# Artificial Intelligence

## Lab 9

Artificial Neural Networks
ANNs

# Agenda

- Introduction
- Biological Background
- Building Blocks: Neurons
- Activation Function
- Network Architectures
- Perceptron

# Introduction

- The human brain is amazing at learning new things.
  - Why not use the model of the human brain to build a machine?

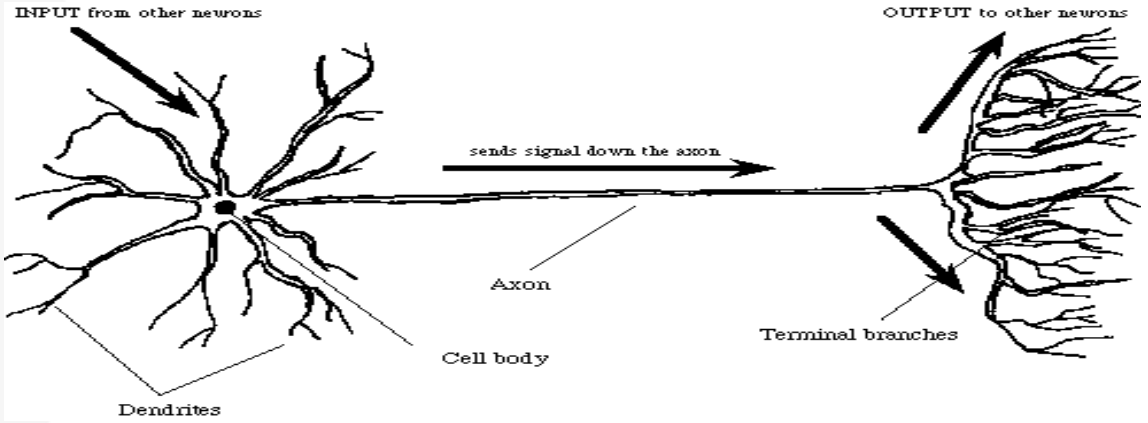
- An artificial Neural Network

  is a model designed to simulate the learning process and generalization ability of the human brain.

# Biological Background

- Our brain has about 100 billions nerve cells (**neurons**)
- A neuron may connect to as many as 100,000 other neurons
- Neuron consists of:
  - Cell body
  - Dendrites
  - Axon
  - Synapses

# Biological Background




- Signals “move” via electrochemical signals

- The synapses release a chemical transmitter – the sum of which can cause a threshold to be reached – causing the neuron to “fire”

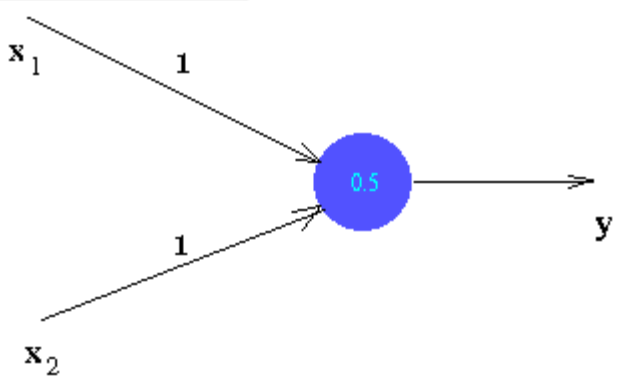
# Advantage and Application

Inherent Advantages of the Brain:

- Parallel processing speeds
- Adaptivity
- Fault tolerance
- Ability to generalize

# Building Blocks: Neurons

- First, we have to talk about neurons, the basic unit of a neural network.
- A neuron takes inputs, does some math with them, and produces one output. Here's what a 2-input neuron looks like:




**Finally, the sum is passed through an activation function:**

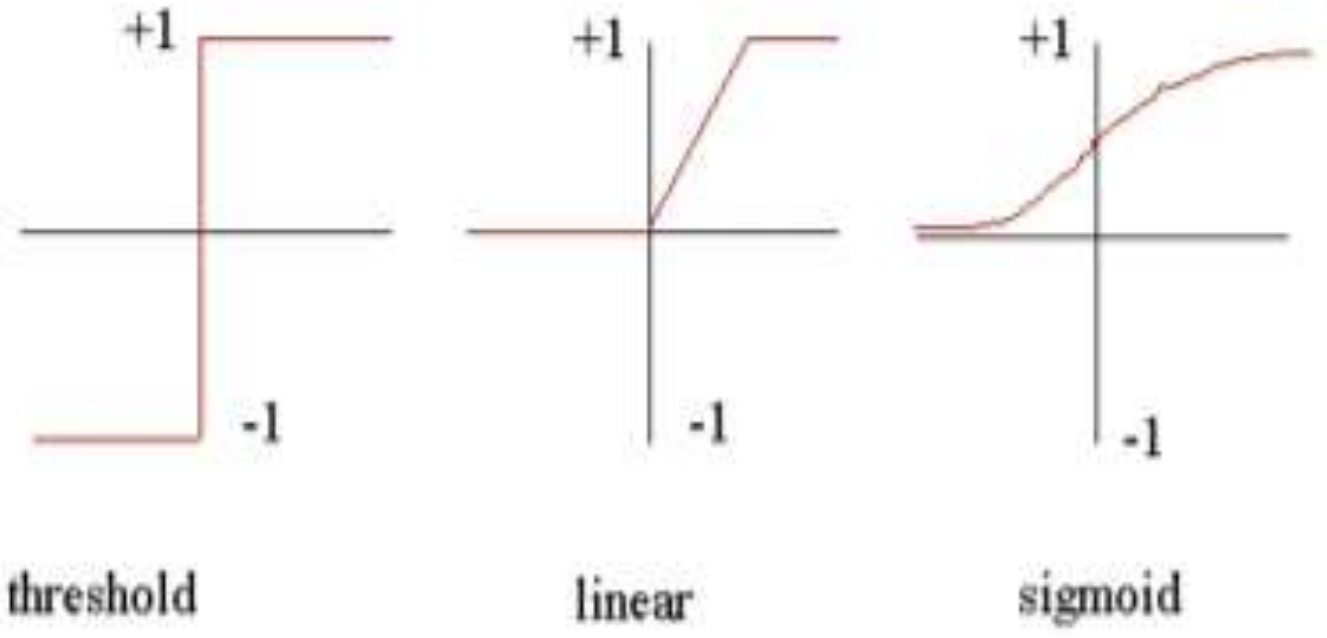
# Activation Function

- Their main purpose is to convert a input signal of a node in a ANN to an output signal.
- Output of neuron depends on the weighted sum of its input and activation function

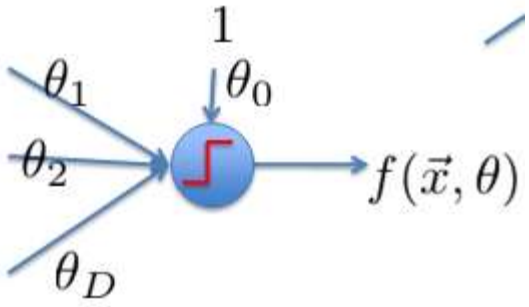
**The question arises that what happened if we don't apply activation function to the weighted sum of input?**

- the output signal would simply be a simple linear function
- A Neural Network without Activation function would simply be a Linear regression Model, which has limited power and does not performs good most of the times.

# Types of activation functions F

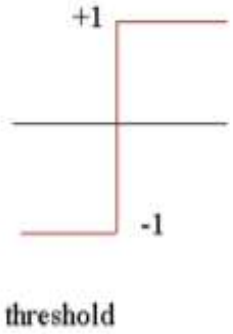


# Types of Activation Function




- Threshold function
  - The activation of a neuron is binary. That is, the neuron either fires (activation of one) or does not fire (activation of zero).

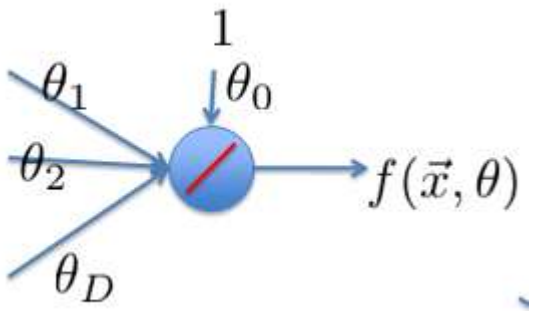
$$X = \sum_{i=1}^{n} x_i w_i \qquad Y = \begin{cases} +1, & \text{if } X \geq \theta \\ -1, & \text{if } X < \theta \end{cases}$$

