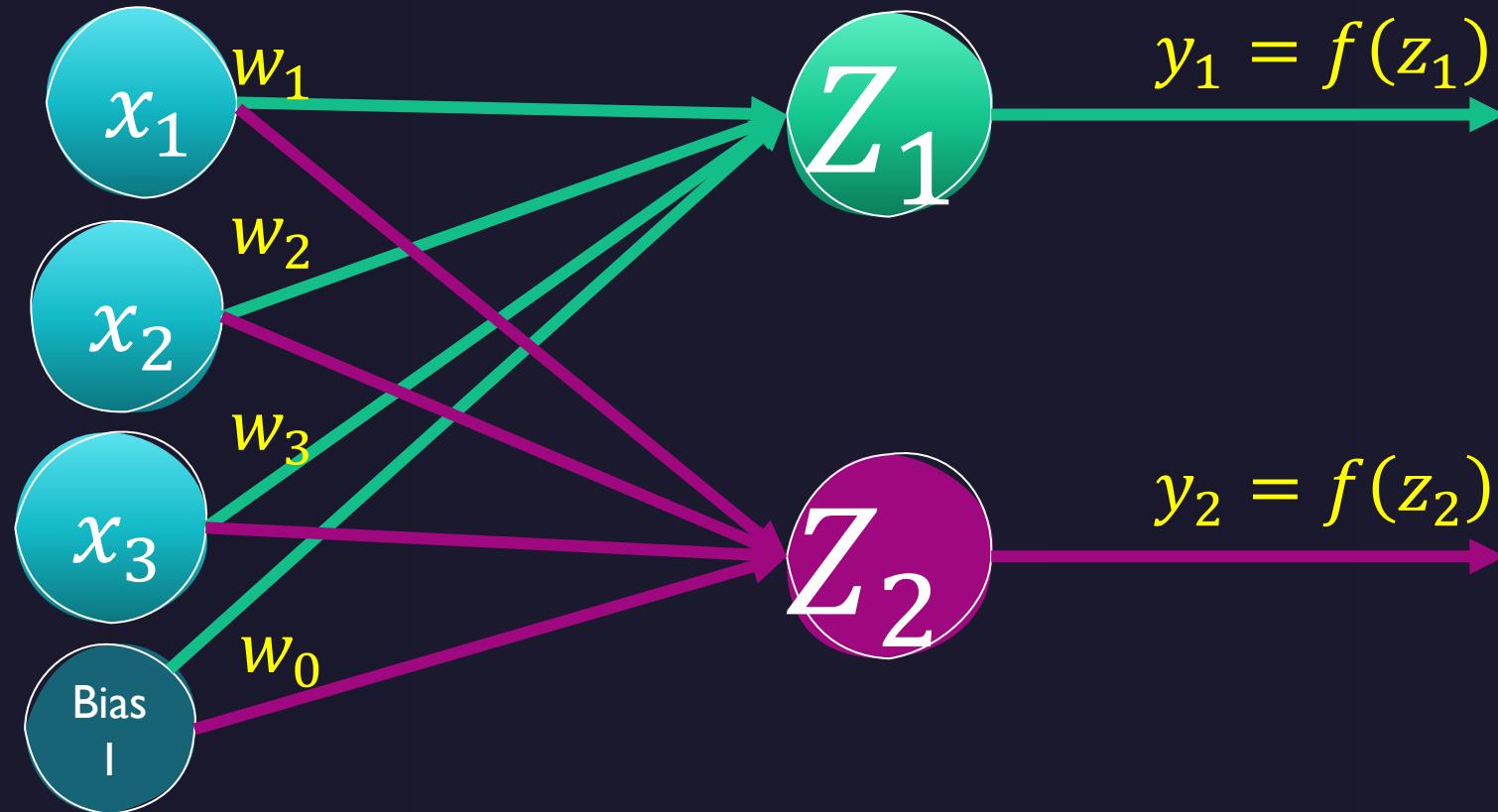
Perceptron

building block

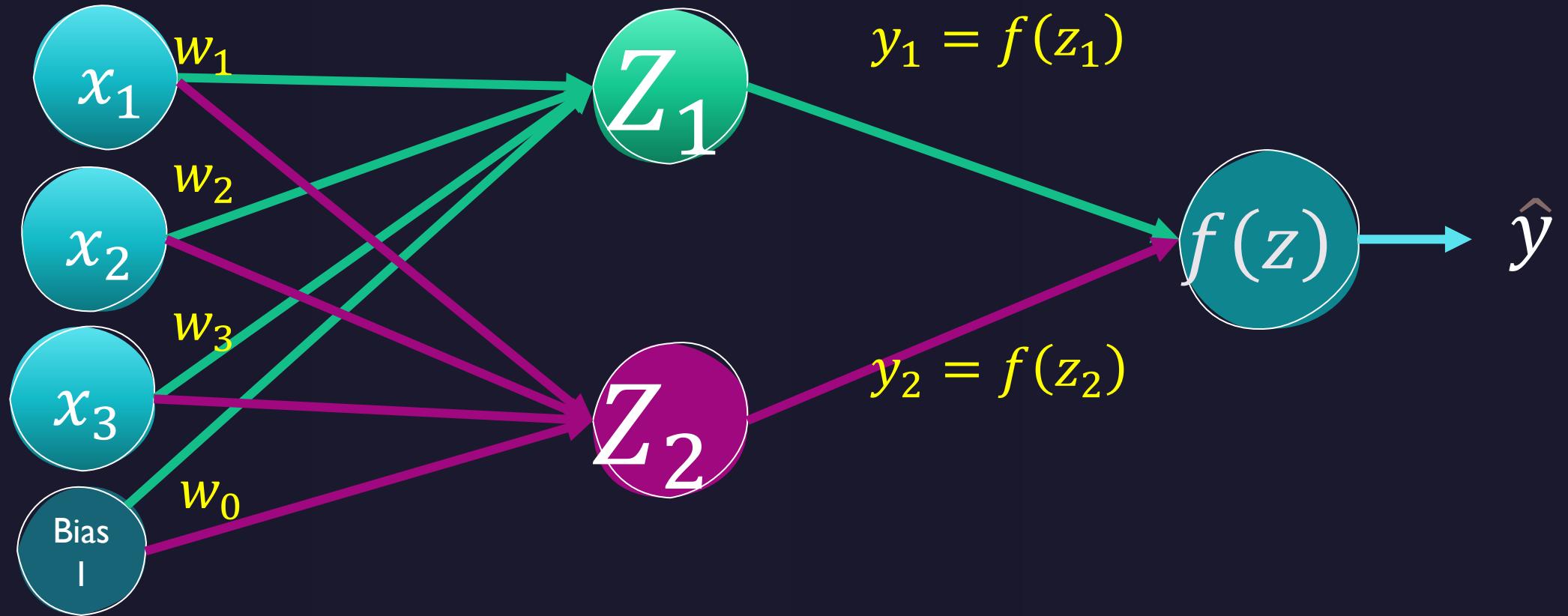


Perceptron

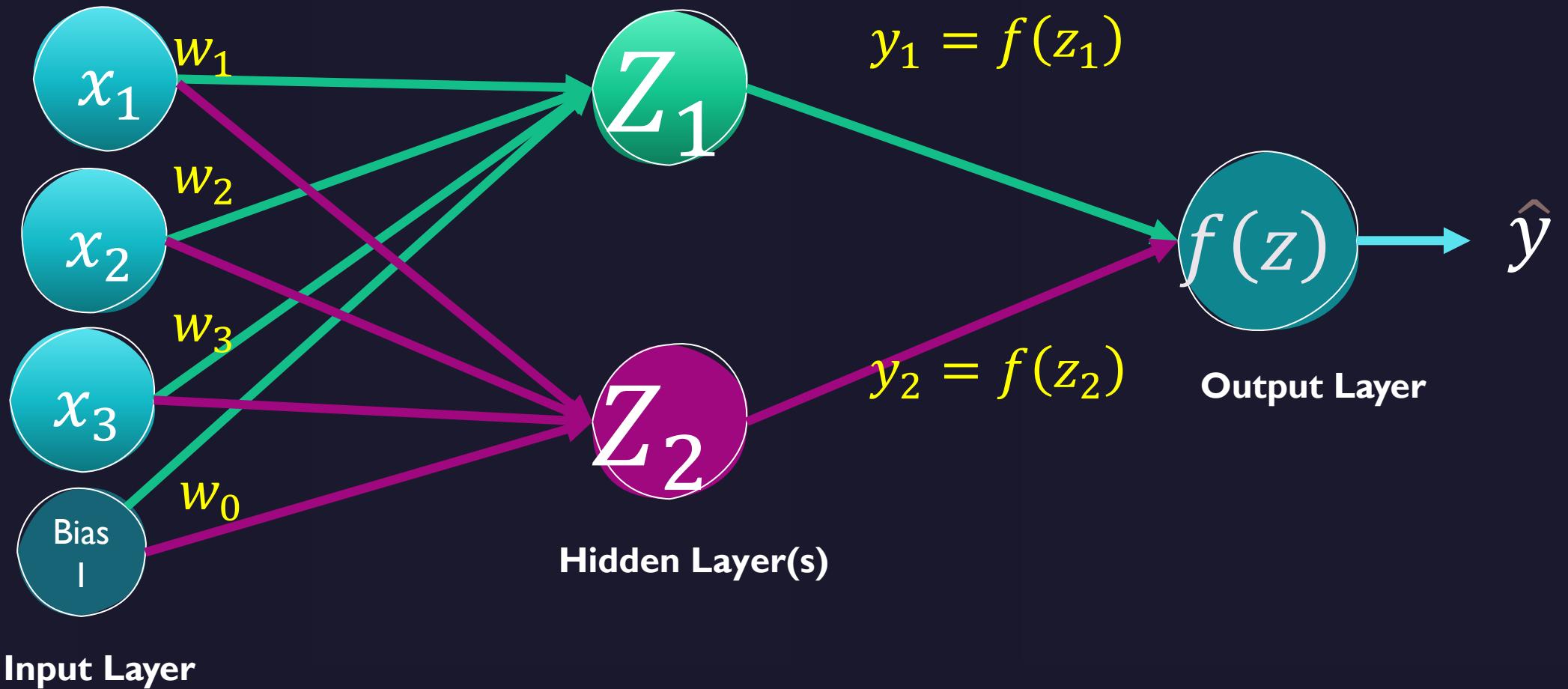
building block

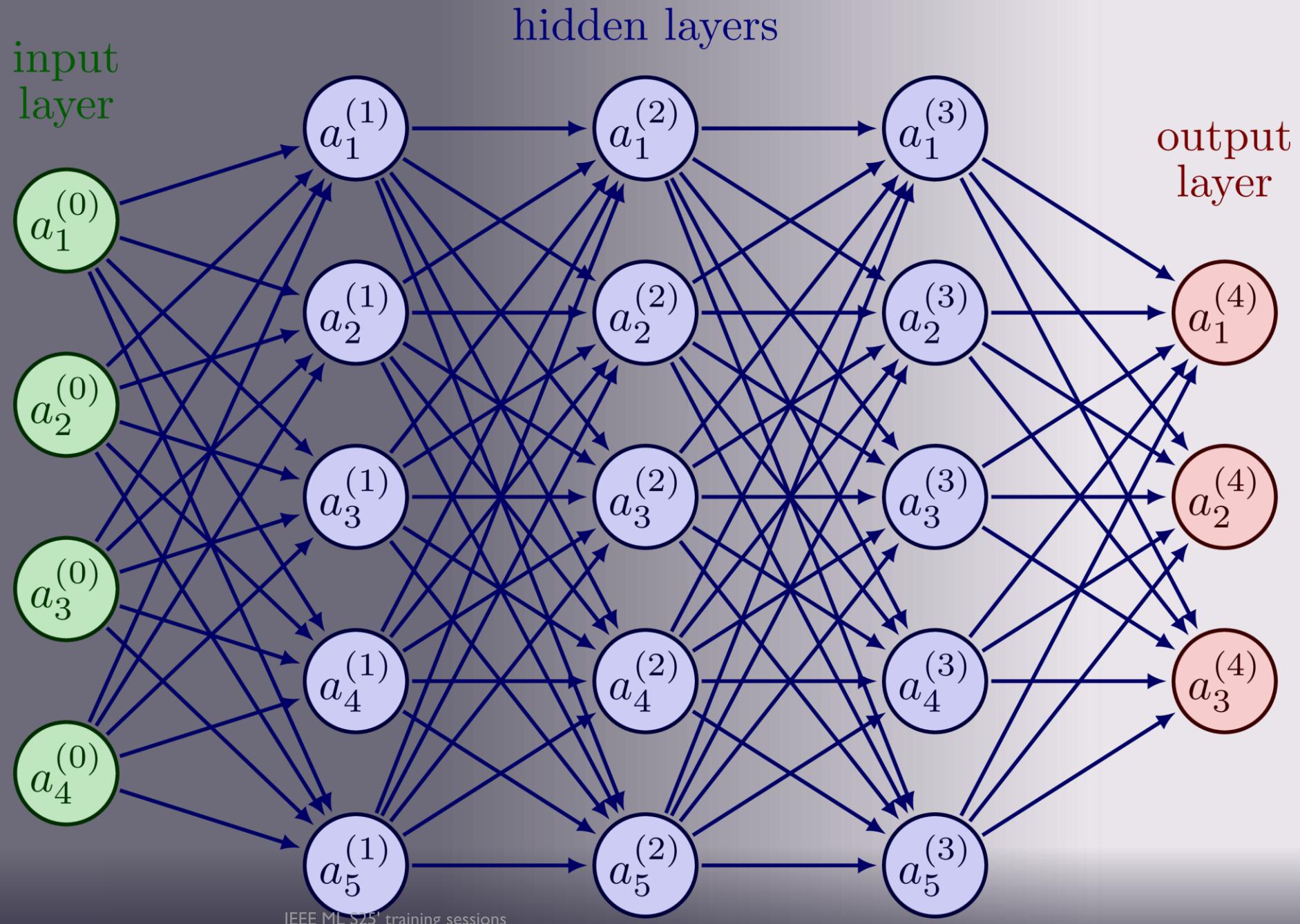


Simple NN



Simple NN

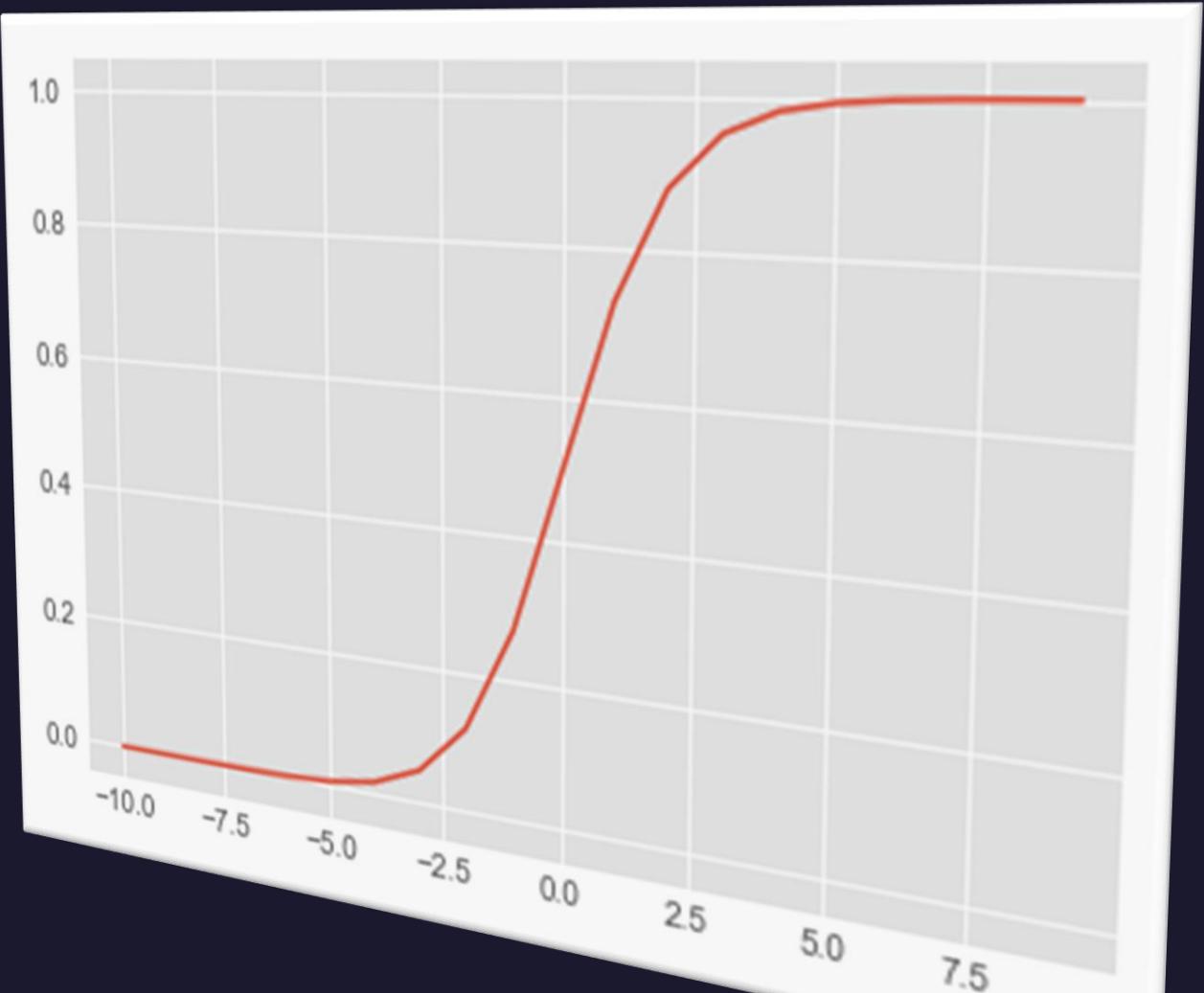




Activation functions

Logistic

```
def Logistic(x):  
    return 1/(1+np.e**(-x))  
inputs = [x for x in range(-10,10)]  
outputs = [Logistic(x) for x in inputs]  
plt.plot(inputs, outputs)  
# range from 0 ot 1
```



Logistic

When to use it?

- The output layer in binary classification
- Don't use it in the hidden layers as activation function

Pros

- It produces outputs in the range of 0 to 1, which can be interpreted as probabilities.
- Good for binary classification

Cons

- It suffers from the **vanishing gradient problem**, which can slow down the learning process

