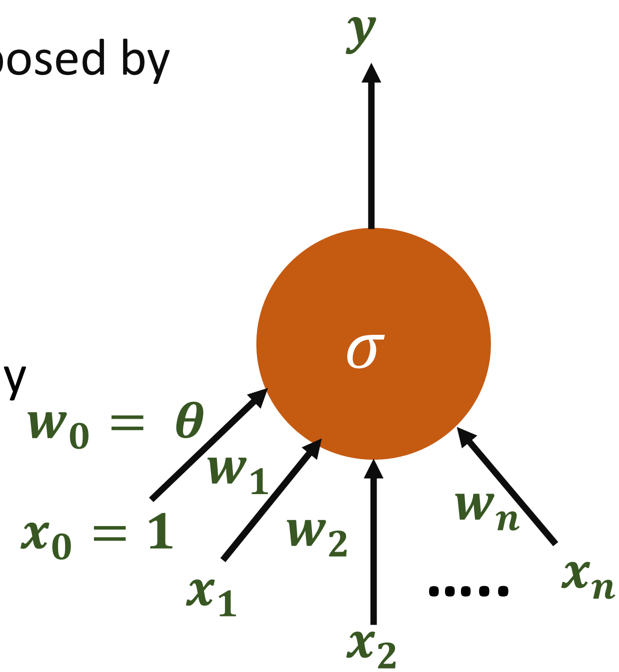
Multi-Layer Perceptron

The Rosenblatt's Perceptron (1957)

- A perceptron is a simple binary classification algorithm, proposed by Frank Rosenblatt.
- It can process non-Boolean inputs
- Different weights can be assigned to each input automatically
- The threshold θ is assigned automatically

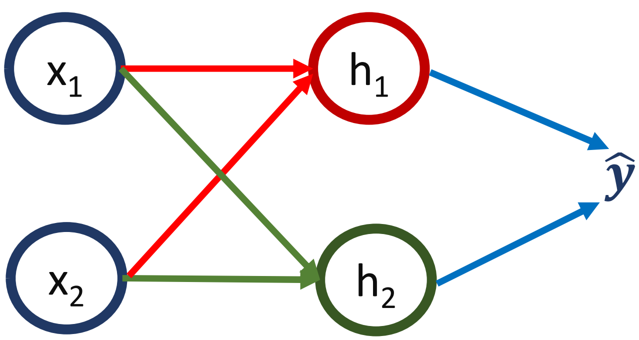
$$y = 1 \;\; if \; \sum_{i=0}^{n} w_i * x_i \geq 0$$

$$= 0 \;\; if \; \sum_{i=0}^{n} w_i * x_i < 0$$



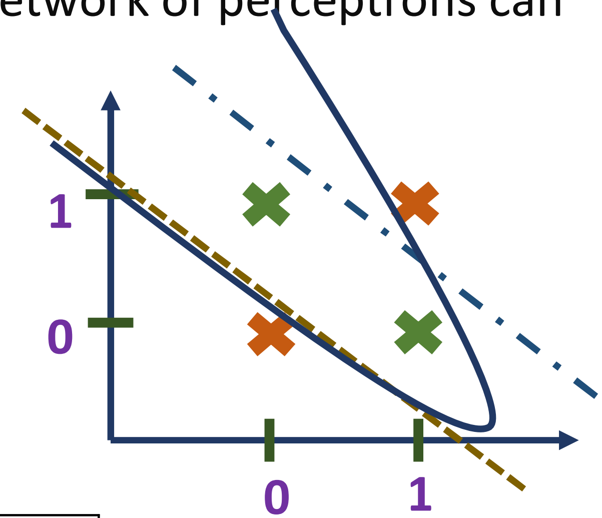
Perceptron

- A single perceptron cannot deal with nonlinear data, however a network of perceptrons can indeed deal with such data.
- Was proved for XOR Logical Function.



$\cancel{\hat{y} = \sigma(w_1x_1 + w_2x_2 + w_0)}$

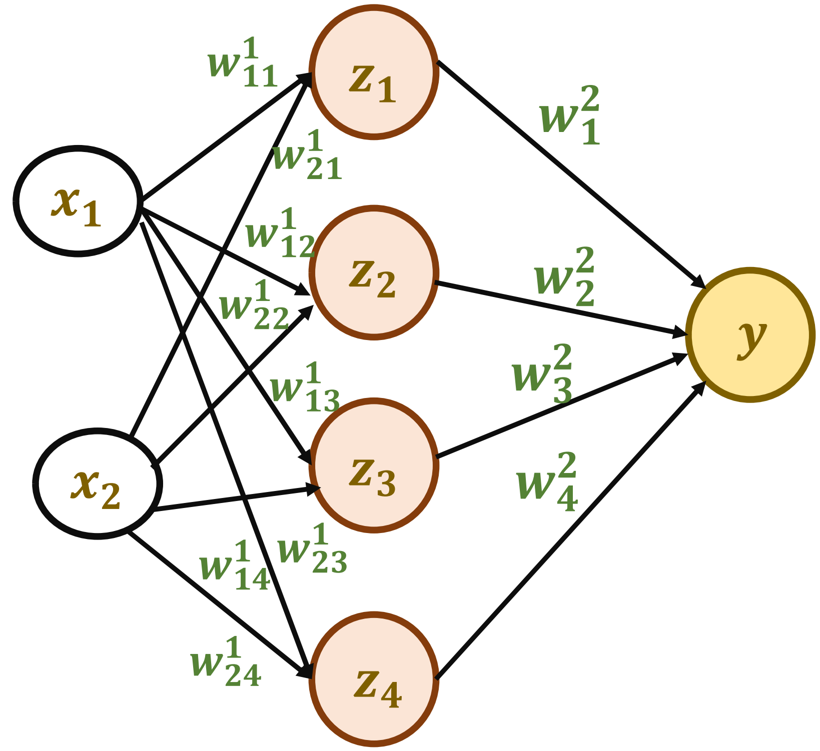
x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0



