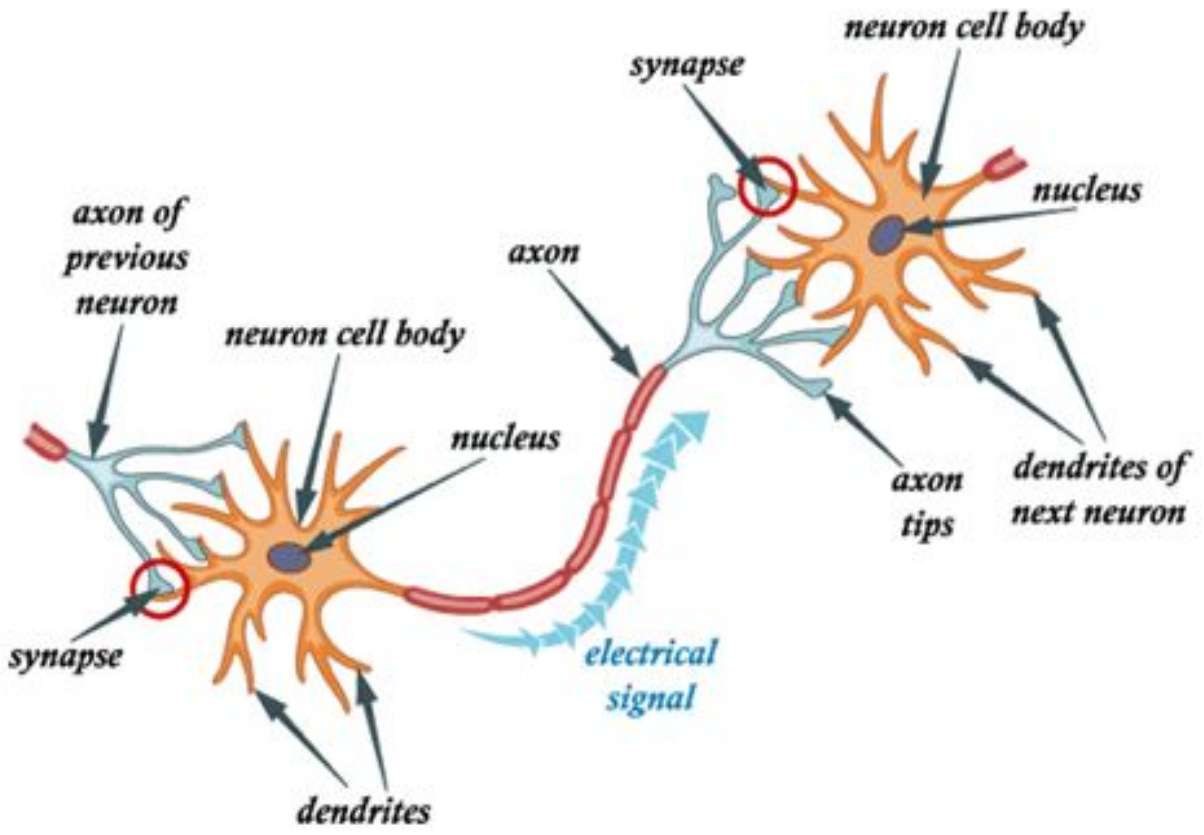
Artificial Neural Networks

Brain Processes



- External information / stimuli is received by the **dendrites** of the neuron
- processed in **the neuron cell body**
- converted to an output and passed through the **Axon** to the next neuron.
- The next neuron can choose to either accept it or reject it depending on the strength of the signal.

