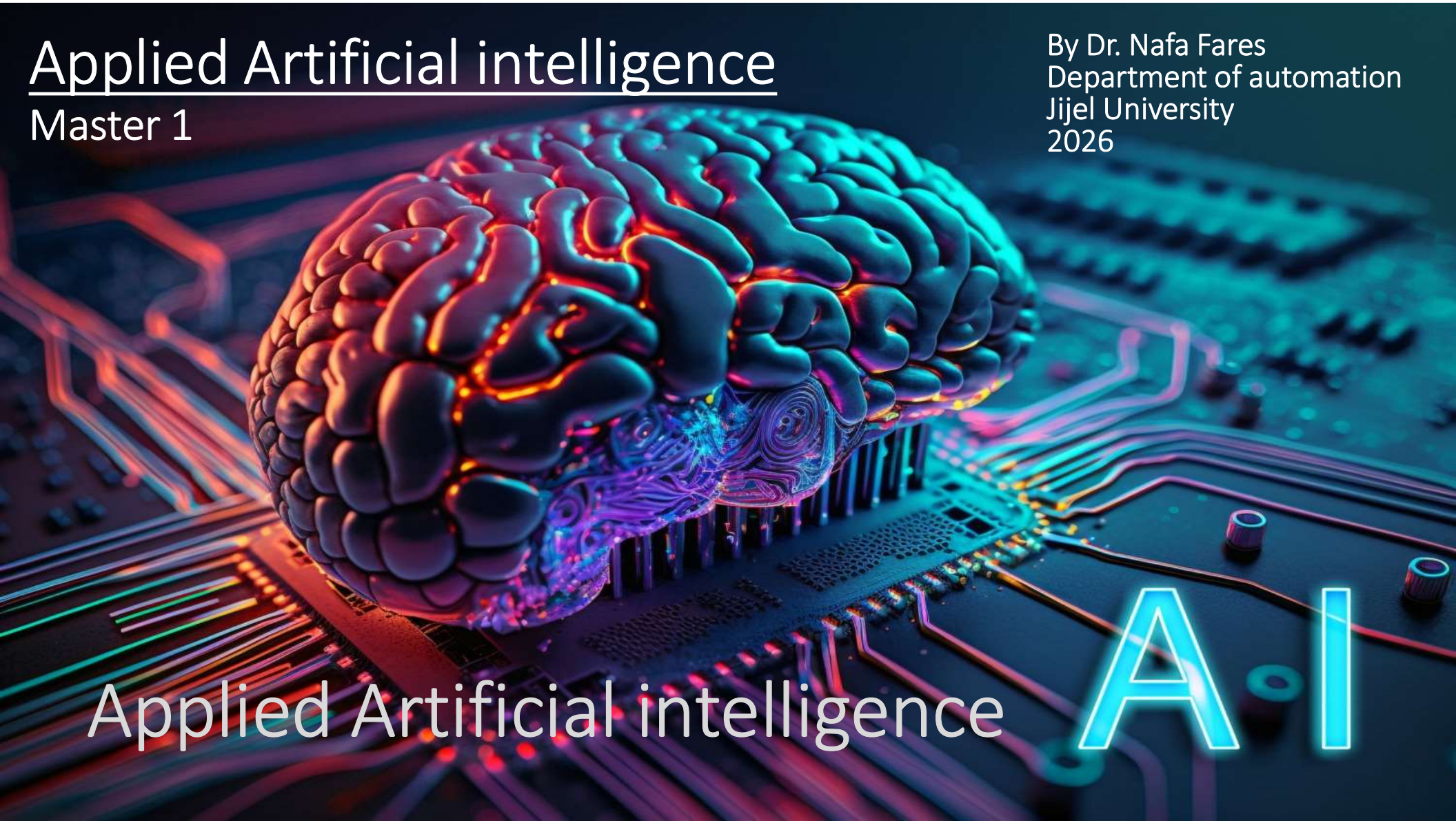
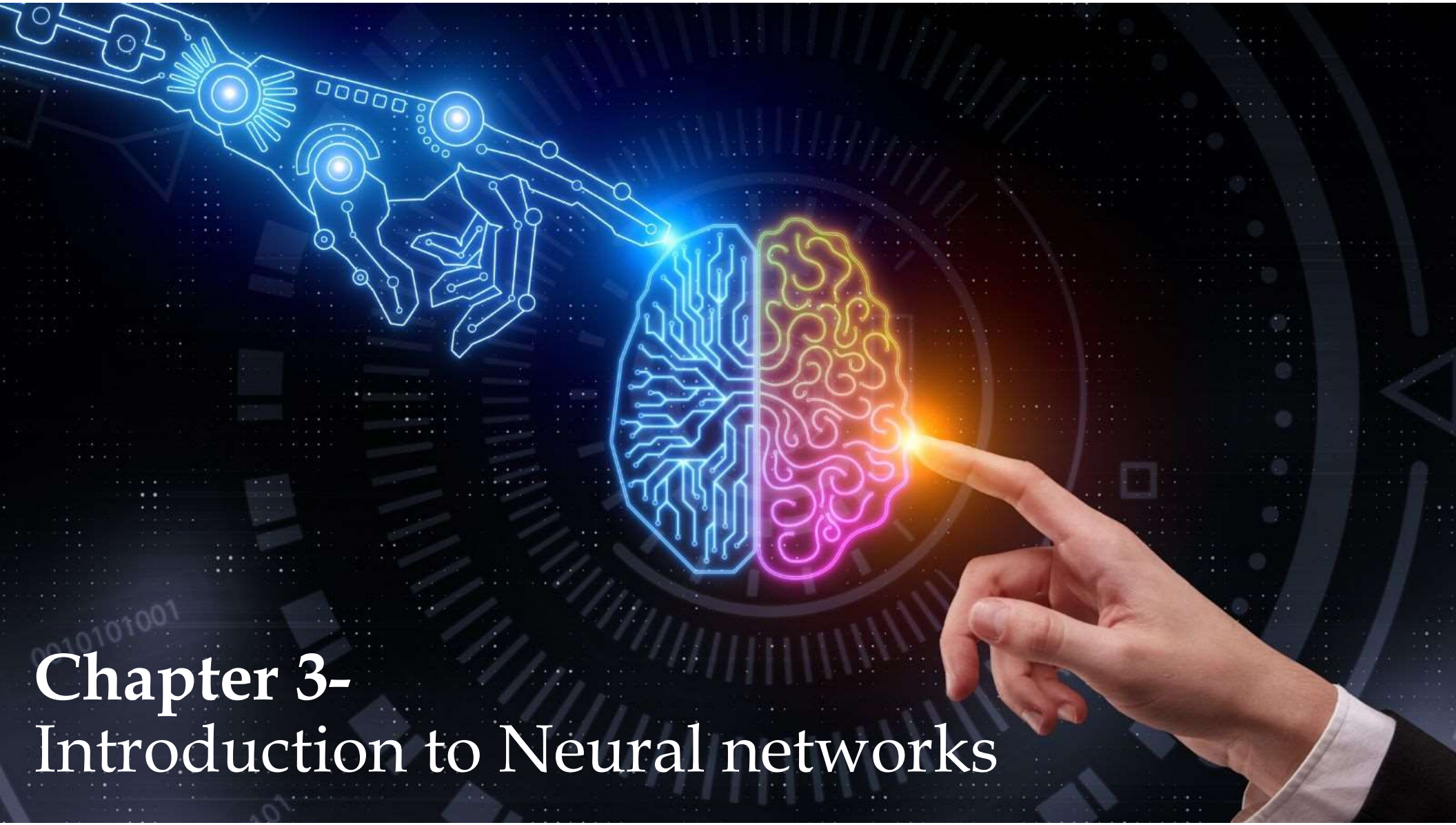
Applied Artificial intelligence

Master 1

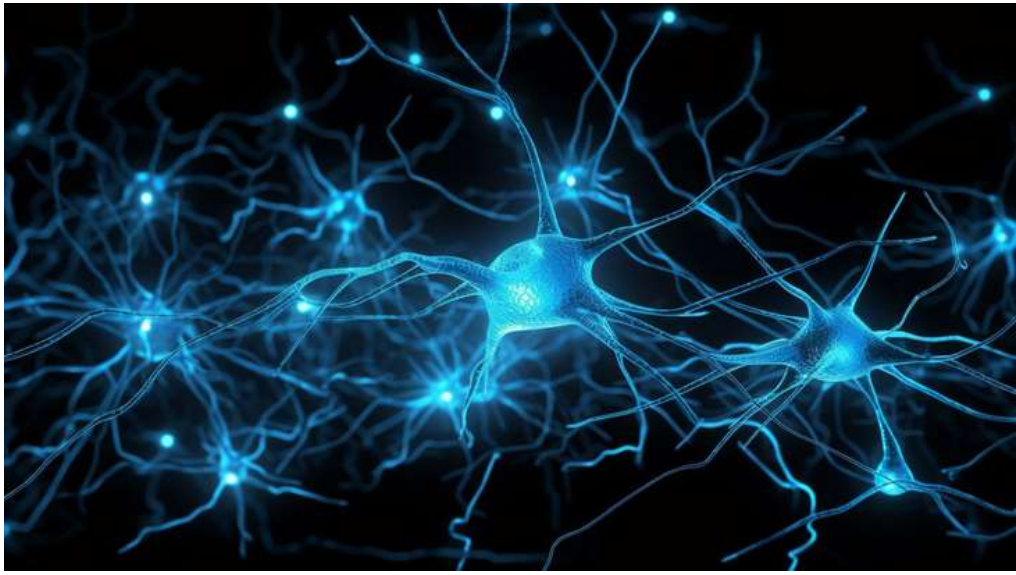
By Dr. Nafa Fares
Department of automation
Jijel University
2026





Chapter 3-
Introduction to Neural networks

1. Neural Networks



- Neural Networks are networks of interconnected neurons, for example in human brains.
- *Artificial Neural Networks* are highly connected to other neurons, and performs computations by combining signals from other neurons.
- Outputs of these computations may be transmitted to one or more other neurons.
- The neurons are connected together in a specific way to perform a particular task.

A neural network is a function.

- It consists of basically:

a. Neurons: which pass input values through functions and output the result.
b. Weights: which carry values (real-number) between neurons.

- Neurons can be categorized into layers:

a. Input Layer
b. Hidden Layer
c. Output Layer

2. Neurophysiology

The human nervous system can be divided into three stages:

a. Receptors:

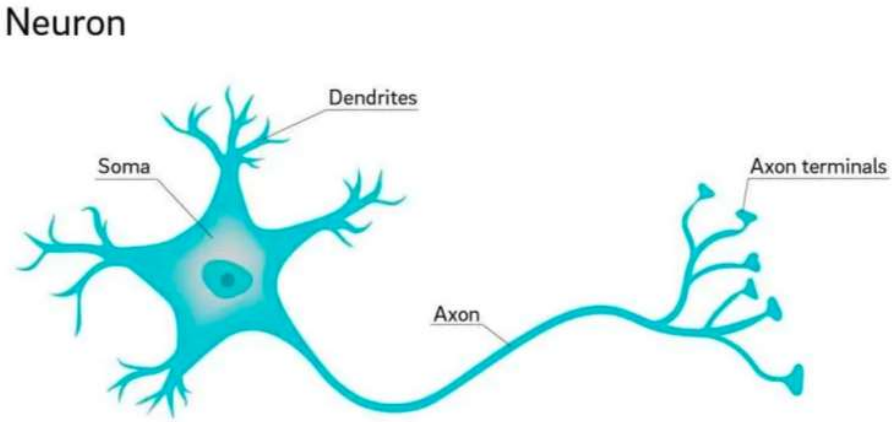
- Convert stimuli from the external environment into electrical impulses
- Rods and Cones of eyes,
- Pain, touch, hot and cold receptors of skin.

b. Neural Net:

- Receive information, process it and make appropriate decisions.
- Brain

c. Effectors:

- Convert electrical impulses generated by the neural net (brain) into responses to the external environment.
- Muscles and glands, speech generators.



