Multi-Layer Perceptron (MLP)

AI with Deep Learning
EE4016

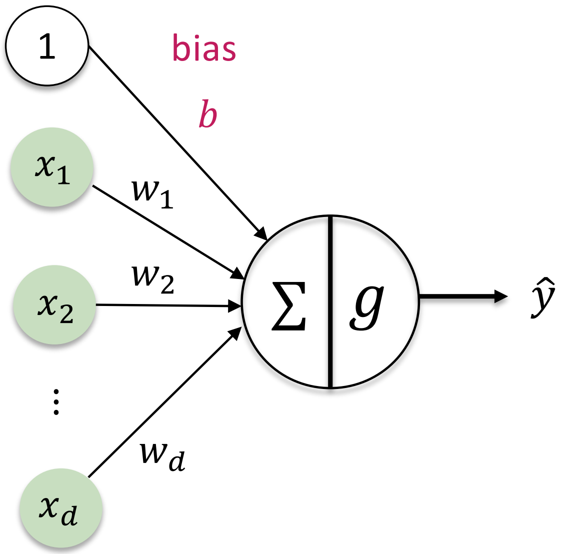
Prof. Lai-Man Po
Department of Electrical Engineering
City University of Hong Kong

Content

- **The XOR Problem of Perceptron**
  - Solving XOR problem by Two-Layer Neural Networks
- **Multi-Layer Perceptron (MLP) Architectures**
  - Matrix Representation of MLPs
  - Activation Functions: Identify (Linear), Binary Step, Sigmoid, Tanh, ReLU, Leaky ReLU, and Softmax
  - Why Deeper Neural Networks is Better?
- **Loss, Cost and Objective Functions**
  - Square loss, Absolute loss, MSE, MAE
  - Binary Cross-Entropy Loss and Categorial Cross-Entropy Loss
- **Gradient Descent**
- **Colab Examples**: MPG Regression and IRIS Flower Classification

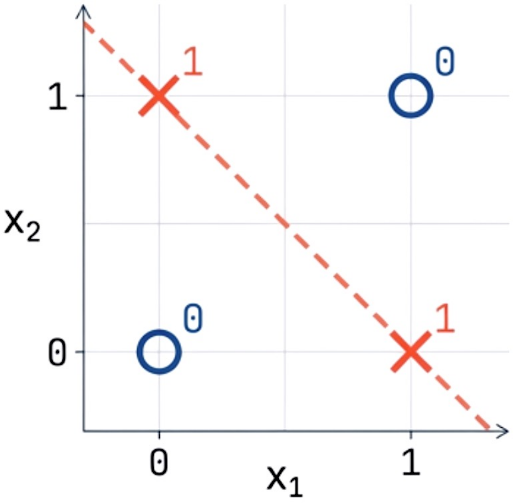
Recap: The Linearity Limit

The Precursor: Single-Layer Perceptron



Limitation: Can only model linear decision boundaries.

The XOR Problem: Linearly Inseparable



XOR

(x_1, x_2)	(z_1, z_2)	(a_1, a_2)	$\hat{y}$
(0, 0)	(-1, 1)	(0, 1)	0
(0, 1)	(0, 0)	(1, 1)	1
(1, 0)	(0, 0)	(1, 1)	1
(1, 1)	(1, -1)	(1, 0)	0

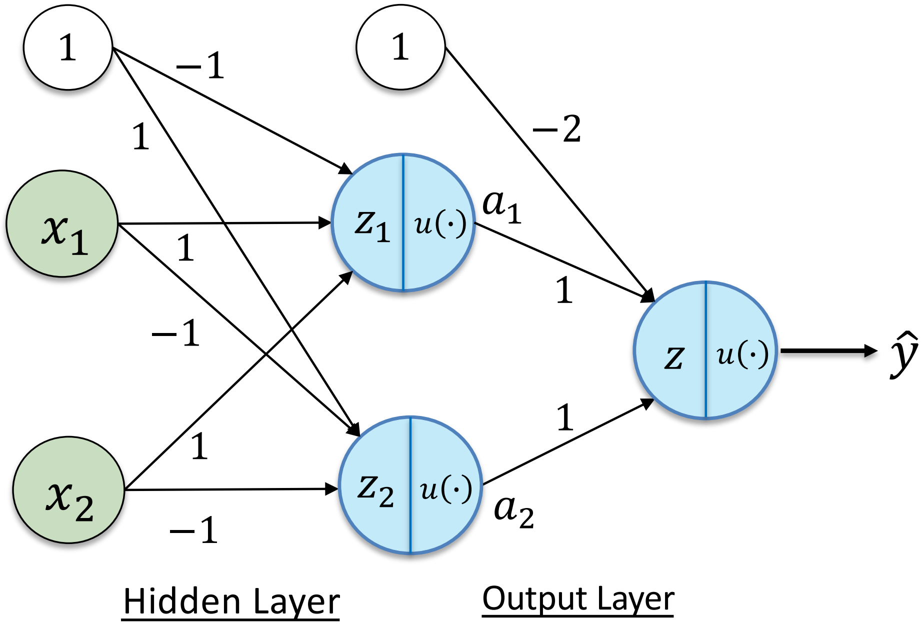
Activation function is the unit step function u(z)

$$g(z) = u(z) = \begin{cases} 1, & z \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

Key Insight: No single straight line can separate the true values from the false values in an XOR function. To solve this, the decision boundary must be bent.

Solving XOR Using a Two-Layer Neural Network

- A two-layer neural network introduces **a hidden layer** with multiple neurons, which allows the network to create **nonlinear** decision boundaries.
- These **hidden layer neurons can create intermediate representations** that enable the XOR function to be modelled.



(x_1, x_2)	(z_1, z_2)	(a_1, a_2)	$\hat{y}$ **XOR**
$(0, 0)$	$(-1, 1)$	$(0, 1)$	0
$(0, 1)$	$(0, 0)$	$(1, 1)$	1
$(1, 0)$	$(0, 0)$	$(1, 1)$	1
$(1, 1)$	$(1, -1)$	$(1\ , 0)$	0

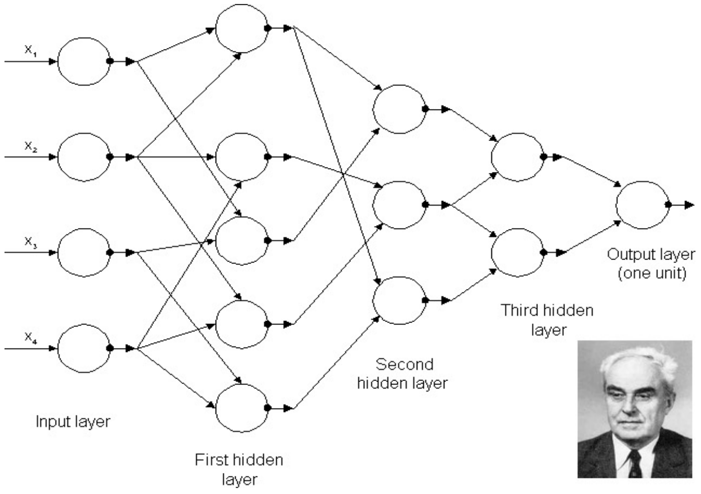
Activation function is the unit step function $u(z)$

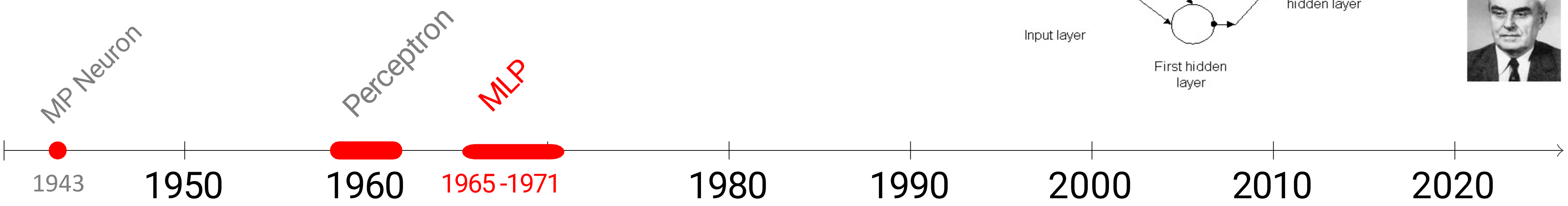
$$g(z) = u(z) = \begin{cases} 1, & z \geq 0 \\ 0, & \text{otherwise} \end{cases}$$

Multilayer Perceptron (MLP)
or
Feedforward Network (FFN)

Multi-Layer Perception (1971)

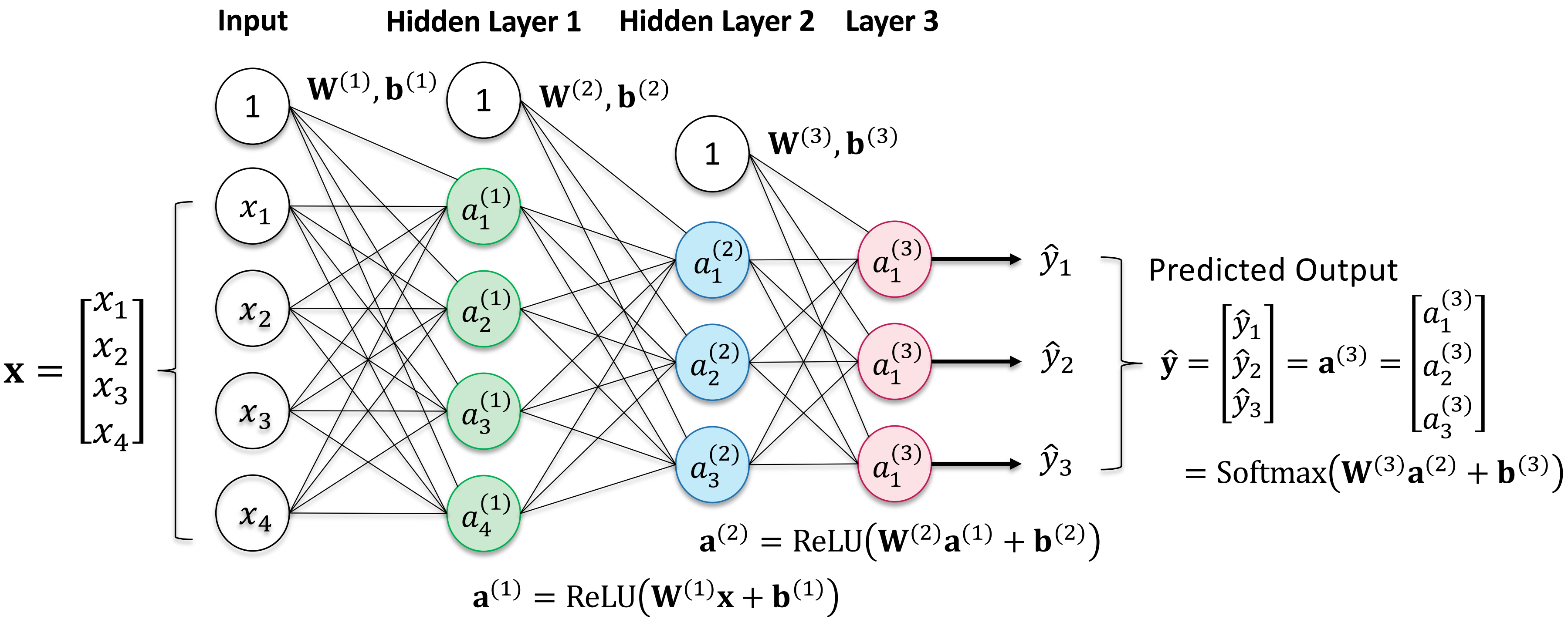
- The XOR problem revealed the necessity of **multilayer neural networks**, which is also known as **Multi-Layer Perceptron (MLP).**
- MLPs contain one or more hidden layers between the input and output layers that enable modeling of **nonlinear functions**.
- The first generation of MLPs was introduced by **A. G. Ivakhnenko** and **V. Lapa**.
  - They published the first general, working learning algorithm for supervised deep feedforward MLPs in 1971.





Feedforward Networks (FFNs)

- Today the term ”**Feedforward Network**" is more commonly used in deep learning, referring to **Multi-Layer Perceptron (MLP)**.
- In addition, the layer of FFN is also referred as **a Fully Connected (FC) Layer.**



$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

$$\hat{\mathbf{y}} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \hat{y}_3 \end{bmatrix} = \mathbf{a}^{(3)} = \begin{bmatrix} a_1^{(3)} \\ a_2^{(3)} \\ a_3^{(3)} \end{bmatrix}$$

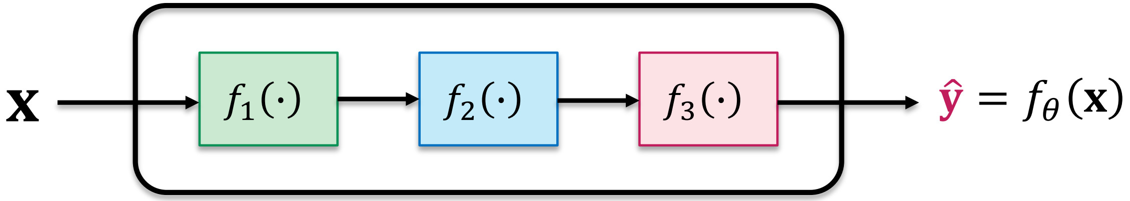
$$= \mathrm{Softmax}\big(\mathbf{W}^{(3)}\mathbf{a}^{(2)} + \mathbf{b}^{(3)}\big)$$

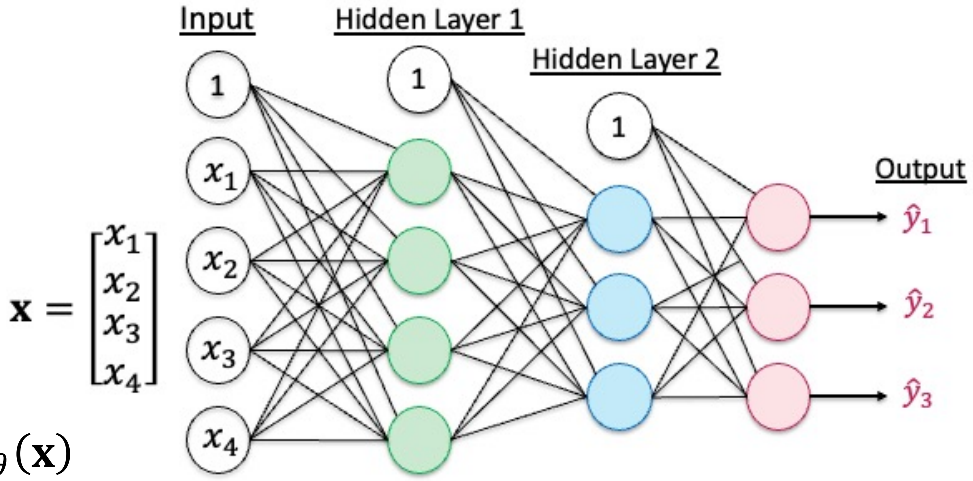
$$\mathbf{a}^{(2)} = \mathrm{ReLU}\big(\mathbf{W}^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)}\big)$$

$$\mathbf{a}^{(1)} = \mathrm{ReLU}\big(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\big)$$

Feedforward Networks (FFNs)

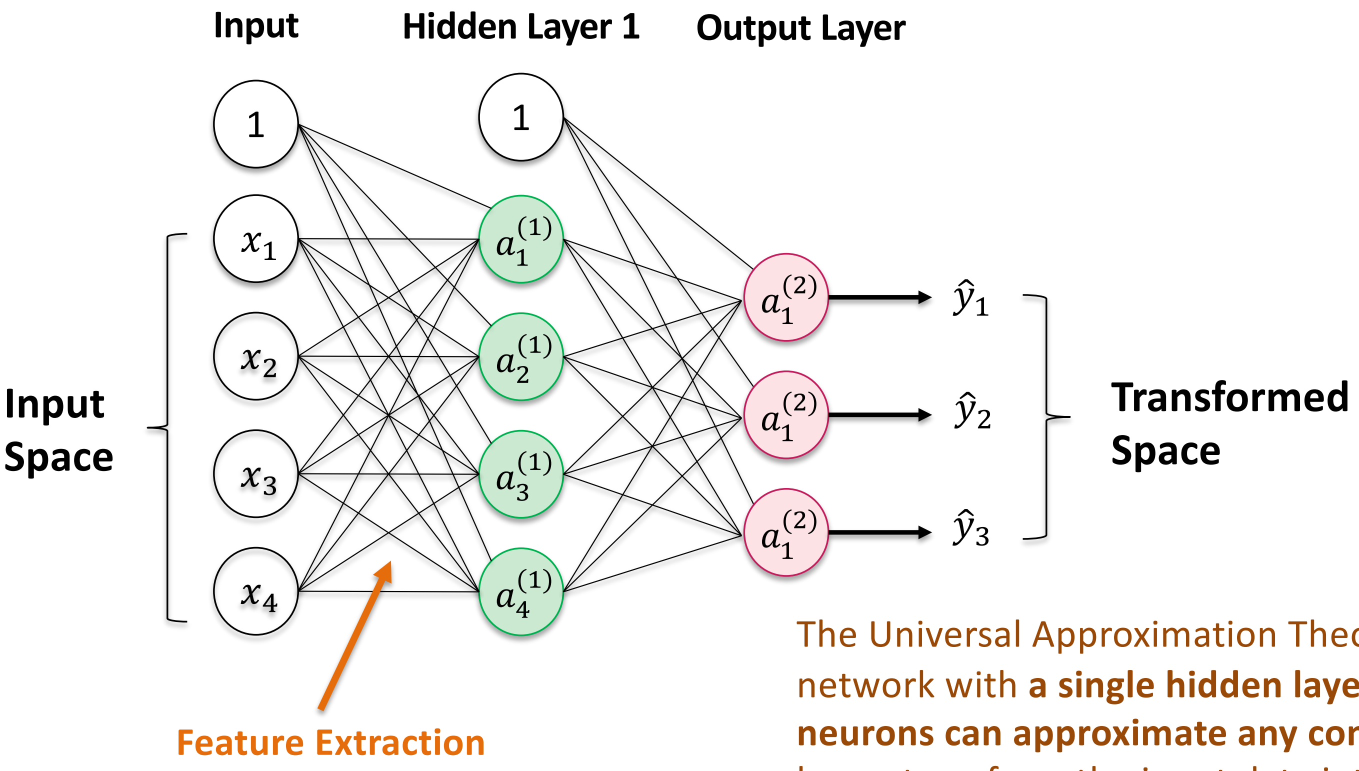
- The number of layers in the FFNs (excluding the input layer) is known as depth
- **Each layer** can be seen as a **vector-to-vector function** which takes a vector of inputs from the previous layer and computes a scalar value.
- Below network can be seen as a composition of functions
- $\hat{\mathbf{y}} = f_\theta(\mathbf{x}) = f_3\left(f_2\left(f_1(\mathbf{x})\right)\right)$
  - f_1 being the first hidden layer,
  - f_2 being the second hidden layer,
  - f_3 being the final output layer.





The Universal Approximation Theorem

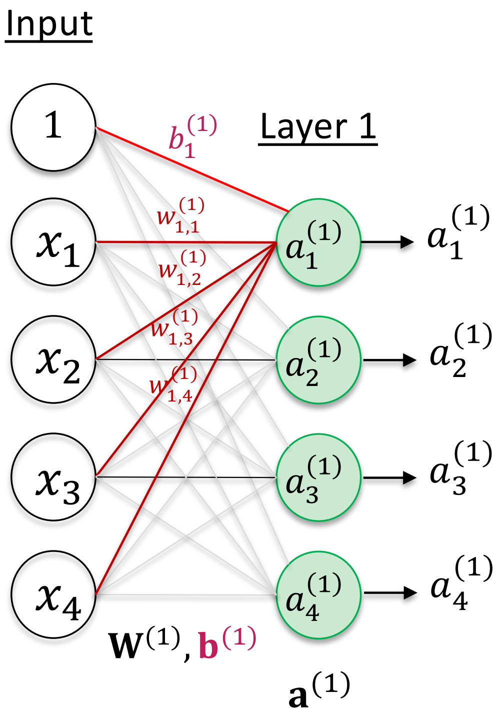
SOLVING NON-LINEARITY WITH HIDDEN LAYERS



The Universal Approximation Theorem states that a feedforward network with **a single hidden layer containing a finite number of neurons can approximate any continuous function**. The hidden layers transform the input data into a higher-dimensional space where patterns become linearly separable.