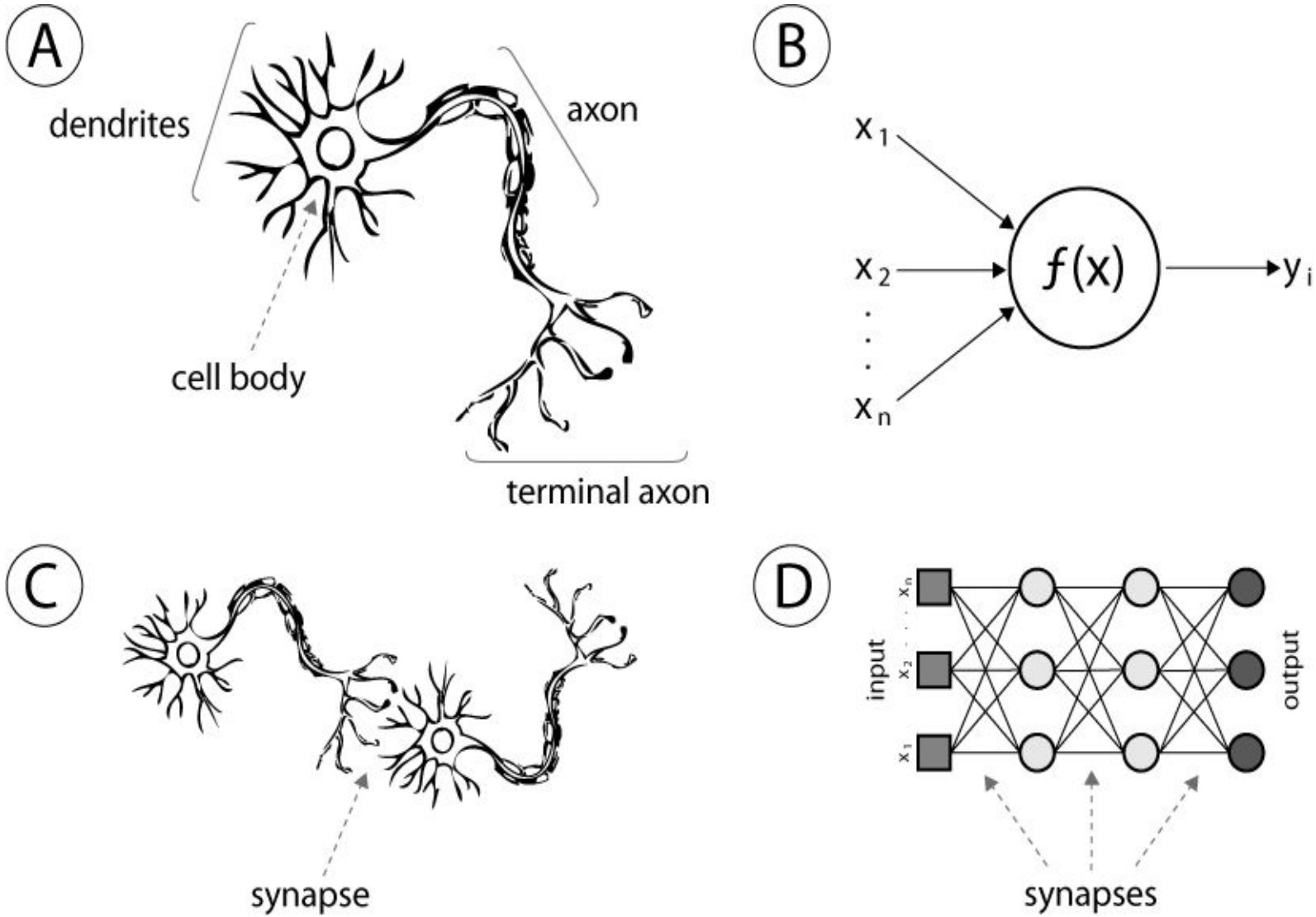
Step 1: External signal received by dendrites

Step 2: External signal processed in the neuron cell body

Step 3: Processed signal converted to an output signal and transmitted through the Axon

Step 4: Output signal received by the dendrites of the next neuron through the synapse

Neural Network continued..