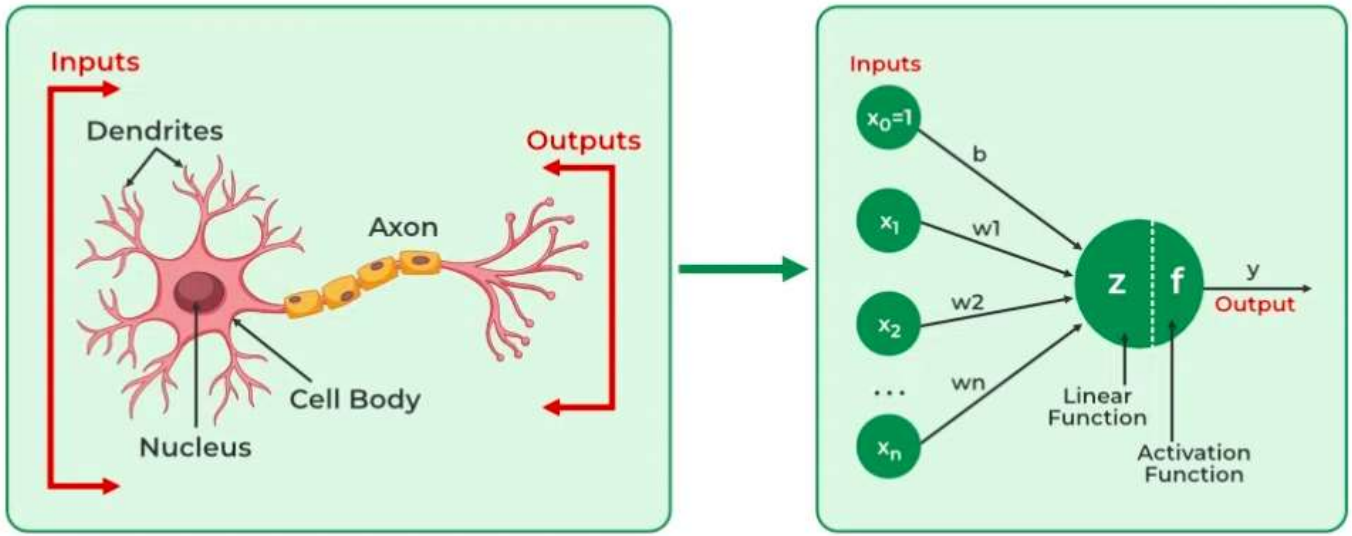
3. Basic Components of Biological Neurons

The basic components of a biological neuron are:

- Cell Body (Soma) processes the incoming activations and converts them into output activations.
- Neuron Nucleus contains the genetic material (DNA).
- Dendrites form a fine filamentary bush each fiber thinner than an axon.
- Axon: Long thin cylinder carrying impulses from soma to other cells
- Synapses: The junctions that allow signal transmission b/w the axons and dendrites.

Computation in Biological Neurons

- Incoming signals from synapses are summed up at the soma.
- On crossing a threshold, the cell fires generating an action potential in the axon hillock region.



4. The Perceptron Model

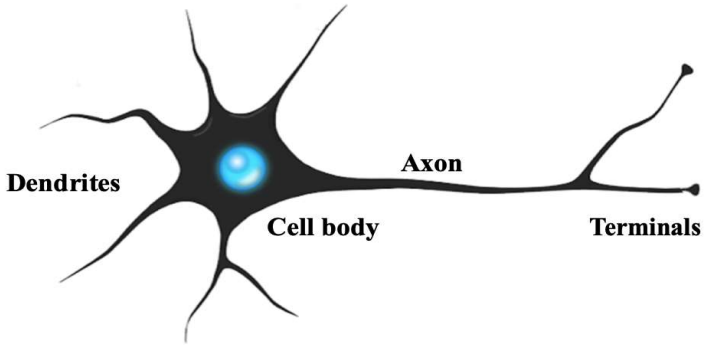
- Motivated by the biological neuron.
- A perceptron is a computing element where inputs are associated with the weights and the cell having a threshold value.

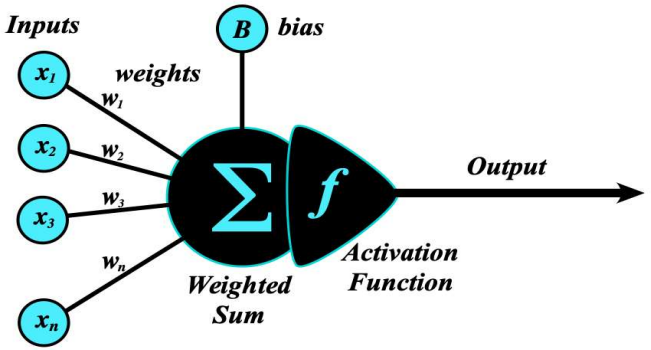
$$y = \begin{cases} 1, & \text{if } \sum w_i x_i > \text{threshold} \\ 0 & \text{otherwise} \end{cases}$$

Rewrite Σ wi.xi as w.x

- Replace threshold = -b
- **b:** Bias, a prior inclination towards some decision.

$$y = \begin{cases} 1, & \text{if } w.x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

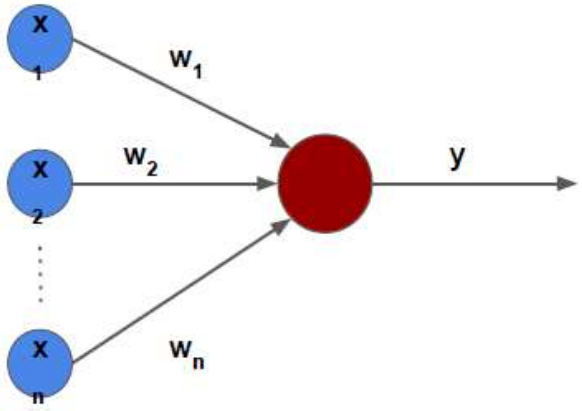




A simple decision via Perceptron

Whether should you go to watch movie this weekend?

- The decision variables are:
  - Is there any extra lecture this weekend? (x_1)
  - Does your friend want to go with you? (x_2)
  - Do you have pending assignments due on the weekend? (x_3)



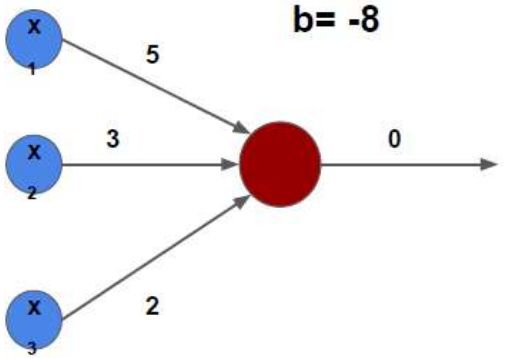
4. The Perceptron Model

A simple decision via Perceptron

Whether should you go to watch movie this weekend?

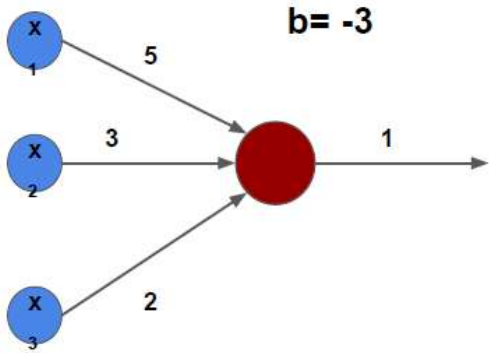
The decision variables are:

- Is there any extra lecture this weekend? ($x_1=1$)
- Does your friend want to go with you? ($x_2=0$)
- Do you have pending assignments due on the weekend? ($x_3=1$)

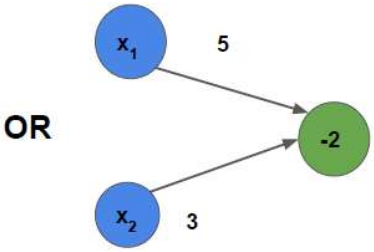


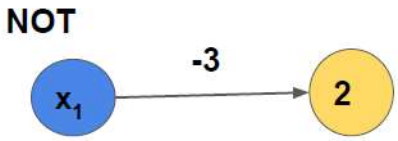
The decision variables are:

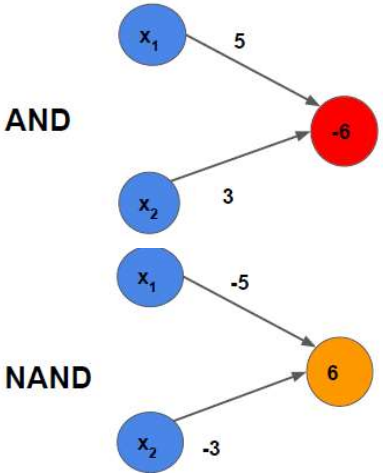
- Is there any extra lecture this weekend? ($x_1=1$)
- Does your friend want to go with you? ($x_2=0$)
- Do you have pending assignments due on the weekend? ($x_3=0$)



Emulating Logical Gates with Perceptron





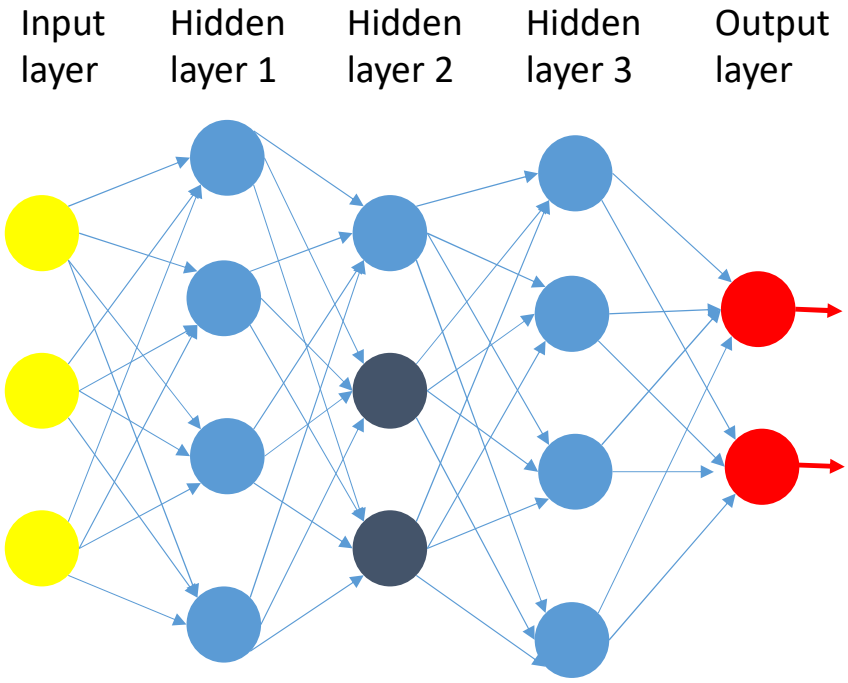


5. Artificial Neural Networks from Scratch

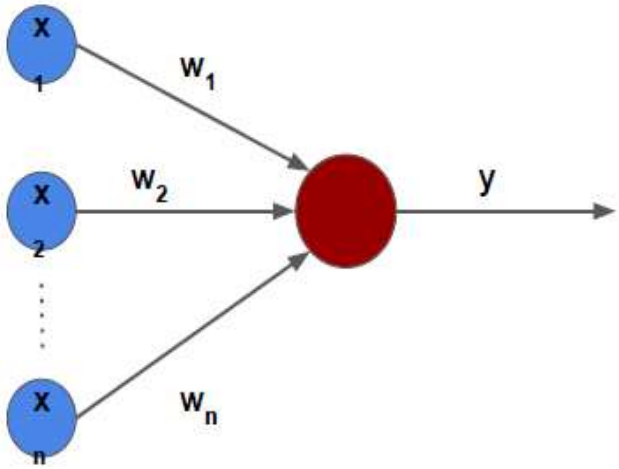
- *Learn to build neural network from scratch.*
  - *Focus on multi-level feedforward neural networks (multi-level perceptrons)*
- *Training large neural networks is one of the most important workload in large scale parallel and distributed systems*
  - *Programming assignments throughout the semester will use this.*

Artificial neural network example

A neural network consists of layers of artificial neurons and connections between them.
Each connection is associated with a weight.
Training of a neural network is to get to the right weights (and biases) such that the error across the training data is minimized.

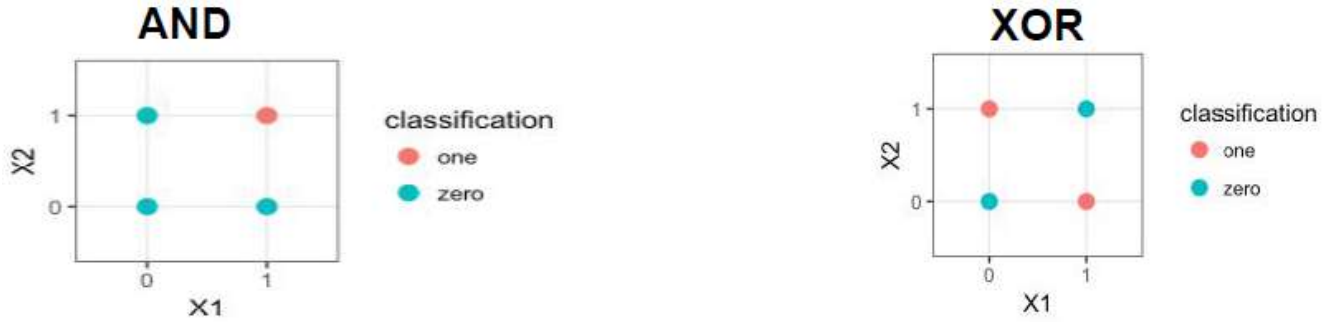


5. Perceptron Training principle



Step-1: Absorb bias b as weight.
Step-2: Start with a random value of weight $\mathbf{w}_j$
Step-3: Predict for each input $\mathbf{x}_i$ **:** If the prediction is correct ∀x, *then Return* **w**
Step-4: On a mistake for given input **x** , update as follows:
Mistake on positive ($y=1$), update $\mathbf{w}_{j+1} \leftarrow \mathbf{w}_j + \mathbf{x}$
● Mistake on negative ($y=0$), update $\mathbf{w}_{j+1} \leftarrow \mathbf{w}_j - \mathbf{x}$

Convergence of Perceptron Training



(source: https://jarvmiller.github.io/2017/10/14/neural-nets-pt1/)

● Whatever be the initial choice of weights and whatever be the input vector, PTA converges if the vectors are from a linearly separable function.
● If the weight repeats while training the perceptron, then the function is not linearly separable.