Matrix Representation of FFN

- **Net input** of the **Hidden Layer 1**: $\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$



$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \quad \mathbf{W}^{(1)} = \begin{bmatrix} w_{1,1}^{(1)} & w_{1,2}^{(1)} & w_{1,3}^{(1)} & w_{1,4}^{(1)} \\ w_{2,1}^{(1)} & w_{2,2}^{(1)} & w_{2,3}^{(1)} & w_{2,4}^{(1)} \\ w_{3,1}^{(1)} & w_{3,2}^{(1)} & w_{3,3}^{(1)} & w_{3,4}^{(1)} \\ w_{4,1}^{(1)} & w_{4,2}^{(1)} & w_{4,3}^{(1)} & w_{4,4}^{(1)} \end{bmatrix} \quad \mathbf{b}^{(1)} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \\ b_3^{(1)} \\ b_4^{(1)} \end{bmatrix}$$

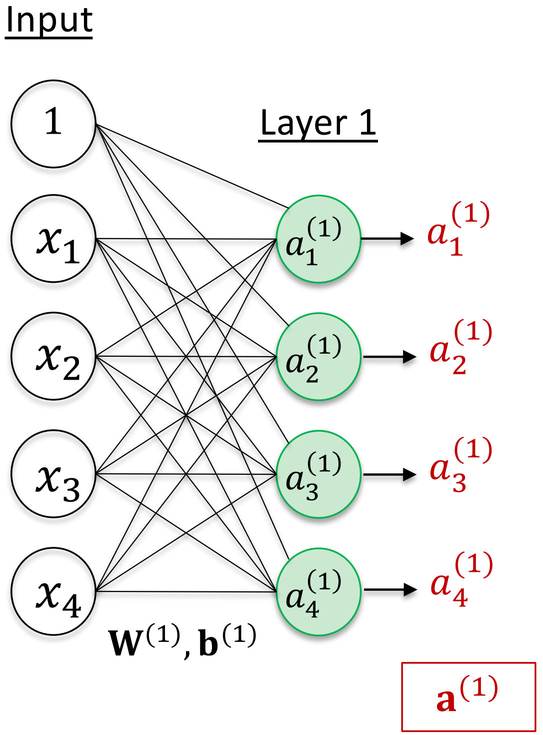
$$\mathbf{z}^{(1)} = \begin{bmatrix} z_1^{(1)} \\ z_2^{(1)} \\ z_3^{(1)} \\ z_4^{(1)} \end{bmatrix} = \begin{bmatrix} w_{1,1}^{(1)} & w_{1,2}^{(1)} & w_{1,3}^{(1)} & w_{1,4}^{(1)} \\ w_{2,1}^{(1)} & w_{2,2}^{(1)} & w_{2,3}^{(1)} & w_{2,4}^{(1)} \\ w_{3,1}^{(1)} & w_{3,2}^{(1)} & w_{3,3}^{(1)} & w_{3,4}^{(1)} \\ w_{4,1}^{(1)} & w_{4,2}^{(1)} & w_{4,3}^{(1)} & w_{4,4}^{(1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \\ b_3^{(1)} \\ b_4^{(1)} \end{bmatrix}$$

$$z_1^{(1)} = w_{1,1}^{(1)} x_1 + w_{1,2}^{(1)} x_2 + w_{1,3}^{(1)} x_3 + w_{1,4}^{(1)} x_4 + b_1^{(1)}$$

$\mathbf{z}^{(1)} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$ is called Net Input (Inputs of the activation function)

Matrix Representation of FFN

- **Activations** of the **Hidden Layer 1** : $\mathbf{a}^{(1)} = g^{(1)}\left(\mathbf{z}^{(1)}\right) = g^{(1)}\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right)$



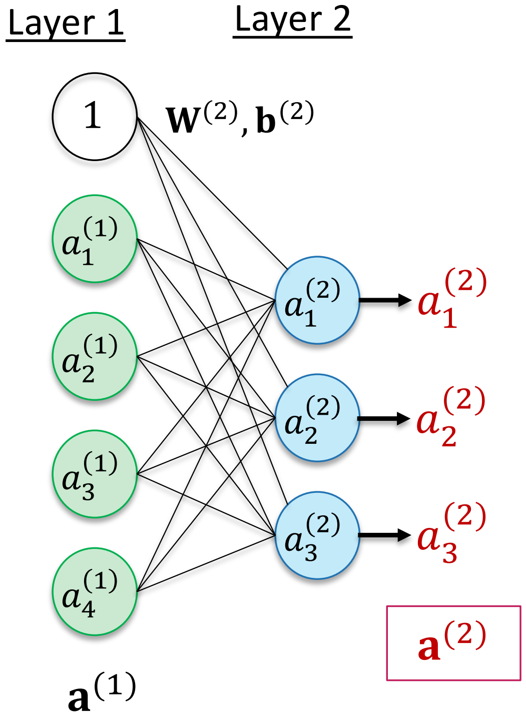
$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} \quad \mathbf{W}^{(1)} = \begin{bmatrix} w_{1,1}^{(1)} & w_{1,2}^{(1)} & w_{1,3}^{(1)} & w_{1,4}^{(1)} \\ w_{2,1}^{(1)} & w_{2,2}^{(1)} & w_{2,3}^{(1)} & w_{2,4}^{(1)} \\ w_{3,1}^{(1)} & w_{3,2}^{(1)} & w_{3,3}^{(1)} & w_{3,4}^{(1)} \\ w_{4,1}^{(1)} & w_{4,2}^{(1)} & w_{4,3}^{(1)} & w_{4,4}^{(1)} \end{bmatrix} \quad \mathbf{b}^{(1)} = \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \\ b_3^{(1)} \\ b_4^{(1)} \end{bmatrix} \quad \mathbf{z}^{(1)} = \begin{bmatrix} z_1^{(1)} \\ z_2^{(1)} \\ z_3^{(1)} \\ z_4^{(1)} \end{bmatrix}$$

$$\mathbf{a}^{(1)} = \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \\ a_3^{(1)} \\ a_4^{(1)} \end{bmatrix} = g^{(1)}\left(\mathbf{z}^{(1)}\right) = g^{(1)}\left(\begin{bmatrix} w_{1,1}^{(1)} & w_{1,2}^{(1)} & w_{1,3}^{(1)} & w_{1,4}^{(1)} \\ w_{2,1}^{(1)} & w_{2,2}^{(1)} & w_{2,3}^{(1)} & w_{2,4}^{(1)} \\ w_{3,1}^{(1)} & w_{3,2}^{(1)} & w_{3,3}^{(1)} & w_{3,4}^{(1)} \\ w_{4,1}^{(1)} & w_{4,2}^{(1)} & w_{4,3}^{(1)} & w_{4,4}^{(1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} + \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \\ b_3^{(1)} \\ b_4^{(1)} \end{bmatrix}\right)$$

Activation function of layer 1

Matrix Representation of FFN

Activations of the **Layer 2**: $\mathbf{a}^{(2)} = g^{(2)}\left(\mathbf{W}^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)}\right) = g^{(2)}\left(\mathbf{W}^{(2)}g^{(1)}\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right) + \mathbf{b}^{(2)}\right)$

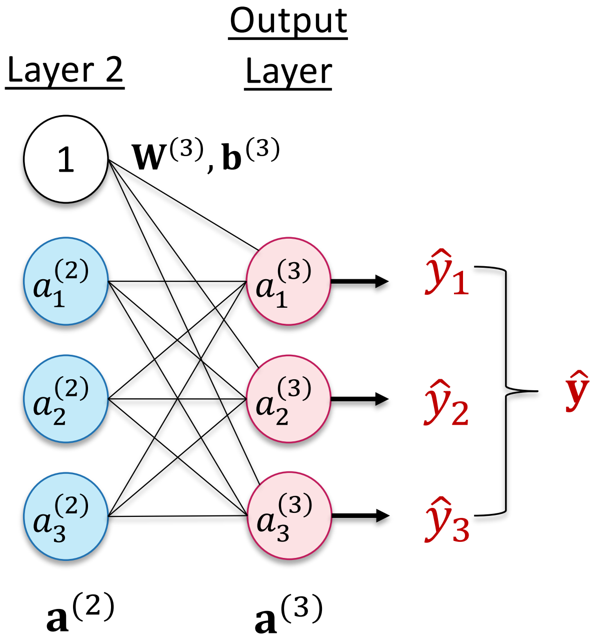


$$\mathbf{a}^{(1)} = \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \\ a_3^{(1)} \\ a_4^{(1)} \end{bmatrix} \qquad \mathbf{W}^{(2)} = \begin{bmatrix} w_{1,1}^{(2)} & w_{1,2}^{(2)} & w_{1,3}^{(2)} & w_{1,4}^{(2)} \\ w_{2,1}^{(2)} & w_{2,2}^{(2)} & w_{2,3}^{(2)} & w_{2,4}^{(2)} \\ w_{3,1}^{(2)} & w_{3,2}^{(2)} & w_{3,3}^{(2)} & w_{3,4}^{(2)} \end{bmatrix} \qquad \mathbf{b}^{(2)} = \begin{bmatrix} b_1^{(2)} \\ b_2^{(2)} \\ b_3^{(2)} \end{bmatrix}$$

$$\mathbf{a}^{(2)} = \begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \\ a_3^{(2)} \end{bmatrix} = g^{(2)}\left(\begin{bmatrix} w_{1,1}^{(2)} & w_{1,2}^{(2)} & w_{1,3}^{(2)} & w_{1,4}^{(2)} \\ w_{2,1}^{(2)} & w_{2,2}^{(2)} & w_{2,3}^{(2)} & w_{2,4}^{(2)} \\ w_{3,1}^{(2)} & w_{3,2}^{(2)} & w_{3,3}^{(2)} & w_{3,4}^{(2)} \end{bmatrix} \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \\ a_3^{(1)} \\ a_4^{(1)} \end{bmatrix} + \begin{bmatrix} b_1^{(2)} \\ b_2^{(2)} \\ b_3^{(2)} \end{bmatrix} \right)$$

Matrix Representation of FFN

- **Activations** of the **Output Layer**: $\hat{\mathbf{y}} = \mathbf{a}^{(3)} = g^{(3)}\left(\mathbf{W}^{(3)}\mathbf{a}^{(2)} + \mathbf{b}^{(3)}\right)$



$$\mathbf{a}^{(2)} = \begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \\ a_3^{(2)} \end{bmatrix} \qquad \mathbf{W}^{(3)} = \begin{bmatrix} w_{1,1}^{(3)} & w_{1,2}^{(3)} & w_{1,3}^{(3)} \\ w_{2,1}^{(3)} & w_{2,2}^{(3)} & w_{2,3}^{(3)} \\ w_{3,1}^{(3)} & w_{3,2}^{(3)} & w_{3,3}^{(3)} \end{bmatrix} \qquad \mathbf{b}^{(3)} = \begin{bmatrix} b_1^{(3)} \\ b_2^{(3)} \\ b_3^{(3)} \end{bmatrix}$$

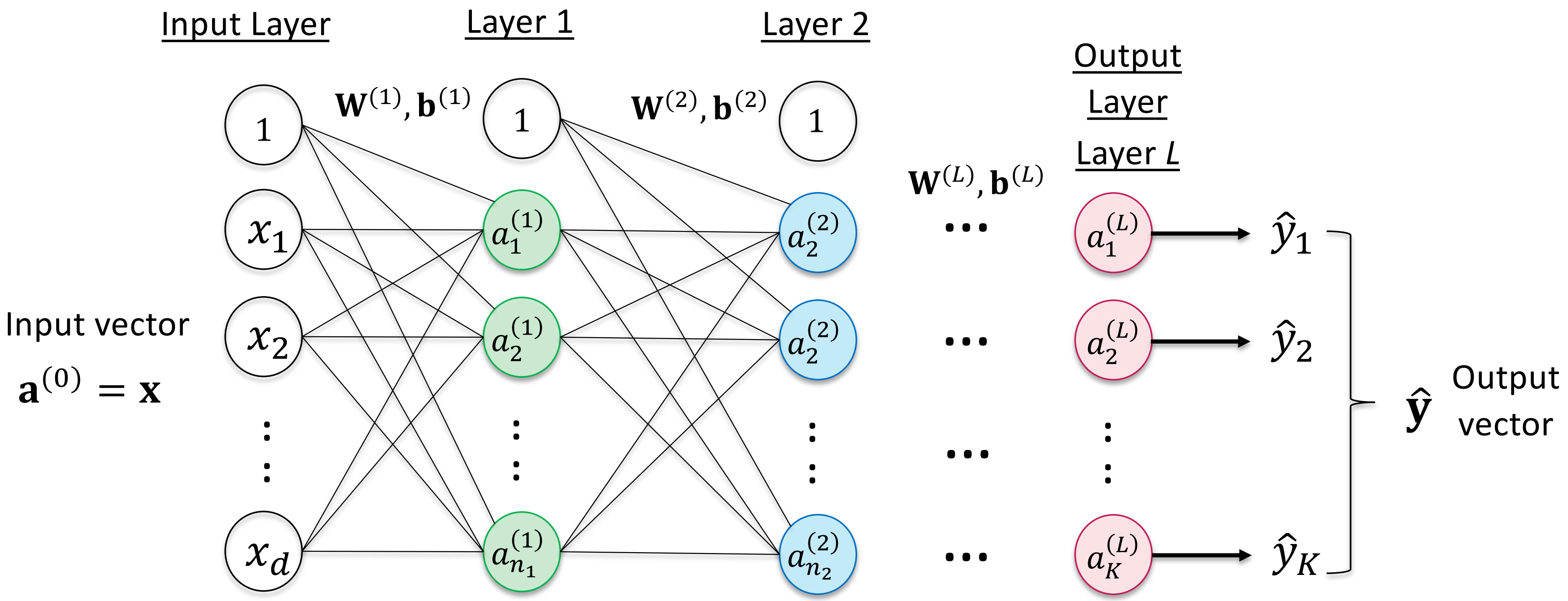
$$\hat{\mathbf{y}} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \hat{y}_2 \end{bmatrix} = \begin{bmatrix} a_1^{(3)} \\ a_2^{(3)} \\ a_3^{(3)} \end{bmatrix} = g^{(3)}\left(\begin{bmatrix} w_{1,1}^{(3)} & w_{1,2}^{(3)} & w_{1,3}^{(3)} \\ w_{2,1}^{(3)} & w_{2,2}^{(3)} & w_{2,3}^{(3)} \\ w_{3,1}^{(3)} & w_{3,2}^{(3)} & w_{3,3}^{(3)} \end{bmatrix} \begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \\ a_3^{(2)} \end{bmatrix} + \begin{bmatrix} b_1^{(3)} \\ b_2^{(3)} \\ b_3^{(3)} \end{bmatrix} \right)$$

$$\hat{\mathbf{y}} = \mathbf{a}^{(3)} = g^{(3)}\left(\mathbf{W}^{(3)}\mathbf{h}^{(2)} + \mathbf{b}^{(3)}\right)$$

$$\hat{\mathbf{y}} = g^{(3)}\left(\mathbf{W}^{(3)} g^{(2)}\left(\mathbf{W}^{2} g^{(1)}\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right) + \mathbf{b}^{(2)}\right) + \mathbf{b}^{(3)}\right)$$

Feedforward neural network is just a function of input vector $\mathbf{x}$.

Feedforward Neural Network Formulation

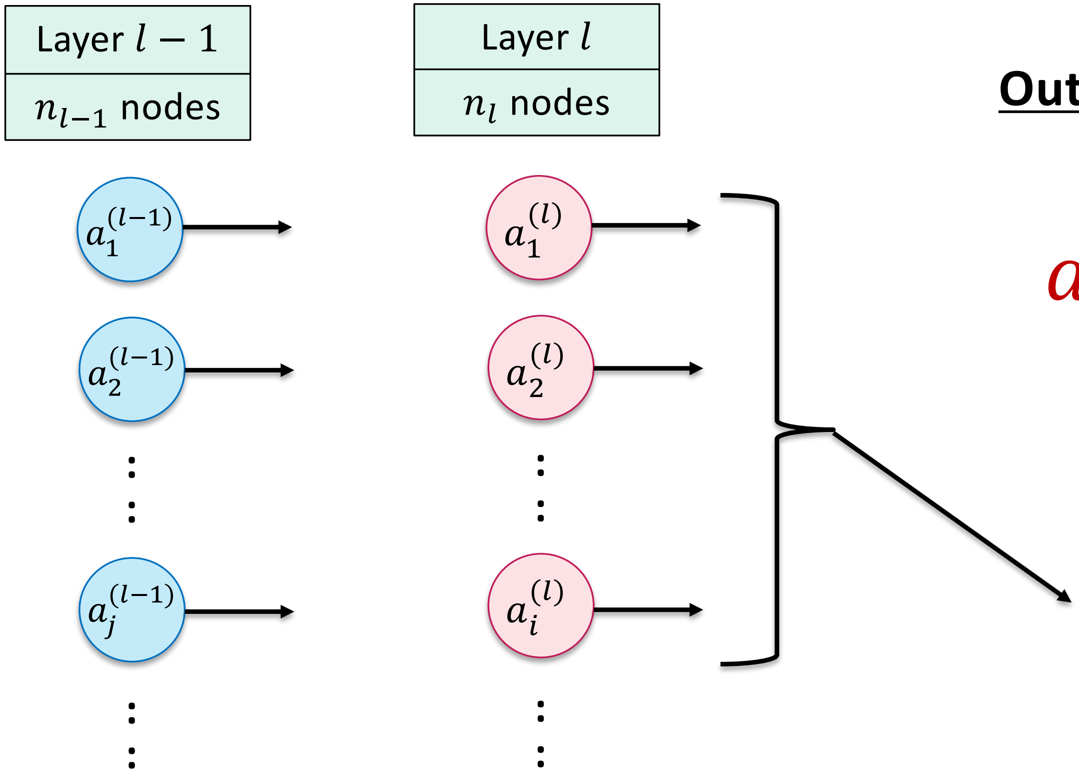


$\mathbf{a}^{(1)}$ $= g^{(1)}\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right)$	$\mathbf{a}^{(2)}$ $= g^{(2)}\left(\mathbf{W}^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)}\right)$

$$\boxed{\mathbf{a}^{(l)} = g^{(l)}\left(\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}\right)}$$

$$\hat{\mathbf{y}} = f_\theta(\mathbf{x}) = g^{(L)}\left(\mathbf{W}^{(L)} \cdots g^{(2)}\left(\mathbf{W}^{(2)} g^{(1)}\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right) + \mathbf{b}^{(2)}\right) \cdots + \mathbf{b}^{(L)}\right)$$

Notation Definition for Hidden Layers



Output of a neuron:

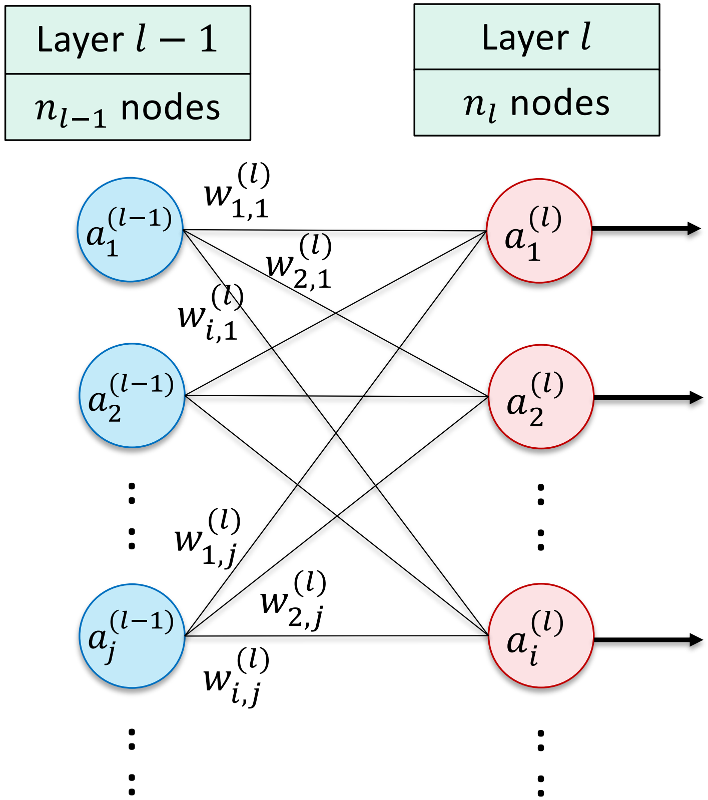
$a_i^{(l)}$

Layer l

neuron i

$$\mathbf{a}^{(l)} = \begin{bmatrix} a_1^{(l)} \\ a_2^{(l)} \\ \vdots \\ a_{n_l}^{(l)} \end{bmatrix}$$

Notation Definition for Weights



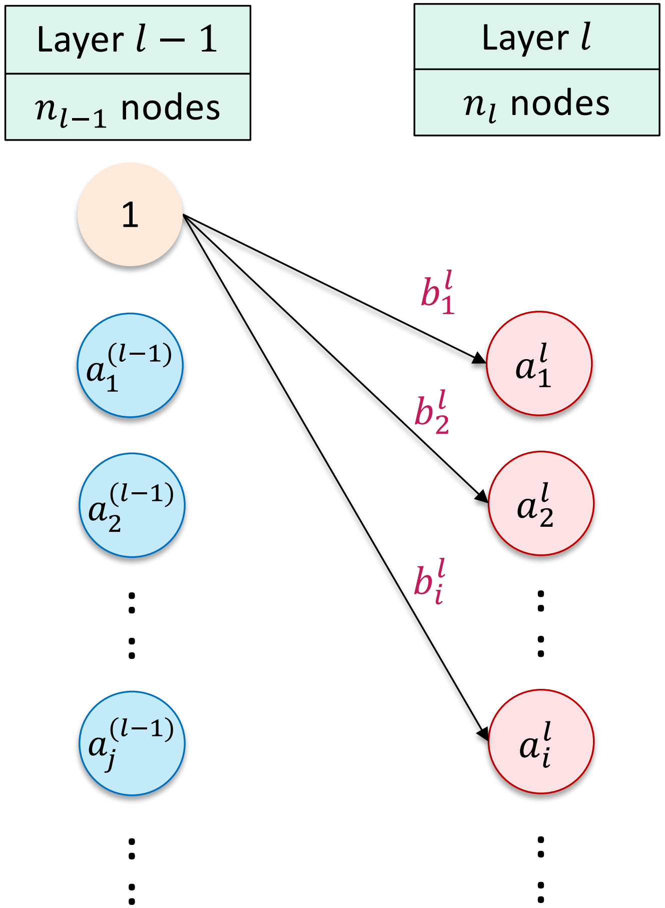
$w_{i,j}^{(l)}$ → Layer $(l-1)$ to Layer l

→ From neuron j of Layer $l-1$ to neuron i of Layer l

$$\mathbf{W}^{(l)} = \underbrace{\begin{bmatrix} w_{1,1}^{(l)} & w_{1,2}^{(l)} & \cdots & w_{1,n_{l-1}}^{(l)} \\ w_{2,1}^{(l)} & w_{2,2}^{(l)} & \cdots & w_{2,n_{l-1}}^{(l)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n_l,1}^{(l)} & w_{n_l,2}^{(l)} & \cdots & w_{n_l,n_{l-1}}^{(l)} \end{bmatrix}}_{n_{l-1}} \Bigg\} n_l$$

Weights between two layers is **a matrix**

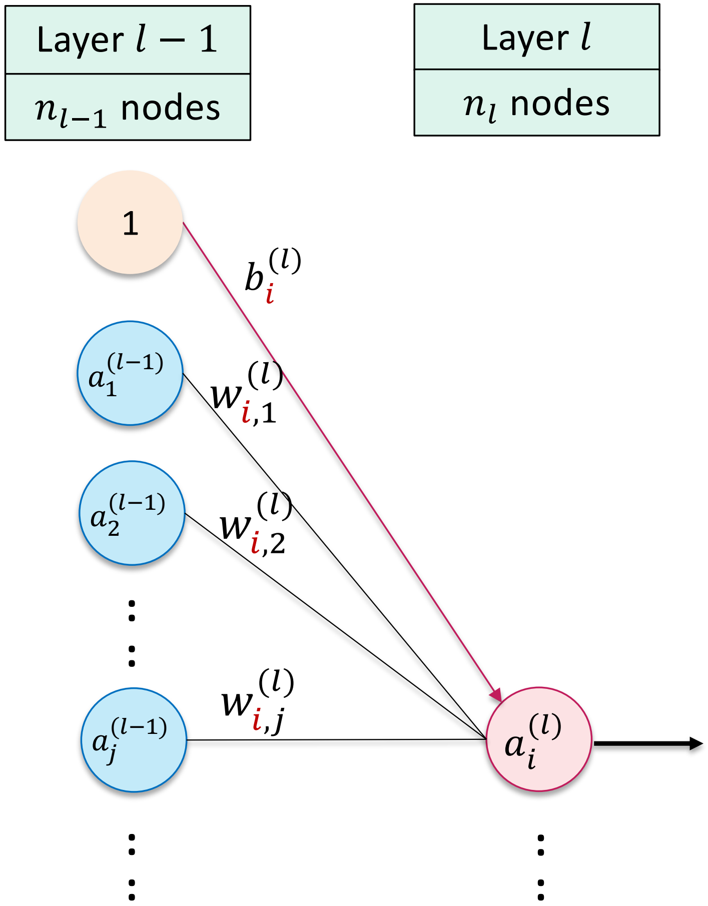
Notation Definition for Biases



$b_i^{(l)}$: Bias for neuron i at layer l

$$\mathbf{b}^{(l)} = \begin{bmatrix} b_1^{(l)} \\ b_2^{(l)} \\ \vdots \\ b_{n_l}^{(l)} \end{bmatrix}$$

Notation Definition



Layer $l-1$	Layer l
n_{l-1} nodes	n_l nodes

$z_i^{(l)}$: input of the activation function for neuron i at layer l (**Net Input**)

$$z_i^{(l)} = w_{i,1}^{(l)} a_1^{(l-1)} + w_{i,2}^{(l)} a_2^{(l-1)} + \cdots + w_{i,N_{l-1}}^{(l)} a_{n_{l-1}}^{(l-1)} + b_i^{(l)}$$

$$z_i^{(l)} = \sum_{j=1}^{n_{l-1}} w_{i,j}^{(l)} a_j^{(l-1)} + b_i^{(l)} \qquad \mathbf{z}^{(l)} = \begin{bmatrix} z_1^{(l)} \\ z_2^{(l)} \\ \vdots \\ z_{n_l}^{(l)} \end{bmatrix}$$

Activation function input at each layer is a vector

$$a_i^{(l)} = g^{(l)}\left(z_i^{(l)}\right) \quad \mathbf{a}^{(l)} = \begin{bmatrix} a_1^{(l)} \\ a_2^{(l)} \\ \vdots \\ a_{n_l}^{(l)} \end{bmatrix} = g^{(l)}\left(\begin{bmatrix} z_1^{(l)} \\ z_2^{(l)} \\ \vdots \\ z_{n_l}^{(l)} \end{bmatrix}\right)$$

Notation Summary

$a_i^{(l)}$: Output of the i-th neuron in layer l.