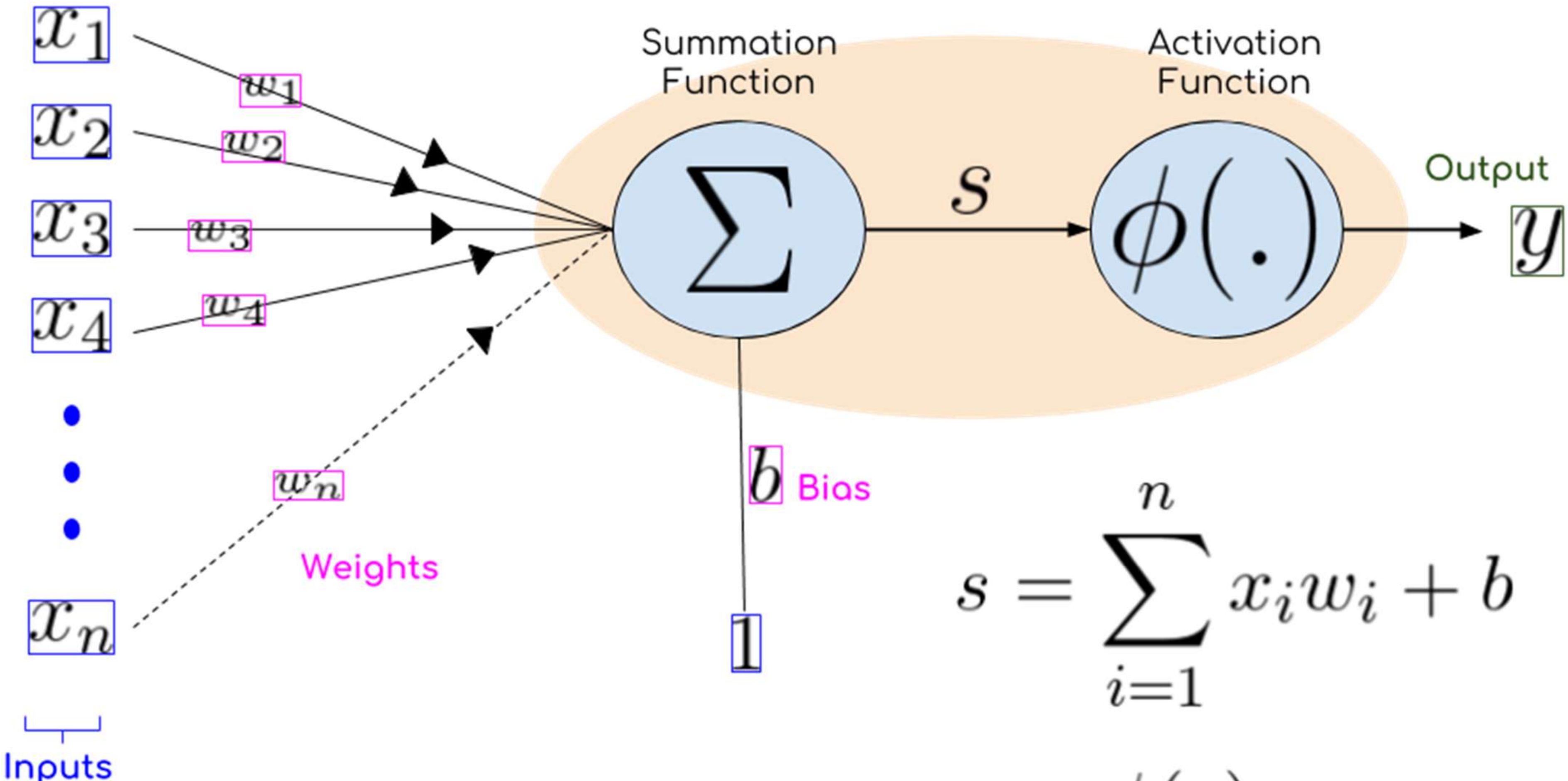
x_1	x_2	h_1	h_2	$\hat{y}$
0	0	0	1	0
0	1	1	1	1
1	0	1	1	1
1	1	1	0	0

Multilayer Perceptrons

- The layer containing the inputs (x_1, x_2) is called the **input layer.**
- The middle layer containing the 4 perceptrons is called the **hidden layer.** The outputs of the 4 perceptrons in the hidden layer are denoted by $(\mathbf{z}_1, \mathbf{z}_2, \mathbf{z}_3, \mathbf{z}_4)$.
- The weights represent the relative importance of each of the nodes to the final decision.
- The final layer containing one output neuron is called the **output layer.**



- An MLP with two or more hidden layers is called a Deep Neural Network



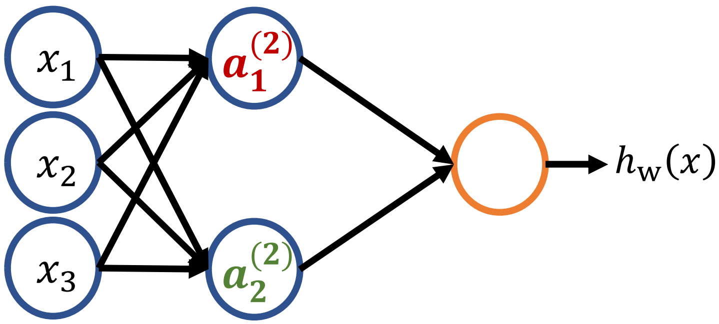
$$s = \sum_{i=1}^{n} x_i w_i + b$$

$$y = \phi(s)$$

ACTIVATION FUNCTIONS

- Activation function performs a complex non-linear transformation of the summed weighted input (neuron)
- Converts linear input signals from perceptron to a linear/non-linear output signal
- Decides whether to activate a node or not
- Must be monotonic, continuously differentiable, and quickly converging
- Squash input data into a narrow range

Neural network – Activation Function

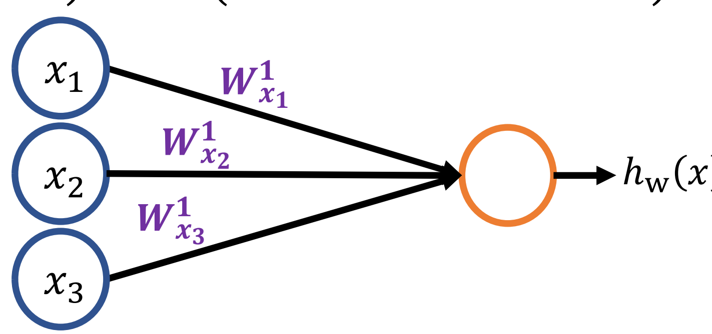


$$a_1^{(2)} = w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2 + w_{13}^{(1)}x_3 + b_1^1$$

$$a_2^{(2)} = w_{21}^{(1)}x_1 + w_{22}^{(1)}x_2 + w_{23}^{(1)}x_3 + b_2^1$$

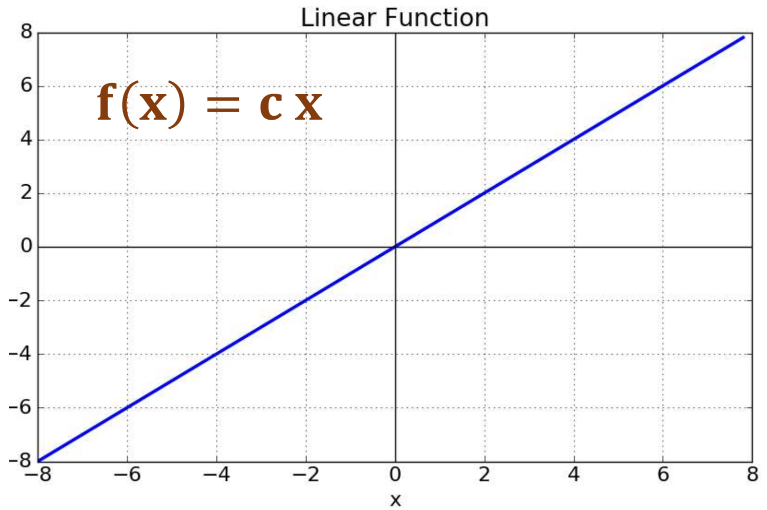
$$h_w(x) = w_{11}^{(2)}a_1^{(2)} + w_{12}^{(2)}a_2^{(2)} + b_1^2$$