5. Activation Functions

- Activation function decide whether a neuron should be activated or not.
- It helps the network to use the useful information and suppress the irrelevant information.
- Usually a nonlinear function.
    - What if we choose a linear?
    - Linear classifier
    - Limited capacity to solve complex problems.

1. Sigmoid

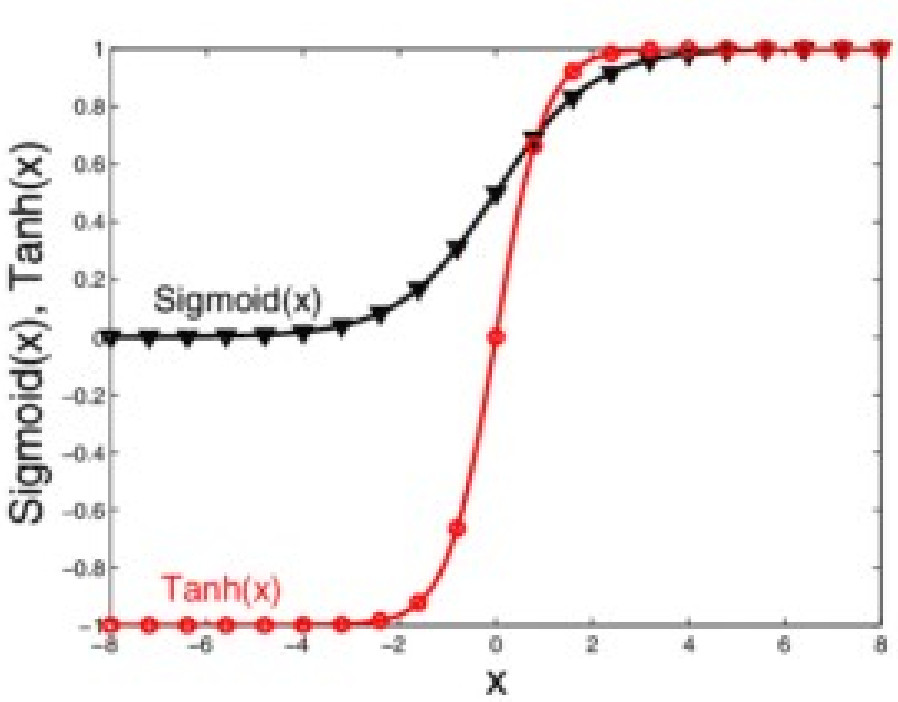
$$f(x) = \frac{1}{1+e^{-x}}$$

- **continuously differentiable**
- **ranges from 0-1**
- **not symmetric around the origin**

2. Tanh

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

- **scaled version of the sigmoid**
- **symmetric around the origin**
- **vanishing gradient**



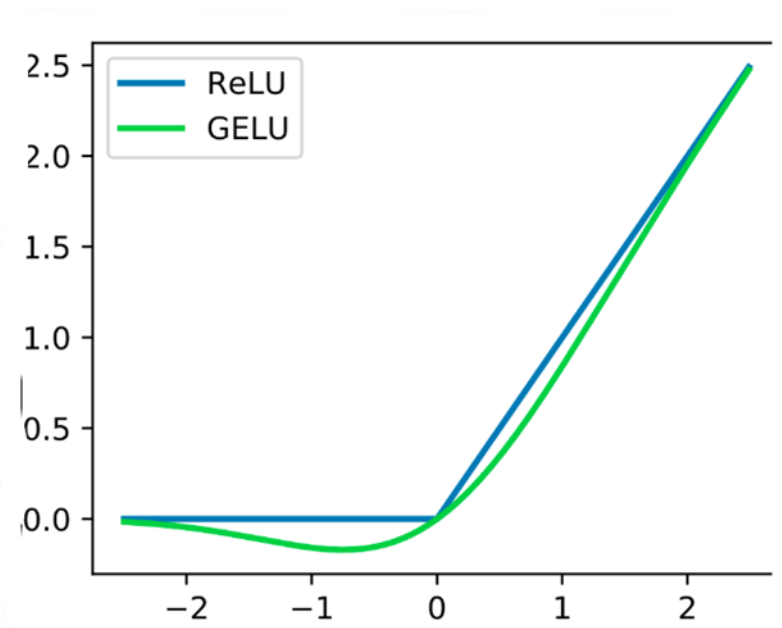
5. Activation Functions

3. Relu

The **rectifier** or **ReLU (rectified linear unit) activation function** is defined as the non-negative part of its argument, i.e., the ramp function:

$$\mathrm{ReLU}(x) = x^+ = \max(0, x) = \frac{x + |x|}{2} = \begin{cases} x & \text{if } x > 0, \\ 0 & x \leq 0 \end{cases}$$

- Also called piecewise linear function because rectified function is linear for half of the input domain and nonlinear for the other half.
- trivial to implement
- sparse representation
- avoid the problem of vanishing gradients
- dead neurons



4. Gelu (Gaussian Error Linear Unit)

- **Formula:** $f(x)=x\cdot\Phi(x)$,

where : $\Phi(x)$ is the cumulative distribution function of the standard normal distribution.

- **Behavior:** A smooth, probabilistic function that weights inputs by their value (approximates ReLU for large inputs, but allows small negative values through).
- **Pros:** Smooth and non-monotonic, avoids dying neurons, performs well in Transformers (LLMs).
- **Cons:** More computationally complex than ReLU

6. Training a neural network

- A neural network is trained with m training samples

$$(x^{(1)}, y^{(1)}), (x^{(2)}, y^{(2)}), \ldots\ldots (x^{(m)}, y^{(m)})$$

  $x^{(i)}$ is an input vector, $y^{(i)}$ is an output vector

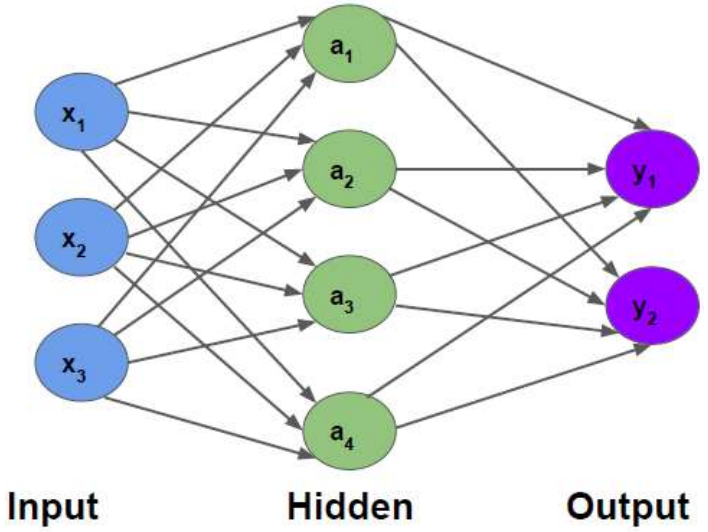
- Training objective: minimize the prediction error (loss)

$$min \sum_{i=1}^{m} (y^{(i)} - f_W(x^{(i)}))^2$$

  $f_W(x^{(i)})$ is the predicted output vector for the input vector $x^{(i)}$

- Approach: Gradient descent (stochastic gradient descent, batch gradient descent, mini-batch gradient descent).
- Use error to adjust the weight value to reduce the loss.

The adjustment amount is proportional to the contribution of each weight to the loss – Given an error, adjust the weight a little to reduce the error.



7. Stochastic gradient descent

- Given one training sample $(x^{(i)}, y^{(i)})$
- **Training objective**: minimize the prediction error (loss) – there are different ways to define error. The following is an example:

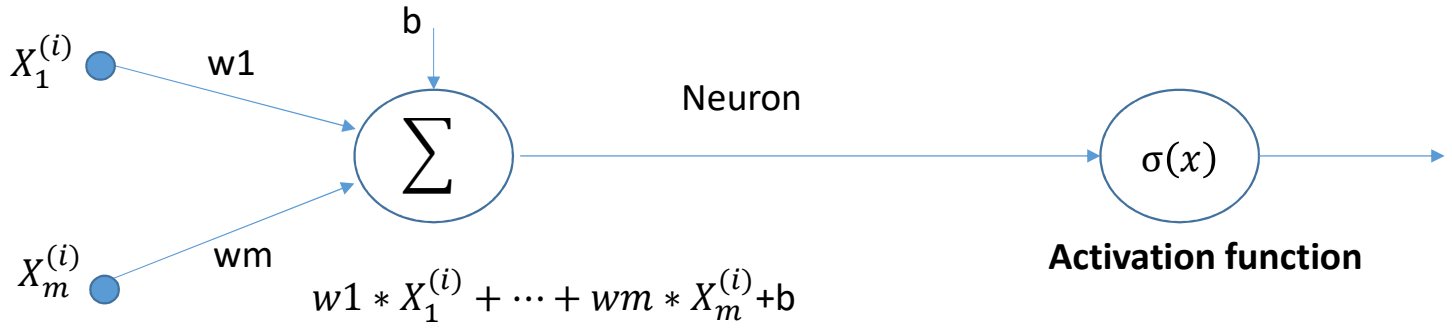
$$E = \frac{1}{2}(y^{(i)} - f_{\vec{W}}(x^{(i)}))^2$$

- Estimate how much each weight w_k in $\vec{W}$ contributes to the error: $\frac{\partial E}{\partial w_k}$
- Update the weight w_k by $w_{k+1} = w_k - \alpha \frac{\partial E}{\partial w_k}$. Here α is the learning rate.

Algorithm for learning artificial neural network

- Initialize the weights $\vec{W} = [W_0, W_1, \ldots\ldots, W_k]$
- **Training**
  - ❑ For each training data $(x^{(i)}, y^{(i)})$, Using forward propagation to compute the neural network output vector $f_{\vec{W}}(x^{(i)})$
  - ❑ Compute the error E (various definitions)
  - ❑ Use backward propagation to compute the output of the neural network $f_{\vec{W}}(x^{(i)})$
- $\frac{\partial E}{\partial W_k}$ for each weight W_k
  - ❑ Update $W_k = W_k - \alpha \frac{\partial E}{\partial W_k}$
  - ❑ Repeat until E is sufficiently small.

8. A single neuron training



- An artificial neuron has two components: (1) weighted sum and activation function. Many activation functions: Sigmoid, ReLU, etc.