- Logistic function derivative $\frac{d}{dx} \sigma(x) :$
- $\frac{d}{dx} \sigma(x) = \sigma(x)(1 - \sigma(x))$

- Logistic function range is $(0,1)$, its domain is \mathbb{R} , it's symmetric around $x = 0$, it's differentiable everywhere.

Softmax

```
def softmax(x):
    return np.exp(x)/np.exp(x).sum()
inputs = [0.1, 0.2, 0.3]
outputs = softmax(inputs)
print(outputs)
print(outputs.sum())
```

```
[0.30060961 0.33222499 0.3671654 ]
1.0000000000000002
```

- Softmax function

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_j^k e^z_j}$$

- Softmax function derivative

$$\frac{d}{dz_j} \sigma_i = \begin{cases} \sigma_i(1 - \sigma_i) & \text{if } i = j \\ -\sigma_i \sigma_j & \text{if } i \neq j \end{cases}$$

Softmax

When to use it?

- The output layer in multi-class classification
- Don't use it in the hidden layers as activation function

$$\sigma_{\text{sigmoid}}(x) = \frac{\exp(x_j)}{\exp(x_j) + 1}$$

Pros

- It can be used for **multi-class classification** problems.
- It produces outputs in the range of 0 to 1, which can be **interpreted as probabilities** that sum up to 1.
- Generalization of logistic with classes = 2

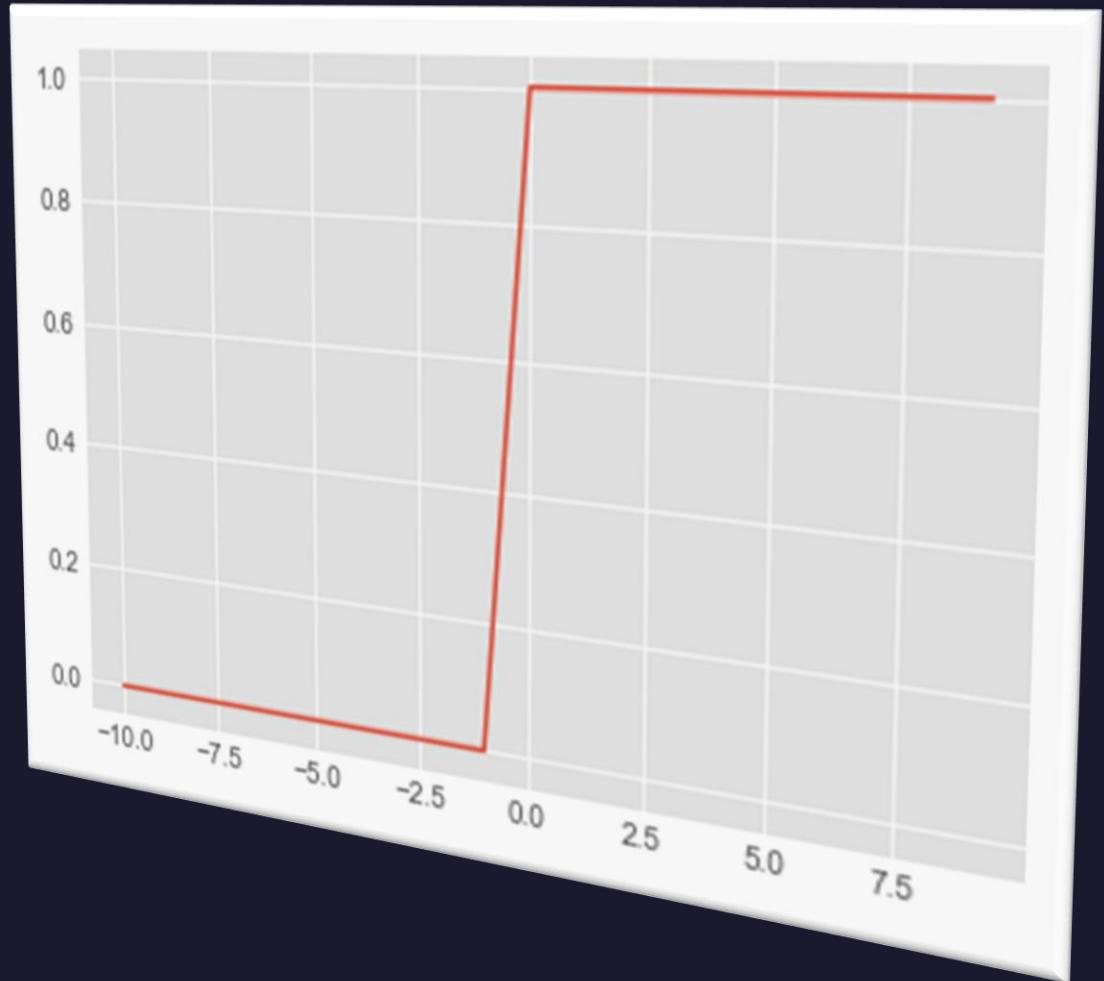
Cons

- It suffers from the **vanishing gradient problem**, which can slow down the learning process
- Derivatives can go to very small numbers almost zero confusing the GD and make it slow the Learning rate

$$\sigma_{\text{softmax}}(x_j) = \frac{\exp(x_j)}{\sum_{k=0}^{K-1} \exp(x_k)}$$

Binary Step

```
def bin_step(x):  
    if x < 0:  
        return 0  
    else:  
        return 1  
inputs = [x for x in range(-10,10)]  
outputs = [bin_step(x) for x in inputs]  
plt.plot(inputs, outputs)  
plt.show()
```



Binary step

When to use it?