$$\begin{aligned} h_w(x) &= w_{11}^{(2)}a_1^{(2)} + w_{12}^{(2)}a_2^{(2)} + b_1^2 \\ &= w_{11}^{(2)}\left(w_{11}^{(1)}x_1 + w_{12}^{(1)}x_2 + w_{13}^{(1)}x_3 + b_1^1\right) + w_{12}^{(2)}\left(w_{21}^{(1)}x_1 + w_{22}^{(1)}x_2 + w_{23}^{(1)}x_3 + b_2^1\right) + b_1^2 \\ &= \left(w_{11}^{(2)}w_{11}^{(1)} + w_{12}^{(2)}w_{21}^{(1)}\right)\mathbf{x_1} + \left(w_{11}^{(2)}\ w_{12}^{(1)} + w_{12}^{(2)}w_{22}^{(1)}\right)\mathbf{x_2} + \left(w_{11}^{(2)}w_{13}^{(1)} + w_{12}^{(2)}w_{23}^{(1)}\right)\mathbf{x_3} \\ &\quad + (b_1^1 + b_2^1 + b_1^2) \\ &= \boldsymbol{W_{x_1}^1}\mathbf{x_1} + \boldsymbol{W_{x_2}^1}\mathbf{x_2} + \boldsymbol{W_{x_3}^1}\mathbf{x_3} + \mathbf{B} \end{aligned}$$



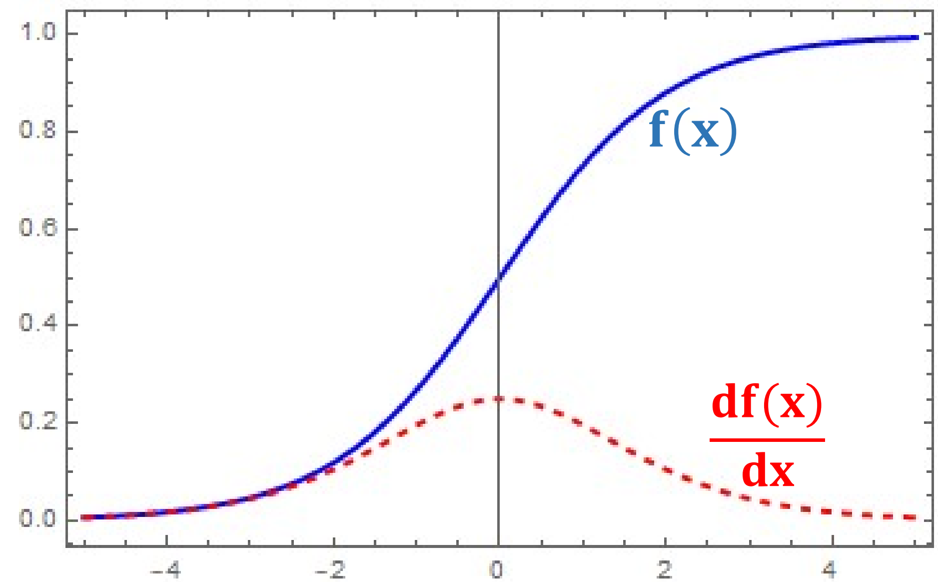
Activation – Linear Function

- **Linear function** means that the output signal is proportional to the input signal to the neuron
- If the value of the constant *c* is 1, it is also called **identity activation function**
- This activation type is used in regression problems
  - E.g., the last layer can have linear activation function, in order to output a real number (and not a class membership)
- Constant gradient with the gradient not depending on the change in the input, $\frac{df(x)}{dx} = c$



Activation - Sigmoid Function

- Squashes the neuron's pre-activation between 0 and 1
- Always positive and bounded
- The output can be interpreted as the firing rate of a biological neuron
  - Not firing = 0; Fully firing = 1
- When the neuron's activation are 0 or 1, sigmoid neurons saturate
  - Gradients at these regions are almost zero (almost no signal will flow)



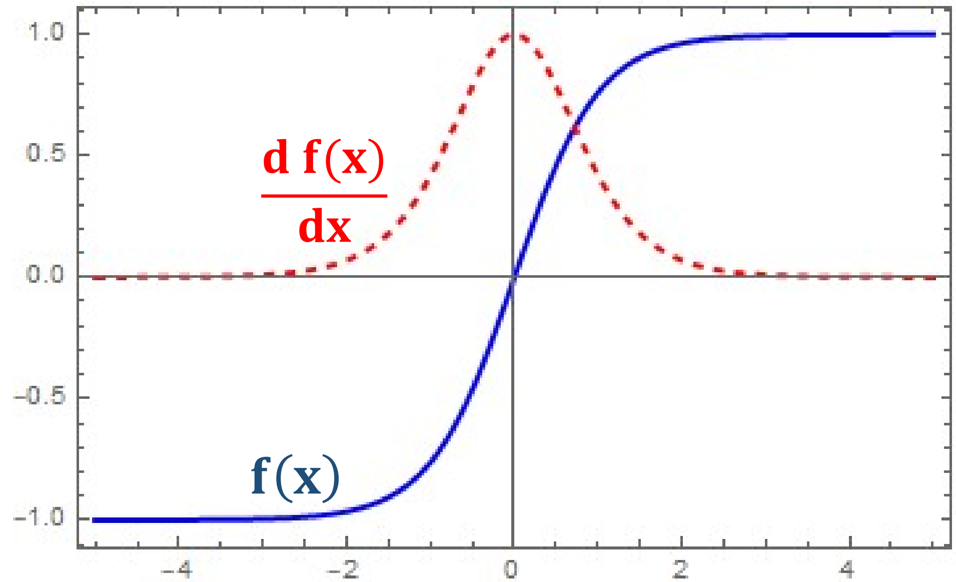
$$\mathbf{f(x)} = \frac{\mathbf{1}}{\mathbf{1 + e^{-x}}}$$

$$\frac{\mathbf{df(x)}}{\mathbf{dx}} = \mathbf{f(x)(1 - f(x))}$$

Activation - Hyperbolic Tangent (tanh)