Example - Sigmoid function

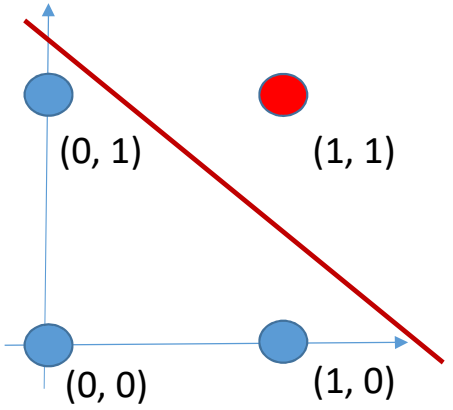
- $\sigma(x) = \frac{1}{1+e^{-x}}$
- The derivative of the sigmoid function: $\sigma'(x) = \frac{d}{dx}\sigma(x) = \sigma(x)(1-\sigma(x))$

8. Training for the logic AND with a single neuron

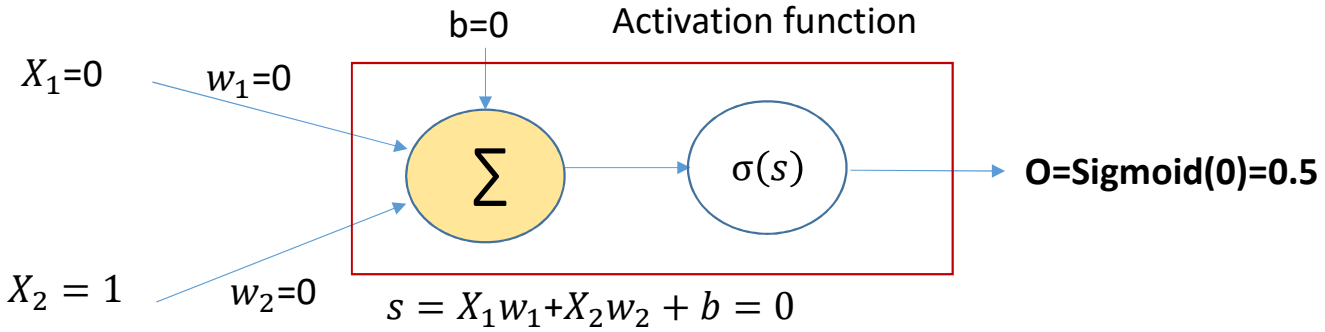
- In general, one neuron can be trained to realize a linear function.
- Logic AND function is a linear function:

$x1$	$x2$	$x1 \wedge x2$
0	0	0
0	1	0
1	0	0
1	1	1

Logic AND ($\wedge$) operation



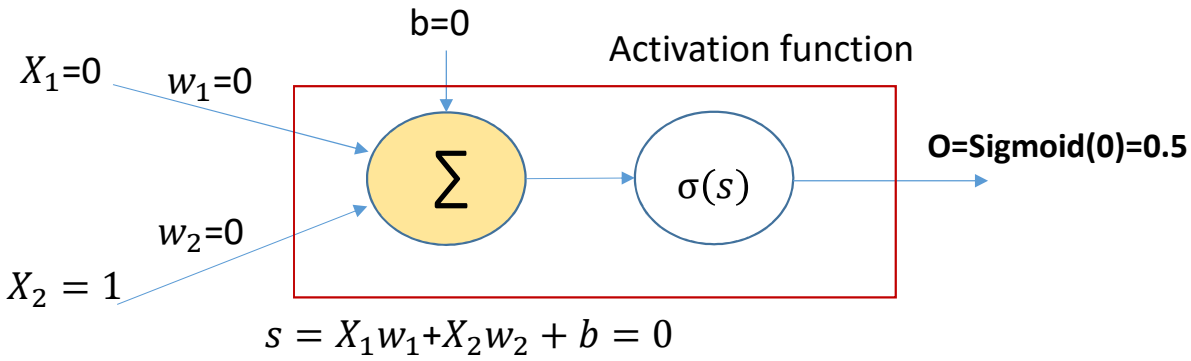
- Consider training data input (X_1=0, $X_2 = 1$), output Y=0.
- NN Output = 0.5
- Error: $E = \frac{1}{2}(Y - O)^2 = 0.125$
- To update w_1, w_2, and b, gradient descent needs to compute $\frac{\partial E}{\partial w_1}$, $\frac{\partial E}{\partial w_2}$, and $\frac{\partial E}{\partial b}$



8. Training for the logic AND with a single neuron

Chain rules for calculating $\frac{\partial E}{\partial w_1}$, $\frac{\partial E}{\partial w_2}$, and $\frac{\partial E}{\partial b}$

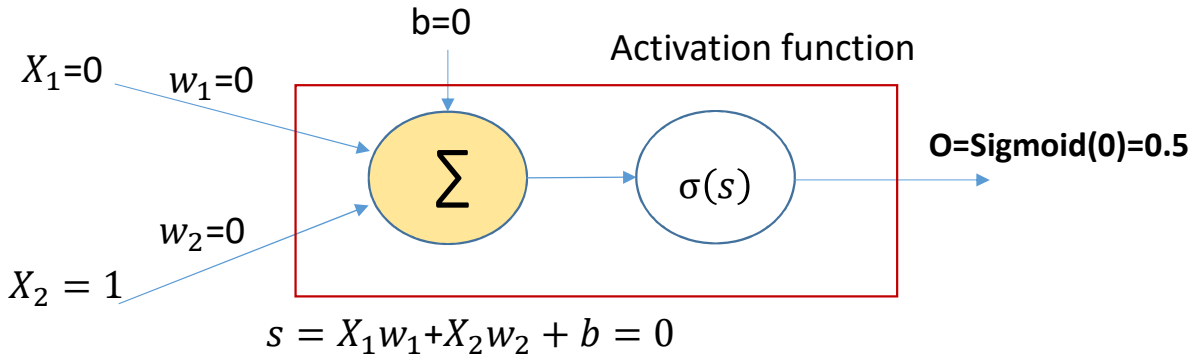
- If a variable z depends on the variable y, which itself depends on the variable x, then z depends on x as well, via the intermediate variable y. The **chain rule** is a formula that expresses the derivative as : $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$
- $\boldsymbol{\frac{\partial E}{\partial W_1} = \frac{\partial E}{\partial O}\frac{\partial O}{\partial s}\frac{\partial s}{\partial W_1}}$, $\frac{\partial E}{\partial W_2} = \frac{\partial E}{\partial O}\frac{\partial O}{\partial s}\frac{\partial s}{\partial W_2}$



Assume : rate = 0.1			
$\frac{\partial E}{\partial O} = \frac{\partial(\frac{1}{2}(Y-O)^2)}{\partial O}$	$= O - Y = 0.5 - 0 = 0.5$	$\frac{\partial E}{\partial O} = \frac{\partial(\frac{1}{2}(Y-O)^2)}{\partial O}$	$=O - Y = 0.5 - 0 = 0.5$
$\frac{\partial O}{\partial s} = \frac{\partial \sigma(s)}{\partial s} = \sigma(s)\,(1-\sigma(s))$	$=0.5\,(1-0.5) = 0.25$	$\frac{\partial O}{\partial s} = \frac{\partial \sigma(s)}{\partial s} = \sigma(s)\,(1-\sigma(s))$	$=0.5\,(1-0.5) = 0.25$
$\frac{\partial s}{\partial W_1} = \frac{\partial(X_1w_1+X_2w_2+b)}{\partial W_1}$	$= X_1 = 0$	$\frac{\partial s}{\partial W_2} = \frac{\partial(X_1w_1+X_2w_2+b)}{\partial W_2}$	$= X_2 = 1$
$\frac{\partial E}{\partial W_1} = \frac{\partial E}{\partial O}\frac{\partial O}{\partial s}\frac{\partial s}{\partial W_1}$	$=0.1*0.5*0.25*0$	$\frac{\partial E}{\partial W_2} = \frac{\partial E}{\partial O}\frac{\partial O}{\partial s}\frac{\partial s}{\partial W_2}$	$=0.1*0.5*0.25*1=0.0125$
$w_1 = w_1 - rate * \frac{\partial E}{\partial W_1}$	$=0 - 0.1*0.5*0.25*0 = 0$	$w_2 = w_2 - rate * \frac{\partial E}{\partial W_2}$	$=0 - 0.1*0.5*0.25*1 = -0.0125$

8. Training for the logic AND with a single neuron

- $\frac{\partial E}{\partial b} = \frac{\partial E}{\partial O}\frac{\partial O}{\partial s}\frac{\partial s}{\partial b}$:
  - $\frac{\partial E}{\partial O} = \frac{\partial(\frac{1}{2}(Y-O)^2)}{\partial O} = O - Y = 0.5 - 0 = 0.5$
  - $\frac{\partial O}{\partial s} = \frac{\partial \sigma(s)}{\partial s} = \sigma(s)\,(1-\sigma(s)) = 0.5\,(1-0.5) = 0.25,$
  - $\frac{\partial s}{\partial b} = \frac{\partial(X_1 w_1 + X_2 w_2 + b)}{\partial b} = 1$



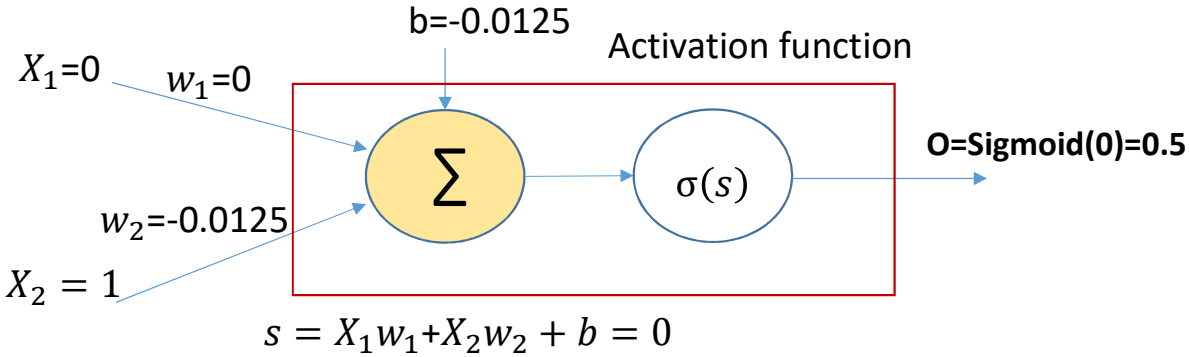
- **To update b:**

$b = b - rate * \frac{\partial E}{\partial b} = 0 - 0.1*0.5*0.25*1 = -0.0125$

This process is repeated until the error is sufficiently small

The initial weight should be randomized.

Gradient descent can get stuck in the local optimal. for training the logic AND operation with a single neuron.

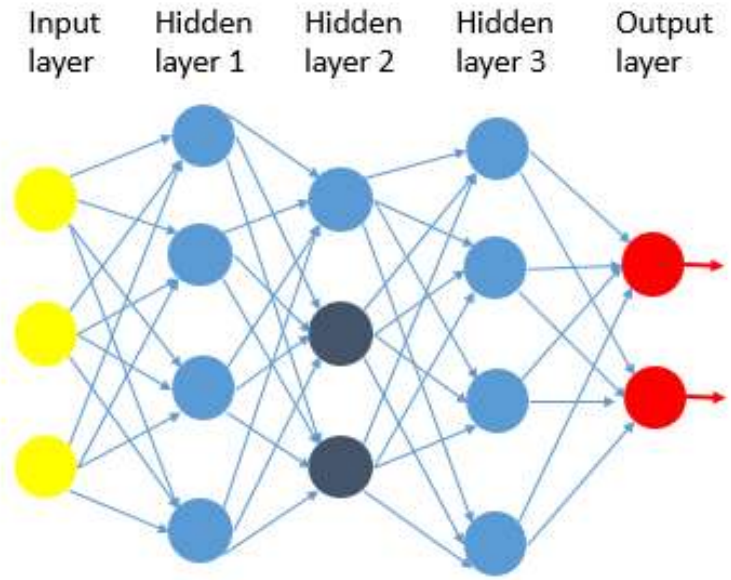


9. Multi-level feedforward neural networks

- A multi-level feedforward neural network is a neural network that consists of multiple levels of neurons. Each level can have many neurons and connections between neurons in different levels do not form loops.
  - Information moves in one direction (forward) from input nodes, through hidden nodes, to output nodes.
- One artificial neuron can only realize a linear function
- Many levels of neurons can combine linear functions can train arbitrarily complex functions.
  - One hidden layer (with infinite number of neurons) can train for any continuous function.

Example

A layer of neurons that do not directly connect to outputs is called a hidden layer.



10. Build a 3-level neural network from scratch

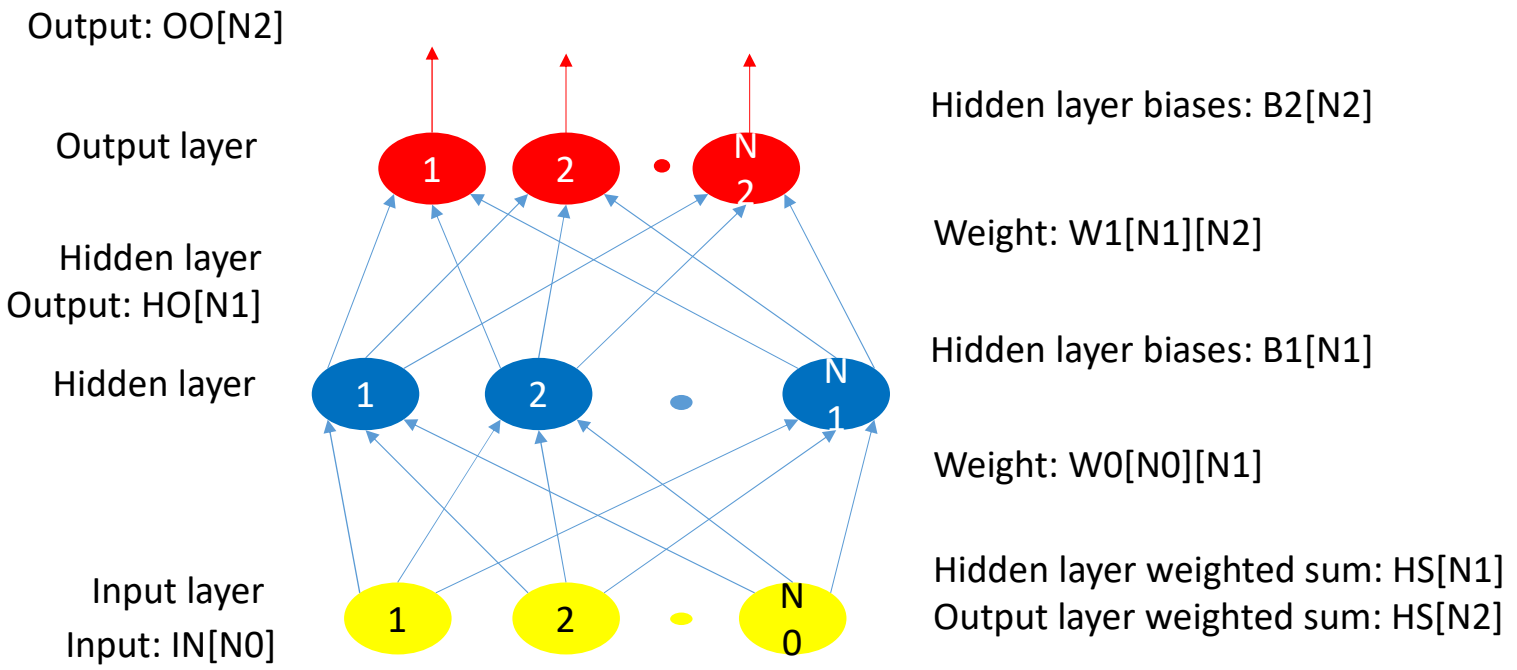
- **3 levels**

Input level, hidden level, output level

- Other assumptions: fully connected between layers, all neurons use sigmoid (σ) as the activation function.