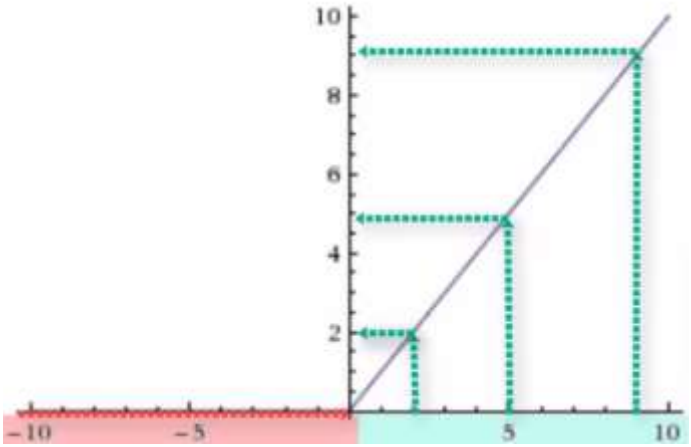



where X is the total input signal (weighted sum of input )received θ is the threshold for Y

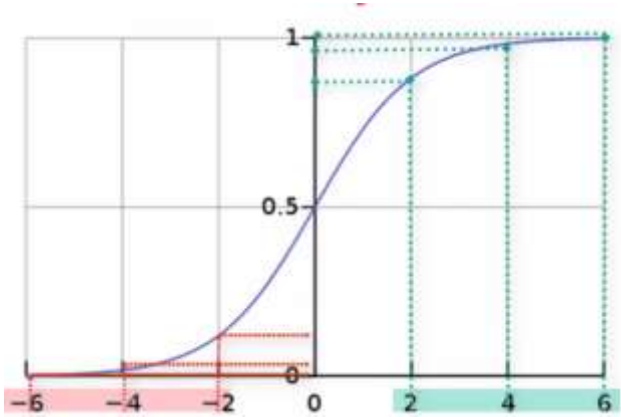
# Types of Activation Function

- Linear function
  - A straight line function where activation is proportional to input ( which is the weighted sum from neuron ).
  - Disadvantage: it is limited power



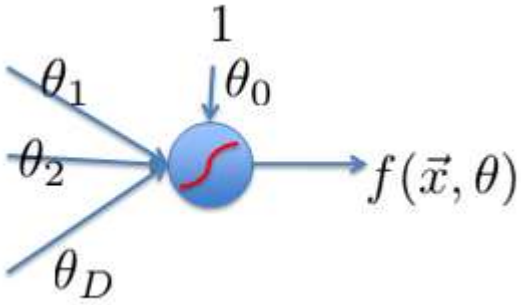



# Types of Activation Function




## Sigmoid Function

$$A = \frac{1}{1+e^{-x}}$$




Sigmoid functions are one of the most widely used activation functions today, but its problem is towards either end of the sigmoid function, the Y values tend to respond very less to changes in X.

# Activation Function (Example)

- For example, let us consider a simple neuron that has just two inputs. Each of these inputs has a weight associated with it, as follows: w1 = 0.8 and w2 = 0.4
- The inputs to the neuron are *x*1 and *x*2:

  x1 = 0.7 and x2 = 0.9
- So, the summed weight of these inputs is
- (0.8 x 0.7) + (0.4 x 0.9) = 0.92
-
- The activation level *Y*, is defined for this neuron as

$$Y = \begin{cases} +1 & for\ X > t \\ 0 & for\ X \leq t \end{cases}$$

- Hence, if *t* is less than 0.92, then this neuron will fire with this particular set of inputs. Otherwise, it will have an activation level of zero.

# Activation Function

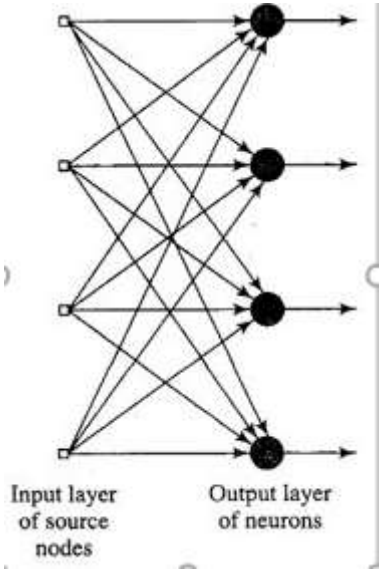
- A neural network consists of a set of neurons that are connected together.
- The connections between neurons have weights associated with them, and each neuron passes its output on to the inputs of the neurons to which it is connected.
- This **output depends on the application of the activation function to the inputs it receives**.
- In this way, an input signal to the network is processed by the entire network and an output (or multiple outputs) produced. There is **no central processing** or control mechanism – **the entire network is involved in every piece of computation** that takes place.