- Squashes the neuron's pre-activation between -1 and 1
- Can be positive or negative and bounded
- Tanh neurons saturate like sigmoid
- Tanh is a scaled sigmoid:

$$\tanh x = 2\sigma(2x) - 1$$

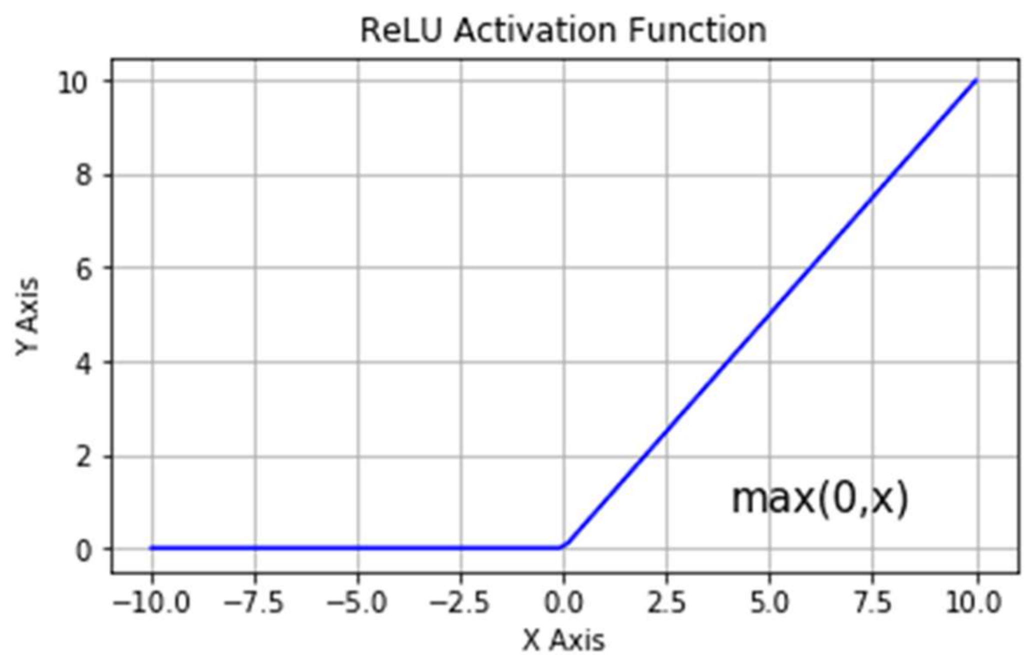


$$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$\frac{d\,f(x)}{dx} = 1 - f(x)^2$$

Activation - Rectified Linear(ReLu)

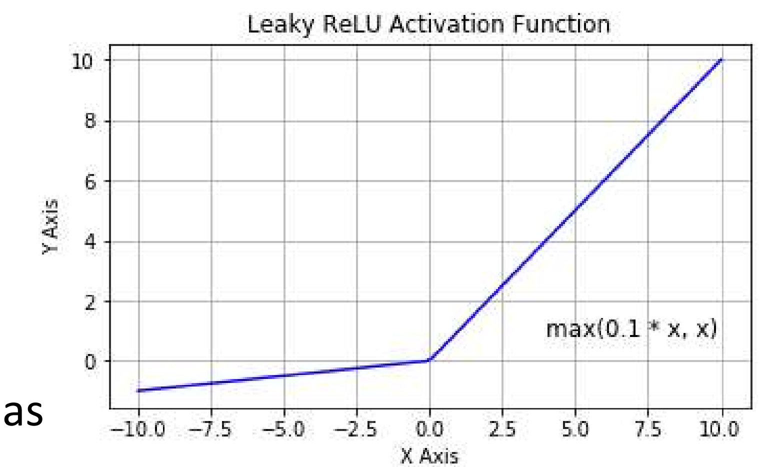
- Bounded below by 0
- Always non-negative
- Not upper bounded
- Most modern deep NNs use ReLU activations
- ReLU is fast to compute, compared to sigmoid and tanh
- Helps prevent the gradient vanishing problem



$$\mathbf{f(x) = relu(x) = max(0, x)}$$

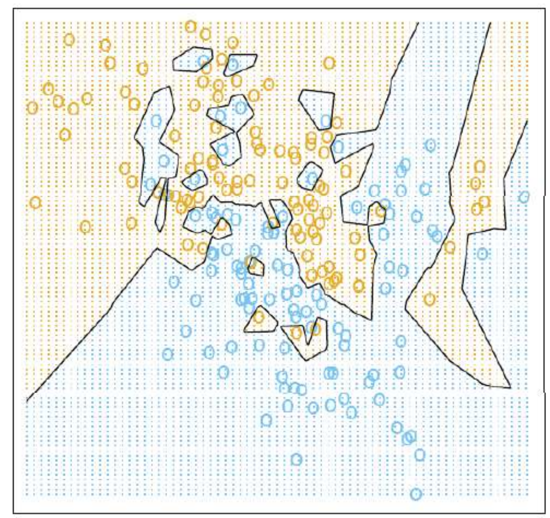
Activation: Leaky ReLU

- The problem of ReLU activations: they can “die”
  - ReLU could cause weights to update in a way that the gradients can become zero and the neuron will not activate again on any data
  - e.g., when a large learning rate is used
- ***Leaky ReLU*** activation function is a variant of ReLU
  - Instead of the function being 0 when $x < 0$, a leaky ReLU has a small negative slope (e.g., α = 0.01, or similar)