- Don't

Pros

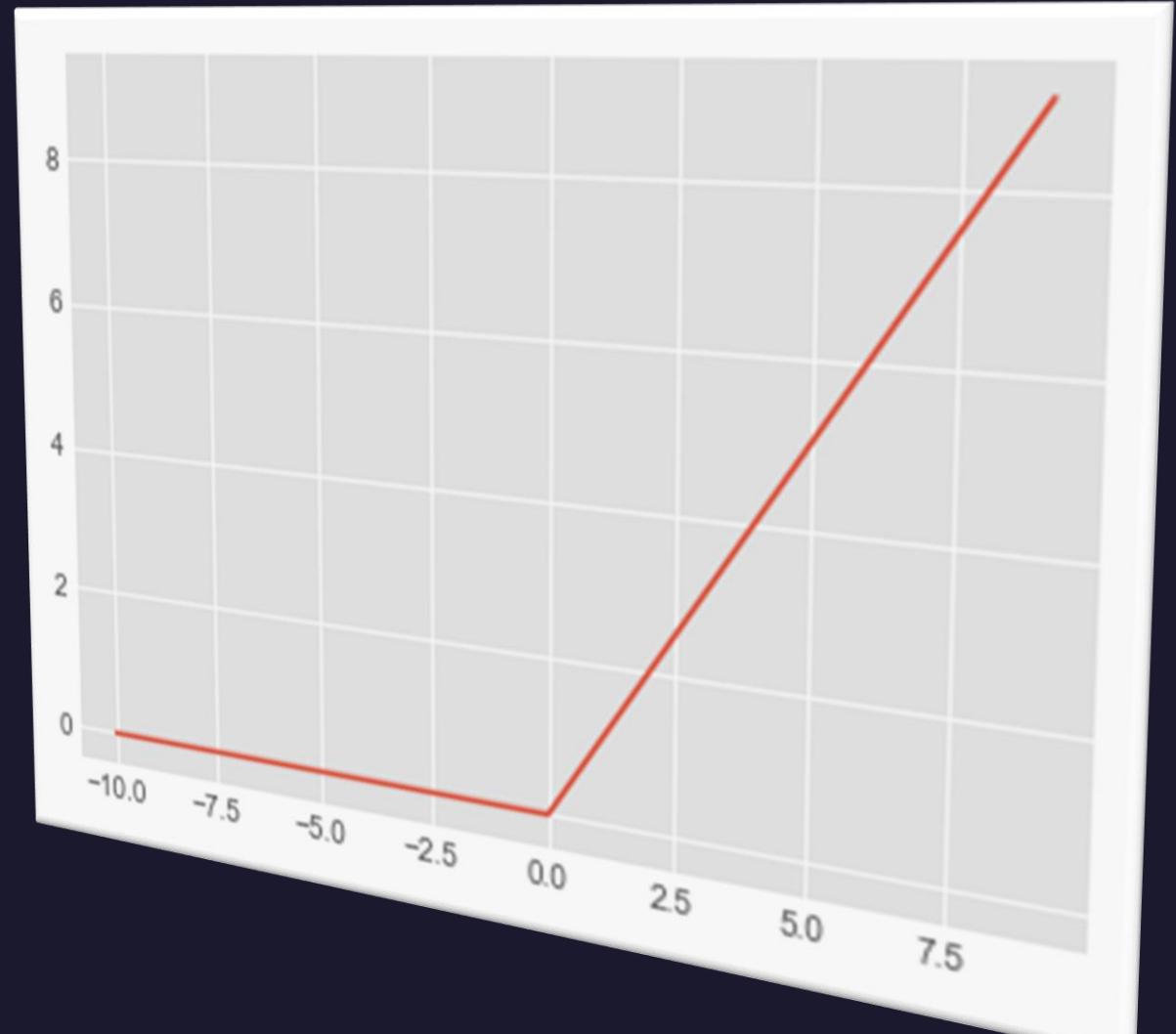
- It is **computationally efficient** and easy to implement.
- It is **computationally efficient** and easy to implement.
- It can be useful for thresholding the output of a neuron to produce binary outputs.

Cons

- It is **not differentiable** at the point where the output switches from 0 to 1, which can make it difficult to use in gradient-based optimization algorithms.
- It does not provide information about the magnitude of the input, which can limit its usefulness in some applications.
- It can suffer from the **vanishing gradient problem**, which occurs when the gradients become very small or zero, making it difficult to update the weights during training

ReLU

```
def rectified(x):  
    return max(0.0, x)  
inputs = [x for x in range(-10,10)]  
outputs = [rectified(x) for x in  
inputs]  
plt.plot(inputs, outputs)  
plt.show()
```



ReLU

When to use it?

- Hidden Layers
- Commonly used
- Default in most cases

Pros

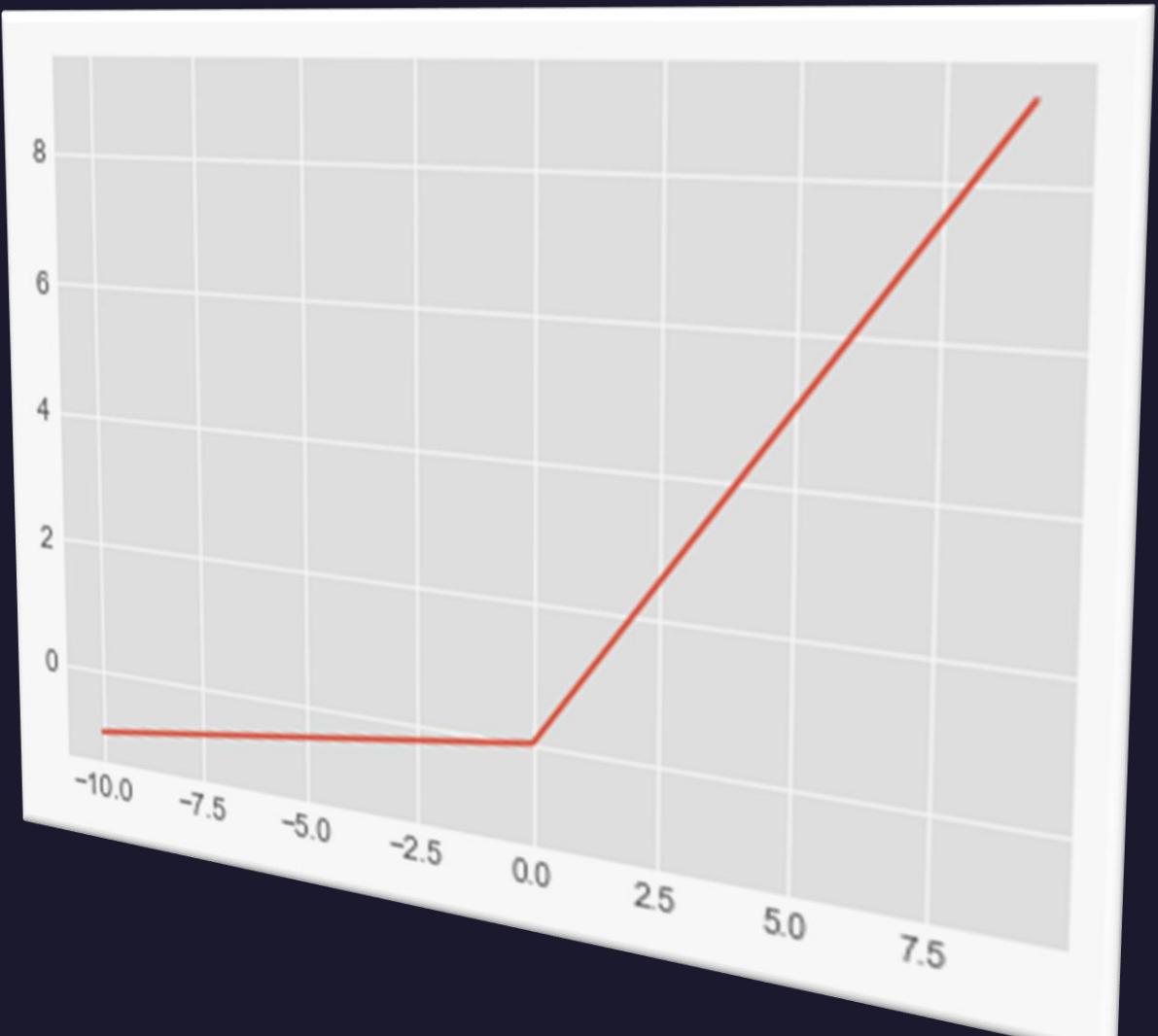
- It is **computationally efficient** and easy to implement.
- ReLU can induce **sparsity** in neural networks by setting negative values to zero, which can help **reduce overfitting**.
- Avoids vanishing gradient problem

Cons

- Not always differentiable: ReLU is **not differentiable at zero**, which can cause problems during backpropagation. (solved with leaky)
- ReLU can result in "dead" neurons, where the neuron always outputs zero, which can happen if the neuron's weights are set such that the input is always negative
- ReLU has an output range of $[0, \infty)$, which may not be suitable for all types of problems.