$\mathbf{a}^{(l)}$: Output vector of a layer l.

$z_i^{(l)}$: Net Input of the i-th neuron in layer l.
(Inputs of the activation function)

$\mathbf{z}^{(l)}$: Net input vector of activation function in layer l.

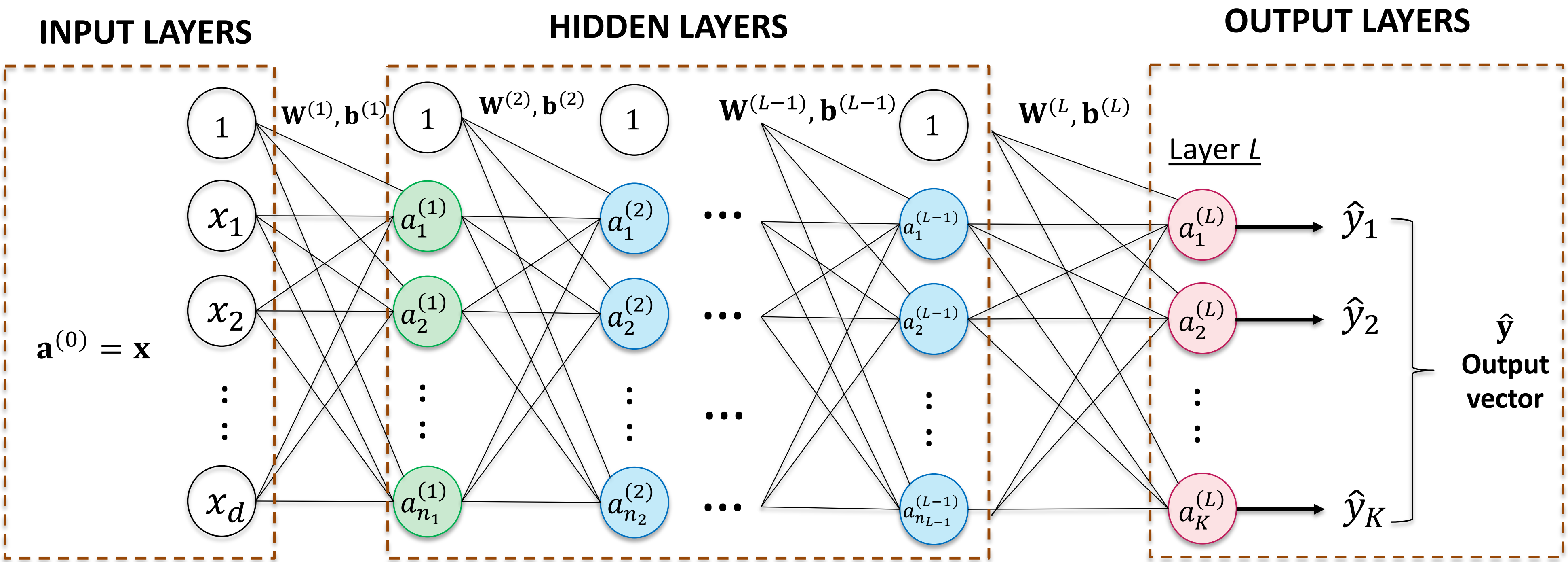
$w_{i,j}^{(l)}$: the weight connecting the j-th neuron in layer $(l-1)$ to the i-th neuron in layer l.

$\mathbf{W}^{(l)}$: the weight matrix connecting layer $(l-1)$ to layer l.

$b_i^{(l)}$: the bias of i-th neuron in layer l.

$\mathbf{b}^{(l)}$: a bias vector of neurons in layer l.

Anatomy of the Architecture



$$\mathbf{a}^{(l)} = g^{(l)}\big(\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}\big)$$

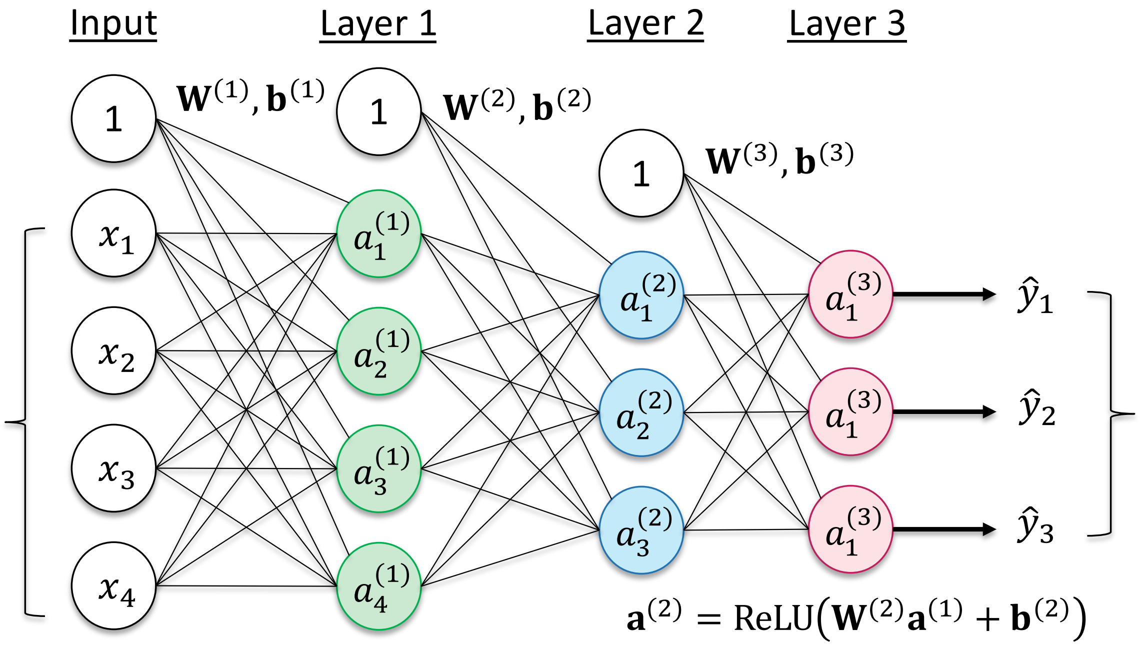
Receives raw data (features). Denoted as vector $\mathbf{x}$.

The engine of the network. Performs transformations via weights and activations to learn intermediate representations.

Produces the final prediction $\hat{\mathbf{y}}$ (probability or value).

Example Feedforward Networks

$$\mathbf{a}^{(0)} = \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$



Predicted Output

$$\hat{\mathbf{y}} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \hat{y}_3 \end{bmatrix} = \mathbf{a}^{(3)} = \begin{bmatrix} a_1^{(3)} \\ a_2^{(3)} \\ a_3^{(3)} \end{bmatrix}$$

$$= \text{Softmax}\big(\mathbf{W}^{(3)}\mathbf{a}^{(2)} + \mathbf{b}^{(3)}\big)$$

$$\mathbf{a}^{(2)} = \text{ReLU}\big(\mathbf{W}^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)}\big)$$

$$\mathbf{a}^{(1)} = \text{ReLU}\big(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\big)$$

$$\hat{\mathbf{y}} = f_\theta(\mathbf{x}) = \text{Softmax}\Big(\mathbf{W}^{(3)}\text{ReLU}\big(\mathbf{W}^{(2)}\text{ReLU}\big(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\big) + \mathbf{b}^{(2)}\big) + \mathbf{b}^{(3)}\Big)$$

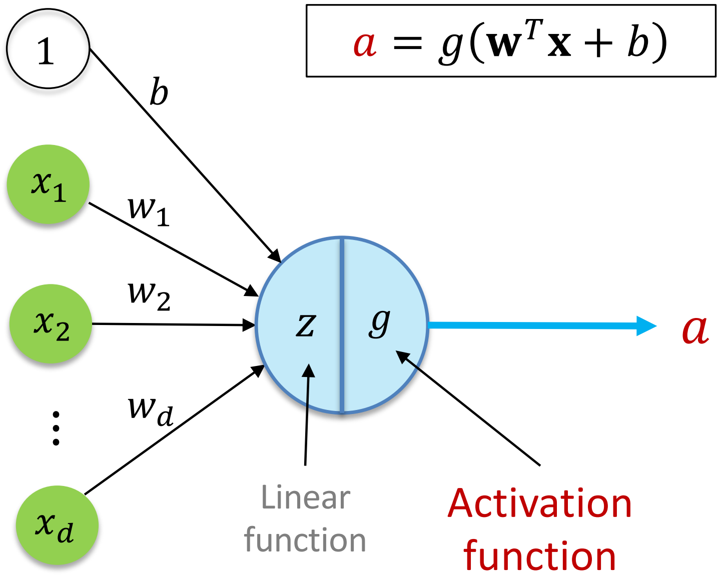
$$\boxed{\mathbf{a}^{(l)} = g^{(l)}\big(\mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)}\big)}$$

Model Parameters (Weights and Biases): $\theta = \big\{\big(\mathbf{W}^{(1)}, \mathbf{b}^{(1)}\big), \big(\mathbf{W}^{(2)}, \mathbf{b}^{(2)}\big), \big(\mathbf{W}^{(3)}, \mathbf{b}^{(3)}\big)\big\}$

Traditional Activation Functions

What are Activation Functions?

- **An Activation Function** $g(z)$ decides whether a neuron should be activated or not.
- Activation functions introduce non-linearity into the network, which is essential for modeling complex relationships in data.
- Without non-linearity, the model would essentially be a linear model, which cannot approximate complex tasks



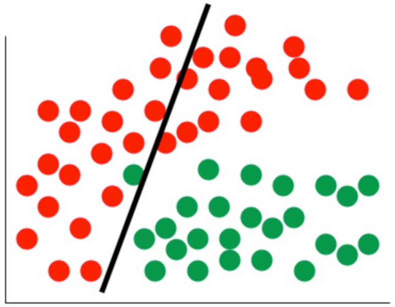
$$a = g(\mathbf{w}^T \mathbf{x} + b)$$

Why Add Non-Linear Activation Function?

- **No Nonlinearity:** A network of linear layers collapses into a single linear transformation, rendering deep architectures useless.
  - $g\left(\mathbf{W}^{(2)}\, g\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right) + \mathbf{b}^{(2)}\right) = \mathbf{W}^{(2)}\mathbf{W}^{(1)}\mathbf{x} + \left(\mathbf{W}^{(2)}\mathbf{b}^{(1)} + \mathbf{b}^{(2)}\right)$

    $= \mathbf{W}'\mathbf{x} + \mathbf{b}'$

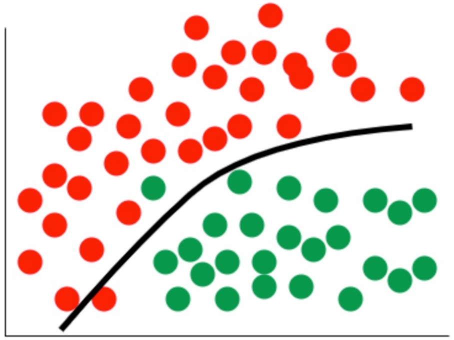
    Drop or use linear activation function such as $g(z) = z$ is equivalent to **a single linear layer.**



Linear functions produce linear decisions no matter the network size

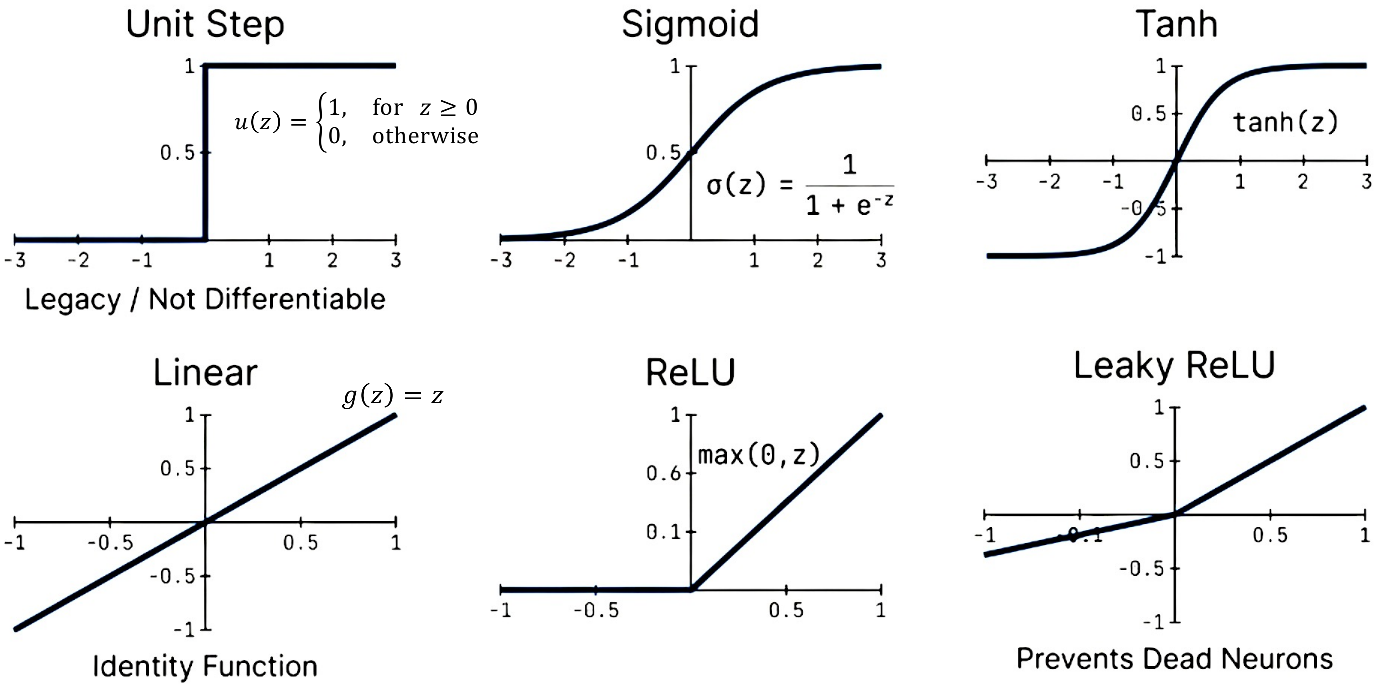
- **Non-linearity** adds capacity to the model to approximate any continuous function to arbitrary accuracy given sufficiently many hidden units.
  - See “universal approximation theorem”

Non-Linear activation functions allow us **to approximate** **arbitrarily complex functions.**



The Non-Linear Spark: Activation Functions

- Activation functions decide whether a neuron 'fires". They **introduce non-linearity,** preventing the network from collapsing into a simple linear regression.



Unit Step

Legacy / Not Differentiable

Sigmoid

Tanh

Linear

Identity Function

ReLU

Leaky ReLU

Prevents Dead Neurons

Binary Step and Linear Activation Functions

- **Linear (or Identity) Function**

$$g(z) = z \text{ , then } \hat{y} = \mathbf{w}^T\mathbf{x} + b$$

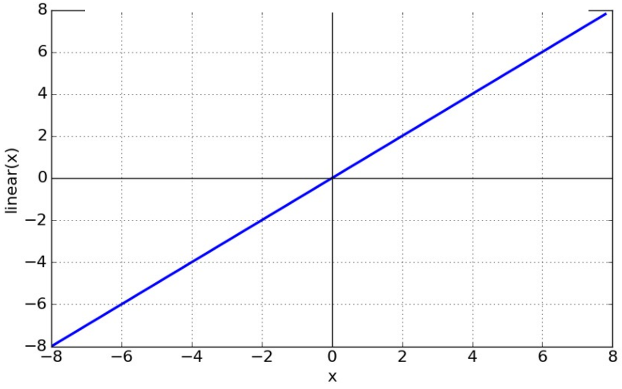
  - The activation is proportional to the input.
  - It is only used in the **output layer** for **Regression** applications.

- **Binary Step Function** (Non-Linear)

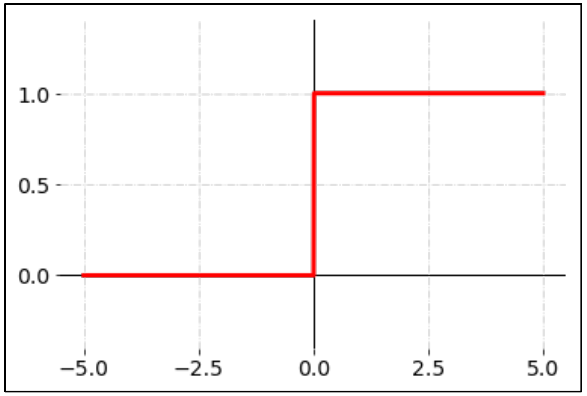
$$u(z) = \begin{cases} 1, & \text{for } \; z \geq 0 \\ 0, & \text{for } \; z < 0 \end{cases} \quad \text{and} \quad \hat{y} = \begin{cases} 1, & \mathbf{w}^T\mathbf{x} + b \geq 0 \\ 0, & \mathbf{w}^T\mathbf{x} + b < 0 \end{cases}$$

  - This is used for **Binary Classification** applications, but it is not a differentiable function and **not used** in modern neural network.

Linear Function

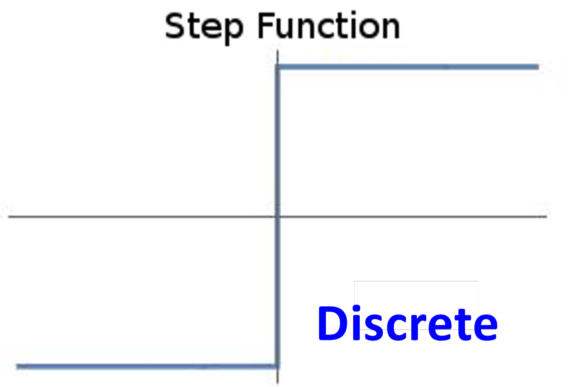


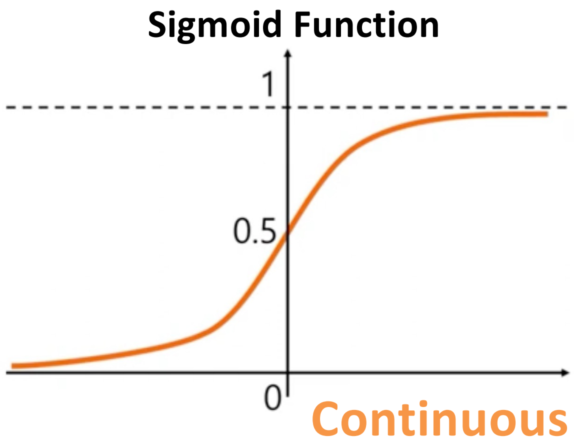
Binary Step Function



Limitation of Binary Step Activation Function

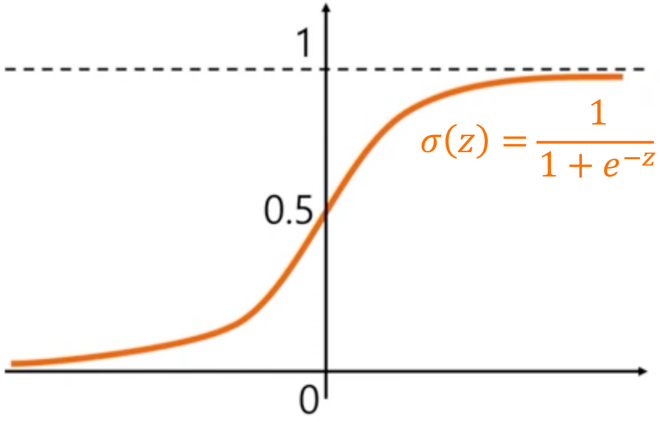
- The original **Perceptron activation is harsh**, firing only when weighted input sum exceeds threshold
    - Thresholding logic means very similar input values can get completely different outputs
    - $z =$ -0.01 and 0.01 get different outputs of 0 and 1
    - Abrupt decision change comes from step function nature of perceptron
- For real applications, want smoother activation that gradually changes from 0 to 1
    - **Sigmoid function** provides continuous smoothness and avoiding harsh cliff of the perceptron





Sigmoid Activation Function

- The **sigmoid function** is an S-shaped curve **(**smoother decision function**)** that always returns an output between 0 and 1 that mapped from the range of $-\infty$ to ∞.
- It is especially used for models where we have to predict the probability as an output. Since probability of anything exists only between the range of 0 and 1, sigmoid is the right choice
  - $\sigma(z) = \frac{1}{1+e^{-z}} = \frac{1}{1+\exp(-z)}$
  - $\sigma'(z) = \frac{d\,\sigma(z)}{dz} = \sigma(z)\big(1-\sigma(z)\big)$



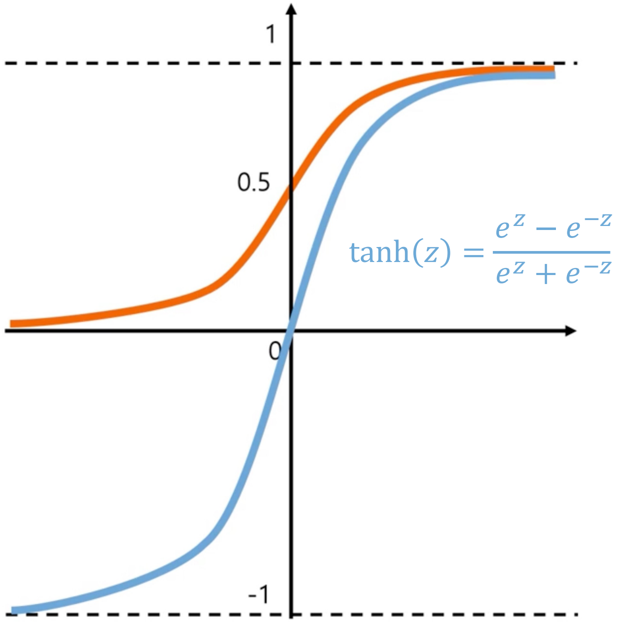
In the 1980s and early 1990s, the sigmoid function was the default activation function for neural networks.

- The sigmoid function is differentiable but has two problems: (1) a very small gradient (slop) for large positive and negative inputs and (2) a lack of zero-centeredness in its output.
- These issues can create challenges during deep learning's backpropagation.

Tanh Activation Function

- **Tanh** is aka as **Hyperbolic Tangent function**. The Tanh function also has an S-shape similar to the sigmoid function while addressing its non-zero-centered problem with output range values in the range of -1 to 1.