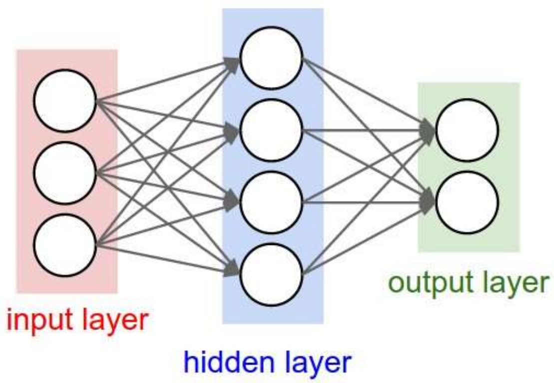


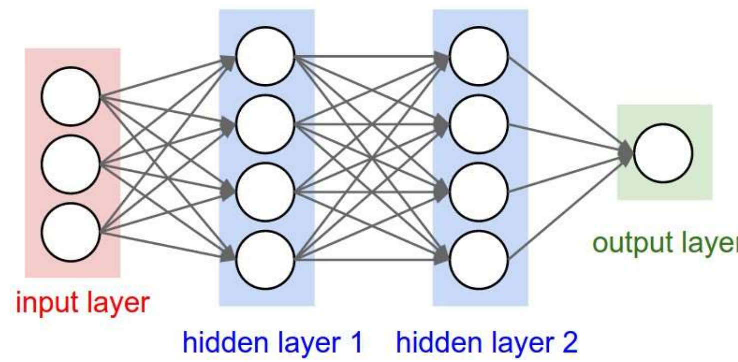
Universal Approximation Theorem

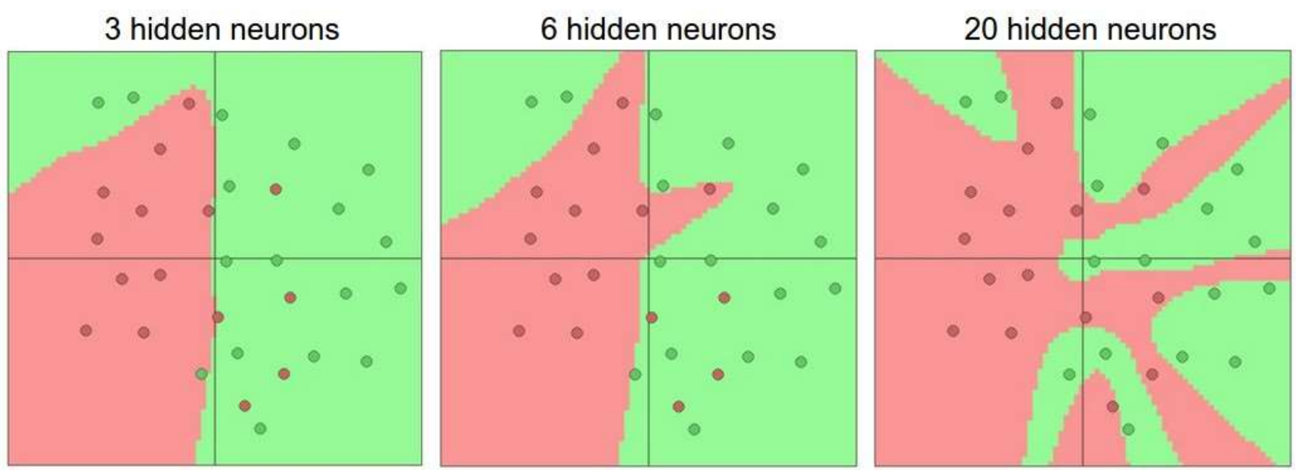
- A neural network with at least one hidden layer and a nonlinear activation function can approximate any continuous function from one finite-dimensional space to another with arbitrary accuracy, provided that the network has enough neurons in the hidden layer.
- NNs use nonlinear mapping of the inputs to the outputs to compute complex decision boundaries
- But then, why use deeper NNs?
  - The fact that deep NNs work better is an empirical observation
  - Mathematically, deep NNs have the same representational power as a one layer NN



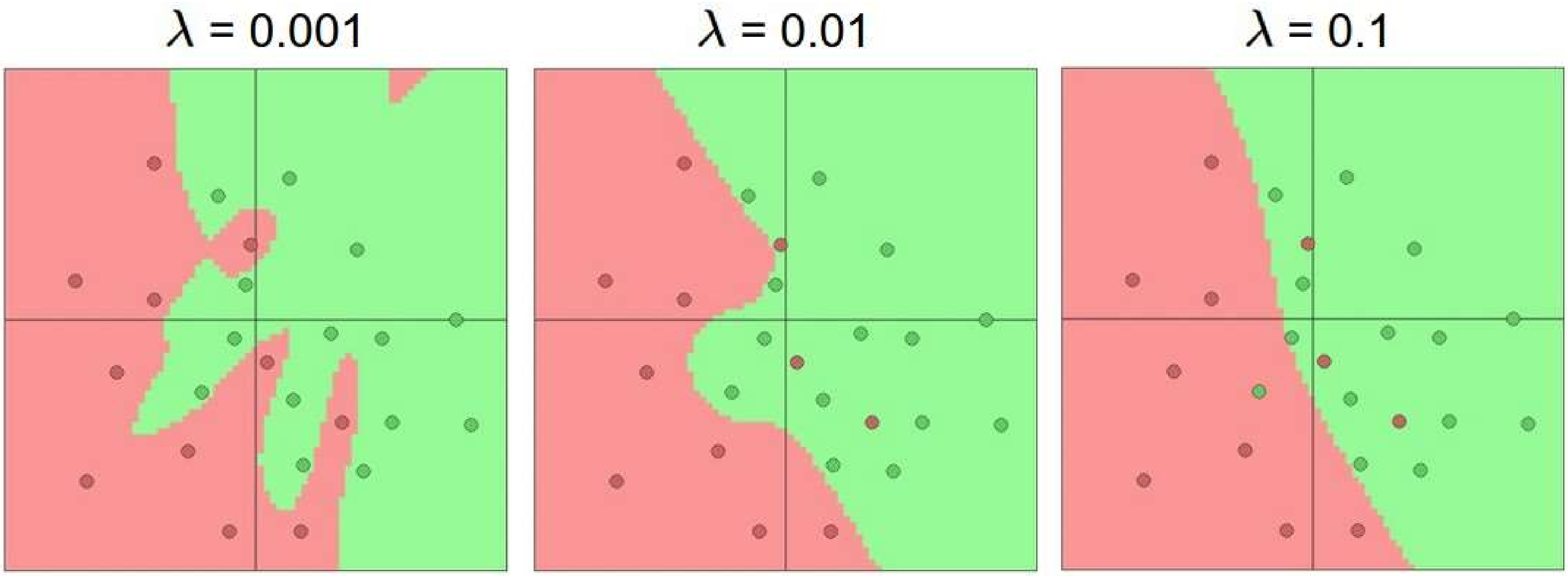
Setting number of layers and their sizes