- **Notations**
  - N_0: size of the input level. Input: $IN[N_0] = [IN_1, IN_2, \dots, IN_{No}]$
  - N_1: size of the hidden layer
  - N_2: size of the output layer. Output: $OO[N_2] = [OUT_1, OUT_2, \dots, OUT_{N2}]$
  - $N_0 \times N_1$ weights from input layer to hidden layer. $W0_{ij}$: the weight from input unit i to hidden unit j. $B0[N_1]$ biases. $B_0[N_1] = [B0_1, B0_2, \dots, B0_{N1}]$
  - $N_1 \times N_2$ weights from hidden layer to output layer. $W1_{ij}$: the weight from hidden unit i to output unit j. $B1[N2]$ biases.
  - $B_1[N_2] = [B1_1, B1_2, \dots, B1_{N2}]$
  - $W_0[N_0][N_1] = \begin{pmatrix} W0_{1,1} & \cdots & W0_{1,N1} \\ \vdots & \ddots & \vdots \\ W0_{N0,1} & \cdots & W0_{N0,N1} \end{pmatrix}$, $W_1[N_1][N_2] = \begin{pmatrix} W0_{1,1} & \cdots & W0_{1,N2} \\ \vdots & \ddots & \vdots \\ W0_{N1,1} & \cdots & W0_{N1,N2} \end{pmatrix}$

10. Build a 3-level neural network from scratch



Forward propagation (compute OO and E)

Compute hidden layer weighted sum: $HS[N1] = [HS_1, HS_2, \dots, HS_{N1}]$

$HS_i = IN_1 \times W0_{1,i} + IN_2 \times W0_{2,i} + \cdots + IN_{N0} \times W0_{N0,i} + B1_i$

In matrix form: $HS = IN \times W0 + B1$

Compute hidden layer output: $HO[N1] = [HO_1, HO_2, \dots, HO_{N1}]$

$HO_i = \sigma(HS_i)$

In matrix form: $HO = \sigma(HS)$

10. Build a 3-level neural network from scratch

Forward propagation

From input (IN[N0]), compute output (OO[N2]) and error E.

Compute output layer weighted sum: $OS[N2] = [OS_1, OS_2, \dots, OS_{N2}]$

$OS_i = HO_1 \times W1_{1,i} + HO_2 \times W1_{2,i} + \cdots + HO_{N1} \times W1_{N1,i} + B2_i$

In matrix form: $HS = HO \times W1 + B2$

Compute final output: $OO[N2] = [OO_1, OO_2, \dots, OO_{N1}]$

$OO_i = \sigma(OS_i)$

In matrix form: $OO = \sigma(OS)$

Let us use mean square error: $E = \frac{1}{N2}\sum_{i=1}^{N2}(OO_i - Y_i)^2$

Backward propagation

To goal is to compute $\frac{\partial E}{\partial W0_{i,j}}$, $\frac{\partial E}{\partial W_{i,j}}$, $\frac{\partial E}{\partial B_i}$, and $\frac{\partial E}{\partial B_i}$.

$$\frac{\partial E}{\partial OO} = \left[\frac{\partial E}{\partial O_1}, \frac{\partial E}{\partial O_2}, \dots, \frac{\partial E}{\partial O_{N2}}\right] = \left[\frac{2}{N2}(OO_1 - Y_1), \frac{2}{N2}(OO_2 - Y_2), \dots, \frac{2}{N2}(OO_{N2} - Y_{N2})\right]$$

In matrix form: $\frac{\partial E}{\partial OO} = \frac{2}{N2}(OO - Y)$

This can be stored in an array dE_OO[N2];

$$\frac{\partial E}{\partial OS} = \left[\frac{\partial E}{\partial O_1}\frac{\partial OO_1}{\partial OS_1}, \frac{\partial E}{\partial OO_2}\frac{\partial OO_2}{\partial O_2}, \dots, \frac{\partial E}{\partial OO_{N2}}\frac{\partial OO_{N2}}{\partial OS_{N2}}\right] = \left[\frac{\partial E}{\partial OO_1}\sigma(OS_1)(1 - \sigma(OS_1)), \dots, \frac{\partial E}{\partial OO_{N2}}\sigma(OS_{N2})(1 - \sigma(OS_{N2}))\right]$$

In matrix form: $\frac{\partial E}{\partial OS} = \frac{2}{N2}(OO - Y) \odot OO \odot (1 - OO)$

This can be stored in an array dE_OS[N2];

10. Build a 3-level neural network from scratch

Backward propagation

To goal is to compute $\frac{\partial E}{\partial W0_{i,j}}$, $\frac{\partial E}{\partial W1_{i,j}}$, $\frac{\partial E}{\partial B1_i}$, and $\frac{\partial E}{\partial B2_i}$.

$\frac{\partial E}{\partial OO}$, $\frac{\partial E}{\partial OS}$ are done

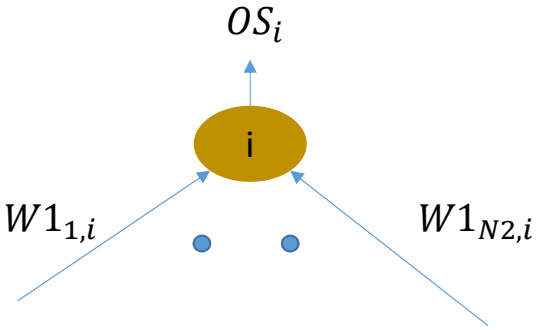
$$\frac{\partial E}{\partial B2} = \left[\frac{\partial E}{\partial OS_1}\frac{\partial OS_1}{\partial B_1}, \frac{\partial E}{\partial OS_2}\frac{\partial OS_2}{\partial B2_2}, \cdots, \frac{\partial E}{\partial OO_{N2}}\frac{\partial O_{N2}}{\partial B2_{N2}}\right]$$

$$OS_i = HO_1 \times W1_{1,i} + HO_2 \times W1_{2,i} + \cdots + HO_{N1} \times W1_{N1,i} + B2_i$$

Hence, $\frac{\partial OS_i}{\partial B2_i} = 1$.

$$\frac{\partial E}{\partial B2} = \left[\frac{\partial E}{\partial O_1}, \frac{\partial E}{\partial O_2}, \cdots, \frac{\partial E}{\partial OO_{N2}}\right] = \frac{\partial E}{\partial OS}$$

$$\frac{\partial E}{\partial W1} = \begin{pmatrix} \frac{\partial E}{\partial OS_1}\frac{\partial OS_1}{\partial W1_{1,1}} & \cdots & \frac{\partial E}{\partial OS_{N2}}\frac{\partial OS_{N2}}{\partial W1_{1,N2}} \\ \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial OS_1}\frac{\partial OS_1}{\partial W1_{N1,1}} & \cdots & \frac{\partial E}{\partial OS_{N2}}\frac{\partial OS_{N2}}{\partial W1_{N1,,N2}} \end{pmatrix}$$



$$OS_i = HO_1 \times W1_{1,i} + HO_2 \times W1_{2,i} + \cdots + HO_{N1} \times W1_{N1,i} + B2_i$$

Hence, $\frac{\partial O_i}{\partial W1j,_i} = HO_j$.

10. Build a 3-level neural network from scratch

Backward propagation

To goal is to compute $\frac{\partial E}{\partial W0_{i,j}}$, $\frac{\partial E}{\partial W1_{i,j}}$, $\frac{\partial E}{\partial B1_i}$, and $\frac{\partial E}{\partial B2_i}$.

$\frac{\partial E}{\partial OO}$, $\frac{\partial E}{\partial OS}$, $\frac{\partial E}{\partial B2}$ are done

$$\frac{\partial E}{\partial W1} = \begin{pmatrix} \frac{\partial E}{\partial OS_1}\frac{\partial OS_1}{\partial W1_{1,1}} & \cdots & \frac{\partial E}{\partial OS_{N2}}\frac{\partial OS_{N2}}{\partial W1_{1,N2}} \\ \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial OS_1}\frac{\partial OS_1}{\partial W1_{N1,1}} & \cdots & \frac{\partial E}{\partial OS_{N1}}\frac{\partial OS_{N2}}{\partial W1_{N1,,N2}} \end{pmatrix} = \begin{pmatrix} \frac{\partial E}{\partial OS_1}HO_1 & \cdots & \frac{\partial E}{\partial OS_{N2}}HO_1 \\ \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial OS_1}HO_{N1} & \cdots & \frac{\partial E}{\partial OS_{N2}}HO_{N1} \end{pmatrix}$$

In matrix form: $\frac{\partial E}{\partial W1} = HO^T \frac{\partial E}{\partial OS}$

$$\frac{\partial E}{\partial HO} = [\frac{\partial E}{\partial HO_1}, \frac{\partial E}{\partial HO_2}, \dots\dots, \frac{\partial E}{\partial HO_{N1}}]$$

$$\frac{\partial E}{\partial HO_i} = \frac{\partial E}{\partial O_1}\frac{\partial OS_1}{\partial HO_i} + \frac{\partial E}{\partial OS_2}\frac{\partial OS_2}{\partial HO_i} + \dots + \frac{\partial E}{\partial O_{N2}}\frac{\partial OS_{N2}}{\partial HO_i} = \frac{\partial E}{\partial OS_1}W1_{i,1} + \frac{\partial E}{\partial OS_2}W1_{i,2} + \dots + \frac{\partial E}{\partial OS_{N2}}W1_{i,N2}$$

$$\frac{\partial E}{\partial HO} = [\frac{\partial E}{\partial HO_1}, \frac{\partial E}{\partial HO_2}, \dots\dots, \frac{\partial E}{\partial HO_{N1}}] = \frac{\partial E}{\partial OS}W1^T$$

Once $\frac{\partial E}{\partial HO}$ is computed, we can repeat the process for the hidden layer by replacing OO with HO, OS with HS, B2 with B1 and W2 with W1, in the differential equation. Also the input is IN[N0] and the output is HO[N1].

11. Feed-forward Neural Network

$$a_1 = f(W_{11}x_1 + W_{12}x_2 + W_{13}x_3)$$

.

.

$$a_3 = f(W_{31}x_1 + W_{32}x_2 + W_{33}x_3)$$

$$y_1 = f(U_{11}a_1 + U_{12}a_2 + U_{13}a_3 + U_{14}a_4)$$

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

Matrix form:

$$z_1 = Wx$$

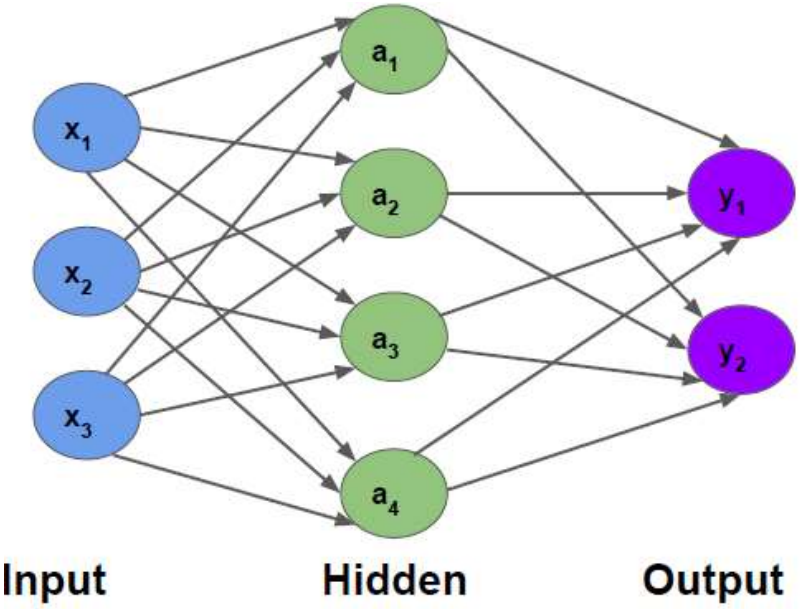
$$a = f(z_1)$$

$$z_2 = Ua$$

$$y = f(z_2)$$

where $x \in \mathbb{R}^{d_i}, W \in \mathbb{R}^{d_1 \times d_i}, a \in \mathbb{R}^{d_1}$,

$U \in \mathbb{R}^{d_o \times d_1}, y \in \mathbb{R}^{d_o}$



Objective Function

The function we want to minimize or maximize is called the objective function or criterion.