# Neural Network Architectures

- Artificial Neural Network is nothing more than a bunch of neurons connected together.
  - Single-Layer Feed-forward Networks
  - Multilayer Feed-forward Networks
  - Recurrent Networks

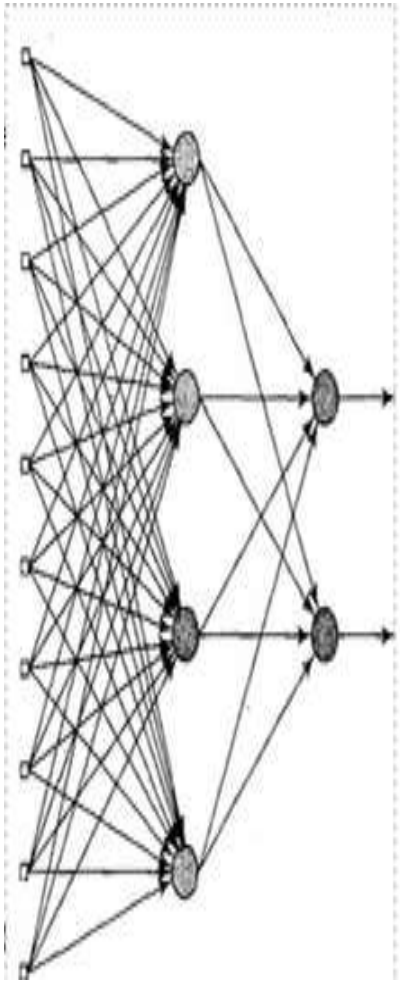
# Single-Layer Feed-forward Networks

- In a layered neural network the neurons are organized in the form of layers
- This network is strictly a feed-forward or acyclic type.
- We do not count the input layer of source nodes because no computation is performed there.



# Multilayer Feed-forward Networks

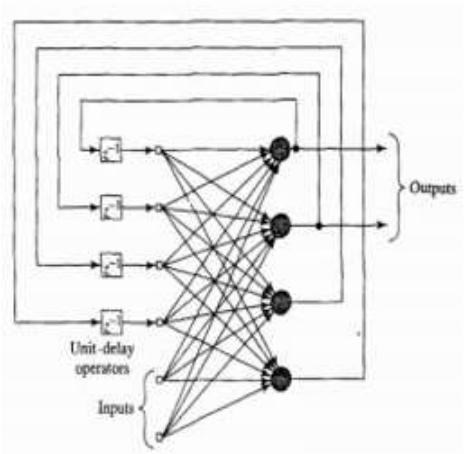
•The second class of a feed-forward neural network distinguishes itself by the presence of one or more hidden layers.
•The source nodes in the input layer of the network supply respective elements of the activation pattern (input vector), which constitute the input signals applied to the neurons (computation nodes) in the second layer (i.e., the first hidden layer).
•The output signals of the second layer are used as inputs to the third layer, and so on for the rest of the network. Typically the neurons in each layer of the network have as their inputs the output signals of the preceding layer only.



# Recurrent Networks

•It has at least one feedback loop.

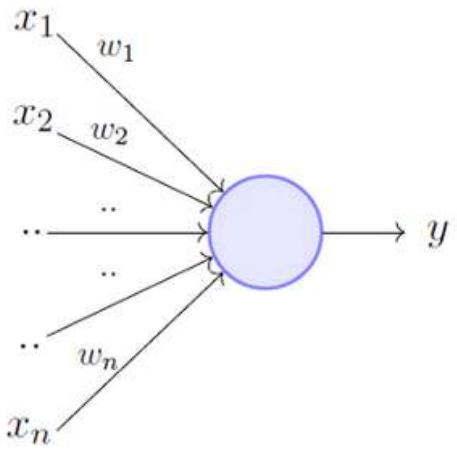
•For example, a recurrent network may consist of a single layer of neurons with each neuron feeding its output signal back to the inputs of all the other neurons; self feedback refers to a situation where the output of a neuron is fed back into its own input.




•The recurrent network may or may not has hidden neurons.