- Basically, Tanh is a shifted and stretched version of the sigmoid function, and the output of Tanh is symmetric around zero, leading to faster convergence
  - $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = \frac{\exp(z) - \exp(-z)}{\exp(z) + \exp(-z)}$
  - $\frac{d\tanh(z)}{dz} = 1 - \tanh(z)^2$

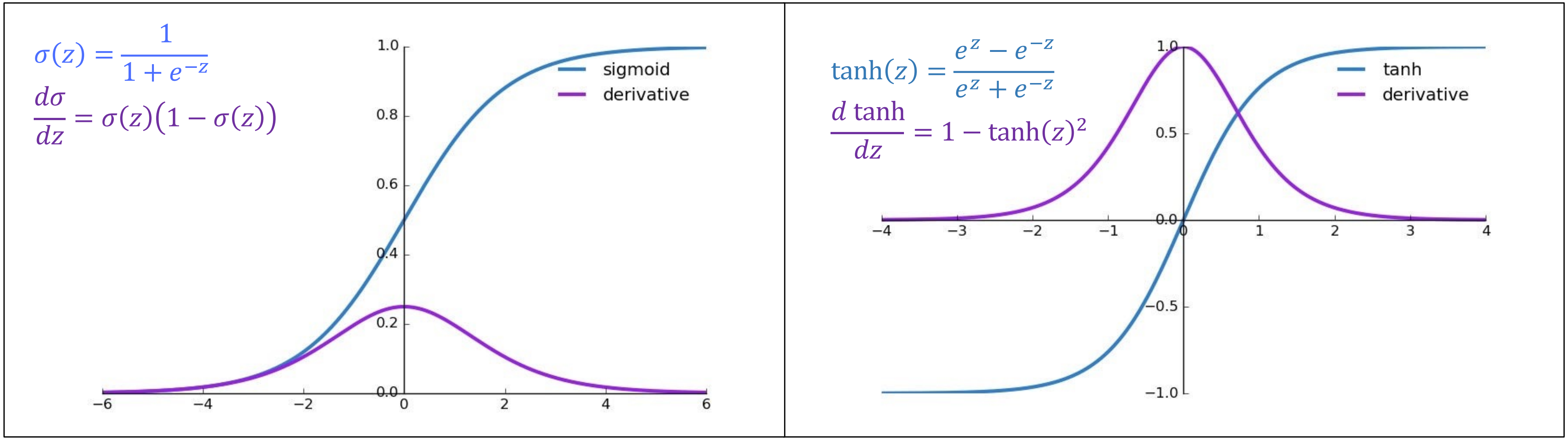


In the late 1990s and early 2000s, the tanh function was a **common choice of activation function** for neural networks.

https://www.youtube.com/watch?v=pfPDTxkXrfM

Shortcomings of Sigmoid and Tanh

- **The derivatives (gradients)** of the both sigmoid and tanh functions are small for large positive and negative inputs, which can cause a **Vanishing Gradient Problem** during backpropagation in neural network training.



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

$$\frac{d\sigma}{dz} = \sigma(z)\big(1-\sigma(z)\big)$$

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

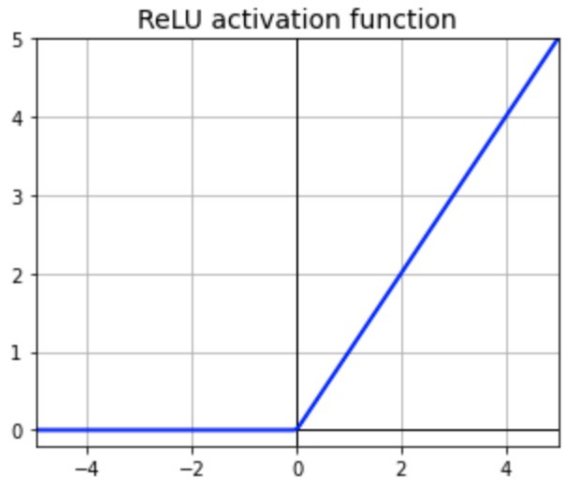
$$\frac{d\tanh}{dz} = 1-\tanh(z)^{2}$$

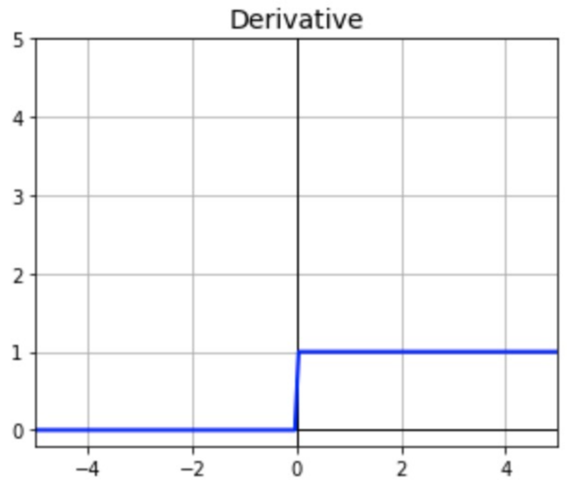
Rectified Linear Unit (ReLU)

- **ReLU** activation function to the rescue. It is a piecewise linear function that outputs the input directly if it is positive, otherwise, it outputs zero.
  - $\mathrm{ReLU}\,(z) = \max(0, z) = \begin{cases} z & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$
  - $\dfrac{d\,\mathrm{ReLU}}{dz} = \begin{cases} 1 & \text{for } z \geq 0 \\ 0 & \text{for } z < 0 \end{cases}$

$$\frac{d\,\mathrm{ReLU}}{dz} = \begin{cases} 1 & \text{for } z \geq 0 \\ 0 & \text{for } z < 0 \end{cases}$$

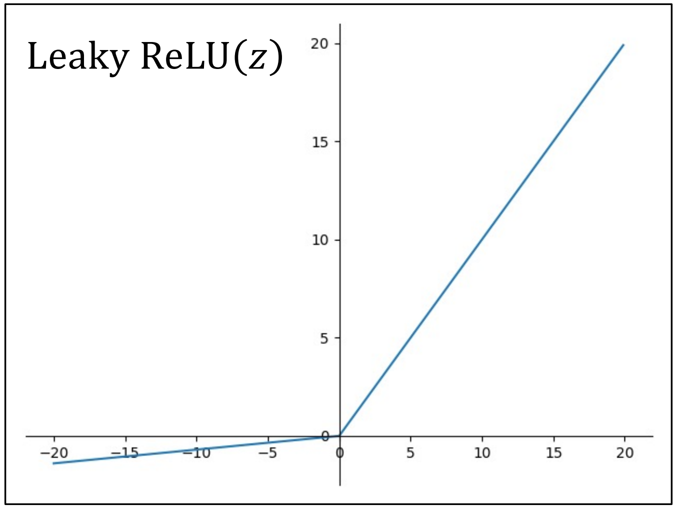
It has become the default activation function for many types of neural networks because **it is easier to train and computationally efficient**.





Leaky ReLU

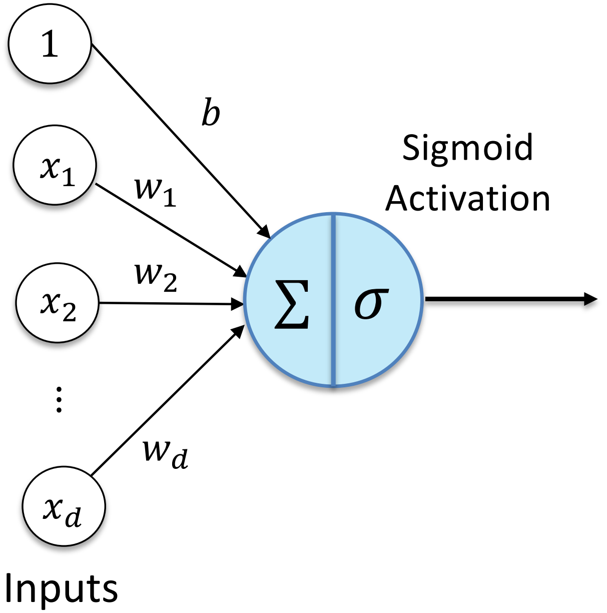
- **Leaky ReLU** is an improvisation of the regular ReLU function that addresses the problem of zero gradient for negative values
- Unlike traditional ReLU functions, which set all negative values to zero, Leaky ReLU allows a small number of negative values to pass through
- $\text{Leaky ReLU}(z) = \max(z, \alpha \cdot z) = \begin{cases} z & \text{for } z \geq 0 \\ \alpha \cdot z & \text{for } z < 0 \end{cases}$
- $\dfrac{d\ \text{LeakyRe}LU}{dz} = \begin{cases} 1 & \text{for } z \geq 0 \\ \alpha & \text{for } z < 0 \end{cases}$
- Commonly used $\alpha = 0.01$



Softmax Activation for Multiclass Classification

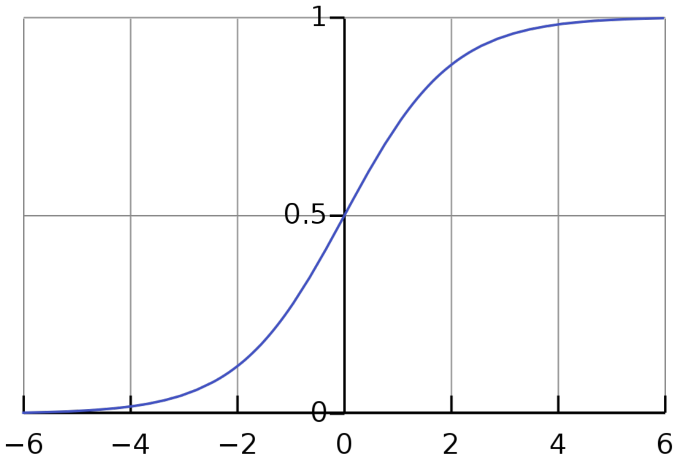
Perceptron for Binary Classification

- Even with modern activation functions, perceptron is fundamentally still a **binary classifier** that outputs a probability $P(y = 1|\mathbf{x})$ representing its confidence that the input $\mathbf{x}$ belongs to the positive class $y = 1$.
- Classification is determined by a **standard threshold** of 0.5: if $\hat{y} \geq 0.5$, the prediction is Positive (1); if $\hat{y} < 0.5$, it is Negative (0).



$$\hat{y} = \sigma(\mathbf{w}^T\mathbf{x} + b)$$
$$= P(y = 1|\mathbf{x})$$

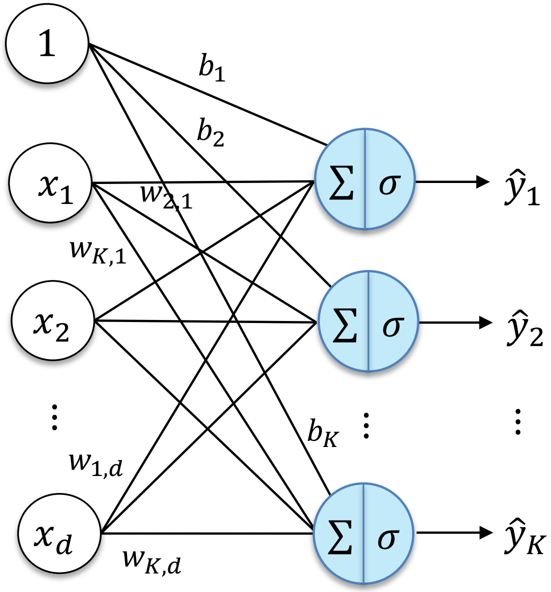
Output is the probability between 0 and 1 for positive (label $y = 1$)



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

Extend Perceptron to Multiclass Classification

- In **multiclass classification**, we can employ K **separate binary perceptron models**, each tailored to a specific class, to estimate the probability of y_j given $\mathbf{x}$ ($P\left(y_j|\mathbf{x}\right)$).
- By selecting the class with the highest probability score, we can determine the predicted class for the purpose of multiclass classification.



$$\begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_K \end{bmatrix} = \begin{bmatrix} \sigma(\mathbf{w}_1^T\mathbf{x} + b_1) \\ \sigma(\mathbf{w}_2^T\mathbf{x} + b_2) \\ \vdots \\ \sigma(\mathbf{w}_K^T\mathbf{x} + b_K) \end{bmatrix}$$

$\mathbf{w}_j^T \in \mathbb{R}^{1\times d}$ and $b \in \mathbb{R}^{K\times 1}$ where K is the number of classes

The outputs of these activations $\hat{y}_j$ are class-membership probabilities (**Not** mutually exclusive classes)

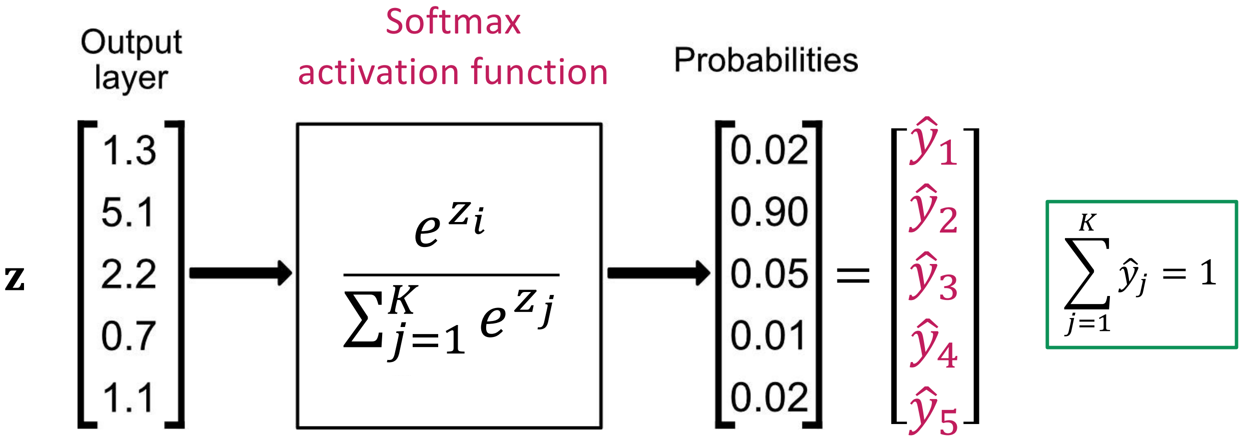
$$\sum_{j=1}^{K} \hat{y}_j \neq 1$$

Softmax Activation: Multinomial Probability Output

- Softmax is just an exponential function that **normalizes** the activations so that they **sum up to 1**
- For example, output layer scores $\mathbf{z} = [z_1, z_2, z_3, z_4, z_5]^T = [1.3, 5.1, 2.2, 0.7, 1.1]^T$

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

for $i = 1,2, \ldots, K$ and K is the number of classes



Output layer

Softmax activation function

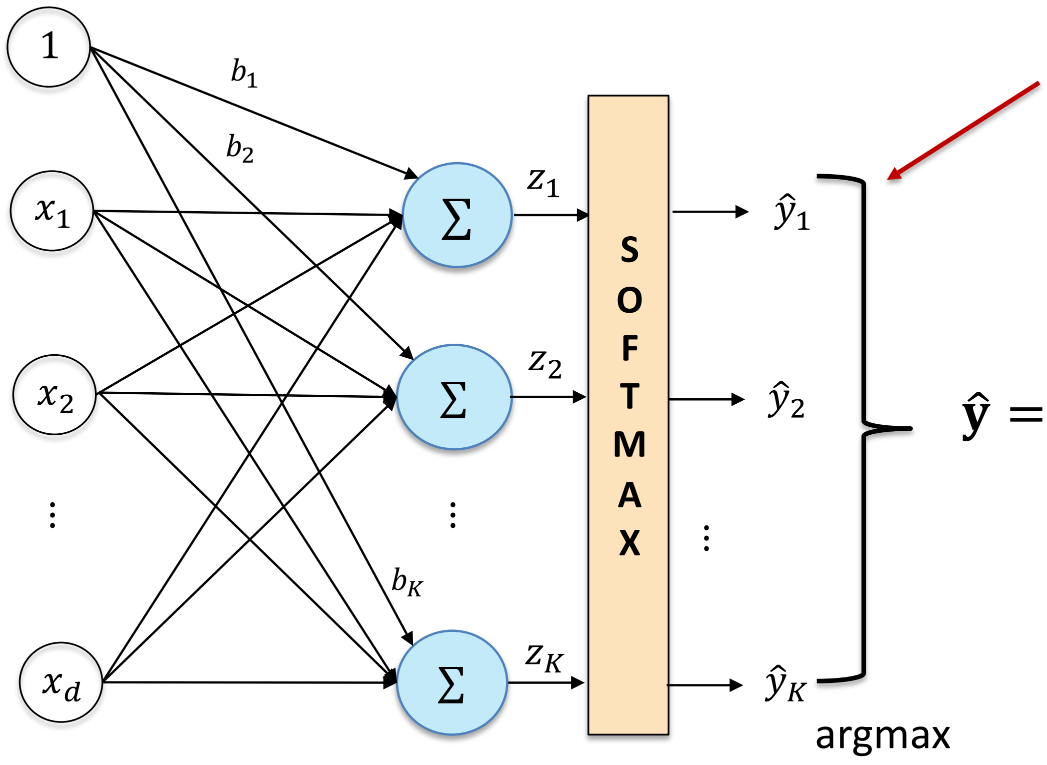
Probabilities

$$\mathbf{z} \begin{bmatrix}1.3\\5.1\\2.2\\0.7\\1.1\end{bmatrix} \longrightarrow \boxed{\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}} \longrightarrow \begin{bmatrix}0.02\\0.90\\0.05\\0.01\\0.02\end{bmatrix} = \begin{bmatrix}\hat{y}_1\\\hat{y}_2\\\hat{y}_3\\\hat{y}_4\\\hat{y}_5\end{bmatrix}$$

$$\boxed{\sum_{j=1}^{K} \hat{y}_j = 1}$$

Softmax Activation: Multinomial Probability Output

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b})$$



Activations are class-membership probabilities (**mutually** exclusive classes)

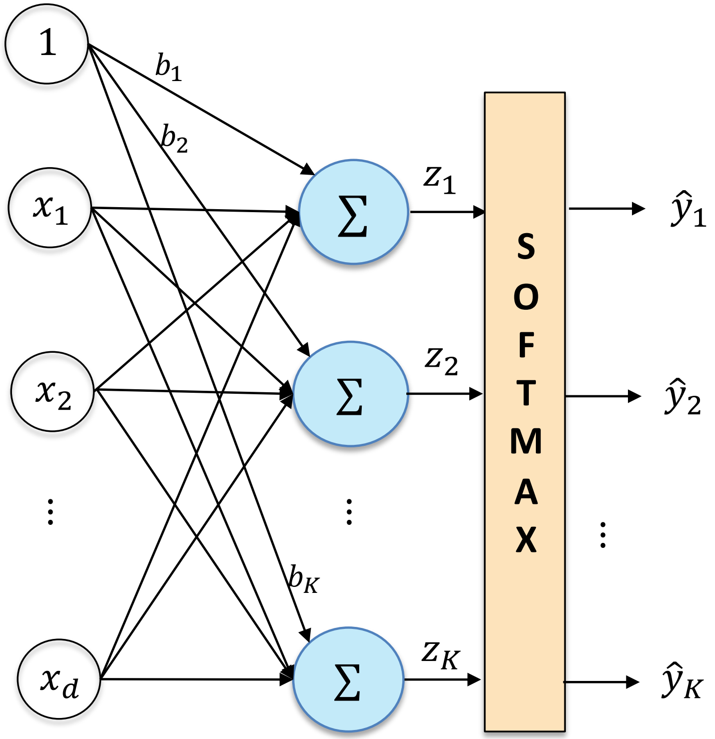
Example with *K*=5

$$\hat{\mathbf{y}} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \hat{y}_3 \\ \vdots \\ \hat{y}_K \end{bmatrix} = \begin{bmatrix} 0.02 \\ 0.90 \\ 0.05 \\ 0.01 \\ 0.02 \end{bmatrix}$$

Predicted Class Probabilities

$$\boxed{\sum_{j=1}^{K} \hat{y}_j = 1}$$

Softmax based Multiclass Classification



$$\mathbf{x} = [x_1, x_2, \dots, x_d]^T \qquad \hat{\mathbf{y}} = [\hat{y}_1, \hat{y}_2, \dots, \hat{y}_K]^T$$

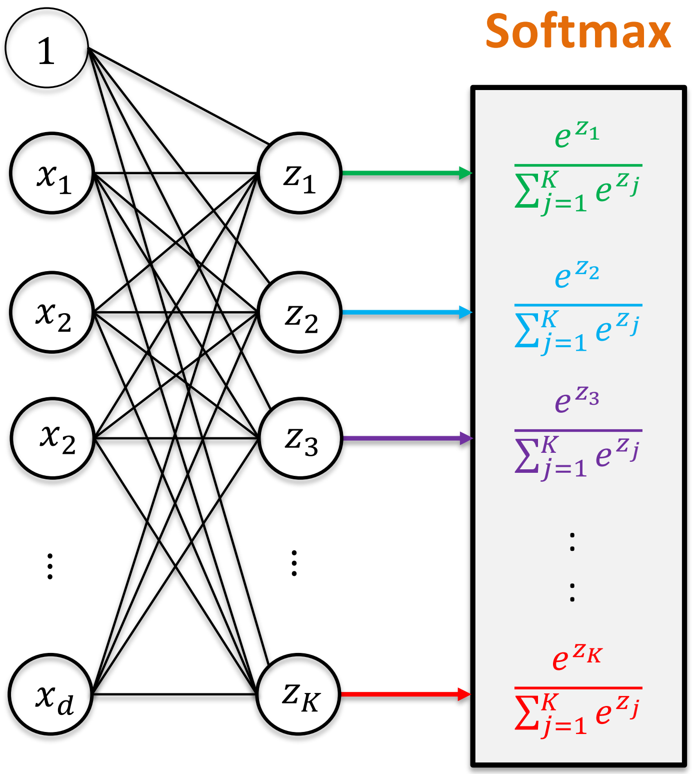
$$\mathbf{W} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,d} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,d} \\ \vdots & \vdots & \ddots & \vdots \\ w_{K,1} & w_{K,2} & \cdots & w_{K,d} \end{bmatrix} \qquad \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_K \end{bmatrix}$$

$\mathbf{W} \in \mathbb{R}^{K\times d}$ and $\mathbf{b} \in \mathbb{R}^{K\times 1}$ where K is the number of classes

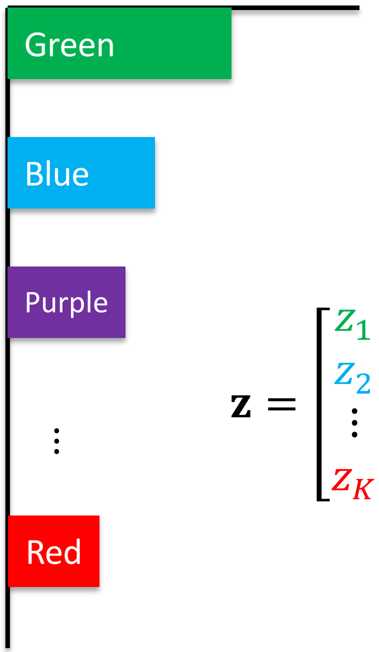
$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

$$\hat{\mathbf{y}} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \vdots \\ \hat{y}_K \end{bmatrix} = \mathrm{softmax}(\mathbf{z}) = \mathrm{softmax}\left(\begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_K \end{bmatrix}\right)$$

Softmax Example


Softmax
1
x_1
x_2
x_2
⋮
x_d
z_1
z_2
z_3
⋮
z_K
$\frac{e^{z_1}}{\sum_{j=1}^{K} e^{z_j}}$
$\frac{e^{z_2}}{\sum_{j=1}^{K} e^{z_j}}$
$\frac{e^{z_3}}{\sum_{j=1}^{K} e^{z_j}}$
⋮
$\frac{e^{z_K}}{\sum_{j=1}^{K} e^{z_j}}$

The output of Softmax represents a **discrete probability distribution** across classes.