- When we are minimizing it, we may also call it the **cost function**, **loss function**, or error function.
- A loss function tells how good our current classifier is.
- Given a dataset:

11. Feed-forward Neural Network _ Objective Function

Objective Function

The function we want to minimize or maximize is called the objective function or criterion.

- When we are minimizing it, we may also call it the **cost function**, **loss function**, or error function.
- A loss function tells how good our current classifier is.
- Given a dataset:

$$\mathbf{x}^{(i)} = \{x_1^{(i)}, x_2^{(i)}, \ldots, x_m^{(i)}\} \in \mathbb{R}^m$$

y^i, The loss function can be written as:

$$\mathcal{L}(\theta) = \frac{1}{n} \sum_{i=1}^{n} L\left(y^{(i)}, f(\mathbf{x}^{(i)}\theta)\right)$$

Mean Squared Error

- Mean Squared Error (MSE), or quadratic, loss function is widely used in linear regression as the performance measure.
- It measures the average of the squares of the errors—that is, the average squared difference between the estimated values and the actual value.
- It is always non-negative, and values closer to zero are better.

$$\mathcal{L} = \frac{1}{n} \sum_{i=1}^{n} (y^{(i)} - \hat{y}^{(i)})^2$$

11. Feed-forward Neural Network _ Objective Function

Mean Absolute Error

○ Mean Absolute Error (MAE) is a quantity used to measure how close forecasts or predictions are to the eventual outcomes.
○ Both MSE and MAE are used in predictive modeling.
○ MSE has nice mathematical properties which makes it easier to compute the gradient.

$$\mathcal{L} = \frac{1}{n}\sum_{i=1}^{n}(|y^{(i)} - \hat{y}^{(i)}|)$$

Cross-entropy

○ Coss-entropy comes from the field of information theory and has the unit of "bits."
○ The cross-entropy between a "true" distribution p and an estimated distribution q is defined as:

$$H(p,q) = -\sum_x p(x)\log q(x)$$

○ Cross-entropy can be re-written in terms of the entropy and Kullback-Leibler divergence between the two distributions

$$H(p,q) = H(p) + D_{KL}(p||q)$$

11. Feed-forward Neural Network _ Objective Function

Cross-entropy

○ Assuming a ground truth probability distribution that is **1** at the right class and **0** everywhere else ***p* = *[0,…,0,1,0,…0]*** and our computed probability is ***q*.**

○ Kullback-Leibler divergence can be written as:

$$
\begin{aligned}
D_{KL}(p||q) &= \sum_x p(x) \log \frac{p(x)}{q(x)} \\
&= \sum_x p(x) \log p(x) - \sum_x p(x) \log q(x) \\
&= -H(p) + H(p,q)
\end{aligned}
$$

or

$$
H(p,q) = H(p) + D_{KL}(p||q)
$$

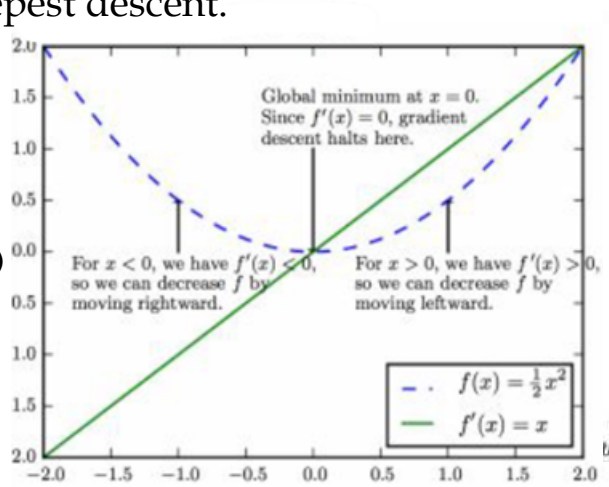
11. Feed-forward Neural Network _ Optimization

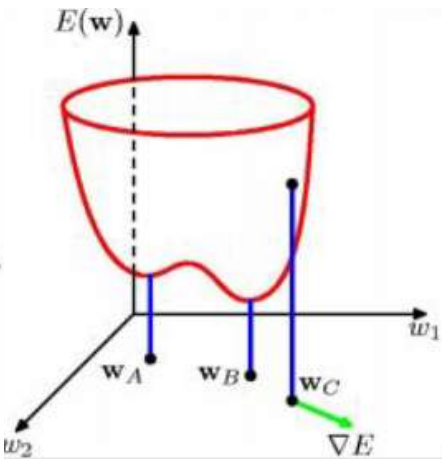
The goal of optimization is to find parameter (weights) that minimizes the loss function.

- How to find such weights?
  - Random Search
    - Very bad idea.
  - Random Local Search
    - Start with a random weight w and generate random perturbations $\mathbf{\Delta w}$ to it and if the loss at the perturbed $\mathbf{w+\Delta w}$ is lower, we will perform an update.
    - Computationally expensive
  - Follow the Gradient
    - No need to search for a good direction.
    - We can compute the best direction along which we should change our weight vector that is mathematically guaranteed to be the direction of the steepest descent.

Find w which minimizes the chosen error function $E(w)$

- wA : a local minimum
- wB : a global minimum
- At point wC local gradient is given by vector $\Delta E(w)$
- It points in direction of greatest rate of increase of $E(w)$
- Negative gradient points to rate of greatest decrease





13. Gradient and Hessian

First derivative of a scalar function $E(\boldsymbol{w})$ with respect to a vector $\boldsymbol{w}=[w_1,w_2]^T$ is a vector called the Gradient of $E(\boldsymbol{w})$

$$\nabla E(\boldsymbol{w}) = \frac{d}{d\boldsymbol{w}} E(\boldsymbol{w}) = \begin{bmatrix} \frac{\partial E}{\partial w_1} \\ \frac{\partial E}{\partial w_2} \end{bmatrix}$$

If there are M elements in the vector then Gradient is a $M \times 1$ vector

Second derivative of a scalar function $E(\boldsymbol{w})$ with respect to a vector $\boldsymbol{w}=[w_1,w_2]^T$ is a matrix called the Hessian of $E(\boldsymbol{w})$

$$H = \nabla\nabla E(\boldsymbol{w}) = \frac{d^2}{d\boldsymbol{w}^2} E(\boldsymbol{w}) = \begin{bmatrix} \frac{\partial^2 E}{\partial w_1^2} & \frac{\partial^2 E}{\partial w_1 \partial w_2} \\ \frac{\partial^2 E}{\partial w_2 \partial w_1} & \frac{\partial^2 E}{\partial w_2^2} \end{bmatrix}$$

Gradient Descent Optimization

Determine weights $\boldsymbol{w}$ from labeled set of training samples.

- Take a small step in the direction of the negative gradient

$$w_{new} = w_{old} - \eta \, \Delta E(w_{old})$$

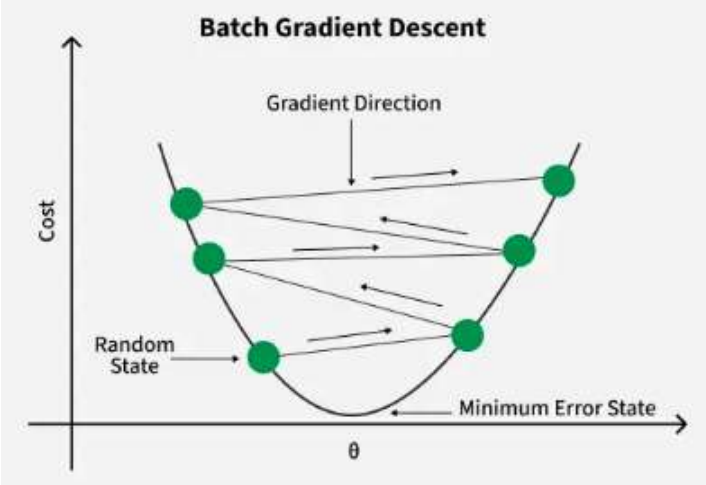
- After each update, the gradient is re-evaluated for the new weight vector and the process is repeated
- This size of steps η taken to reach the minimum or bottom is called **Learning Rate**.

14. Gradient Descent Variants

- **Batch gradient descent**

○ Vanilla gradient descent, aka batch gradient descent, computes the gradient of the cost function w.r.t. to the parameters w for the entire training dataset.

$$w_{new} = w_{old} - \eta \, \Delta E(w_{old})$$

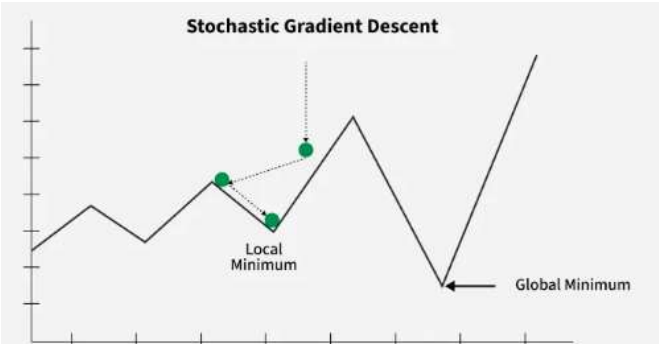


○ Guaranteed to converge to global minimum for convex error surfaces and to a local minimum for non-convex surfaces.
○ Need to calculate the gradients for the whole dataset to perform just one update.
○ Very slow and is intractable for datasets that don't fit in memory

- **Stochastic gradient descent**

○ Stochastic gradient descent (SGD) in contrast performs a parameter update for each training example, say (x_i, y_i):

$$w_{new} = w_{old} - \eta \, \Delta E(w_{old}; x_i ; y_i)$$



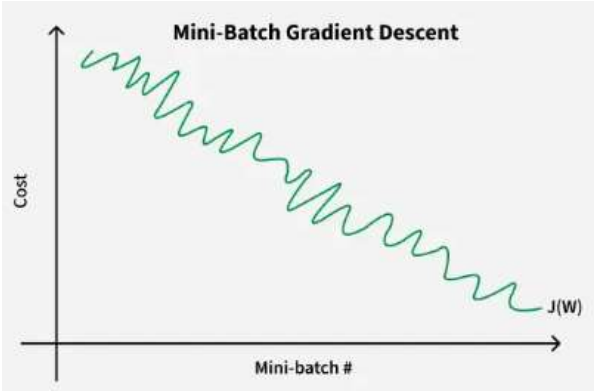
○ Much faster (avoid redundancy as exist in Batch gradient descent)
○ While slowly decreasing the learning rate, SGD shows the same convergence behaviour as batch gradient descent.
○ It performs frequent updates with a high variance that cause the objective function to fluctuate heavily.

14. Gradient Descent Variants

- **Mini-batch gradient descent**
  - Performs update for every mini-batch of n examples.

$$w_{new} = w_{old} - \eta \, \Delta E(w_{old}; x_{i:i+n}; y_{i:i+n})$$

  - Reduces variance of updates.
  - Algorithm of choice
  - Mini-batch size is a hyperparameter. Common sizes are 50-256.