Leaky ReLU

```
def rectified_mod(x, w=0.1):  
    return max(w*x, x)  
inputs = [x for x in range(-10,10)]  
outputs = [rectified_mod(x) for x in inputs]  
plt.plot(inputs, outputs)  
plt.show()
```



Leaky ReLU

Pros

- Avoids dead neurons
- Computationally efficient

When to use it?

- Not commonly used although

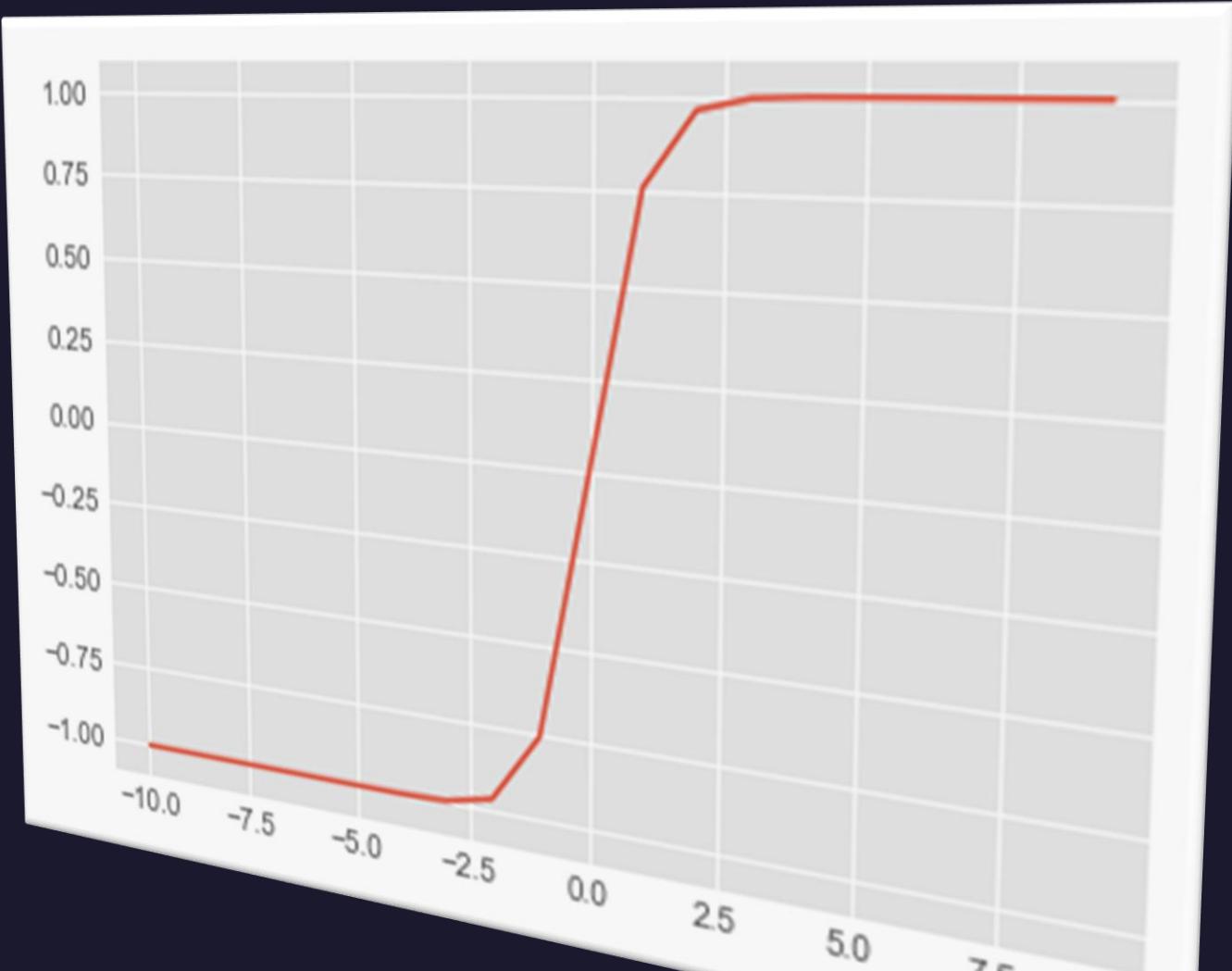
Allows a small, non-zero gradient when the input is negative, instead of setting the output to zero as ReLU does.

Cons

- Not always differentiable: Like ReLU, Leaky ReLU is not differentiable at zero, which can cause problems during backpropagation. However, this issue can be addressed by using a smoothed version of Leaky ReLU such as the exponential linear unit (ELU)

Tanh

```
def tanh(x):
    return(np.exp(x)-np.exp(-x))/(np.exp(x)+np.exp(-x))
inputs = [x for x in range(-10,10)]
outputs = [tanh(x) for x in inputs]
plt.plot(inputs, outputs)
#  $e^x = np.exp(x) = np.e^{**x}$ 
# tanh range from -1 to 1
```



Tanh

hyperbolic tangent

When to use it?

- Instead of logistic in the hidden layers more ranges from -1 to 1 and centered about he zero not 0.5

$$\tanh x = \frac{\sinh x}{\cosh x} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1}.$$

Pros

- Tanh is a **symmetric function** that **maps the input to the range (-1, 1)**. This can be useful in some cases where the output needs to be symmetric around zero.
- **Smooth gradient:** The gradient of the tanh function is smooth and continuous

Cons

- It suffers from the **vanishing gradient problem**.
- **Computationally expensive**
- **Saturation:** Tanh can become saturated when the input values are very large or very small, which can cause the gradients to become very small. This can make it difficult to train deep neural networks with many layers.

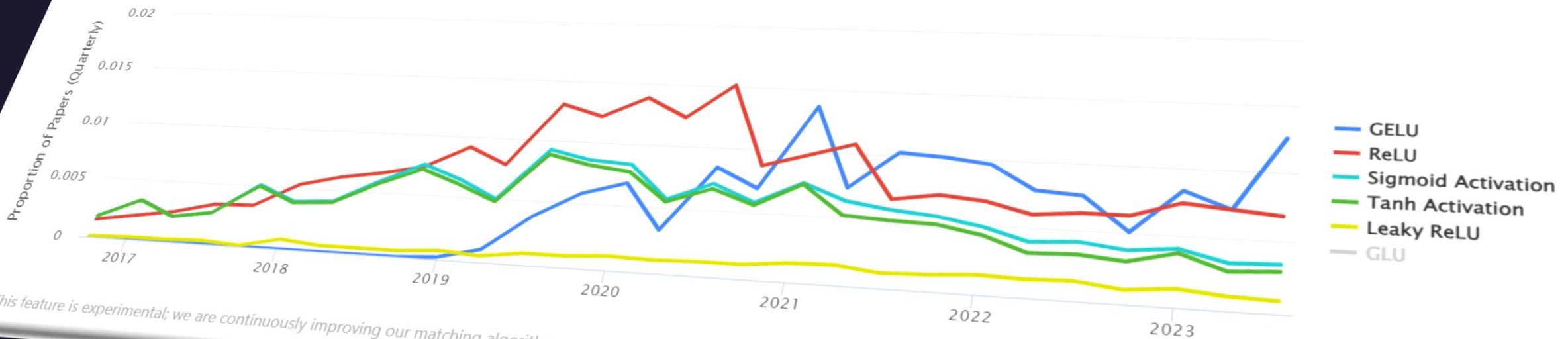
Tanh

hyperbolic tangent

$$\tanh x = \frac{\sinh x}{\cosh x} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1}.$$

- Saturation: Tanh can become saturated when the input values are very large or very small, which can cause the gradients to become very small. This can make it difficult to train deep neural networks with many layers.
- saturation refers to a situation where the output of an activation function becomes "stuck" at a certain value due to the input being very large or very small. When the output of an activation function is saturated, the derivative (or gradient) of the function becomes very small, which can make it difficult to train the network.
- **weight initialization** and **batch normalization** can be used to help prevent saturation and improve the performance of deep neural networks
- **batch normalization** :It is a form of normalization that is applied to the activations of the hidden layers of the network.