# Perceptron

- It takes an input
- aggregates it (weighted sum)
- returns 1 only if the aggregated sum is more than some threshold else returns 0.
- Update weights



# Perceptron

Steps

1-Initialize weights

2. Present a pattern and target output

3. Compute output $Y = Step\left(\sum_{i=0}^{n} w_i x_i\right)$

4. Update weights

Error: $E = y - f(x)$

Update weights: $W_j \leftarrow W_j + \alpha x_j E$

Where y is the expected output , f(x) is real output α learning rate and E error

5. Repeat step 2 until acceptable level of error

# Perceptron: Example (Logical Or)

- **Let us examine a simple example:** we will see how a perceptron can learn to represent the **logical-OR** function for two inputs. We will use a **threshold of zero** ($t = 0$) and a **learning rate of 0.2**.

- First, the weight associated with each of the two inputs is initialized to a random value between -1 and +1:

  **w1 = -0.2** and **w2 = 0.4**

- Now, the first epoch is run through. The training data will consist of the four combinations of 1's and 0's possible with two inputs.
  Hence, our first piece of training data is **x1 = 0 , x2 = 0** and our expected output is **x1 v x2 = 0**. We apply our formula for Y:

$$Y = Step\left(\sum_{i=0}^{n} w_i x_i\right)$$
$$= Step((0 \times -0.2) + (0 \times 0.4))$$

# Perceptron: Example (Logical Or)

- Hence, the output Y is as expected, and the error, e, is therefore 0. So the weights do not change.

- Now, for x1 = 0 and x2 = 1:
  Y = Step ((0 x -0.2) + (1 x 0.4))
  Y = Step (0.4) = 1
  Again, this is correct, and so the weights do not need to change.

- For x1 = 1 and x2 = 0:
  Y = Step ((1 x -0.2) + (0 x 0.4)) = Step (-0.2) = 0

This is incorrect because 1 v 0 = 1, so we should expect Y to be 1 for this set of inputs. Hence, the weights are adjusted.

# Perceptron: Example (Logical Or)

Error: $E = y - f(x)$

Update weights: $W_j \leftarrow W_j + \alpha x_j E$

- We will use the perceptron training rule to assign new values to the weights:

- Our **learning rate is 0.2**, and in this case, **the *e* is 1**, so we will assign the following value to *w*1:
  **w1 = -0.2 + (0.2 x 1 x 1) = -0.2 + (0.2) = 0**
  We now use the same formula to assign a new value to *w2*:
  **w2 = 0.4 + (0.2 x 0 x 1) = 0.4**
  Because *w2* did not contribute to this error, it is not adjusted.

- The final piece of training data is now used (***x*1 = 1** and ***x*2= 1**):
  **Y = Step ((0 x 1) + (0.4 x 1)) = Step (0 + 0.4) = Step (0.4) = 1**
  This is correct, and so the weights are not adjusted.

**This is the end of the first epoch, and at this point the method runs again and continues to repeat until all four pieces of training data are classified correctly.**

# Perceptron

| Epoch | X1 | X2 | Expected Y | Actual Y | Error | w1 | w2 |
|---|---|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 0 | −0.2 | 0.4 |
| 1 | 0 | 1 | 1 | 1 | 0 | −0.2 | 0.4 |
| **1** | **1** | **0** | **1** | **0** | **1** | **0** | **0.4** |
| 1 | 1 | 1 | 1 | 1 | 0 | 0 | 0.4 |
| 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0.4 |
| 2 | 0 | 1 | 1 | 1 | 0 | 0 | 0.4 |
| **2** | **1** | **0** | **1** | **0** | **1** | **0.2** | **0.4** |
| 2 | 1 | 1 | 1 | 1 | 0 | 0.2 | 0.4 |
| 3 | 0 | 0 | 0 | 0 | 0 | 0.2 | 0.4 |
| 3 | 0 | 1 | 1 | 1 | 0 | 0.2 | 0.4 |
| 3 | 1 | 0 | 1 | 1 | 0 | 0.2 | 0.4 |
| 3 | 1 | 1 | 1 | 1 | 0 | 0.2 | 0.4 |

# Hands on

Open Perceptron template to complete the function and implement Or Logical Function using Step function

# Hands on

Threshold = 0

Learning rate = 0.2

Training set input= [0, 0, 1],
[1, 1, 1],
[1, 0, 1],
[0, 1, 1],
[0, 0, 0]

Output=[1,1,1,1,0]

Test with [1,1,0] expected output =1

# Questions?