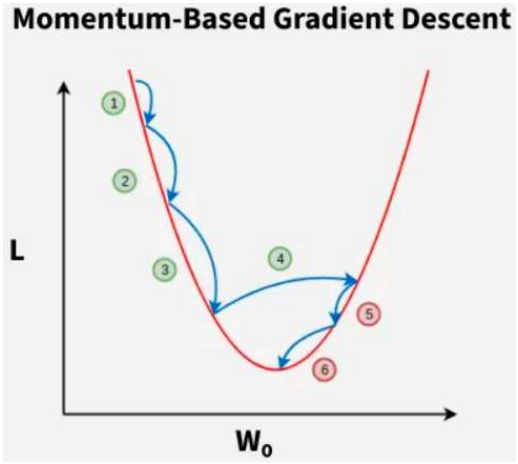


- **Momentum-Based Gradient Descent**

It is an enhancement of standard gradient descent algorithm that aims to accelerate convergence particularly in the presence of high curvature, small but consistent gradients or noisy gradients. It introduces a velocity term that accumulates the gradient of the loss function over time thereby smoothing the path taken by the parameters. The update rule for Momentum-Based Gradient Descent is:

$$v_t = \gamma v_{t-1} + \eta \nabla E(\theta_t)$$
$$\theta_{t+1} = \theta_t - v_t$$

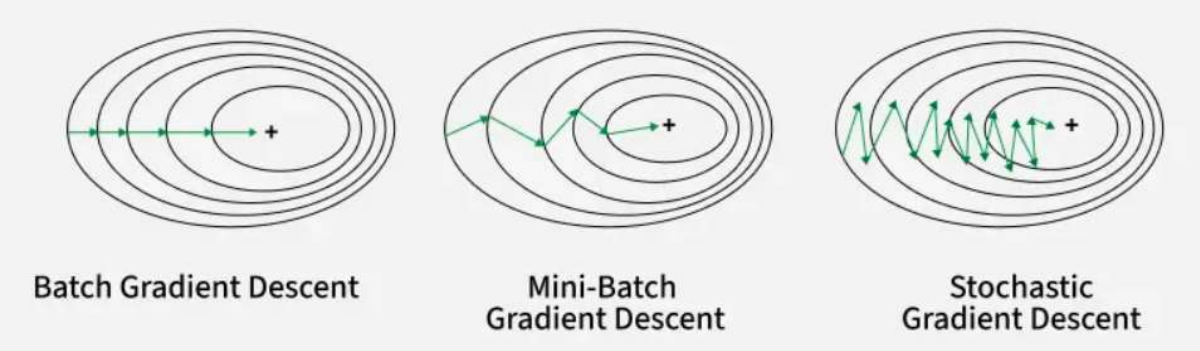
where:
- v_t is the velocity at iteration t,
- γ is the momentum term (typically between 0 and 1),
- H is the learning rate,
- $\nabla E(\theta t)$ is the gradient of the loss function $J(\theta)$ with respect to θ at iteration t.



14. Gradient Descent Variants

• Comparison between the variants of Gradient Descent

Variant	Data Used	Convergence	Memory Usage	Efficiency	Key Advantage
Batch Gradient Descent	Entire dataset	Stable but slow	High (entire dataset)	Computationally expensive	Stable convergence, global view of data
Stochastic Gradient Descent	One example per iteration	Fast but noisy	Low (one example)	Less efficient	Faster convergence, good for online learning
Mini-Batch Gradient Descent	Mini-batch of data	Faster and smoother	Medium (mini-batch)	Efficient, parallelizable	Balance of speed and stability
Momentum-Based Gradient Descent	Entire dataset or mini-batch	Faster and smoother	Medium (like Mini-Batch)	Efficient with momentum	Accelerated



15. Python Implementation of Gradient Descent Variants

- **Batch gradient descent**

```
def batch_gradient_descent(X, y, theta, lr=0.01,
epochs=100):
    m = len(y)
    for _ in range(epochs):
        gradients = (1 / m) * X.T @ (X @ theta - y)
        theta -= lr * gradients
    return theta
```

- **Stochastic gradient descent**

```
import numpy as np


def stochastic_gradient_descent(X, y, theta, lr=0.01,
epochs=100):
    m = len(y)
    for _ in range(epochs):
        for i in range(m):
            xi = X[i:i + 1]
            yi = y[i:i + 1]
            gradient = xi.T @ (xi @ theta - yi)
            theta -= lr * gradient
    return theta
```

15. Python Implementation of Gradient Descent Variants

- **Mini-batch gradient descent**

```
def mini_batch_gradient_descent(X, y, theta, lr=0.01,
epochs=100, batch_size=32):
    m = len(y)
    for _ in range(epochs):
        indices = np.random.permutation(m)
        X_shuffled, y_shuffled = X[indices], y[indices]
        for i in range(0, m, batch_size):
            xb = X_shuffled[i:i + batch_size]
            yb = y_shuffled[i:i + batch_size]
            gradient = (1 / len(yb)) * xb.T @ (xb @
theta - yb)
            theta -= lr * gradient
    return theta
```

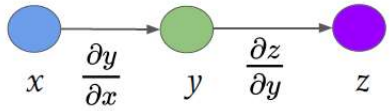
- **Momentum-Based Gradient Descent**

```
def momentum_gradient_descent(X, y, theta, lr=0.01,
epochs=100, gamma=0.9):
    m = len(y)
    v = np.zeros_like(theta)
    for _ in range(epochs):
        gradient = (1 / m) * X.T @ (X @ theta - y)
        v = gamma * v + lr * gradient
        theta -= v
    return theta
```

15. Backpropagation Algorithm

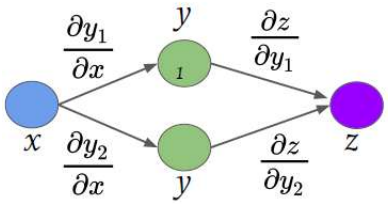
- This algorithm is used to train artificial neural networks, it can update the weights very efficiently.
- It is a computationally efficient approach to compute the derivatives of a complex cost function.
- Goal is to use those derivatives to learn the weight coefficients for parameterizing a multi-layer ANN.
- It compute the gradient of a cost function with respect to all the weights in the network, so that the gradient is fed to the gradient descent method which in turn uses it to update the weights in order to minimize the cost function.

Chain Rule



- Single Path

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$



- Multiple Path

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x}$$

$$\frac{\partial z}{\partial x} = \sum_{t=1}^{T} \frac{\partial z}{\partial y_t}\frac{\partial y_t}{\partial x}$$

The total error in the network for a single input is given by the following equation:

$$E = \frac{1}{2}\sum_{k=1}^{K}(a_k - t_k)^2$$

where

a_k : predicted output/activation of a node k

t_k : actual output of a node k

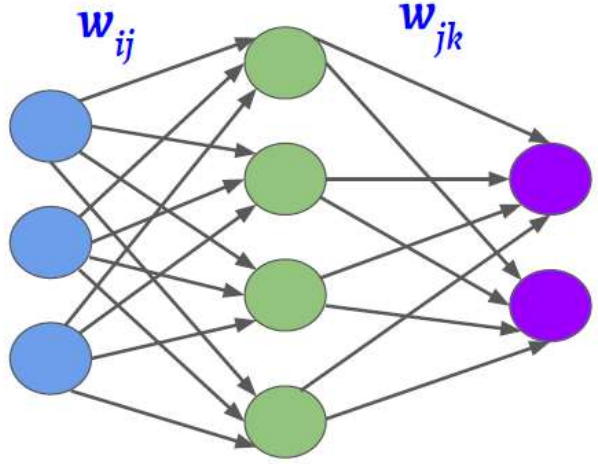
15. Backpropagation Algorithm

There are two sets of weights in our network:

- ○ *wij* : from the input to the hidden layer.
- ○ *wjk* : from the hidden to the output layer.

● We want to adjust the network's weights to reduce this overall error.

$$\triangle W \propto -\frac{\partial E}{\partial W}$$



$$a_k = f_k\Big(\overbrace{\sum_j f_j\underbrace{\Big(\sum_i a_i w_{ij}\Big)}_{z_j} w_{jk}}^{z_k}\Big)$$

a_j

Backpropagation – for outermost layer

○ outermost layer parameters directly affect the value of the error function.

○ only one term of the E summation will have a non-zero derivative: the one associated with the particular weight we are considering.

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial a_k}\frac{\partial a_k}{\partial z_k}\frac{\partial z_k}{\partial w_{jk}}$$

$$\frac{\partial E}{\partial a_k} = \frac{\partial}{\partial a_k}\left(\frac{1}{2}\sum_{k\in K}(a_k - t_k)^2\right)$$
$$= (a_k - t_k)$$

$$\frac{\partial a_k}{\partial z_k} = \frac{\partial}{\partial z_k}(f_k(z_k))$$
$$= f_k'(z_k)$$

15. Backpropagation Algorithm

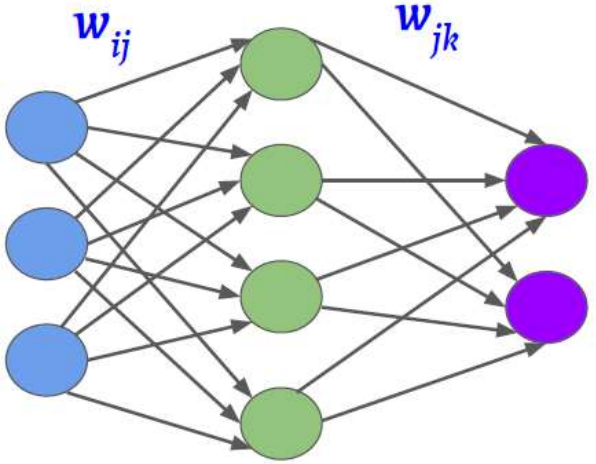
$$\frac{\partial z_k}{\partial w_{jk}} = \frac{\partial}{\partial w_{jk}}\left(\sum_j a_j w_{jk}\right)$$
$$= a_j$$

$$\frac{\partial E}{\partial w_{jk}} = \underbrace{(a_k - t_k) f_k'(z_k)}_{\delta_k} a_j$$

For sigmoid activation function

$$\frac{\partial E}{\partial w_{jk}} = (a_k - t_k) a_k (1 - a_k) a_j$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\left(\frac{1}{2}\sum_{k\in K}(a_k - t_k)^2\right)$$
$$= \sum_{k\in K}(a_k - t_k)\frac{\partial}{\partial w_{ij}} a_k$$
$$= \sum_{k\in K}(a_k - t_k)\frac{\partial}{\partial w_{ij}}(f_k(z_k))$$
$$= \sum_{k\in K}(a_k - t_k) f_k'(z_k)\frac{\partial}{\partial w_{ij}} z_k$$

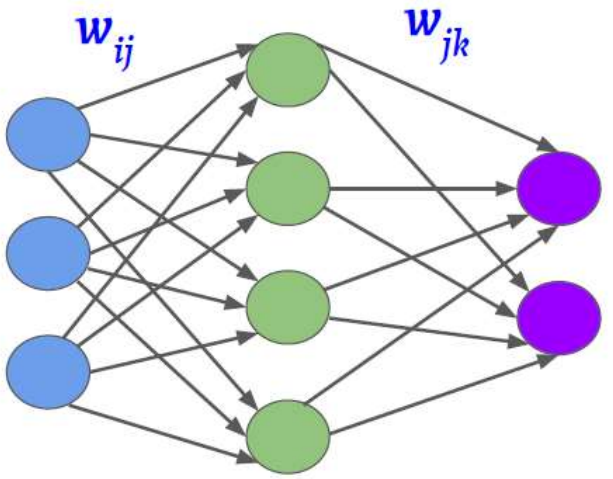


$$a_k = f_k\Big(\overbrace{\textstyle\sum_j f_j\underbrace{(\textstyle\sum_i a_i w_{ij})}_{z_j} w_{jk}}^{z_k}\Big)$$
$$\underbrace{\qquad\qquad}_{a_j}$$

15. Backpropagation Algorithm

$$
\begin{aligned}
\frac{\partial z_k}{\partial w_{ij}} &= \frac{\partial z_k}{\partial a_j}\frac{\partial a_j}{\partial w_{ij}} \\
&= \frac{\partial}{\partial a_j} a_j w_{jk} \frac{\partial a_j}{\partial w_{ij}} \\
&= w_{jk}\frac{\partial a_j}{\partial w_{ij}} \\
&= w_{jk}\frac{\partial f_j(z_j)}{\partial w_{ij}} \\
&= w_{jk} f'_j(z_j)\frac{\partial z_j}{\partial w_{ij}} \\
&= w_{jk} f'_j(z_j)\frac{\partial}{\partial w_{ij}}\left(\sum_i a_i w_{ij}\right) \\
&= w_{jk} f'_j(z_j) a_i
\end{aligned}
$$

$$
\begin{aligned}
\frac{\partial E}{\partial w_{ij}} &= \sum_{k\in K}(a_k - t_k) f'_k(z_k)\underbrace{w_{jk} f'_j(z_j) a_i}_{\partial z_k/\partial w_{ij}} \\
&= f'_j(z_j) a_i \sum_{k\in K}\underbrace{(a_k - t_k) f'_k(z_k)}_{\delta_k} w_{jk} \\
&= \underbrace{a_i f'_j(z_j)\sum_{k\in K}\delta_k w_{jk}}_{\delta_j} \\
&= \delta_j a_i
\end{aligned}
$$

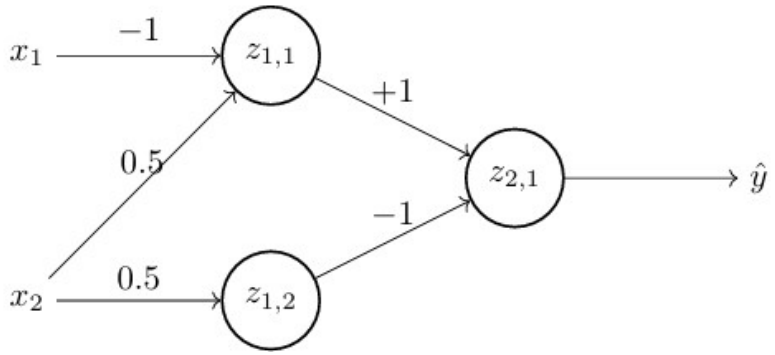


$$
a_k = f_k\Big(\overbrace{\sum_j \underbrace{f_j\Big(\sum_i a_i w_{ij}\Big)}_{z_j} w_{jk}}^{z_k}\Big)
$$

Exercices

Exercise 1.