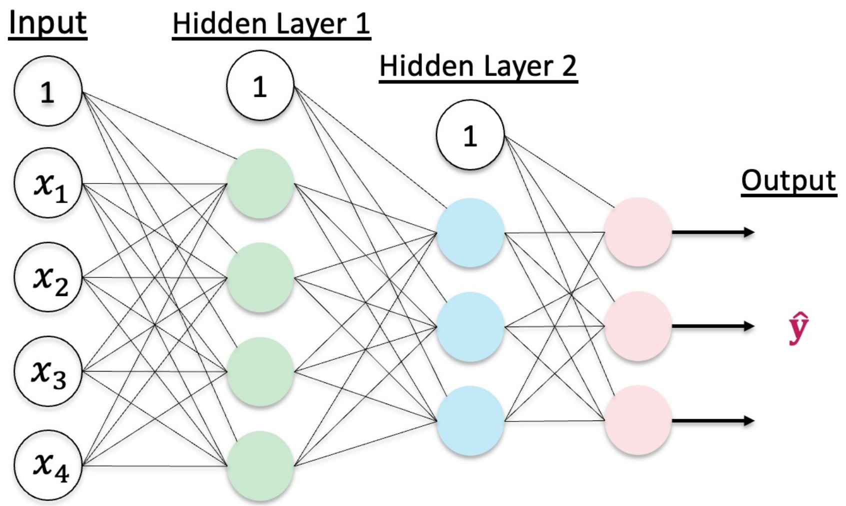
$$\mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_K \end{bmatrix} = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,d} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,d} \\ \vdots & \vdots & \ddots & \vdots \\ w_{K,1} & w_{K,2} & \cdots & w_{K,d} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_d \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_K \end{bmatrix}$$

The softmax activation function is widely preferred as the output layer choice for classification applications.

Hyperparameters of MLPs (or FFNs)

- In MLPs, neurons are organized into layers. Hidden layers take inputs from neurons and pass their activations to other neurons
  - Note that when we count the number of layers, **the input layer is NOT counted**.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$



Hyperparameters of the MLP Model :

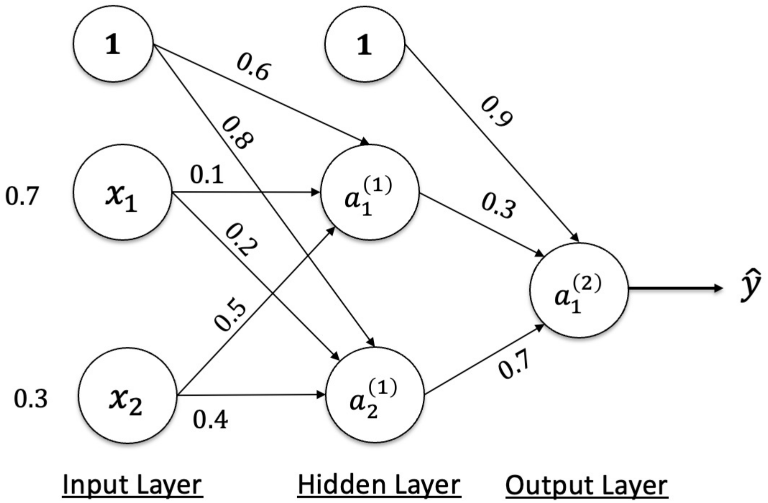
- No. of Layers: 3 ($L = 3$)
- Input Layer: 4 neurons ($d = 4$)
- Layer 1: 4 neurons ($n_1 = 4$)
- Layer 2: 3 neurons ($n_2 = 3$)
- Output Layer: 3 neuron ($K = n_3 = 3$)
- Activation function: Sigmoid σ

$$\hat{\mathbf{y}} = f_\theta(\mathbf{x}) = \sigma\left(\mathbf{W}^{(3)}\sigma\left(\mathbf{W}^{(2)}\sigma\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right) + \mathbf{b}^{(2)}\right) + \mathbf{b}^{(3)}\right)$$

Model Parameters (Weights and Biases) $\theta := \left\{\left(\mathbf{W}^{(1)}, \mathbf{b}^{(1)}\right), \left(\mathbf{W}^{(2)}, \mathbf{b}^{(2)}\right), \left(\mathbf{W}^{(3)}, \mathbf{b}^{(3)}\right)\right\}$

MLP Exercise 1 for MLP using Sigmoid

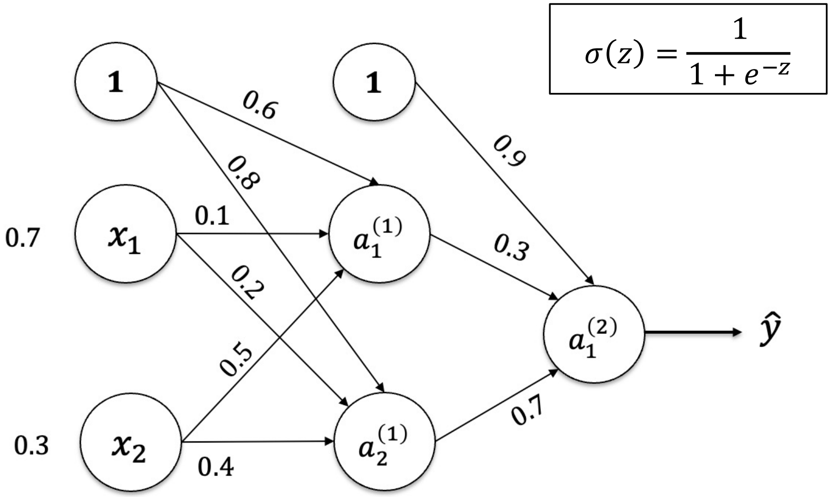
- Given a two-layer feedforward neural network (or MLP) using sigmoid activation functions, determine the output $\hat{y}$ by representing the network in matrix form.
- Include the intermediate results of net inputs $z_i^{(l)}$ and activations $a_i^{(l)}$ of the hidden layer and output layer.



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

Solution of Exercise 1 for MLP using Sigmoid

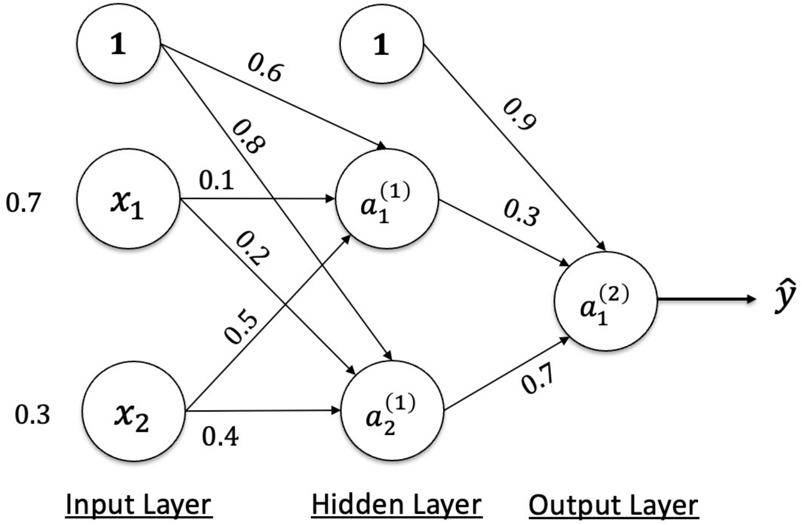
$$\sigma(z) = \frac{1}{1+e^{-z}}$$



- The net input vector $\mathbf{z}^{(1)}$ of the hidden layer is given by
  - $\mathbf{z}^{(1)} = \begin{bmatrix} z_1^{(1)} \\ z_2^{(1)} \end{bmatrix} = \mathbf{W}^{(1)}\,\mathbf{x} + \mathbf{b}^{(1)} = \begin{bmatrix} w_{1,1}^{(1)} & w_{1,2}^{(1)} \\ w_{2,1}^{(1)} & w_{2,1}^{(1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \end{bmatrix}$
  - $\begin{bmatrix} z_1^{(1)} \\ z_2^{(1)} \end{bmatrix} = \begin{bmatrix} 0.1 & 0.5 \\ 0.2 & 0.4 \end{bmatrix} \begin{bmatrix} 0.7 \\ 0.3 \end{bmatrix} + \begin{bmatrix} 0.6 \\ 0.8 \end{bmatrix} = \begin{bmatrix} 0.1 \times 0.7 + 0.5 \times 0.3 + 0.6 \\ 0.2 \times 0.7 + 0.4 \times 0.3 + 0.8 \end{bmatrix} = \begin{bmatrix} 0.82 \\ 1.06 \end{bmatrix}$
- The activation vector $\mathbf{a}^{(1)}$ of the hidden layer is given by
  - $\mathbf{a}^{(1)} = \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \end{bmatrix} = \sigma(\mathbf{z}^{(1)}) = \sigma\left(\begin{bmatrix} 0.82 \\ 1.06 \end{bmatrix}\right) = \begin{bmatrix} 1/(1+e^{-0.82}) \\ 1/(1+e^{-1.06}) \end{bmatrix} = \begin{bmatrix} 0.6942 \\ 0.7427 \end{bmatrix}$
- The net input vector $\mathbf{z}^{(2)}$ of the output layer is given by
  - $\mathbf{z}^{(2)} = \begin{bmatrix} z_1^{(2)} \end{bmatrix} = \mathbf{W}^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)} = \begin{bmatrix} w_{1,1}^{(2)} & w_{1,2}^{(2)} \end{bmatrix} \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \end{bmatrix} + \begin{bmatrix} b_1^{(2)} \end{bmatrix} = \begin{bmatrix} 0.3 & 0.7 \end{bmatrix} \begin{bmatrix} 0.6942 \\ 0.7427 \end{bmatrix} + [0.9] = 0.3 \times 0.6942 + 0.7 \times 0.7427 + 0.9 = 1.6282$
- The activation vector $\mathbf{a}^{(1)}$ of the output layer (output of the network $\hat{y}$) is given by
  - $\hat{y} = \mathbf{a}^{(2)} = \begin{bmatrix} a_1^{(1)} \end{bmatrix} = \sigma(\mathbf{z}^{(2)}) = \sigma(1.6282) = \frac{1}{1+e^{-1.6282}} = 0.8359$
- The matrix representation of the MLP network is given by
  - $\hat{y} = \sigma\left(\mathbf{W}^{(2)}\sigma\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right) + \mathbf{b}^{(2)}\right)$

MLP Exercise 2 for MLP using ReLU

- Given a two-layer feedforward neural network (or MLP) using **ReLU** activation function for the hidden layer and **linear** activation function for the output layer, determine the output $\hat{y}$ by representing the network in matrix form.
- Include the intermediate results of net inputs $z_i^{(l)}$ and activations $a_i^{(l)}$ of the hidden layer and output layer.



$$\text{ReLU}\ (z) = \max(0, z) = \begin{cases} 0 & \text{for } z < 0 \\ z & \text{for } z \geq 0 \end{cases}$$

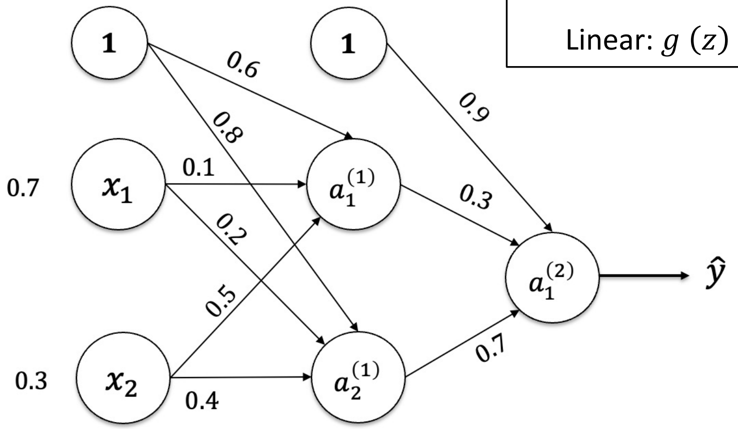
Linear: $g\ (z) = z$

Solution of Exercise 2 for MLP using ReLU

ReLU $(z) = \max(0, z)$

Linear: $g\ (z) = z$

- The net input vector $\mathbf{z}^{(1)}$ of the hidden layer is given by
  - $\mathbf{z}^{(1)} = \begin{bmatrix} z_1^{(1)} \\ z_2^{(1)} \end{bmatrix} = \mathbf{W}^{(1)}\ \mathbf{x} + \mathbf{b}^{(1)} = \begin{bmatrix} w_{1,1}^{(1)} & w_{1,2}^{(1)} \\ w_{2,1}^{(1)} & w_{2,1}^{(1)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1^{(1)} \\ b_2^{(1)} \end{bmatrix}$
  - $\begin{bmatrix} z_1^{(1)} \\ z_2^{(1)} \end{bmatrix} = \begin{bmatrix} 0.1 & 0.5 \\ 0.2 & 0.4 \end{bmatrix} \begin{bmatrix} 0.7 \\ 0.3 \end{bmatrix} + \begin{bmatrix} 0.6 \\ 0.8 \end{bmatrix} = \begin{bmatrix} 0.1\times0.7 + 0.5\times0.3 + 0.6 \\ 0.2\times0.7 + 0.4\times0.3 + 0.8 \end{bmatrix} = \begin{bmatrix} 0.82 \\ 1.06 \end{bmatrix}$
- The activation vector $\mathbf{a}^{(1)}$ of the hidden layer is given by
  - $\mathbf{a}^{(1)} = \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \end{bmatrix} = \text{ReLU}(\mathbf{z}^{(1)}) = \text{ReLU}\left(\begin{bmatrix} 0.82 \\ 1.06 \end{bmatrix}\right) = \begin{bmatrix} \max(0, 0.82) \\ \max(0, 1.06) \end{bmatrix} = \begin{bmatrix} 0.82 \\ 1.06 \end{bmatrix}$
- The net input vector $\mathbf{z}^{(2)}$ of the output layer is given by
  - $\mathbf{z}^{(2)} = \begin{bmatrix} z_1^{(2)} \end{bmatrix} = \mathbf{W}^{(2)}\mathbf{a}^{(1)} + \mathbf{b}^{(2)} = \begin{bmatrix} w_{1,1}^{(2)} & w_{1,2}^{(2)} \end{bmatrix} \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \end{bmatrix} + \begin{bmatrix} b_1^{(2)} \end{bmatrix} = \begin{bmatrix} 0.3 & 0.7 \end{bmatrix} \begin{bmatrix} 0.82 \\ 1.06 \end{bmatrix} + [0.9] = 0.3\times0.82 + 0.7\times1.06 + 0.9 = 1.888$
- The activation vector $\mathbf{a}^{(1)}$ of the output layer (output of the network $\hat{y}$) is given by
  - $\hat{y} = \mathbf{a}^{(2)} = \begin{bmatrix} a_1^{(1)} \end{bmatrix} = g(\mathbf{z}^{(2)}) = g(1.888) = 1.888$
- The matrix representation of the MLP network is given by
  - $\hat{y} = \mathbf{W}^{(2)}\text{ReLU}\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right) + \mathbf{b}^{(2)}$

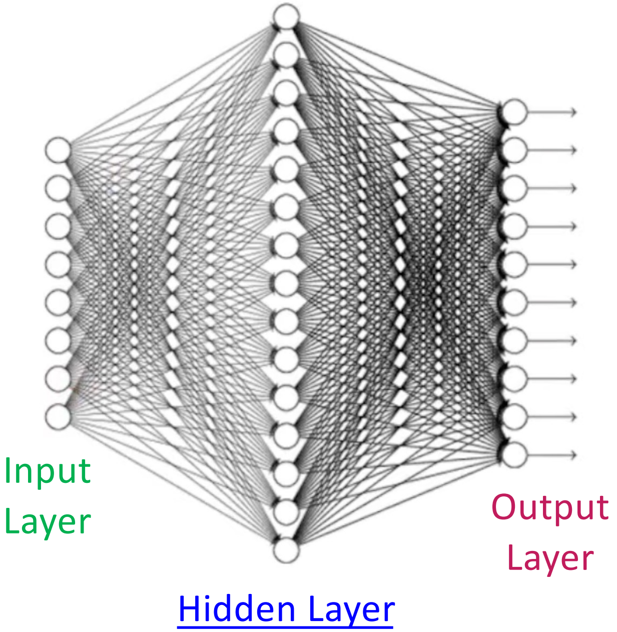


Why Deeper Neural Network is Better?

Universal Approximation Theorem (1989)

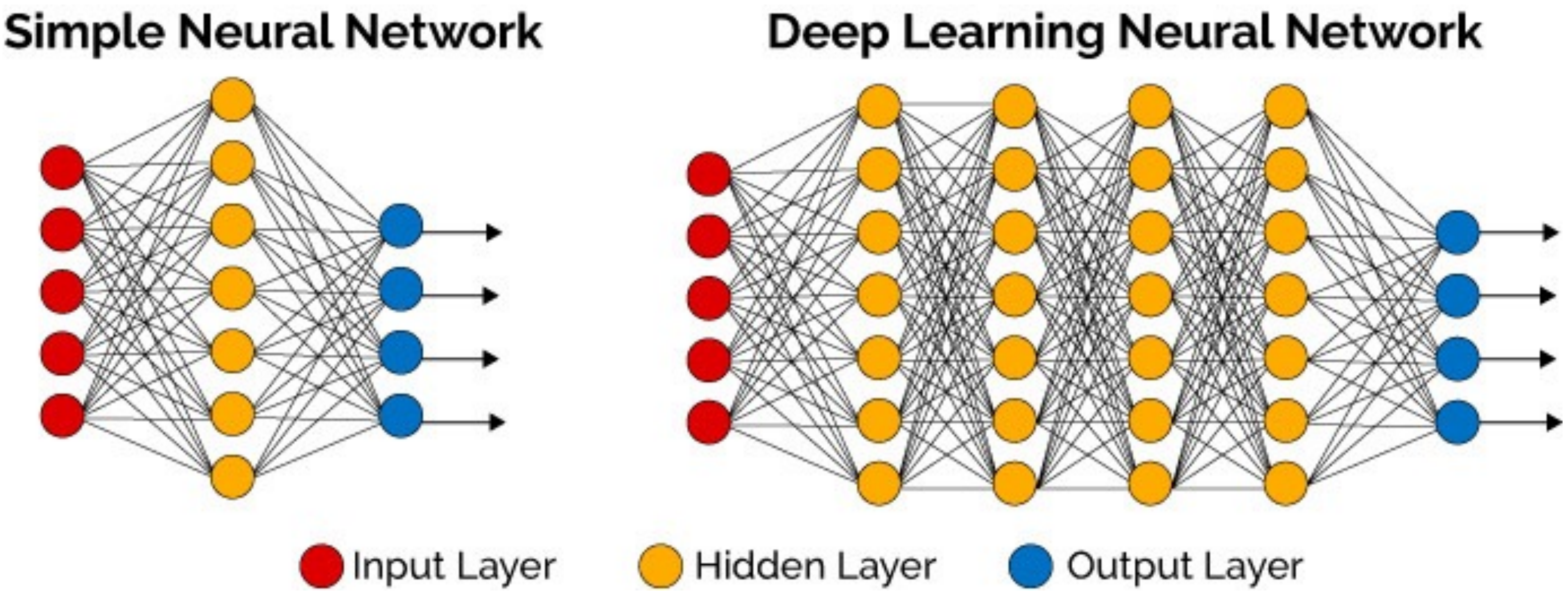
- **Theorem**: A multilayered network of neurons with a single hidden layer can be used to approximate any continuous function to any desired precision
  - $f: \mathbb{R}^m \to \mathbb{R}^n$
- Only **one hidden layer** is enough
  - This refers to a **Two-layer Feedforward Network**
    - one hidden layer and the output layer

Why "deep" not "wide"?



Wide + Shallow vs Thin + Deep

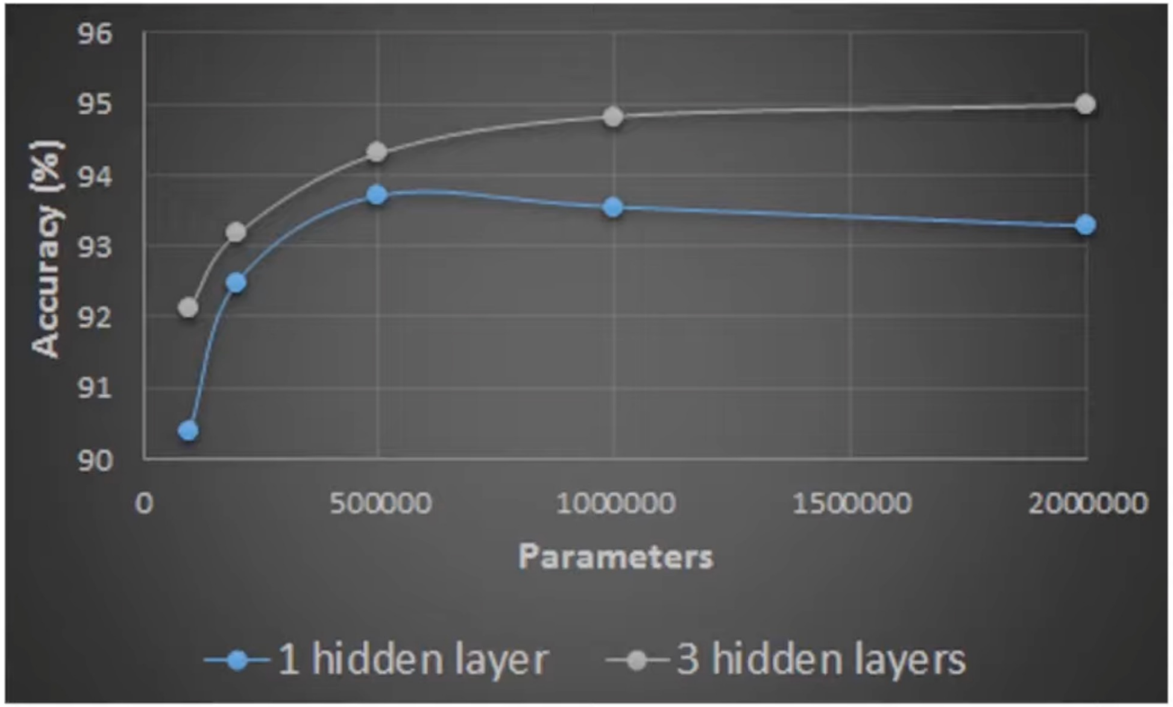
- For two MLP networks with the same number of parameters but different width and depth, **which one is better**?



Wide + Shallow vs Thin + Deep

Is the deeper the better?

- Handwritten digit recognition performance
- The deeper network uses less parameters to achieve the same performance.