Usage Over Time



Example

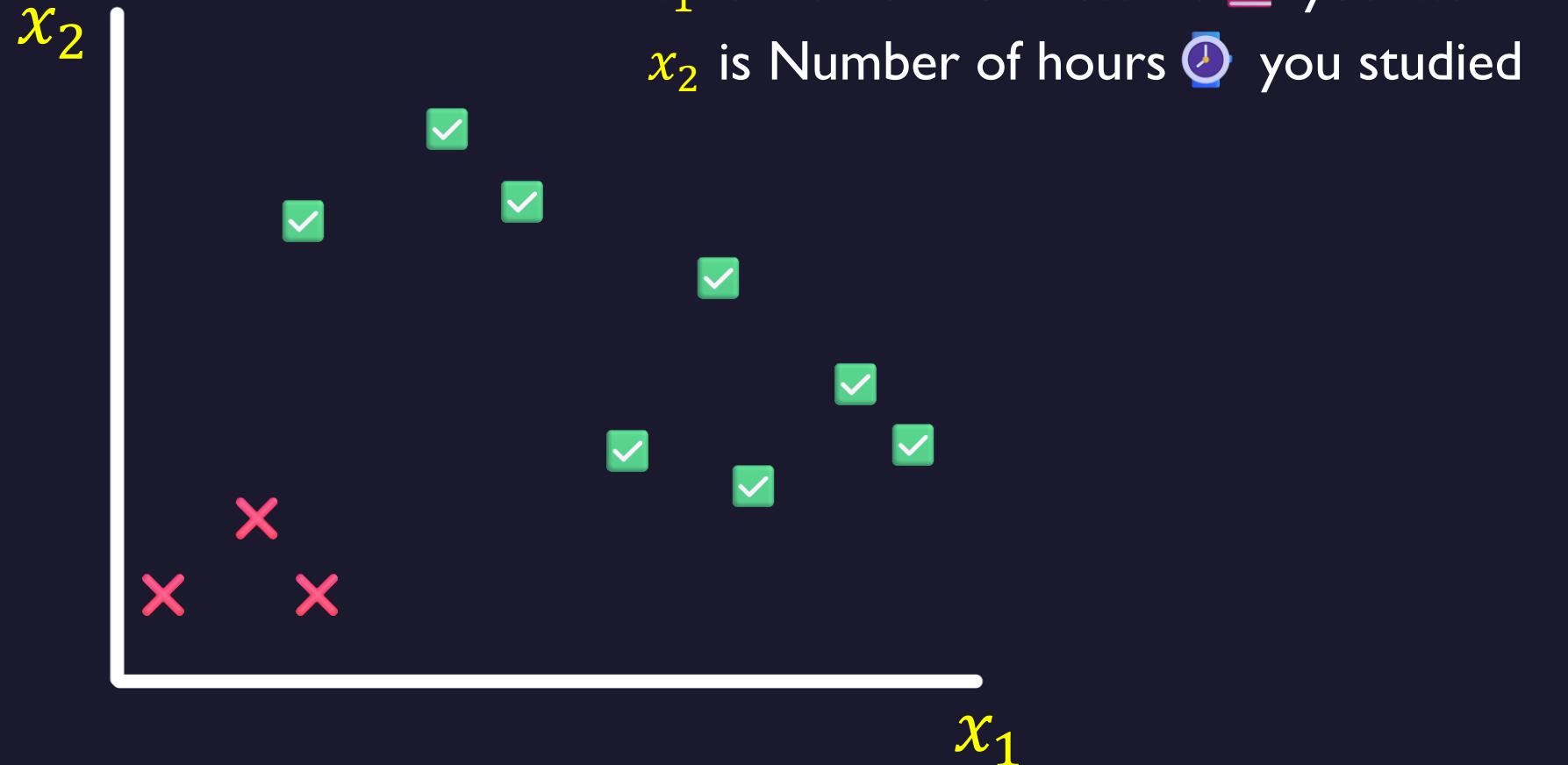
Will you pass an Exam ?