Consider the Neural Network depicted in the below Figure. Bias terms are omitted in this exercise



A. Perform the forward pass for the single input data pointx=(1,2).
B. Given the single training instance x=(1,2), y=1, update all the weights once via back-propagation, using the log-likelihood objective function l=ylog(ˆy)+(1−y)log(1−ˆy), sigmoid activation function, learn rate η=1 and without any regularization.
C. Perform an other forward pass, using the updated weights. Comment on the result.

Exercices

Exercise 2 .

Update the parameters of the neural network in below figure 1 with the gradient descent algorithm based on one data point X = [1,1] with label y = 1.

Consider the following network parameters for your calculations:

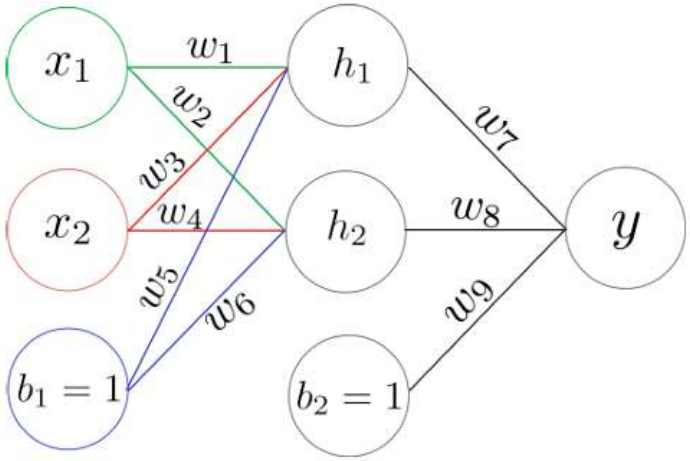
- Weights: $w_1 = 0.5$, $w_2 = 0.3$, $w_3 = 0.3$, $w_4 = 0.1$, $w_5 = 0.8$, $w_6 = 0.3$, $w_7 = 0.5$, $w_8 = 0.9$, $w_9 = 0.2$.
- The hidden neurons use a sigmoid activation function.

The output neuron is computed with a simple linear activation function

a) Forward Pass Calculate the values of the hidden neurons h1,h2 and the output neuron y and evaluate the error L of the prediction result. Use the mean squared error: $L = 1\ 2(\hat{}y - y)2$.

b) Backward Pass and Parameter Update Calculate the partial derivative (with backpropagation) of the prediction error L w.r.t. each weight ∂L ∂wi and update the weights via: $w * i = wi - \eta\ \partial L$ ∂wi with η = 0.2.

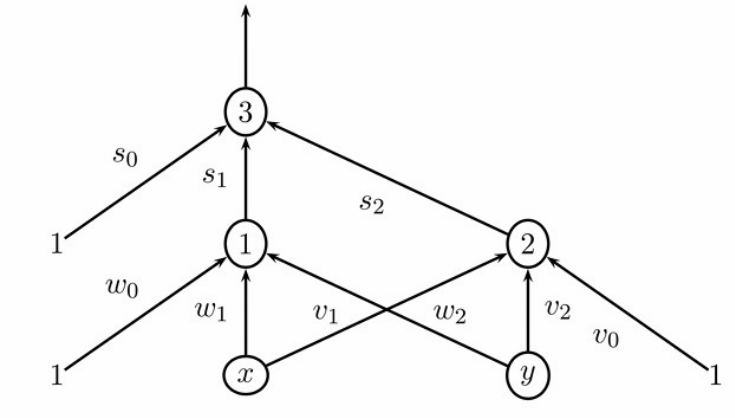
Use the chain rule for obtaining the partial derivatives. Be aware that the derivative of the sigmoid function is: a(t)= a(t) ∗ (1 −a(t)). Verify your result by testing if the prediction error decreased after the weight update.



Exercices

Exercise 2

Draw a neural network that represents the function f(x,y) defined below :



x	y	$f(x,y)$
0	0	10
0	1	-5
1	0	-5
1	1	10

Solution

Nodes labeled by 1 and 2 are simple threshold units while the node labeled by 3 is a linear unit.

A possible setting of the weights is given below. Recall that the simple threshold unit is given by:

$$out = \begin{cases} +1 & \text{if } \sum_i w_i x_i > 0 \\ -1 & \text{otherwise} \end{cases}$$

o_1, which is the output of node labeled by 1 implements the following function

$$o_1 = \begin{cases} +1 & \text{if } \neg x \wedge \neg y \text{ is true} \\ -1 & \text{otherwise} \end{cases}$$

To achieve this, we can use $w_0 = 1$ and $w_1 = w_2 = -2$

o_2, which is the output of node labeled by 2 implements the following function

$$o_2 = \begin{cases} +1 & \text{if } x \wedge y \text{ is true} \\ -1 & \text{otherwise} \end{cases}$$

To achieve this, we can use $v_0 = -2$ and $v_1 = v_2 = 1.5$.

o_3 implements the following function

$$o_3 = \begin{cases} +10 & \text{if } o_1 = +1 \text{ or } o_2 = +1 \\ -5 & \text{otherwise} \end{cases}$$

Note that since 3 is a linear unit, we need it to obey the following constraints:
$s_0 + s_1 - s_2 = 10$ (if $o_1 = +1$ and $o_2 = -1$)
$s_0 - s_1 + s_2 = 10$ (if $o_1 = -1$ and $o_2 = +1$)
$s_0 - s_1 - s_2 = -5$ (if $o_1 = -1$ and $o_2 = -1$)

Notice that the case $o_1 = +1$ and $o_2 = +1$ can never happen.

A solution to the three equations is $s_0 = 10$ and $s_1 = s_2 = 7.5$.

Exercise 3

Perceptron for AND Gate The AND function is a binary logic gate that outputs true (1) only if all inputs are true (1); otherwise, it outputs false (0).

Solution

Configuration (Weights and Bias)

A common set of parameters that solves the AND function are:

- **Weight 1** ($w1$): 0.5
- **Weight 2** ($w2$): 0.5
- **Bias** (b): -0.7
- **Activation Function**: A step function where the output is 1 if the weighted sum plus bias is ≥0 , and 0 otherwise.

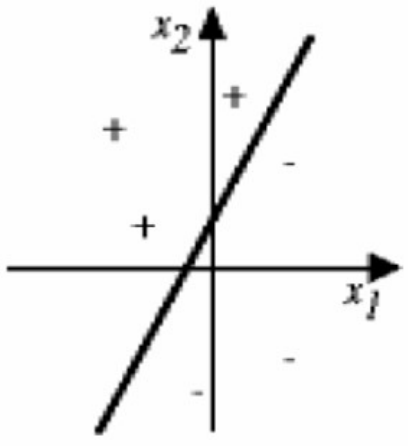
$y = f((w_1 \cdot x_1) + (w_2 \cdot x_2) + b)$:

- For (0, 0): $f((0.5 \cdot 0) + (0.5 \cdot 0) + (-0.7)) = f(-0.7) = 0$
- For (0, 1): $f((0.5 \cdot 0) + (0.5 \cdot 1) + (-0.7)) = f(-0.2) = 0$
- For (1, 0): $f((0.5 \cdot 1) + (0.5 \cdot 0) + (-0.7)) = f(-0.2) = 0$
- For (1, 1): $f((0.5 \cdot 1) + (0.5 \cdot 1) + (-0.7)) = f(0.3) = 1$

As shown, the perceptron correctly produces the output of an AND gate for all possible inputs. The ability to find a set of weights and a bias that works demonstrates that the AND function is a **linearly separable** problem, which single-layer perceptrons are capable of solving.

Exercise 4

- What are the values of weights w0, w1, and w2 for the perceptron whose decision surface is illustrated in the figure? Assume the surface crosses the x1 axis at -1 and the x2 axis at 2.



Solution The output of the perceptron is:

$$o = sgn(w_0 + w_1 x_1 + w_2 x_2)$$

The equation of the decision surface (the line) is

$$w_o + w_1 x_1 + w_2 x_2 = 0$$

We know the coordinates of 2 points of this line: A=(-1,0) and B=(0,2).
Therefore, the equation of the line is

$$\frac{x_1 - x_{1A}}{x_{1B} - x_{1A}} = \frac{x_2 - x_{2A}}{x_{2B} - x_{2A}} \rightarrow \frac{x_1 - (-1)}{0 - (-1)} = \frac{x_2 - 0}{2 - 0} \rightarrow x_1 + 1 = \frac{x_2}{2} \rightarrow 2 + 2x_1 - x_2 = 0$$

So 2, 2, -1 are possible values for the weights w0, w1, and w2, respectively. To check if their signs are correct, consider a point on one side of the line, for instance the origin O=(0,0).
The output of the perceptron for this point has to be negative, but the output of the perceptron using the candidate weights is positive. Therefore, we need to negate the previous values and conclude that w0=− 2, =−2, =1.

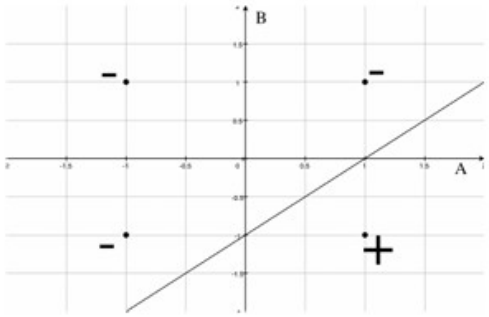
Exercise 5

- (a) Design a two-input perceptron that implements the Boolean function A∧¬B.
  (b) Design the two-layer network of perceptrons that implements A XOR B.

Solution (a) The requested perceptron has 3 inputs: A, B, and the constant 1. The values of A and B are 1 (true) or -1 (false). The following table describes the output O of the perceptron:

A	B	O = A∧¬B
-1	-1	-1
-1	1	-1
1	-1	1
1	1	-1

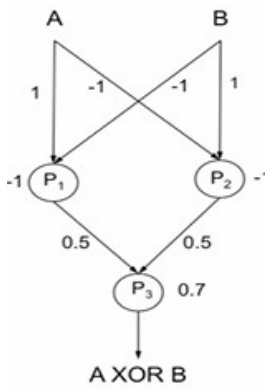
One of the correct decision surfaces (any line that separates the positive point from the negative points would be fine) is shown in the following picture.



The line crosses the A axis at 1 and the B axis -1. The equation of the line is

$$\frac{A-0}{1-0} = \frac{B-(-1)}{0-(-1)} \rightarrow A = B + 1 \rightarrow 1 - A + B = 0$$

- So 1, -1, 1 are possible values for the weights w0, w1, and w2, respectively. Using this values the output of the perceptron for A=1, B=-1 is negative. Therefore, we need to negate the weights and therefore we can conclude that w0=−1, =1, =−1.
- Solution (b) A XOR B cannot be calculated by a single perceptron, so we need to build a two-layer network of perceptrons. The structure of the network can be derived by:
- Expressing A XOR B in terms of other logical connectives: A XOR B = (A∧¬B) ∨(¬A∧B) • Defining the perceptrons P1 and P2 for (A∧¬B) and (¬A∧B)
- Composing the outputs of P1 and P2 into a perceptron P3 that implements o(P1) ∨ o(P2) Perceptron P1 has been defined above. P2 can be defined similarly. P3 is defined in the course slides1. In the end, the requested network is the following:



NB. The number close to each unit is the weight w0.

Quizz

- What has been the original motivation behind artificial neural networks? Neural networks.

Answer - The motivation has been to imitate biological neural networks, namely their learning and adaptation capabilities and parallel information processing. To imitate human learning, one: where no prior knowledge is present (new environment) and two: to overcome complex problems (pattern recognition and image processing)

- Give at least three examples of activation functions.

Answer - Examples of activation functions are: Rectified Linear Unit (ReLU) function, sigmoidal function, tangent hyperbolic, etc. An important property of the activation function is that it maps the activation $z \in (-\infty, \infty)$ onto some interval (e.g. $[-1,1]$ or $[0,\infty]$. Therefore, it is also called a squeezing function.

- Explain the term "training" of a neural network.

Answer- Training adjusts the parameters (weights and biases) of the network to minimize an error cost function. Training is the adaptation of weights in a network such that a certain goal is achieved. By training, the network learns to approximate a certain function from examples of input–output data pairs.