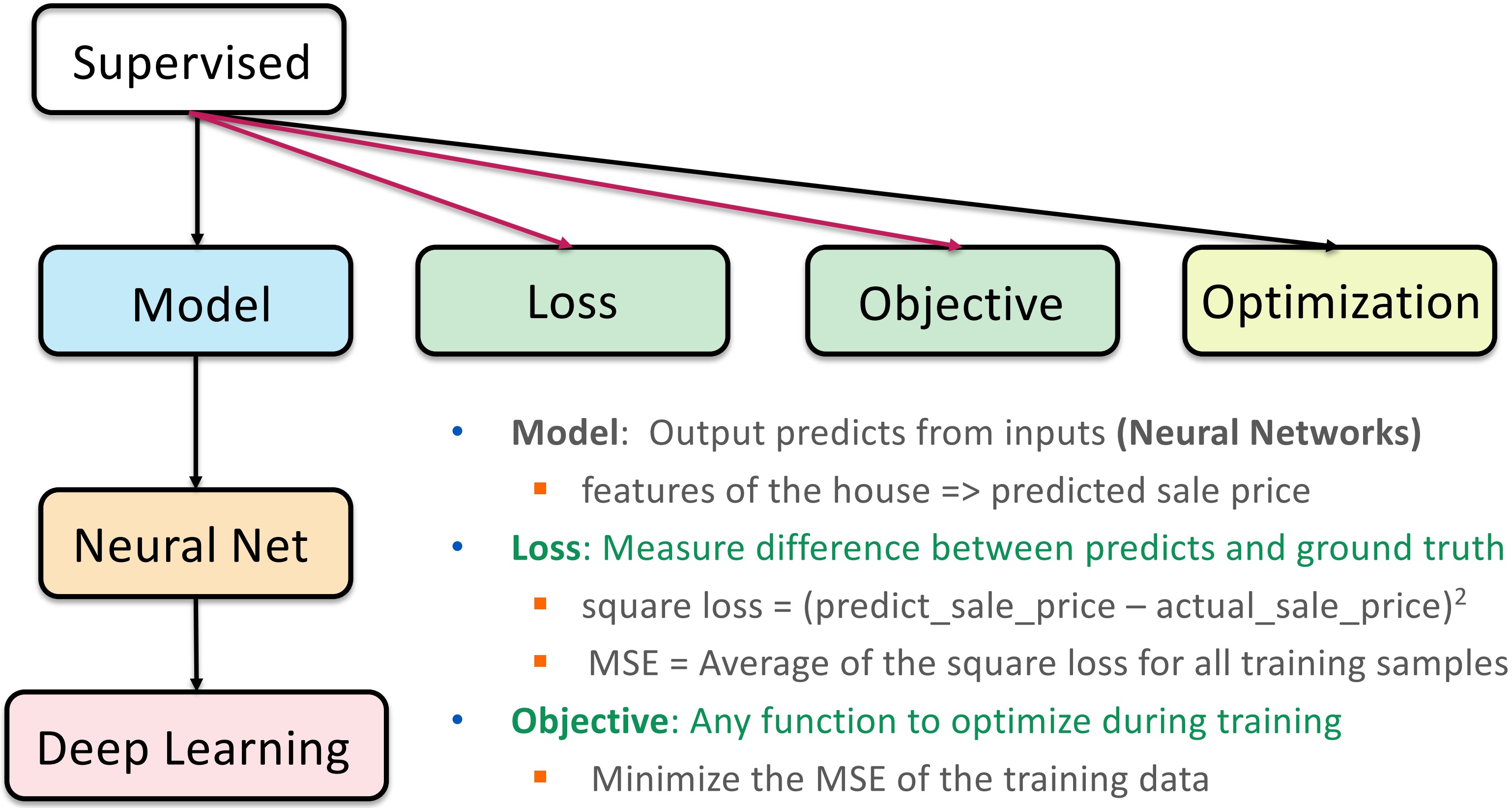
It is better to have **deeper network** than wider network.

Why Deep Networks Outperform with Reduced Parameters?

- **Hierarchical representation**: Deep networks learn progressively more abstract features, capturing intricate patterns and variations in the data.
- **Reusing features**: Deeper networks can share learned features across multiple layers, reducing redundancy and the need for additional parameters.
- **Non-linear transformations**: Deep networks employ non-linear activation functions, enabling them to model complex relationships and reduce the need for a wider network.
- **Regularization effect**: The architecture of deeper networks introduces noise and randomness, acting as a form of regularization, preventing overfitting and improving generalization performance.

Loss, Cost, and Objective Functions

Components in Supervised Training



- **Model**: Output predicts from inputs **(Neural Networks)**
  - features of the house => predicted sale price
- **Loss**: Measure difference between predicts and ground truth labels
  - square loss = (predict_sale_price – actual_sale_price)^2
  - MSE = Average of the square loss for all training samples
- **Objective**: Any function to optimize during training
  - Minimize the MSE of the training data
- **Optimization**: Learn model parameters by solving the objective function

Objective of the MLP Modeling

- In order **to approximate a function** f, we typically leverage **a training dataset** $\mathcal{D}$ consisting of **noisy estimated samples** $\mathbf{x}^{(i)}$ along with their corresponding **target values** $\mathbf{y}^{(i)}$ (labels).
  - $\mathcal{D} := \{(\mathbf{x}^{(1)}, \mathbf{y}^{(1)}), (\mathbf{x}^{(2)}, \mathbf{y}^{(2)}), \ldots, (\mathbf{x}^{(N)}, \mathbf{y}^{(N)})\} = \{(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})\}_{i=1}^{N}$
- Our model, a Multilayer Perceptron (MLP) or Feedforward Networks, utilizes the function $\hat{\mathbf{y}} = f_\theta(\mathbf{x})$ to make predictions $\hat{\mathbf{y}}^{(i)} = f_\theta(\mathbf{x}^{(i)})$ that closely match the target values $\mathbf{y}^{(i)}$.
- The main objective is to determine the optimal **weights** and **biases** parameters $\theta = \{(\mathbf{W}^{(l)}, \mathbf{b}^{(l)})\}_{i=1}^{L}$ for f_θ, aiming to achieve a close approximation to f.

- **The objective is to construct a model** $f_\theta(\mathbf{x}^{(i)})$ where the predicted values $\hat{\mathbf{y}}^{(i)}$ exhibit a strong alignment with the true labels $\mathbf{y}^{(i)}$, which is assessed by a **loss function** to **measure the difference** between the predicted values $\hat{\mathbf{y}}^{(i)}$ and target values $\mathbf{y}^{(i)}$(labels).

Loss Function and Cost Function

- Loss Function: It is denoted as $\ell\left(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}\right)$, is **utilized to quantify the prediction error** between the model's prediction $\hat{\mathbf{y}}^{(i)} = f_\theta\left(\mathbf{x}^{(i)}\right)$ and the true label $\mathbf{y}^{(i)}$ for **a single training example** $(\mathbf{x}^{(i)},\mathbf{y}^{(i)})$ from the dataset $\mathcal{D}: = \left\{(\mathbf{x}^{(i)},\mathbf{y}^{(i)})\right\}_{i=1}^{N}$.
- The loss function can be represented as

$$\ell\left(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}\right) = \ell\left(\mathbf{y}^{(i)}, f_\theta\left(\mathbf{x}^{(i)}\right)\right)$$

- During the training process, the model's parameters (weights and biases) θ are adjusted to minimize the total or average loss for a set of training examples.
- Cost Function: **Average of loss functions over the entire training dataset**. Measures overall model performance and is used for optimizing model parameters.

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N} \ell\left(\mathbf{y}^{(i)}, f_\theta\left(\mathbf{x}^{(i)}\right)\right) = \frac{1}{N}\sum_{i=1N} \ell\left(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}\right)$$

Loss Function vs Cost Function

Loss Function $\ell\left(y^{(i)}, \hat{y}^{(i)}\right)$

- Error for **a single data point** (One sample in training set)
- Calculated many times for every training samples during the training cycle (epoch)
- Has only error terms

$$\ell_{squared}\left(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}\right) = \frac{1}{2}\left\|\mathbf{y}^{(i)} - \hat{\mathbf{y}}^{(i)}\right\|_2^2$$

Cost Function $\mathcal{L}(\theta)$

- Average error of ***N*-samples** in the data (for the whole training dataset).
- Calculated once for entire training set during the training cycle (epoch).
- Can have other terms like regularization, etc.

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N}\ell\left(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}\right) + \lambda\sum_{j=0}^{M}\left(w_j\right)^2$$

Loss term Regularization term

Objective Functions

- The **Cost Function**, $\mathcal{L}(\theta)$, measures a model's performance by averaging errors across all training examples. It is **a specific instance of the broader concept of an objective function**, which defines the overall goal of the machine learning task.
- **Objective Functions** encompass **what the model aims to optimize**, such as minimizing errors, incorporating regularization, or maximizing rewards.
- They formulate optimization problems in various learning scenarios, guiding the model to learn effectively from data.
  - **Minimization:** $\theta^* = \arg\min_{\theta} \mathcal{L}(\theta) = \arg\min_{\theta} \mathcal{L}_{\text{MSE}}(\theta)$ **Objective function of regression problem**
  - **Maximization:** $\theta^* = \arg\max_{\theta} \prod_{i=1}^{N} \mathrm{P}_{\theta}(\mathbf{x}^{(i)})$ **Maximum Likelihood Estimation**

Cost and Loss Functions for Deep Learning

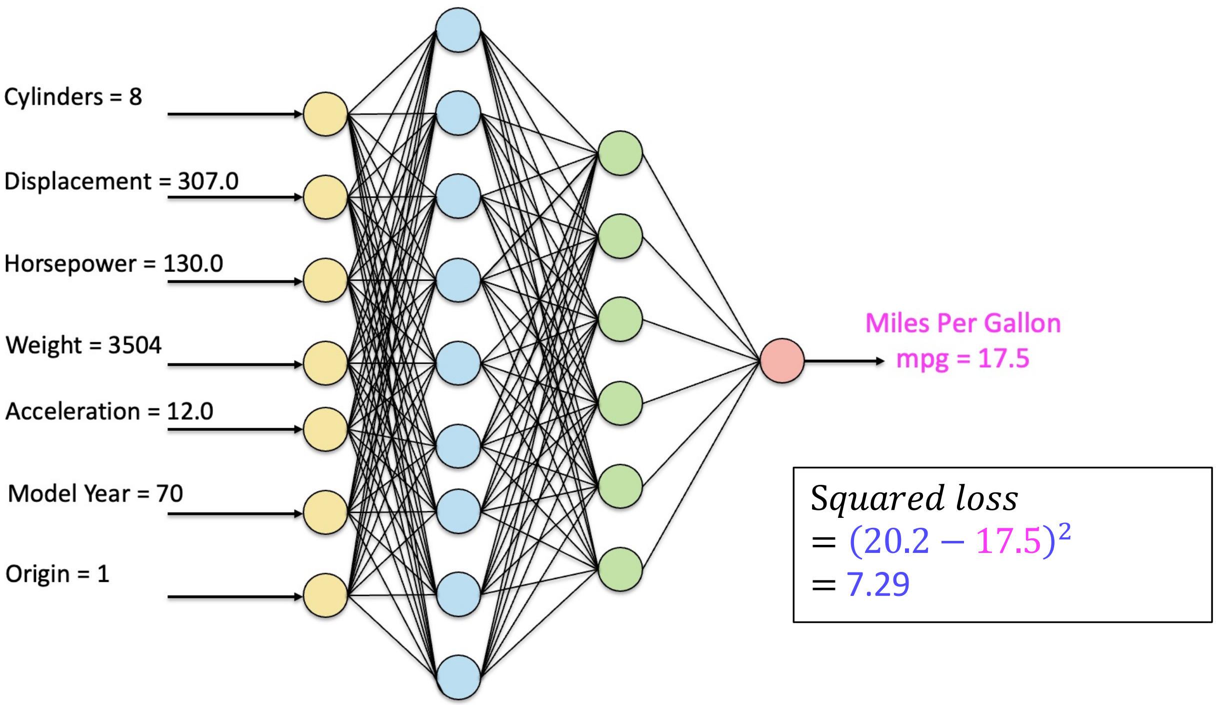
- Choosing the right loss/cost function depends on the specific problem and data, and it's crucial for aligning model behavior with task objectives.
- **Regression Loss**:
  - **Mean Squared Error (MSE):** Measures the average squared difference between the predicted and target values.
  - **Mean Absolute Error (MAE):** Calculates the average absolute difference between the predicted and target values.
- **Classification Loss:**
  - **Binary Cross-Entropy (BCE):** Used in binary classification tasks, it measures the dissimilarity between predicted and target probability distributions.
  - **Categorical Cross-Entropy (CCE):** Suitable for multi-class classification, it quantifies the difference between predicted and target probability distributions.

MSE and MAE Cost Functions for Regressions

- MSE and MAE are commonly used cost functions for Regressions tasks. For a single training example with predicted output $\hat{\mathbf{y}}^{(i)}$ and target $\mathbf{y}^{(i)}$, their losses are defined as
  - **Squared Loss:** $\ell_{squared}(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}) = \frac{1}{2}\|\hat{\mathbf{y}}^{(i)} - \mathbf{y}^{(i)}\|_2^2 = \frac{1}{2}\sum_{j=1}^{d}\left(\hat{y}_j^{(i)} - y_j^{(i)}\right)^2$
  - **Absolute Loss:** $\ell_{abs}(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}) = \sum_{j=1}^{d}\left|y_j^{(i)} - \hat{y}_j{}^{(i)}\right|$
- The **Cost Functions** using squared and absolute losses for a set of n examples are defined as
  - **Mean Square Error (MSE):** $\mathcal{L}_{\text{MSE}} = \frac{1}{N}\sum_{i=1}^{N}\frac{1}{2}\|\hat{\mathbf{y}}^{(i)} - \mathbf{y}^{(i)}\|_2^2 = \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{j=1}^{d}\left(y_j^{(i)} - \hat{y}_j{}^{(i)}\right)^2\right)$
  - **Mean Absolute Error (MAE):** $\mathcal{L}_{\text{MAE}} = \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{j=1}^{d}\left|y_j^{(i)} - \hat{y}_j{}^{(i)}\right|\right)$

Simple Car MPG (Mile Per Gallon) Regression

- A Multi-Layer Perceptron (MLP) model is trained using the Car MPG Dataset to make regression predictions of Mile Per Gallon (continuous values), incorporating various car attributes like horsepower, weight, acceleration, and more.

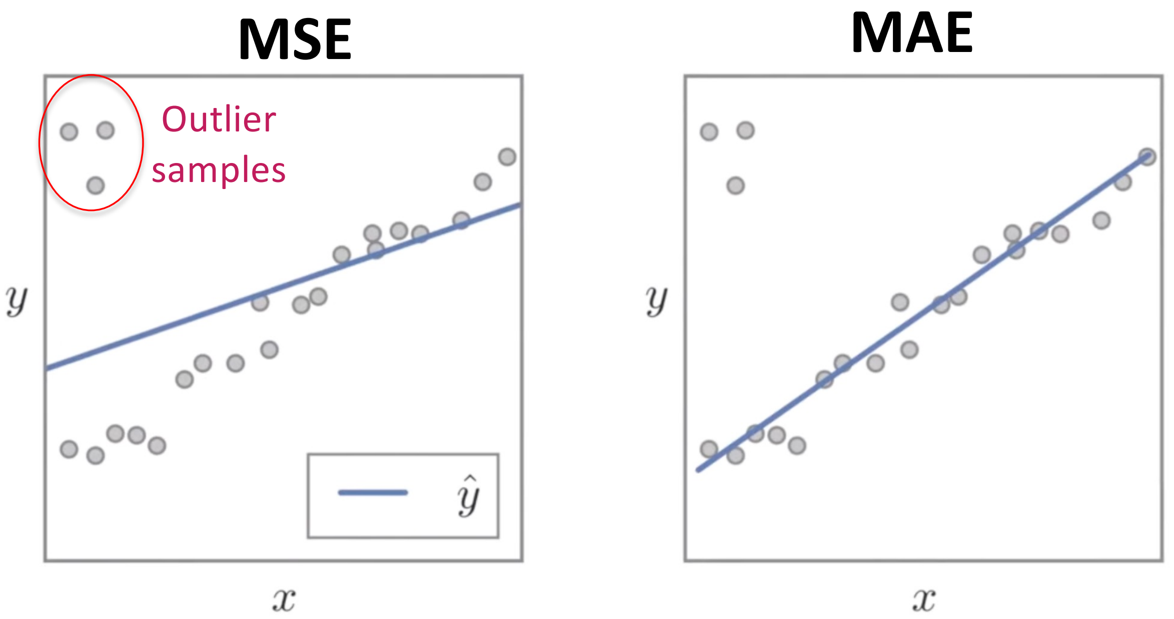


$$\textit{Squared loss} = (20.2 - 17.5)^2 = 7.29$$

https://colab.research.google.com/drive/1QyWO7_TI4-F4vYOD-HhKHBHxcashfMW4

MSE and MAE Cost Functions for Regressions

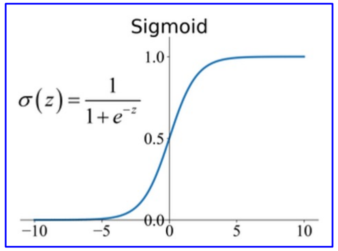
- **MSE penalizes larger prediction errors** more significantly due to the squaring operation. This means that outliers or instances with larger errors contribute more to the overall loss.
- **MSE cost is more sensitive to outliers than MAE cost**



BCE: Loss Functions for Binary Classification

- If we want to classify an input only into two options, **class 0** or **class 1**, we can use a single neuron output layer with sigmoid activation function.

$$\hat{\mathbf{y}} = \sigma\left(\mathbf{W}^{(L)} \cdots g\left(\mathbf{W}^{(2)} g\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right) + \mathbf{b}^{(2)}\right) \cdots + \mathbf{b}^{(L)}\right)$$



Output between 0 and 1

- **Binary Cross Entropy (BCE)** aka Negative Log Loss is commonly used for binary classification. The BCE loss function is defined as

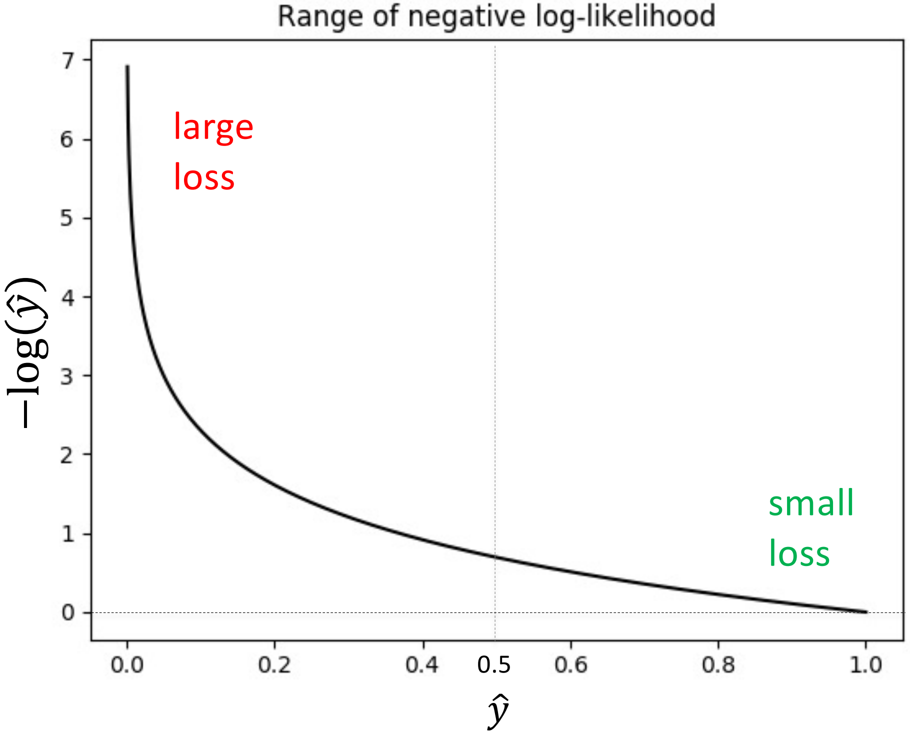
$$\mathrm{BCE} = -y\log(\hat{y}) - (1-y)\log(1-\hat{y}) = \begin{cases} -\log(\hat{y}) & \text{if } y = 1 \\ -\log(1-\hat{y}) & \text{if } y = 0 \end{cases}$$

- Each predicted probability $\hat{y}$ is compared to the actual class output value (y = 0 or 1) and the predicted probability can be calculated by the sigmoid function.

Visualization of Negative Log-Loss Function

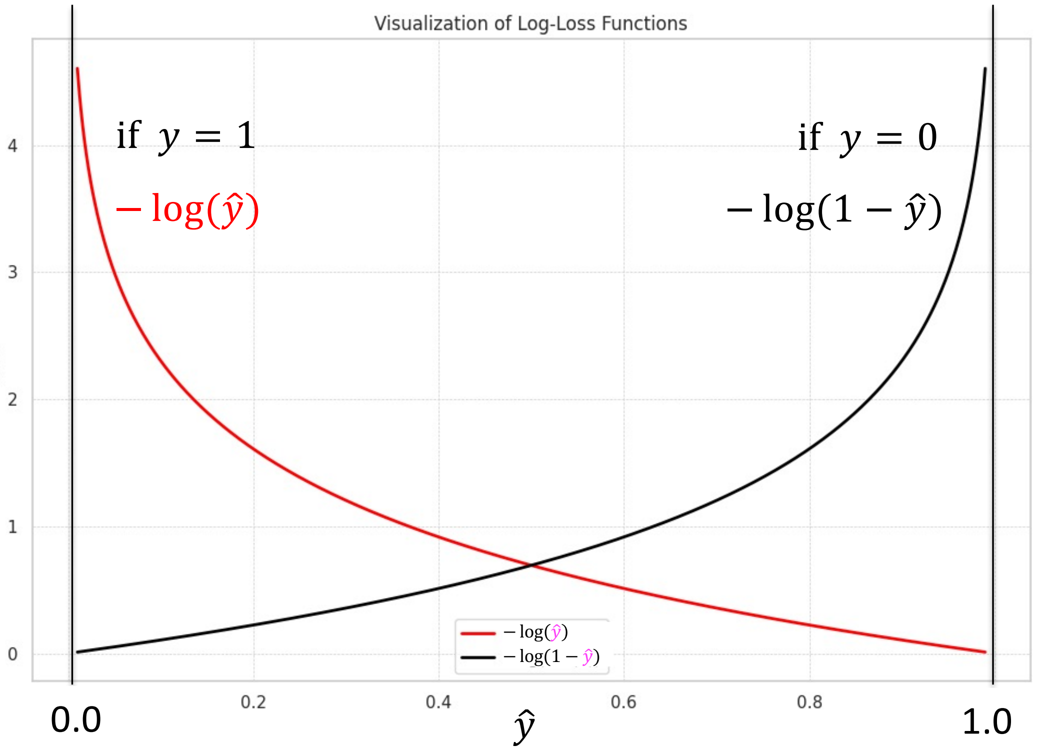
Negative Log-Loss Curve: Demonstrating the increasing penalty as predicted probabilities diverge from true labels. The steeper the curve, the higher the cost of being wrong.

- For positive samples with label $y = 1$
  - $\mathrm{BCE} = -1 \cdot \log(\hat{y}) - (1-1)\log(\hat{y})$
  - $\mathrm{BCE} = -\log(\hat{y})$
    - $\hat{y} = 1$ => $\mathrm{BCE} = -\log(1) = 0$
    - $\hat{y} > 0.5$ => BCE will be small when small when the prediction is correct
    - $\hat{y}$ close to 0 => BCE is a very large when the prediction is wrong.