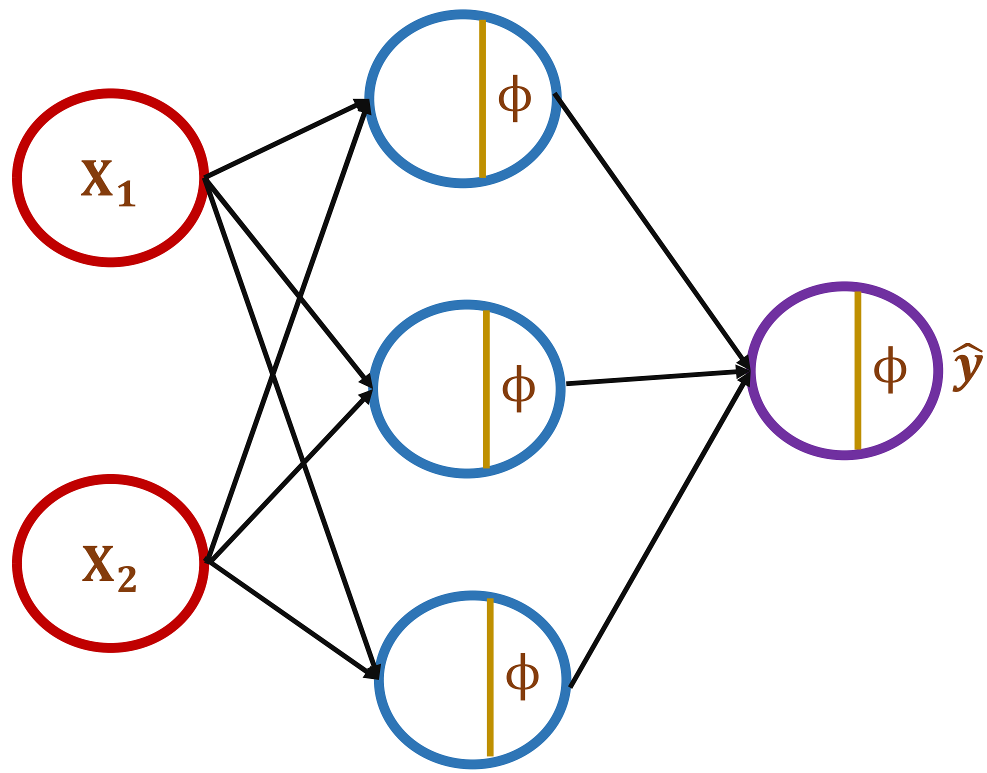
Regularization



Note: You should not be using smaller networks because you are afraid of overfitting. Instead, you should use as big of a neural network as your computational budget allows, and use other regularization techniques to control overfitting.

Training of a Multi-Neuron Network

X_1	X_2	y
7	3	0
4	6	1
9	2	0
3	8	?



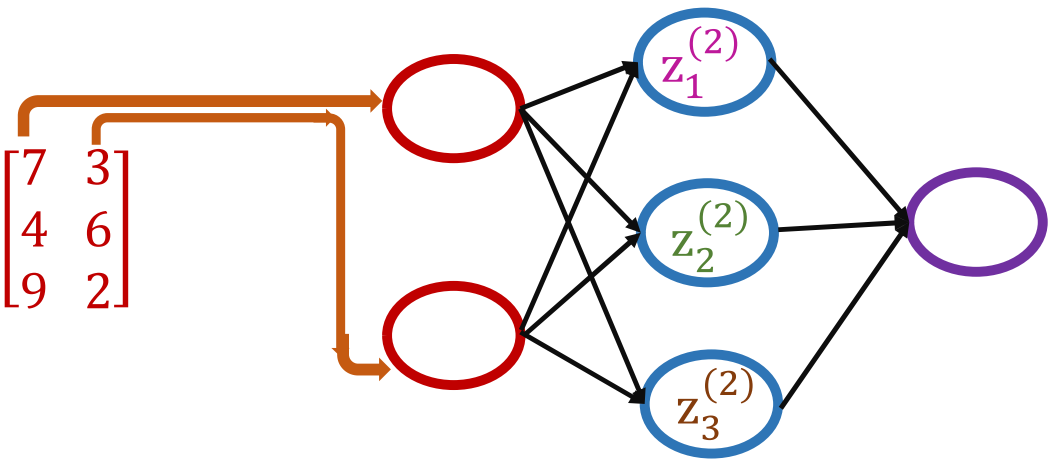
- Problem Statement fixes the input and the output layer.
- The structure of your network, the # of hidden layers and the hidden neurons are the hyperparameters

Training of MLP

- Initialize the network with random weights
- Propagate your input through the network

X_1	X_2	y
7	3	0
4	6	1
9	2	0
3	8	?

$$\Rightarrow \begin{bmatrix} 7 & 3 \\ 4 & 6 \\ 9 & 2 \end{bmatrix}$$



$$z_1^{(2)} = 7\,w_{11}^{(1)} + 3\,w_{12}^{(1)} + b_1^1$$

$$z_2^{(2)} = 7\,w_{21}^{(1)} + 3\,w_{22}^{(1)} + b_2^1$$

$$z_3^{(2)} = 7\,w_{31}^{(1)} + 3\,w_{32}^{(1)} + b_3^1$$

Training of MLP

- Considering $b_1^1, b_2^1, b_3^1 = 0$, for simplicity

$$z_1^{(2)} = 7\,w_{11}^{(1)} + 3\,w_{12}^{(1)}$$

$$z_2^{(2)} = 7\,w_{21}^{(1)} + 3\,w_{22}^{(1)}$$

$$z_3^{(2)} = 7\,w_{31}^{(1)} + 3\,w_{32}^{(1)}$$

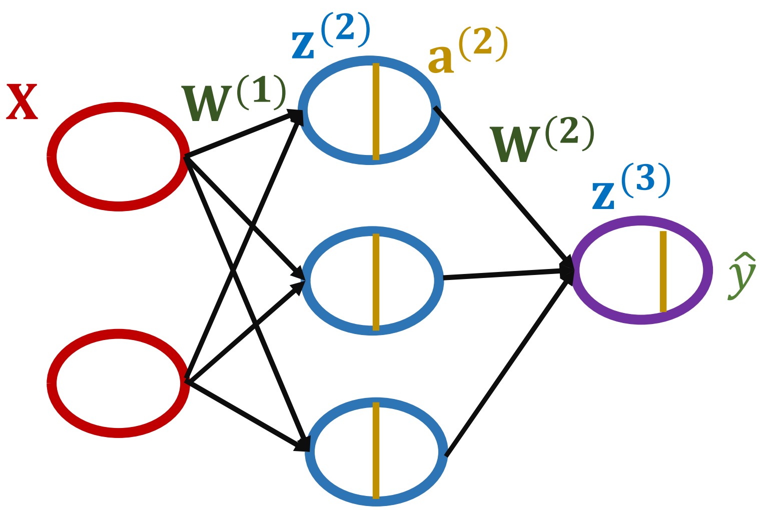
$$[7 \quad 3]\begin{bmatrix} w_{11}^{(1)} & w_{21}^{(1)} & w_{31}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} & w_{32}^{(1)} \end{bmatrix}$$

$$= \begin{bmatrix} 7\,w_{11}^{(1)} + 3\,w_{12}^{(1)} & 7\,w_{21}^{(1)} + 3\,w_{22}^{(1)} & 7\,w_{31}^{(1)} + 3\,w_{32}^{(1)} \end{bmatrix}$$

$$\begin{bmatrix} 7 & 3 \\ 4 & 6 \\ 9 & 2 \end{bmatrix}\begin{bmatrix} w_{11}^{(1)} & w_{21}^{(1)} & w_{31}^{(1)} \\ w_{12}^{(1)} & w_{22}^{(1)} & w_{32}^{(1)} \end{bmatrix} = \begin{bmatrix} \mathbf{7}\,w_{11}^{(1)} + \mathbf{3}\,w_{12}^{(1)} & \mathbf{7}\,w_{21}^{(1)} + \mathbf{3}\,w_{22}^{(1)} & \mathbf{7}\,w_{31}^{(1)} + \mathbf{3}\,w_{32}^{(1)} \\ \mathbf{4}\,w_{11}^{(1)} + \mathbf{6}\,w_{12}^{(1)} & \mathbf{4}\,w_{21}^{(1)} + \mathbf{6}\,w_{22}^{(1)} & \mathbf{4}\,w_{31}^{(1)} + \mathbf{6}\,w_{32}^{(1)} \\ \mathbf{9}\,w_{11}^{(1)} + \mathbf{2}\,w_{12}^{(1)} & \mathbf{9}\,w_{21}^{(1)} + \mathbf{2}\,w_{22}^{(1)} & \mathbf{9}\,w_{31}^{(1)} + \mathbf{2}\,w_{32}^{(1)} \end{bmatrix}$$