Factor 1 x_1 is Number of lectures you attend

Factor 2 x_2 is Number of hours you studied

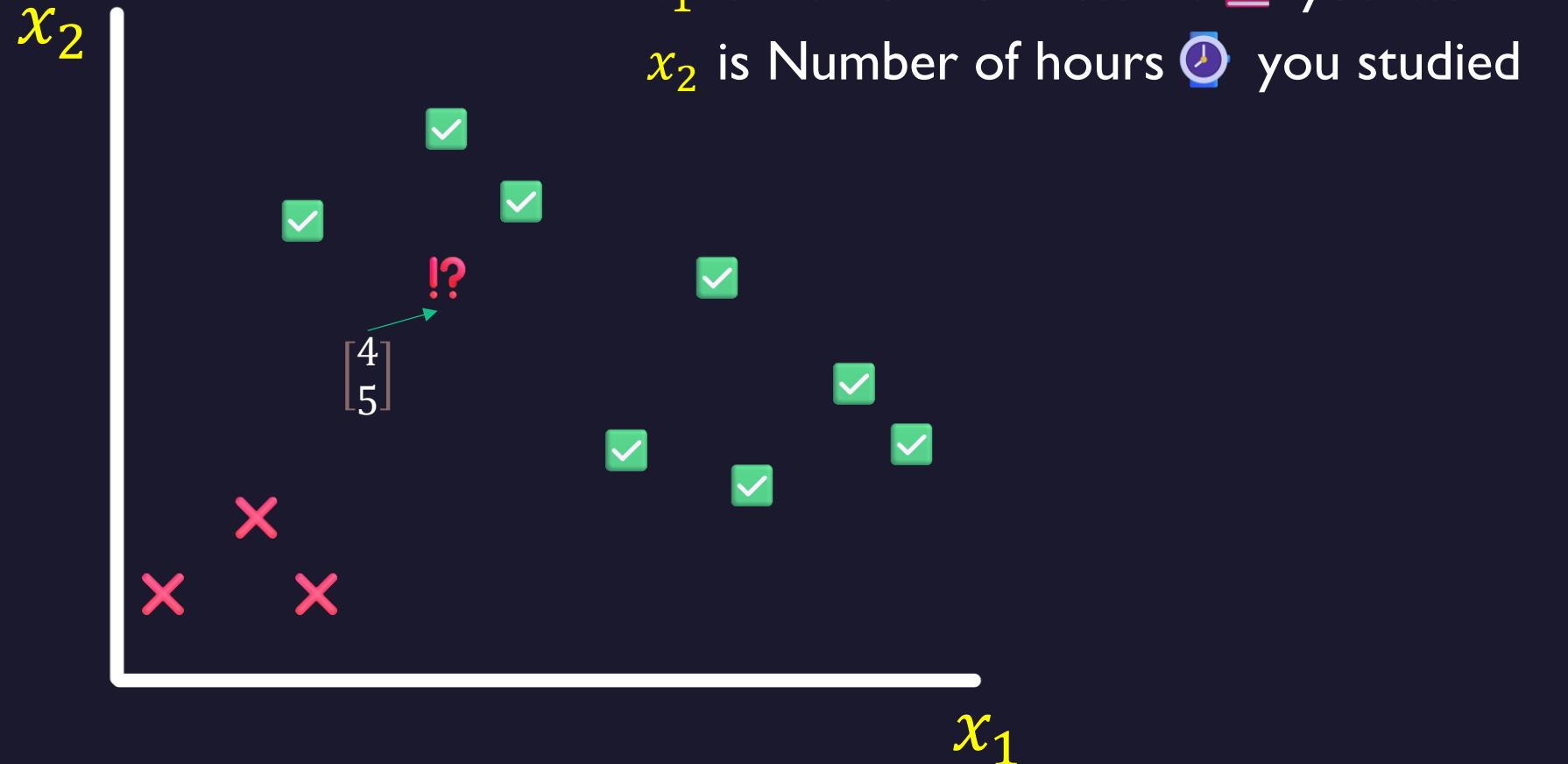
Example

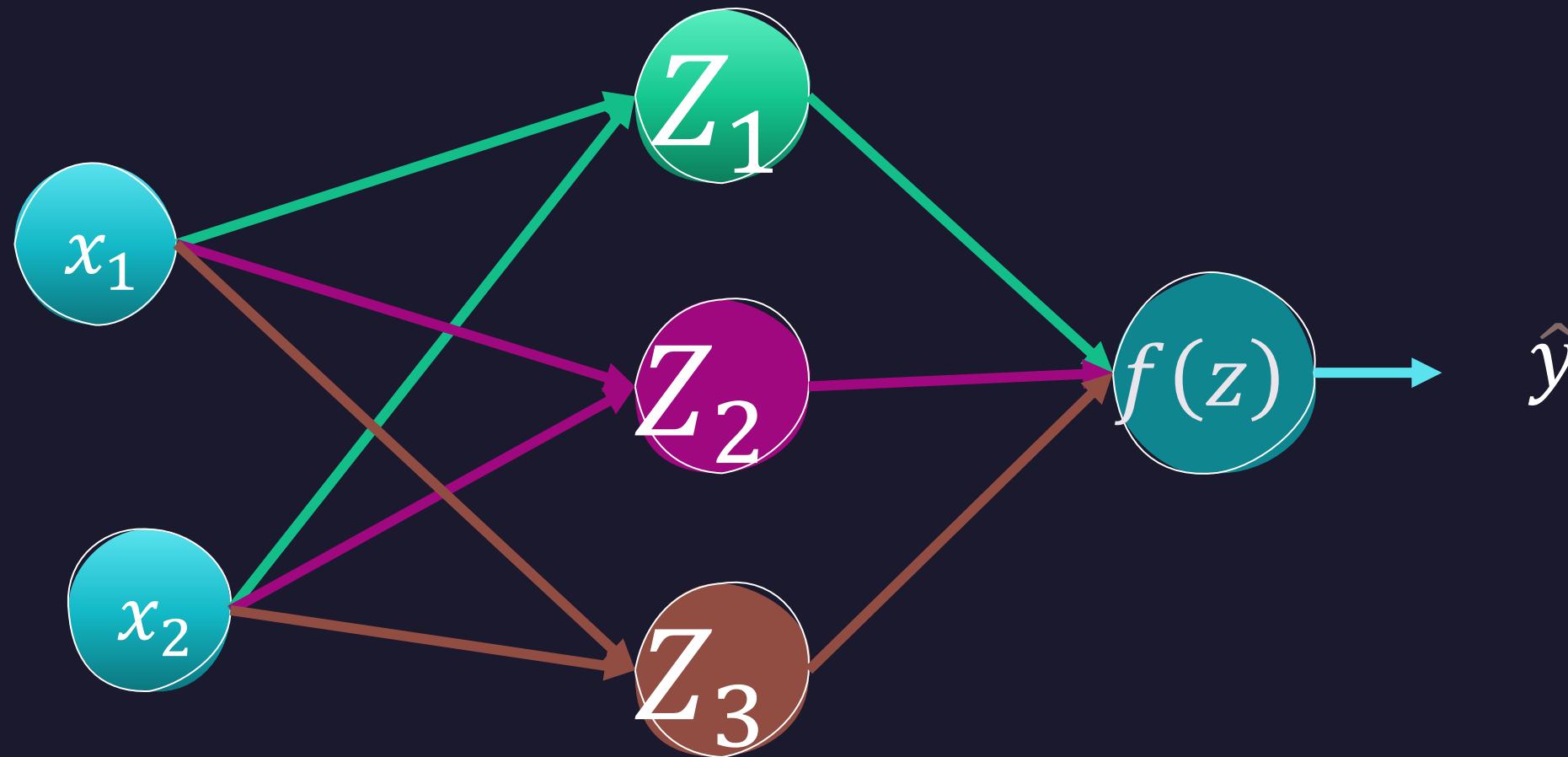
Will you pass an Exam ?



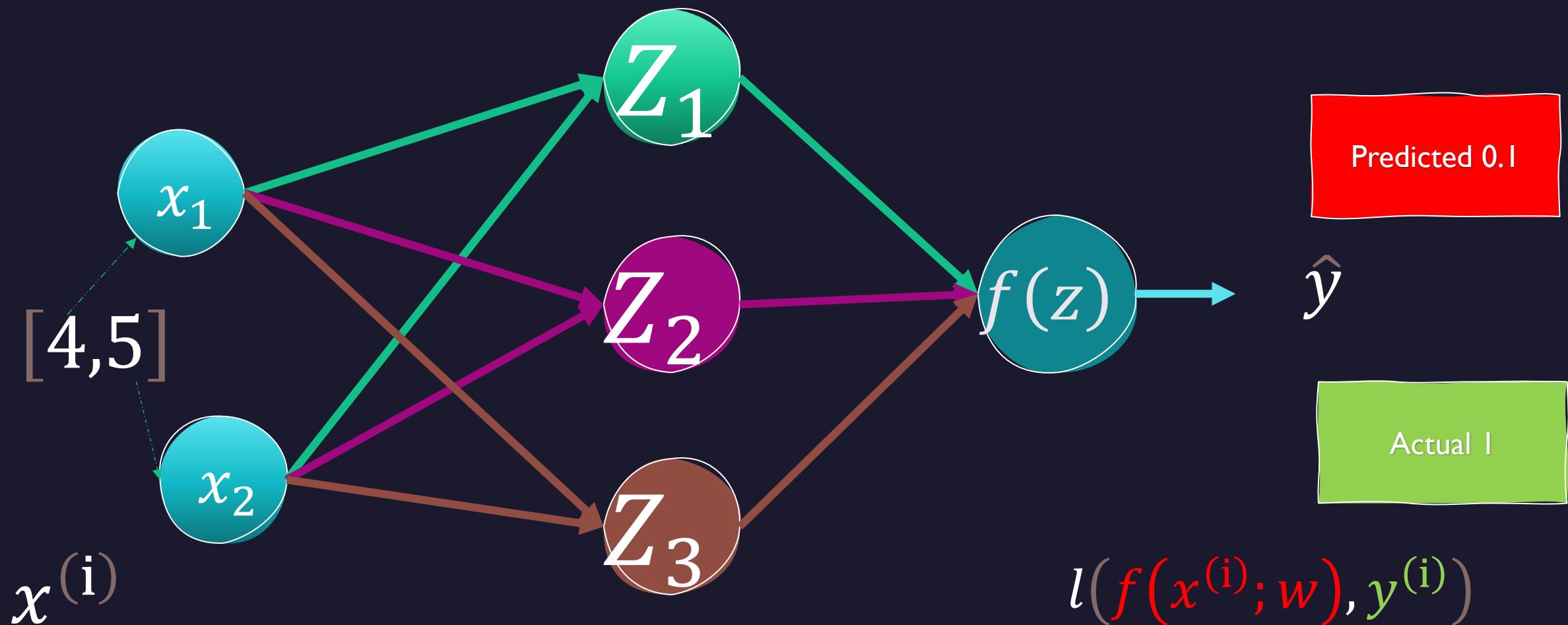
Example

Will you pass an Exam ?