Visualization of BCE Loss Function

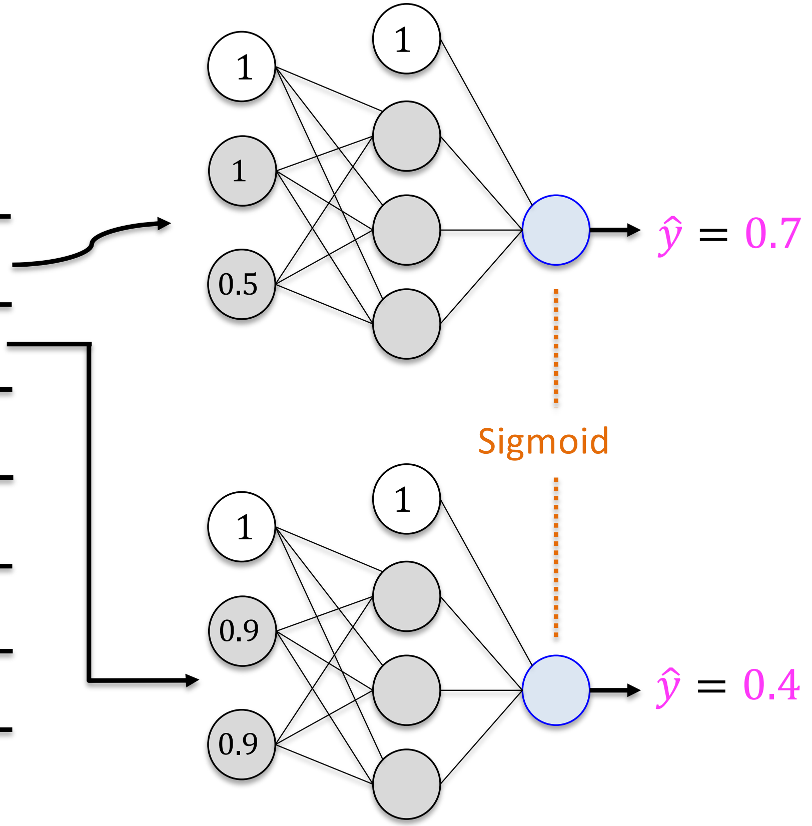
$$\text{BCE} = -y\log(\hat{y}) - (1-y)\log(1-\hat{y}) = \begin{cases} -\log(\hat{y}) & \text{if } y = 1 \\ -\log(1-\hat{y}) & \text{if } y = 0 \end{cases}$$



Binary Cross Entropy (BCE) Example

Training Dataset

$x_1^{(i)}$	$x_2^{(i)}$	$y^{(i)}$
1	0.5	1
0.9	0.9	0
3	0.7	1
2.9	0.9	0
3.5	0.8	1
4	1.2	1



Loss

$y^{(1)} = 1$, so:

BCE = $-y\log(\hat{y}) - (1-y)\log(1-\hat{y})$

$= -\log(0.7) = 0.15$

Binary cross-entropy (BCE)

$y^{(2)} = 0$, so:

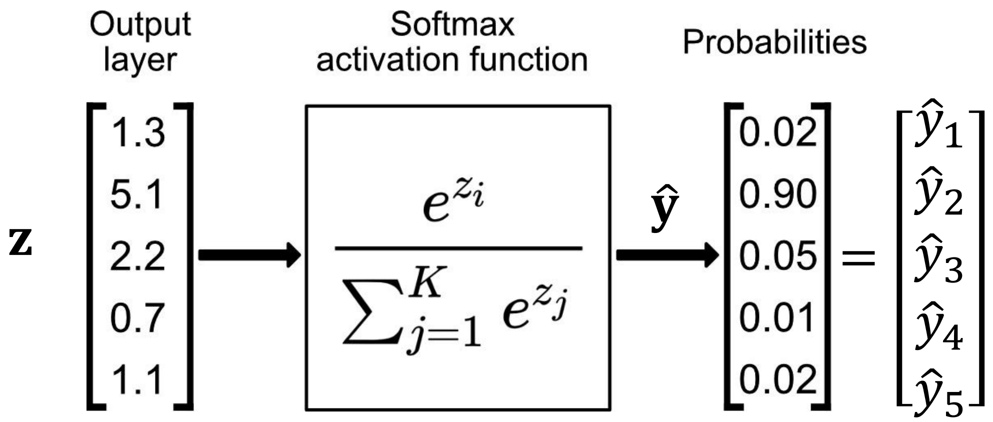
BCE = $-y\log(\hat{y}) - (1-y)\log(1-\hat{y})$

$= -\log(1-0.4) = 0.2$

$$\text{BCE} = -[y\log(\hat{y}) + (1-y)\log(1-\hat{y})] = \begin{cases} -\log(\hat{y}) & \text{for } y = 1 \\ -\log(1-\hat{y}) & \text{for } y = 0 \end{cases}$$

Categorial Cross Entropy Loss (or Softmax Loss)

- It is a Softmax activation plus a cross-entropy loss for **multi-class classification task**


Output layer
Softmax activation function
Probabilities
$\mathbf{z}$
$\begin{bmatrix} 1.3 \\ 5.1 \\ 2.2 \\ 0.7 \\ 1.1 \end{bmatrix}$
$\frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$
$\hat{\mathbf{y}}$
$\begin{bmatrix} 0.02 \\ 0.90 \\ 0.05 \\ 0.01 \\ 0.02 \end{bmatrix} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \hat{y}_3 \\ \hat{y}_4 \\ \hat{y}_5 \end{bmatrix}$

$$\mathrm{CCE} = -\sum_{j=1}^{K} y_j \log(\hat{y}_j)$$

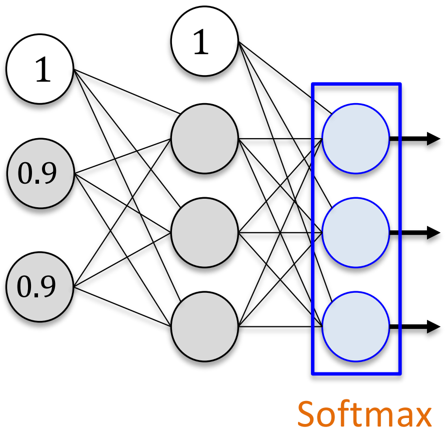
The target label $\mathbf{y}$ is required to be represented by one-hot-encoding: $\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_K \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$

Categorical Cross Entropy (CCE) Example

Training Dataset

$x_1^{(i)}$	$x_2^{(i)}$	$\mathbf{y}^{(i)}$
1	0.5	$[1, 0, 0]^{\mathrm{T}}$
0.9	0.9	$[0, 1, 0]^{\mathrm{T}}$
3	0.7	$[1, 0, 0]^{\mathrm{T}}$
2.9	0.9	$[0, 0, 1]^{\mathrm{T}}$
3.5	0.8	$[1, 0, 0]^{\mathrm{T}}$
4	1.2	$[0, 1, 0]^{\mathrm{T}}$

Class 1: $[1, 0, 0]^{\mathrm{T}}$
Class 2: $[0, 1, 0]^{\mathrm{T}}$
Class 3: $[0, 0, 1]^{\mathrm{T}}$



Output	Label	CCE Loss
$\hat{y}_1^{(2)} = 0.04$	$y_1^{(2)} = 0$	
$\hat{y}_2^{(2)} = 0.95$	$y_2^{(2)} = 1$	$-\log(0.95) = 0.051$
$\hat{y}_3^{(2)} = 0.01$	$y_3^{(2)} = 0$	

$$\mathrm{CCE} = -\sum_{j=1}^{K} y_j \log(\hat{y}_j)$$

Categorical Cross-Entropy (CCE)

Cross-Entropy Cost Function

- The **Cost function** $\mathcal{L}(\theta)$ using Categorial Cross-Entropy Loss for K different class labels and training dataset with N examples:

$$\mathcal{L}(\theta) = \sum_{i=1}^{N} \sum_{k=1}^{K} -y_k^{(i)} \log\left(\hat{y}_k^{(i)}\right)$$

- This assumes one-hot encoded labels.

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ \vdots \\ y_K \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

Cross-Entropy Cost Function for Binary Classification

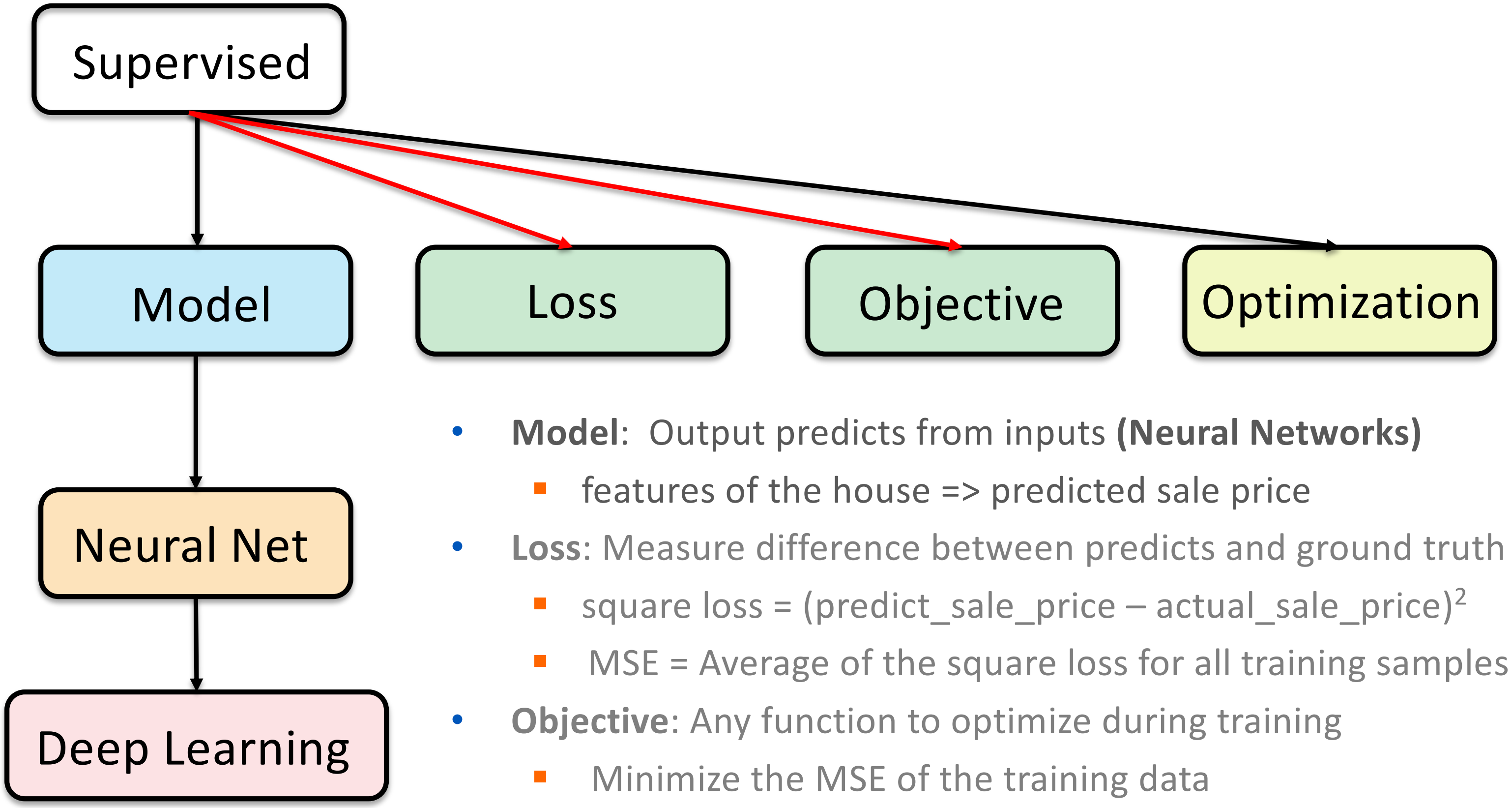
- **Binary Cross-Entropy Cost** with $y^{(i)} \in \{0, 1\}$ (Sigmoid activation is used in the output layer as single output):

$$\mathcal{L}(\theta) = -\sum_{i=1}^{N} \left[y^{(i)} \log\left(\hat{y}_k^{(i)}\right) + \left(1 - y^{(i)}\right) \log\left(1 - \hat{y}_k^{(i)}\right) \right]$$

- **Binary Cross-Entropy Cost** with $\mathbf{y}^{(i)}$ in **one-hot encoding** $\mathbf{y}^{(i)} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$ or $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$

$$\mathcal{L}(\theta) = \sum_{i=1}^{N} \sum_{j=1}^{2} -y_j^{(i)} \log\left(\hat{y}_k^{(i)}\right)$$

Components in Supervised Training



- **Model**: Output predicts from inputs **(Neural Networks)**
  - features of the house => predicted sale price
- **Loss**: Measure difference between predicts and ground truth labels
  - square loss = (predict_sale_price – actual_sale_price)^2
  - MSE = Average of the square loss for all training samples
- **Objective**: Any function to optimize during training
  - Minimize the MSE of the training data
- **Optimization**: Learn model parameters by solving the objective function

Optimization

Gradient Descent

- Gradient descent is an optimization algorithm used to minimize the loss function by iteratively adjusting the network's weights. It is a fundamental technique in training neural networks, including MLPs. The goal is to find the set of weights that minimize the loss function, thereby improving the model's performance.
- The basic idea behind gradient descent is to iteratively update the model's parameters (weights and biases) in the direction that reduces the loss function. This is done by computing the gradient of the loss function with respect to each parameter and then **adjusting the parameters in the opposite direction of the gradient.**

Function = MLP Model + A Set of Model Parameters

- $\hat{\mathbf{y}} = f_\theta(\mathbf{x}) = g\left(\mathbf{W}^{(L)} \cdots g\left(\mathbf{W}^{(2)} g\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right) + \mathbf{b}^{(2)}\right) \cdots + \mathbf{b}^{(L)}\right)$

Function set

Different parameters $\mathbf{W}^{(i)}, \mathbf{b}^{(i)}$ (**Weights** and **biases**) => different

Hyperparameters: No. of Layer L and No. of nodes of each layer N_l
Activation function (Sigmoid, ReLU, Softmax, etc)

- **Formal definition for MLP (or Feedforward Networks)**
  - $f_\theta(\mathbf{x})$ where θ is the model parameter set of Feedforward Networks
    - $\theta := \left\{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)}, \ldots, \mathbf{W}^{(L)}, \mathbf{b}^{(L)}\right\}$

Pick a function $f_\theta(\cdot)$ = Pick a set of model parameters θ.

How to Pick a set of model parameters?

- To approximate a function f, we are generally **given a dataset** $\mathcal{D} := \left\{\left(\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\right)\right\}_{i=1}^{N}$ with labels $\mathbf{y}^{(i)}$ are noisy estimates of the target function $f\left(\mathbf{x}^{(i)}\right)$ at different points $\mathbf{x}^{(i)}$.
  - A MLP neural network defines a function $\hat{y} = f_\theta(\mathbf{x})$ with a set of model parameters $\theta := \left\{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)}, \dots, \mathbf{W}^{(L)}, \mathbf{b}^{(L)}\right\}$ (weights and biases of the MLP model)
  - Goal is to find the parameters θ such that the model function $f_\theta(\mathbf{x})$ best approximates the target $f(\mathbf{x})$.
- How to find the values of the parameters i.e., **Supervised training of the network**?
  - **Gradient Descent** is the most common optimization method used in deep learning to find the best parameter θ^* for a model $f_\theta(\mathbf{x})$ using a loss/cost function:

$$\theta^* = \min_\theta \mathcal{L}(\theta) = \min_\theta \frac{1}{N} \sum_{i=1}^{N} \ell\left(\mathbf{y}^{(i)}, f_\theta\left(\mathbf{x}^{(i)}\right)\right)$$

Gradient Descent Algorithm

Step 0: Randomly initialize weights and biases parameters: $\theta = \{\mathbf{W}^{(1)}, \mathbf{b}^{(1)}, \mathbf{W}^{(2)}, \mathbf{b}^{(2)}, \dots, \mathbf{W}^{(L)}, \mathbf{b}^{(L)}\}$

Step 1: Compute the cost function $\mathcal{L}(\theta)$, which measures how well the model is performing of the dataset.

Step 2: Find the gradients of the cost function with respect to each parameters $\nabla_\theta \mathcal{L}(\theta_t)$

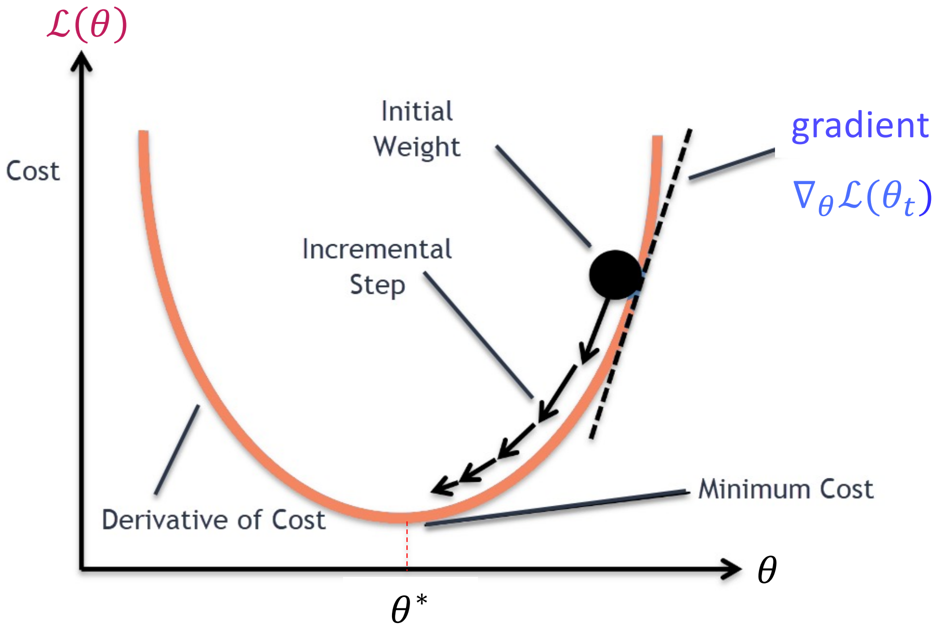
Step 3: Update the parameters by

$$\theta_{t+1} = \theta_t - \eta \cdot \nabla_\theta \mathcal{L}(\theta_t)$$

where η is the learning rate that determines how big the updates should be in each iteration t.

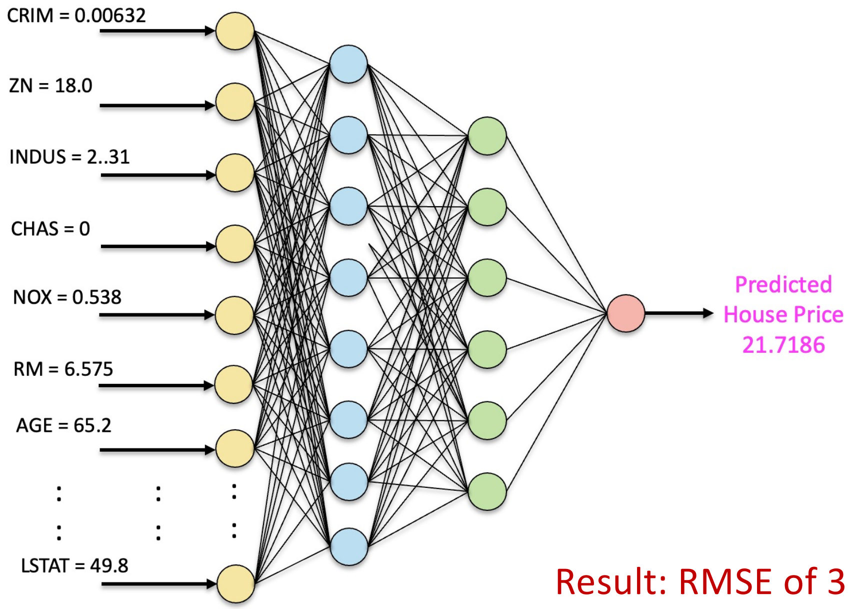
- **Repeat** the above steps 1 to 3, unit the cost is low enough or convergence.

$$\theta^* = \min_\theta \mathcal{L}(\theta) = \min_\theta \frac{1}{N} \sum_{i=1}^{N} \ell\left(y^{(i)}, f_\theta\left(\mathbf{x}^{(i)}\right)\right)$$

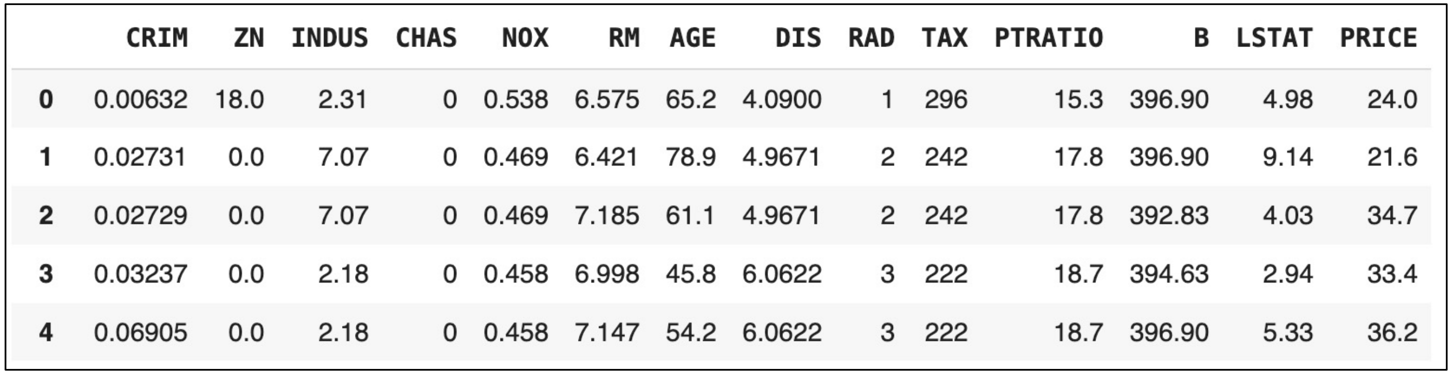


Case Study: Regression
Predicting Boston Housing Prices

- **Dataset:** Boston Housing
- **Input:** 13 features (Crime, Rooms, Age)
- **Output:** Continuous Price.