$$\mathbf{X} \qquad \mathbf{W}^{(1)} \qquad = \qquad \mathbf{z}^{(2)}$$

Forward Propagation



$$z^{(2)} = X\,W^{(1)} \quad (1)$$

$$a^{(2)} = f\left(z^{(2)}\right) \quad (2)$$

$$z^{(3)} = a^{(2)}W^{(2)} \quad (3)$$

$$\hat{y} = f\left(z^{(3)}\right) \quad (4)$$

Forward Propagation

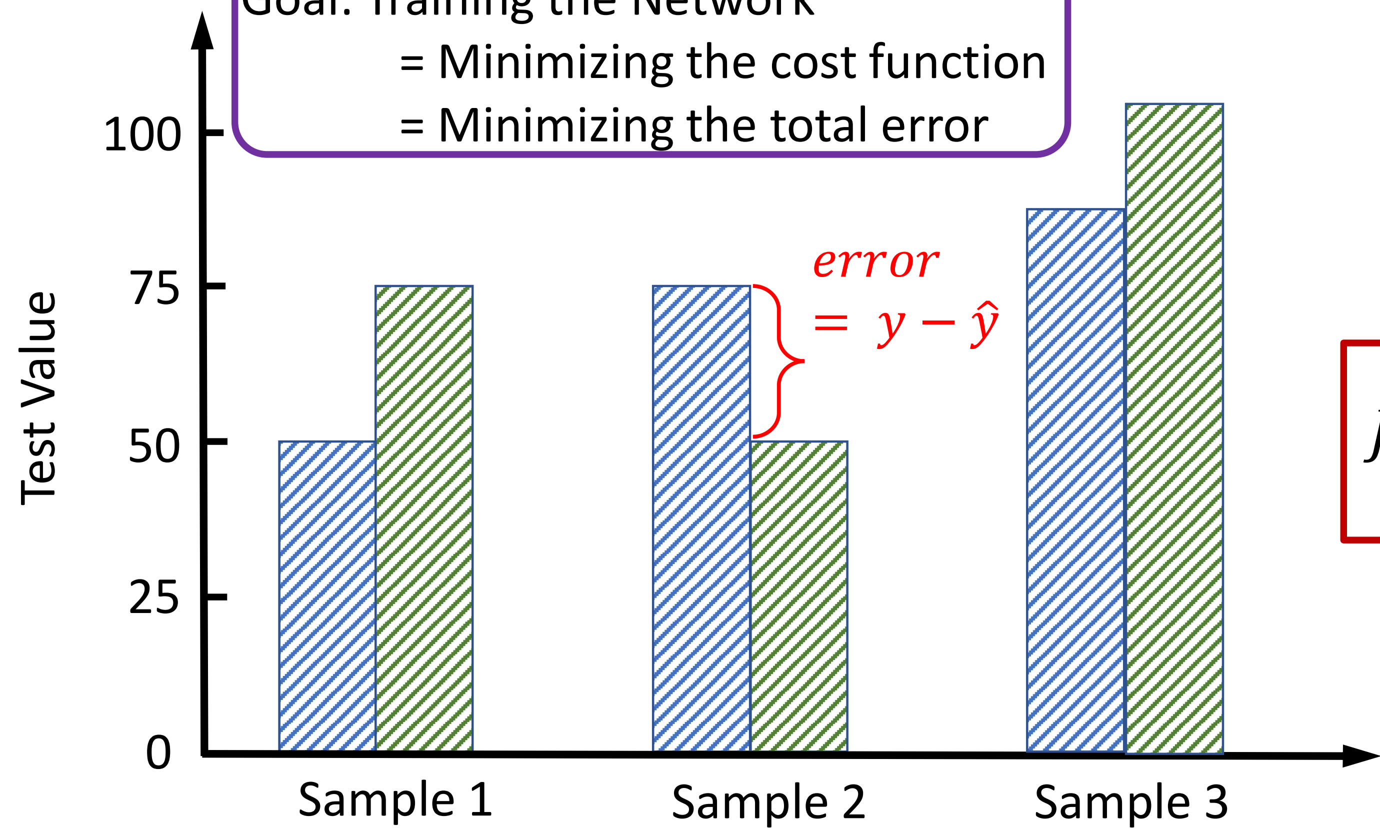
Training - MLP

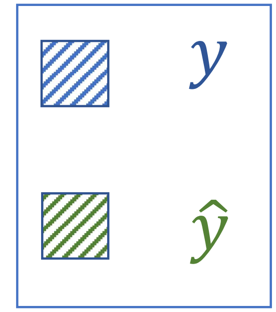
- **Step 1** – We initialized the network with random weights.
- **Step 2** – Perform forward propagation and determine $\hat{y}$.
- **Step 3** – Determine the Loss Function, $|y - \hat{y}|$.
- **Step 4** – Do backward propagation and determine change in weights.
- **Step 5** – Update all weights in all layers.
- **Step 6** – Repeat Steps 2 -5 until convergence.

Loss Function

- Used to capture the difference between the actual and predicted values

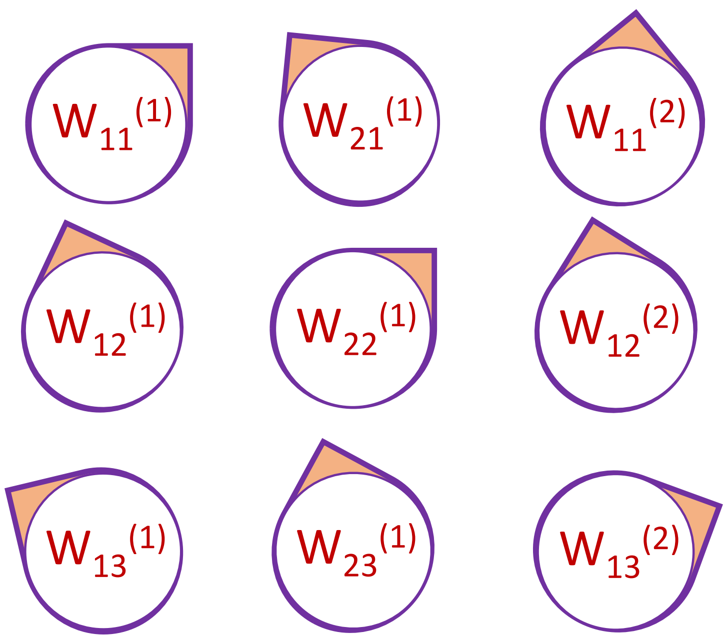
Goal: Training the Network
= Minimizing the cost function
= Minimizing the total error