Neural Networks

- Computational models inspired by the human brain:
  - Algorithms that try to mimic the brain.
  - Massively parallel, distributed system, made up of simple processing units (neurons)
  - Synaptic connection strengths among neurons are used to store the acquired knowledge.
  - Knowledge is acquired by the network from its environment through a learning process

History

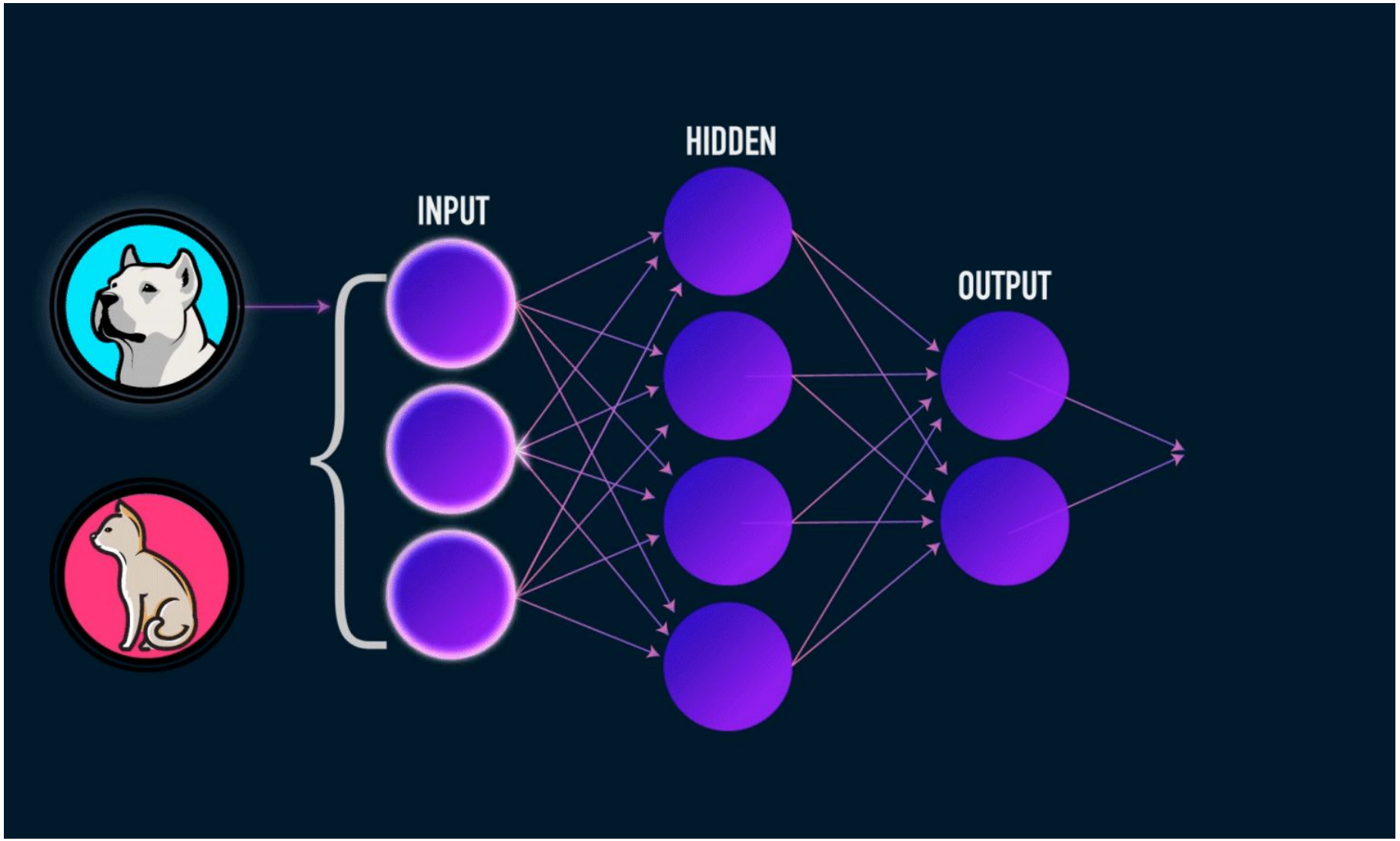
- late-1800's - Neural Networks appear as an analogy to biological systems
- 1960's and 70's – Simple neural networks appear
  - Fall out of favour because the perceptron is not effective by itself, and there were no good algorithms for multilayer nets
- 1986 – Backpropagation algorithm appears
  - Neural Networks have a resurgence in popularity
  - More computationally expensive