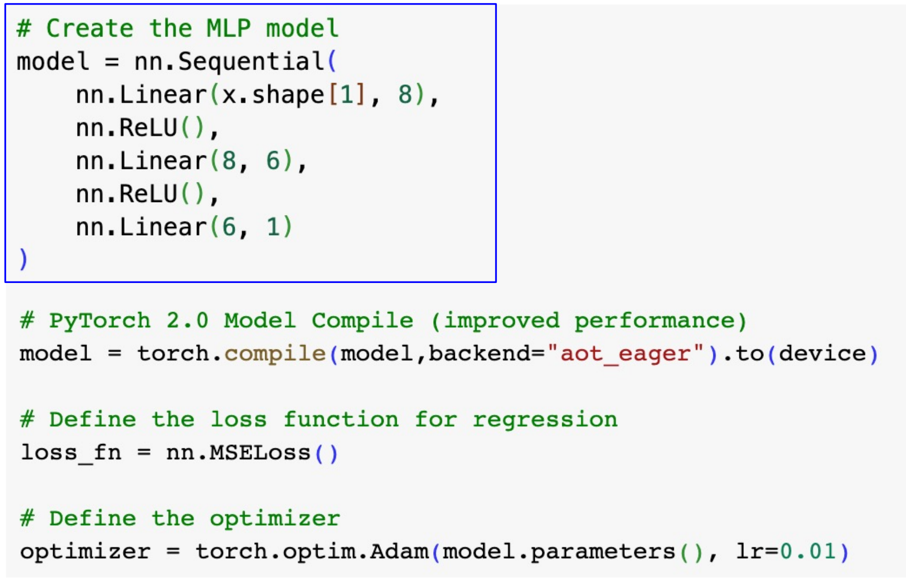


Result: RMSE of 3.4

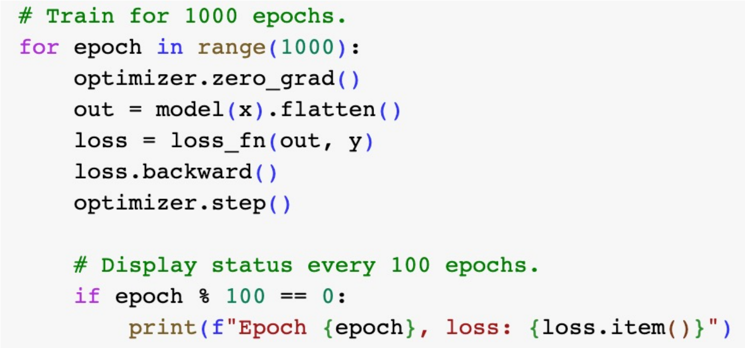


	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	PRICE
0	0.00632	18.0	2.31	0	0.538	6.575	65.2	4.0900	1	296	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0	0.469	6.421	78.9	4.9671	2	242	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0	0.469	7.185	61.1	4.9671	2	242	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0	0.458	6.998	45.8	6.0622	3	222	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0	0.458	7.147	54.2	6.0622	3	222	18.7	396.90	5.33	36.2

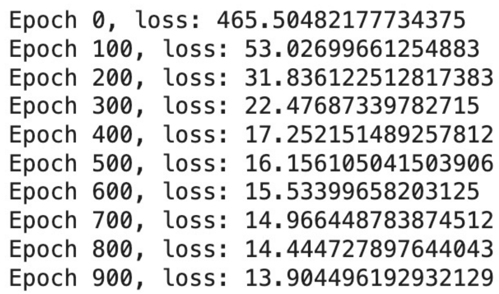
- We define a neural network with inputs equal to the 13 different attributes of houses. These connect to a hidden layer of 8 neurons, which connects to another hidden layer of 6 neurons. Output neuron count must match the input neuron count of the next layer.
- We generally use the Relu activation for hidden layers. The output layer has no activation function since this is a regression task predicting a continuous house prices.



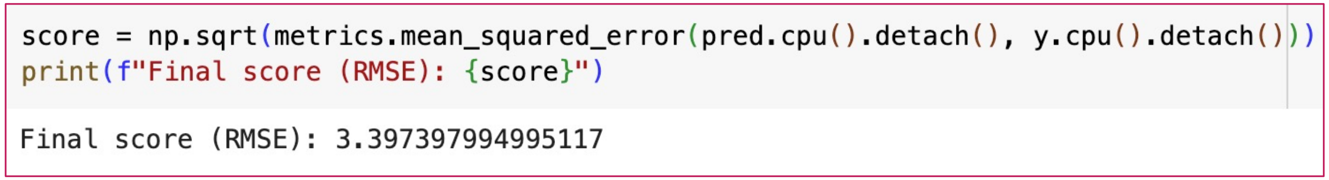
```
# Create the MLP model
model = nn.Sequential(
    nn.Linear(x.shape[1], 8),
    nn.ReLU(),
    nn.Linear(8, 6),
    nn.ReLU(),
    nn.Linear(6, 1)
)

# PyTorch 2.0 Model Compile (improved performance)
model = torch.compile(model,backend="aot_eager").to(device)

# Define the loss function for regression
loss_fn = nn.MSELoss()

# Define the optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
```



```
# Train for 1000 epochs.
for epoch in range(1000):
    optimizer.zero_grad()
    out = model(x).flatten()
    loss = loss_fn(out, y)
    loss.backward()
    optimizer.step()

    # Display status every 100 epochs.
    if epoch % 100 == 0:
        print(f"Epoch {epoch}, loss: {loss.item()}")
```



```
Epoch 0, loss: 465.50482177734375
Epoch 100, loss: 53.02699661254883
Epoch 200, loss: 31.836122512817383
Epoch 300, loss: 22.47687339782715
Epoch 400, loss: 17.252151489257812
Epoch 500, loss: 16.156105041503906
Epoch 600, loss: 15.53399658203125
Epoch 700, loss: 14.966448783874512
Epoch 800, loss: 14.444727897644043
Epoch 900, loss: 13.904496192932129
```



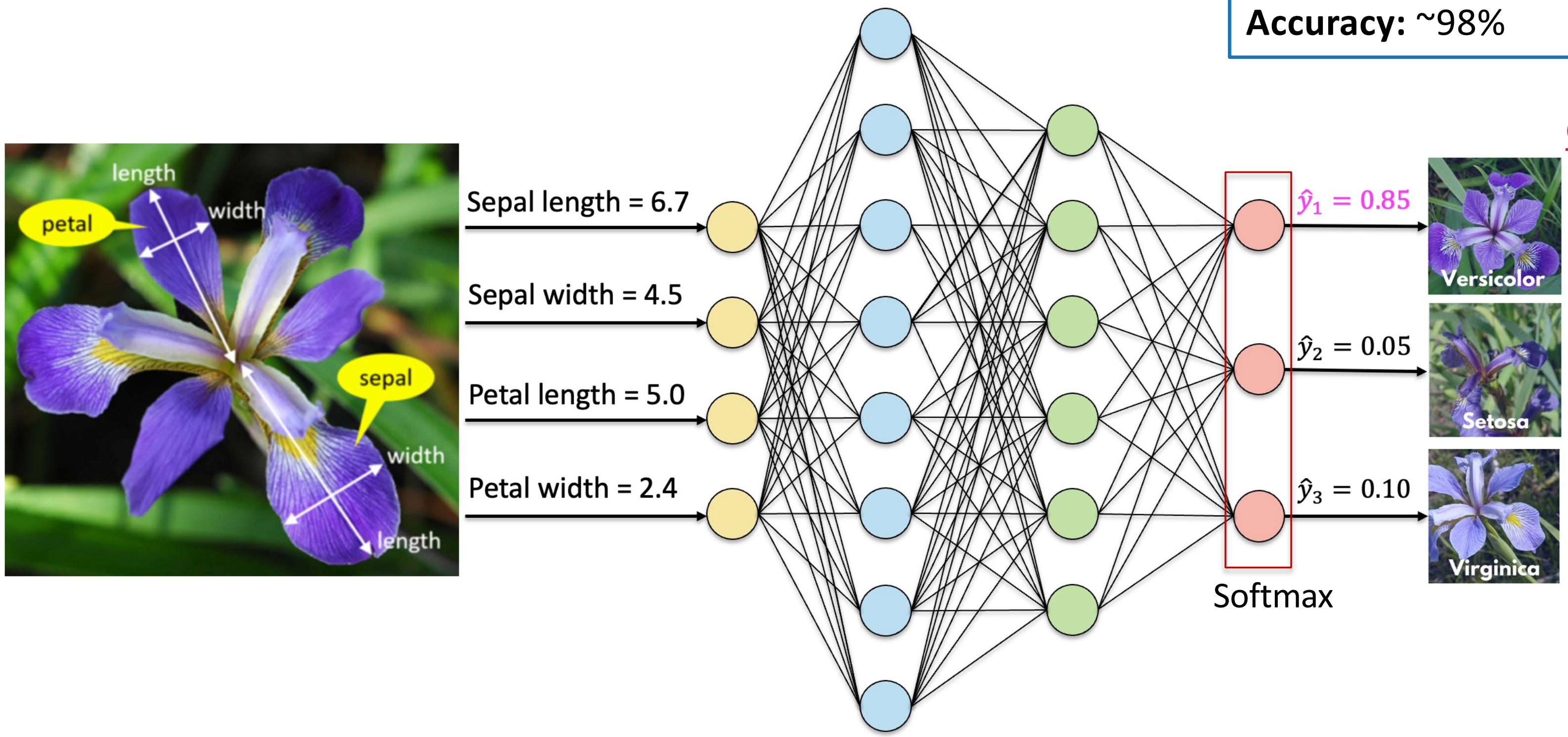
```
score = np.sqrt(metrics.mean_squared_error(pred.cpu().detach(), y.cpu().detach()))
print(f"Final score (RMSE): {score}")
```

```
Final score (RMSE): 3.397397994995117
```

https://colab.research.google.com/drive/1IrjUJ_TuQXVNfPNXze0yuRgLr2ywJGe2?usp=sharing

Case Study: Classification
The Iris Dataset

Dataset: Iris (150 samples).
Input: 4 physical dimensions.
Output: Probability of species
Accuracy: ~98%



Ground Truth

$y_1 = 1$

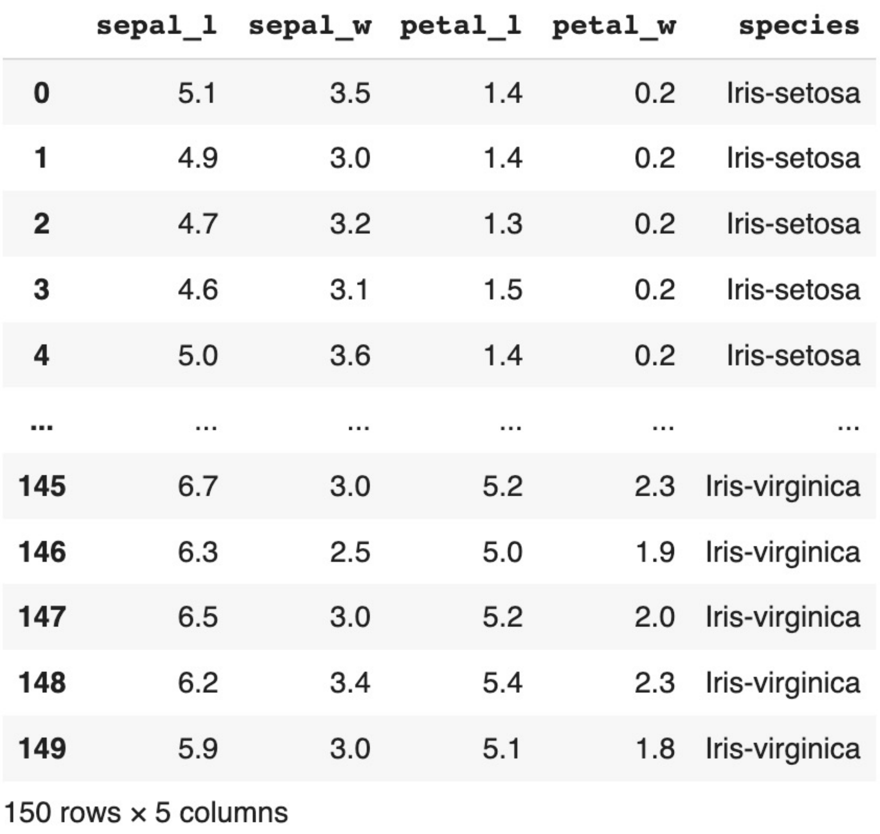
$y_2 = 0$

$y_2 = 0$

The classification labels are one-hot encoder

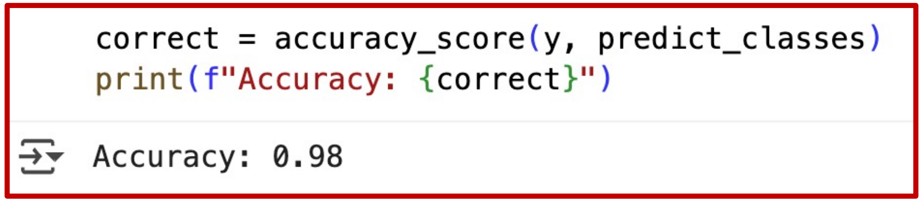
Cross-Entropy Loss = -(1*log(0.85)+0*log(0.05)+0*log(0.10) = -log(0.85) = 0.16

https://colab.research.google.com/drive/1EsatlPEY3fb21qgbMyel4Es_bM8sherJ



	sepal_l	sepal_w	petal_l	petal_w	species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
...	...	...	...	...	...
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

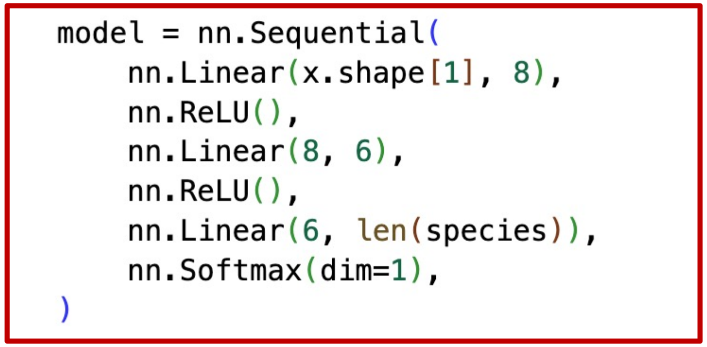
150 rows × 5 columns



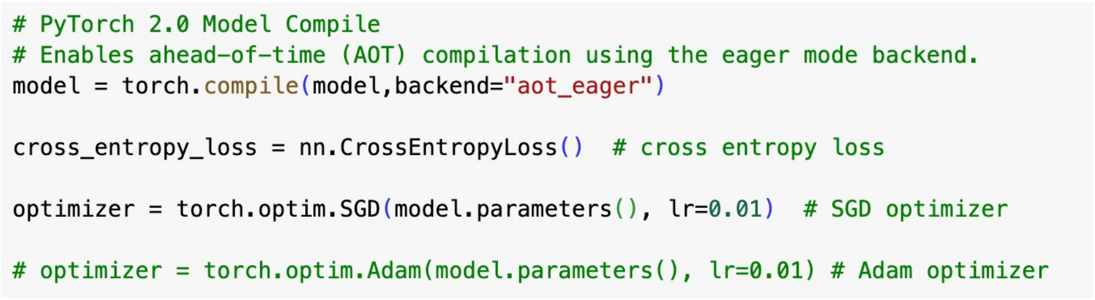
```
correct = accuracy_score(y, predict_classes)
print(f"Accuracy: {correct}")
```

```
Accuracy: 0.98
```

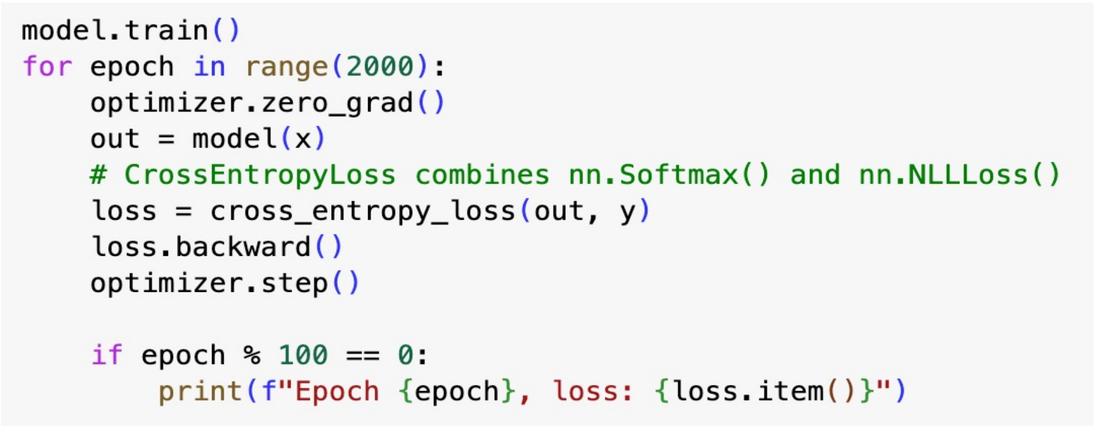
https://colab.research.google.com/drive/1EsatlPEY3fb21qgbMyel4Es_bM8sherJ



```
model = nn.Sequential(
    nn.Linear(x.shape[1], 8),
    nn.ReLU(),
    nn.Linear(8, 6),
    nn.ReLU(),
    nn.Linear(6, len(species)),
    nn.Softmax(dim=1),
)
```



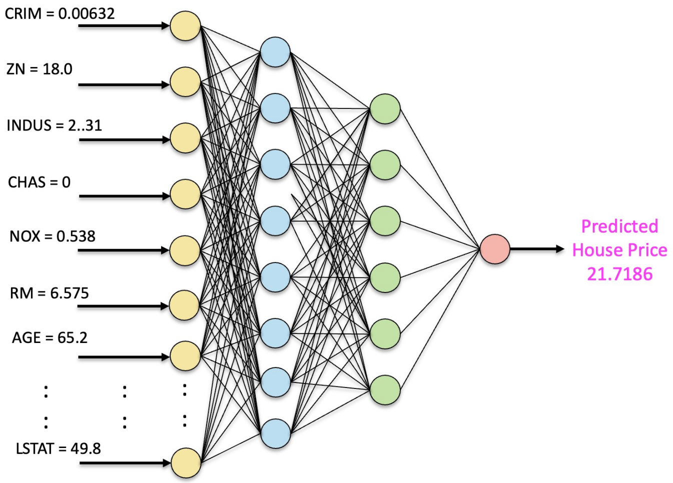
```
# PyTorch 2.0 Model Compile
# Enables ahead-of-time (AOT) compilation using the eager mode backend.
model = torch.compile(model,backend="aot_eager")

cross_entropy_loss = nn.CrossEntropyLoss()  # cross entropy loss

optimizer = torch.optim.SGD(model.parameters(), lr=0.01)  # SGD optimizer

# optimizer = torch.optim.Adam(model.parameters(), lr=0.01) # Adam optimizer
```



```
model.train()
for epoch in range(2000):
    optimizer.zero_grad()
    out = model(x)
    # CrossEntropyLoss combines nn.Softmax() and nn.NLLLoss()
    loss = cross_entropy_loss(out, y)
    loss.backward()
    optimizer.step()

    if epoch % 100 == 0:
        print(f"Epoch {epoch}, loss: {loss.item()}")
```

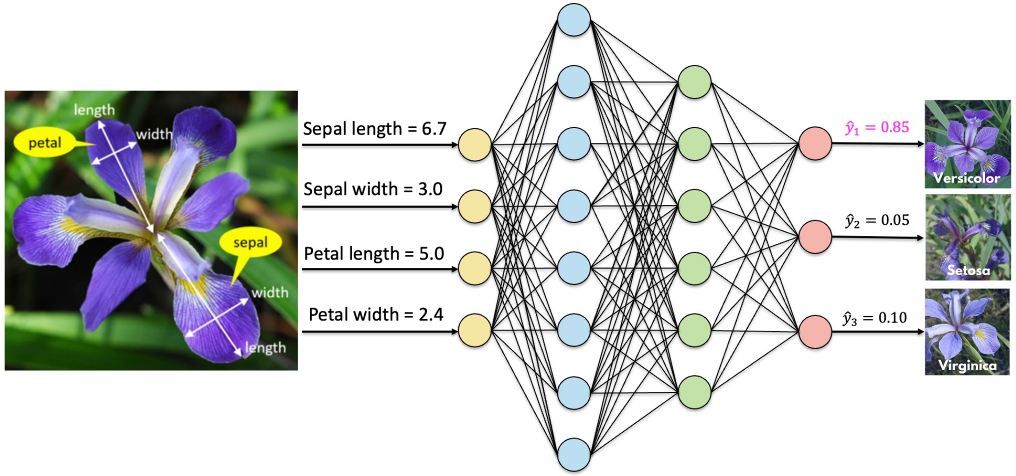
MLPs for Simple Regression and Classification

Regression



- Boston Housing Dataset
  - 13 features and **506** records
  - A 3-Layer MLP (13-8-6-1)
  - **No. of Parameters**: **173**
    - 13*8+8*6+6*1+8+6+1
  - **Performance**: **RMSE = 3.97**

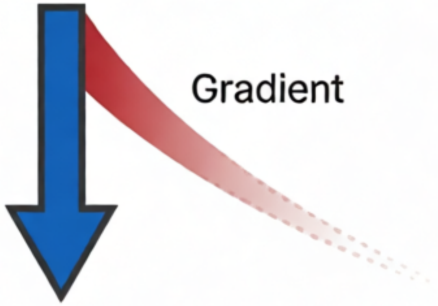
Classification



- Iris Flower Dataset
  - 4 features and **150** records
  - A 3-Layer MLP (4-8-6-3)
  - **No. of Parameters**: **187**
    - 13*8+8*6+6*3+8+6+3
  - **Performance**: **98% Accuracy**

Challenges in Deep Architectures

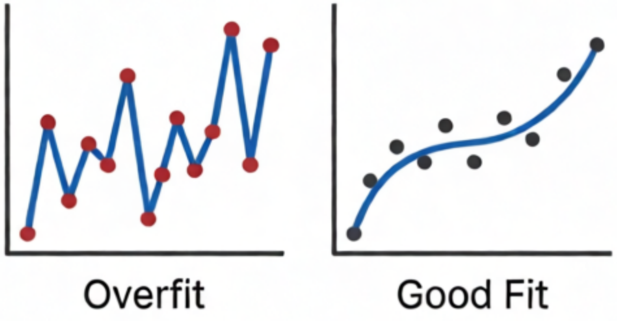
Vanishing Gradient



In deep networks, gradients can shrink exponentially, halting learning.

Fix: Use ReLU activation

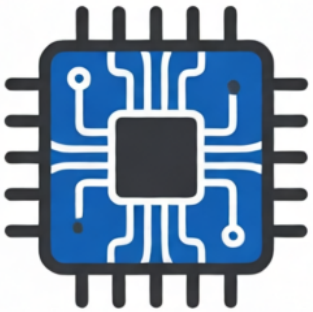
Overfitting



The model memorizes training data but fails on new data.

Fix: Use Dropout, L1/L2 Regularization, Early Stop, Data Augmentation.

Computational Cost



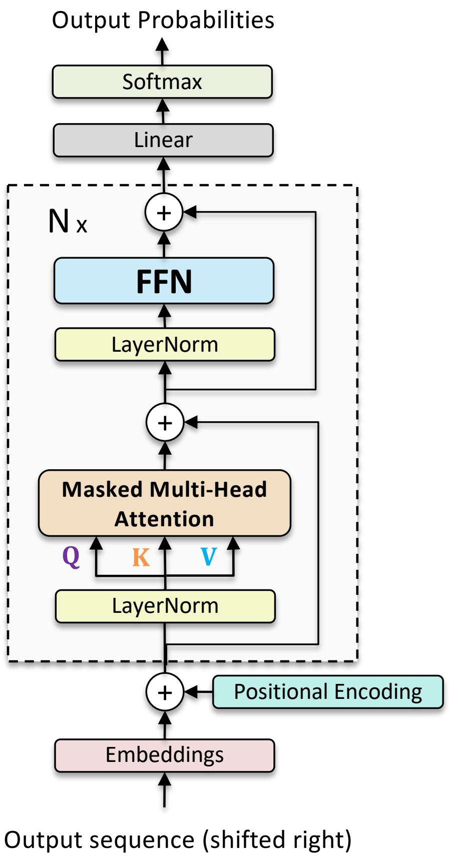
Large matrices require massive parallel processing power.

Fix: GPU acceleration & optimized frameworks (PyTorch/TensorFlow).

The Cornerstone of Deep Learning

- From basic MLPs to Large Language Models (LLMs) such as ChatGPT-5, Gemini 3 Pro, the architecture remains: Layers, Activation, and Backpropagation.

- **MLPs (or FFNs)** paved the way for the AI resolution, proving that machines can learn to approximate any continuous function in our universe.



Output Probabilities
Softmax
Linear
N x
FFN
LayerNorm
Masked Multi-Head Attention
Q
K
V
LayerNorm
Positional Encoding
Embeddings
Output sequence (shifted right)