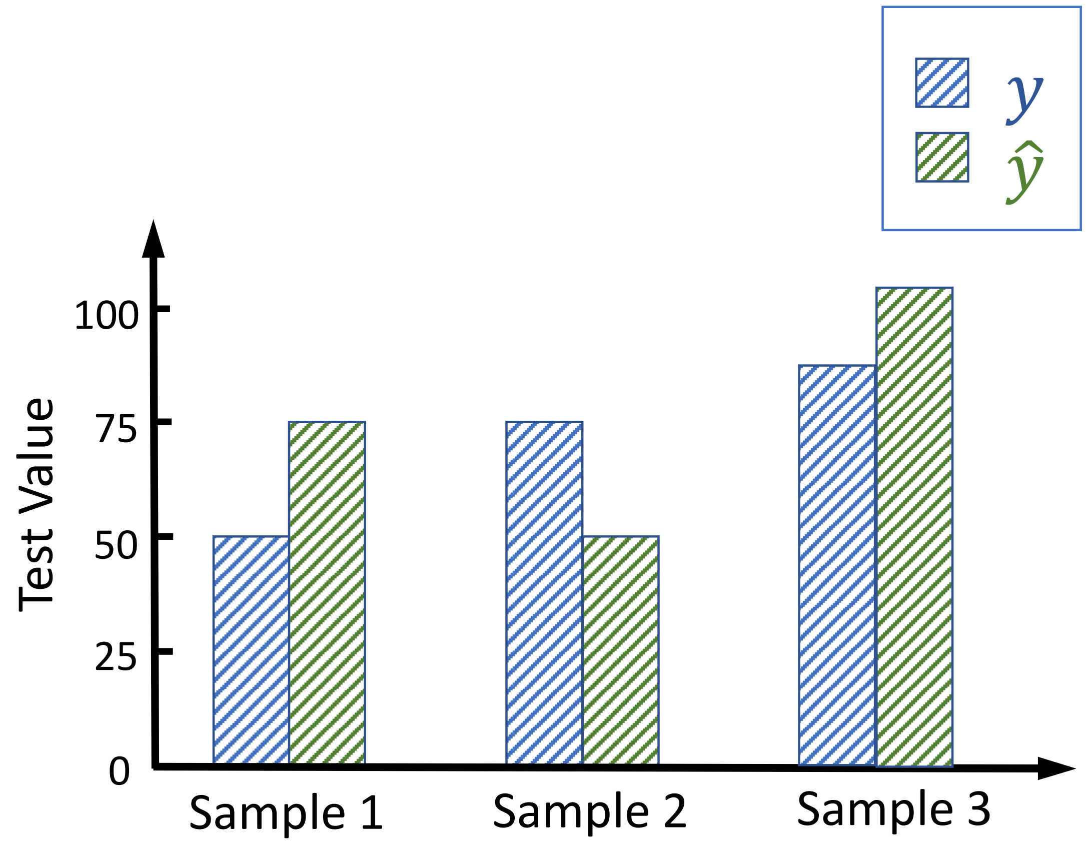




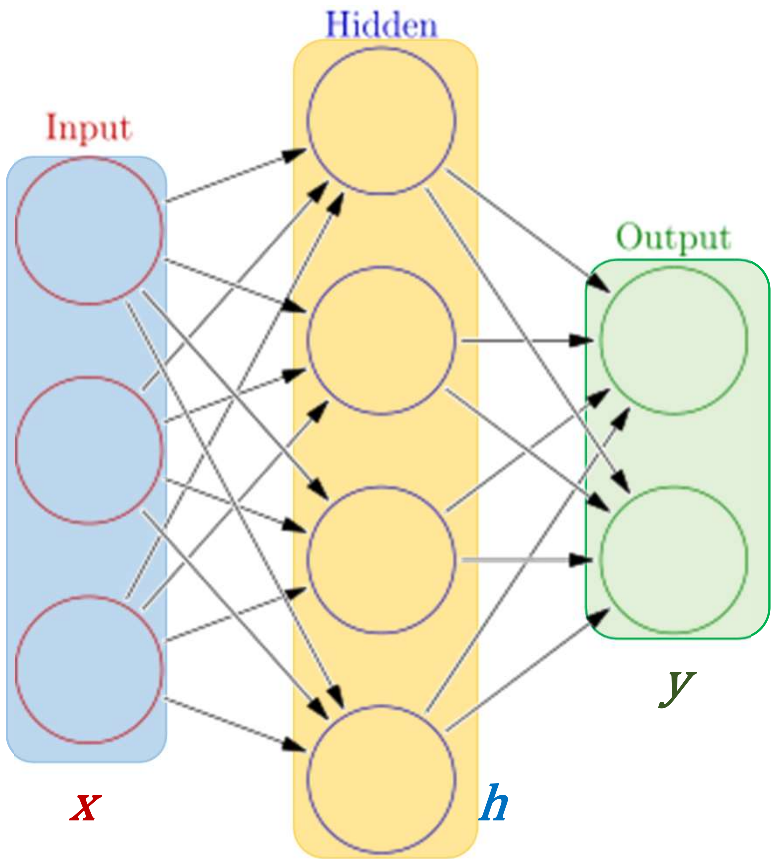
$$J = \frac{1}{2}\sum_{i}(y_i - \hat{y}_i)^2$$

Loss Function





Elements of Neural Networks



$$\mathbf{hidden\ layer\ h} = \boldsymbol{\sigma}(\mathbf{w_i^1 x_i} + \mathbf{w_0})$$

$$\mathbf{output\ layer\ y} = \boldsymbol{\sigma}(\mathbf{w_i^2 x_i} + \mathbf{w'_0})$$

- $\boldsymbol{\sigma}$ → Activation Functions
- $\boldsymbol{w_i^{1,2}}$ → Weights
- $\boldsymbol{w_0}, \boldsymbol{w'_0}$ → Biases

Number of learnable parameters:

$[3 \times 4] + [4 \times 2] = 20$ **weights**

$4 + 2 = 6$ **biases**

$\Longrightarrow$ **26 learnable parameters**

Summary

- An MLP can be seen as a composition of multiple linear models combined

- A single hidden layer MLP with sufficiently large number of hidden units can approximate any function (Hornik, 1991)

- NN can be of different architectures:
  - One hidden layer with m nodes and a single output (e.g. binary classification or single-valued regression)
  - One hidden layer with m nodes and multiple outputs (e.g. multi-class or multi-label classification)
  - Multiple hidden layers with one/multiple outputs