Applications of ANNs

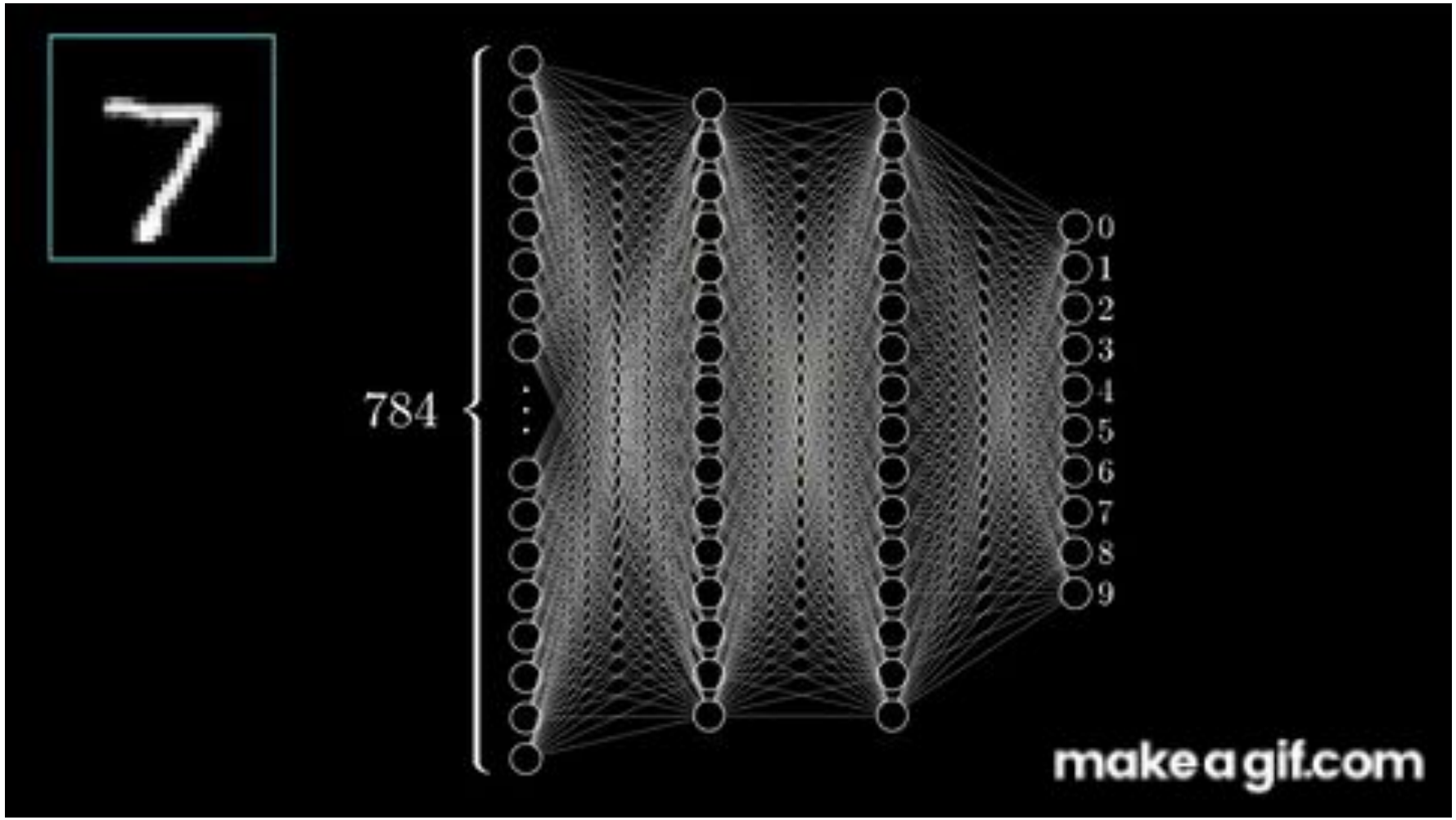
Applications of ANNs

- ANNs have been widely used in various domains for:
  - Handwriting Recognition
    - Optical character recognition for data entry
    - Validation of signatures on a bank cheque
  - Image Compression
  - Data clustering