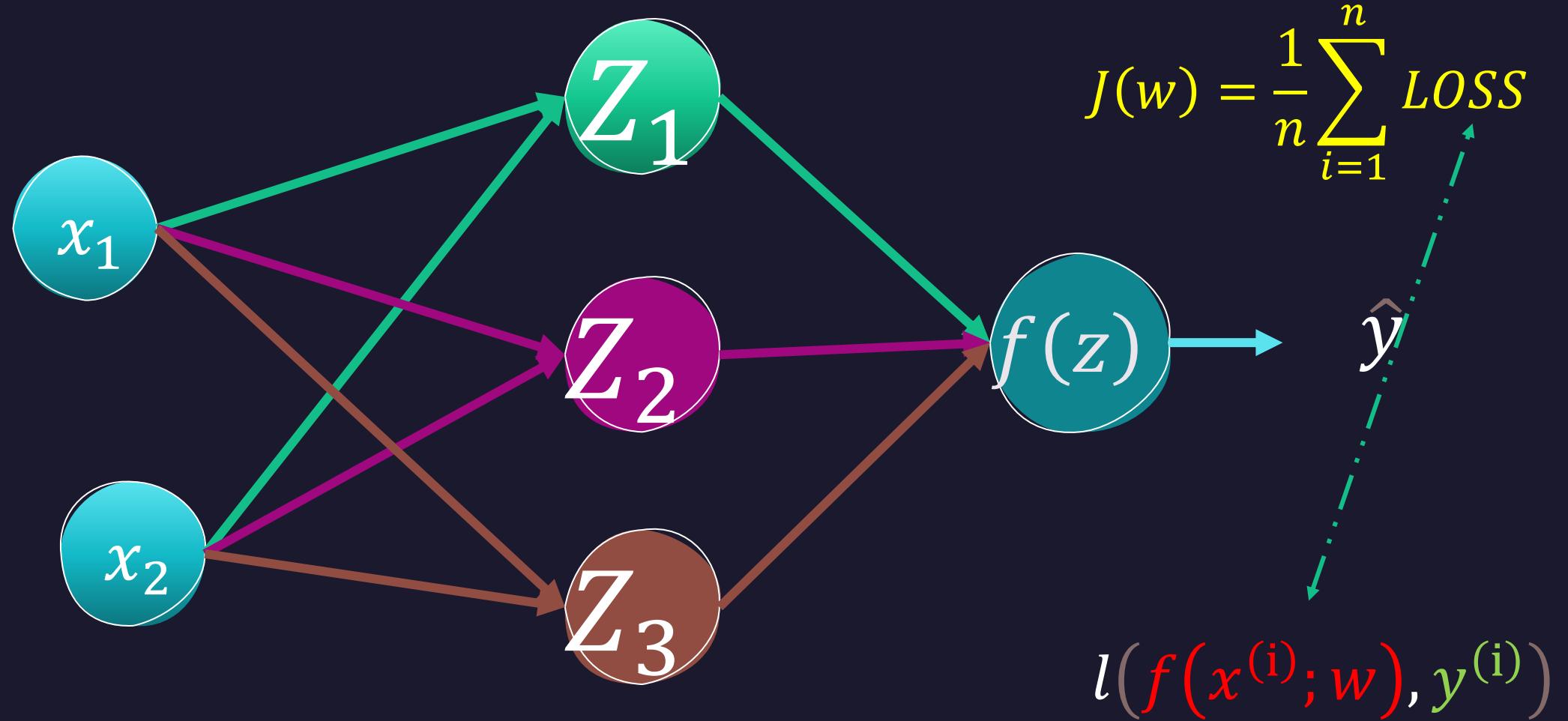




Loss



Cost :overall loss in the entire dataset



Cost :overall loss in the entire dataset

$$l(f(x^{(i)}; w), y^{(i)})$$

$$J(w) = \frac{1}{n} \sum_{i=1}^n LOSS$$

As usual our target is to minimize the error(cost) by finding best estimate of the parameters

Loss functions

- Loss functions for classification
 - Log loss (cross entropy)
- Loss function for Regression
 - MSE
 - MAE
 - RSS

Mean
Error
Squared

$$MSE = \frac{1}{n} \sum_{i=1}^n (Y_i - \hat{Y}_i)^2$$

$$-y_k \ln p_k - (1 - y_k) \ln(1 - p_k).$$

Divide by the total number of data points
 Predicted output value
 Actual output value
 Sum of
 The absolute value of the residual

$$MAE = \frac{1}{n} \sum |y - \hat{y}|$$

Forward propagation

- Forward propagation starts with an input vector, which is passed to the first layer of the neural network.
- Each neuron in the first layer receives the input vector and calculates a weighted sum of the inputs, adding a bias term to the sum.
- The weighted sum is then passed through an activation function, which introduces nonlinearity into the output of the neuron.
- The output of each neuron in the first layer becomes the input to the next layer in the network.
- The process of calculating the weighted sum and passing it through an activation function is repeated for each layer in the network until the final layer is reached.
- The output of the final layer is the predicted output of the neural network for the given input.
- During the forward propagation process, the weights and biases of the network are fixed, and no updates are made to them.
- The output of the network can be compared to the true output to calculate a loss function, which measures the error between the predicted output and the true output.
- The loss function is used during the backpropagation process to update the weights and biases of the network and improve its performance

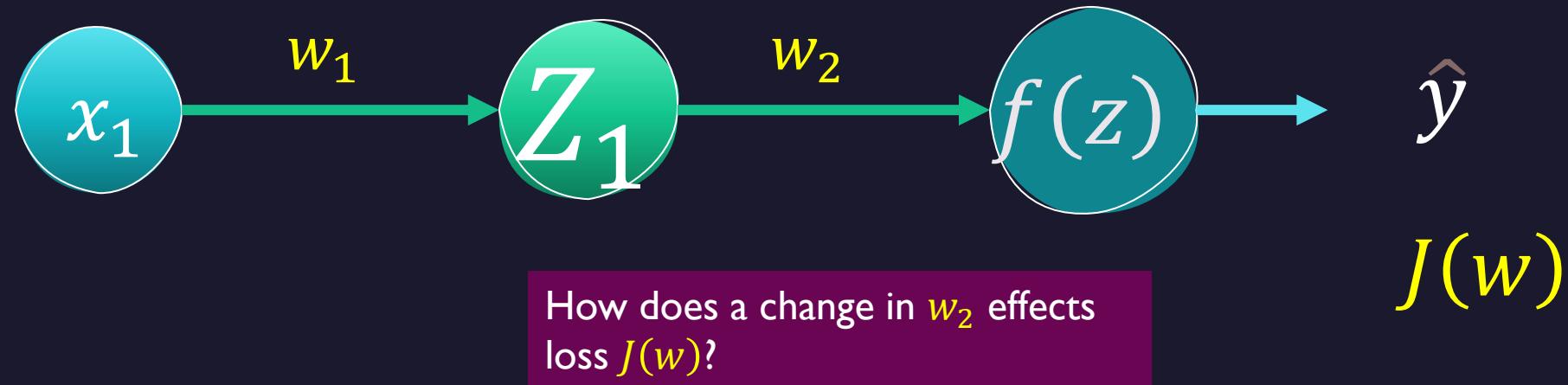
GD and backpropagation

GD

- Initialize weights randomly
- LOOP until convergence

- Compute gradient $\frac{\partial \text{Cost}}{\partial w} = \frac{\partial J(w)}{\partial w}$
- Update weights $W \leftarrow W - \eta \frac{\partial J(w)}{\partial w}$
- Return weights

GD and backpropagation



GD and backpropagation



How does a change in w_2 effects loss $J(w)$?

$J(w)$ is a function in terms of \hat{y}

\hat{y} is a function in terms of w_2

$$\frac{\partial J(w)}{\partial w_2} = \frac{\partial J(w)_*}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_2}$$

$$J(w)$$

GD and backpropagation



How does a change in w_2 effects loss $J(w)$?

$J(w)$ is a function in terms of \hat{y}

\hat{y} is a function in terms of w_2

z_1 is a function in terms of w_1

$$\frac{\partial J(w)}{\partial w_2} = \frac{\partial J(w)_*}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_2}$$

$$J(w)$$

GD and backpropagation



How does a change in w_1 effects loss $J(w)$?

$J(w)$ is a function in terms of \hat{y}

\hat{y} is a function in terms of w_2

z_1 is a function in terms of w_1

$$\frac{\partial J(w)}{\partial w_1} = \frac{\partial J(w)_*}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial w_1}$$

$$J(w)$$

GD and backpropagation



How does a change in w_1 effects loss $J(w)$?

$J(w)$ is a function in terms of \hat{y}

\hat{y} is a function in terms of w_2

Z_1 is a function in terms of w_1

$$\frac{\partial J(w)}{\partial w_1} = \frac{\partial J(w)_*}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial Z_1} \frac{\partial Z_1}{\partial w_1}$$

$$J(w)$$

Optimization

- The idea is not to make the learning rate fixed
- This is called Adaptive Learning Rates
- Examples
 - SGD
 - Adam  use this
 - Adadelta
 - Adagrad
 - RMSProp
- We would use [this](#) to demo how GD works

See

- <https://datascience.stackexchange.com/questions/12851/how-do-you-visualize-neural-network-architectures>
- <https://mathworld.wolfram.com/HyperbolicTangent.html>
- https://github.com/lilipads/gradient_descent_viz ✨
- <https://blog.paperspace.com/intro-to-optimization-in-deep-learning-gradient-descent/>
- <https://xnought.github.io/backprop-explainer/> ✨
- <https://hmkcode.com/ai/backpropagation-step-by-step/>
- <https://mlu-explain.github.io/neural-networks/> ✨



References

- <https://paperswithcode.com/method/gelu> the graph of activation function usage over time
- https://en.wikipedia.org/wiki/Hyperbolic_functions
- <https://arxiv.org/pdf/1502.03167.pdf>
- <http://introtodeeplearning.com/>

The background of the image features a series of vertical, glowing lines that curve downwards from left to right. These lines are colored in a gradient, transitioning from orange on the far left to blue on the far right. The background is a dark, textured surface that appears to be a polished floor or wall.

Thank You