Artificial Neural Network – Example 1



Artificial Neural Network – Example 2



Properties

- Inputs are flexible
  - any real values
  - Highly correlated or independent
- Target function may be discrete-valued, real-valued, or vectors of discrete or real values
  - Outputs are real numbers between 0 and 1
- Resistant to errors in the training data
- Long training time
- Fast evaluation
- The function produced can be difficult for humans to interpret

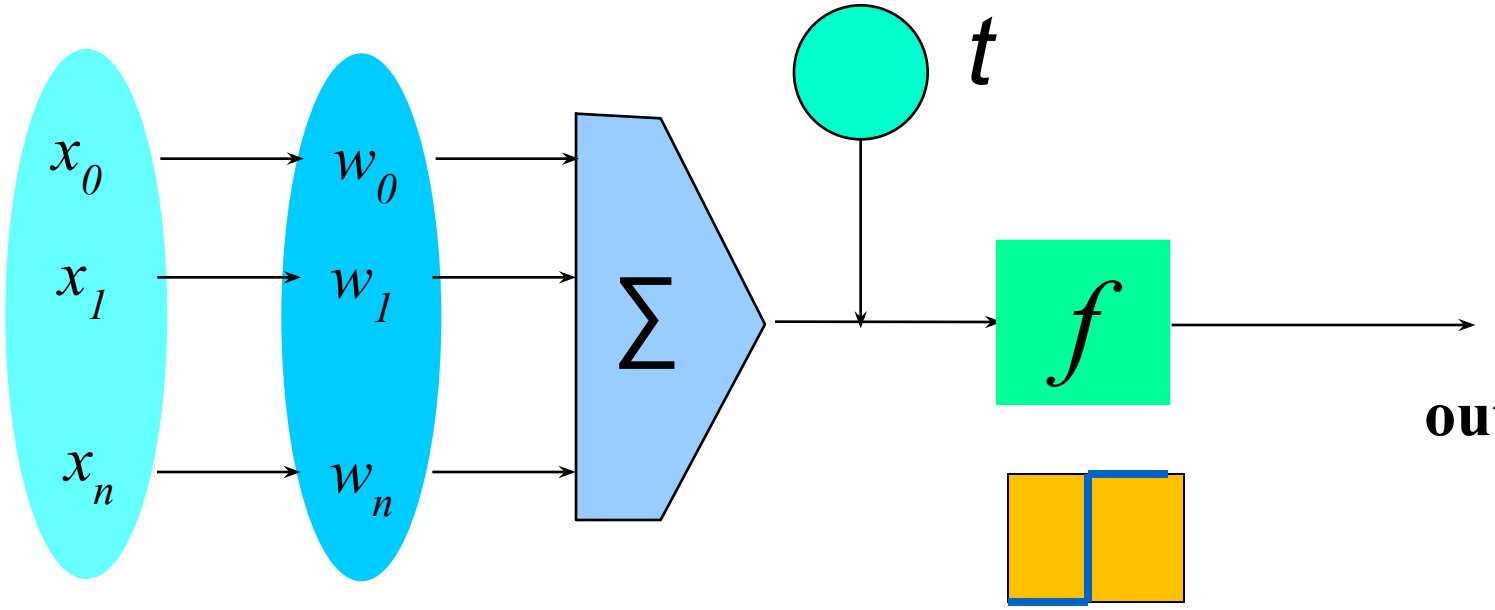
When to consider neural networks

- Input is high-dimensional discrete or raw-valued
- Output is discrete or real-valued
- Output is a vector of values
- Possibly noisy data
- Form of target function is unknown
- Human readability of the result is not important

Examples:

- Speech phoneme recognition
- Image classification
- Financial prediction

A Neuron (= a perceptron)



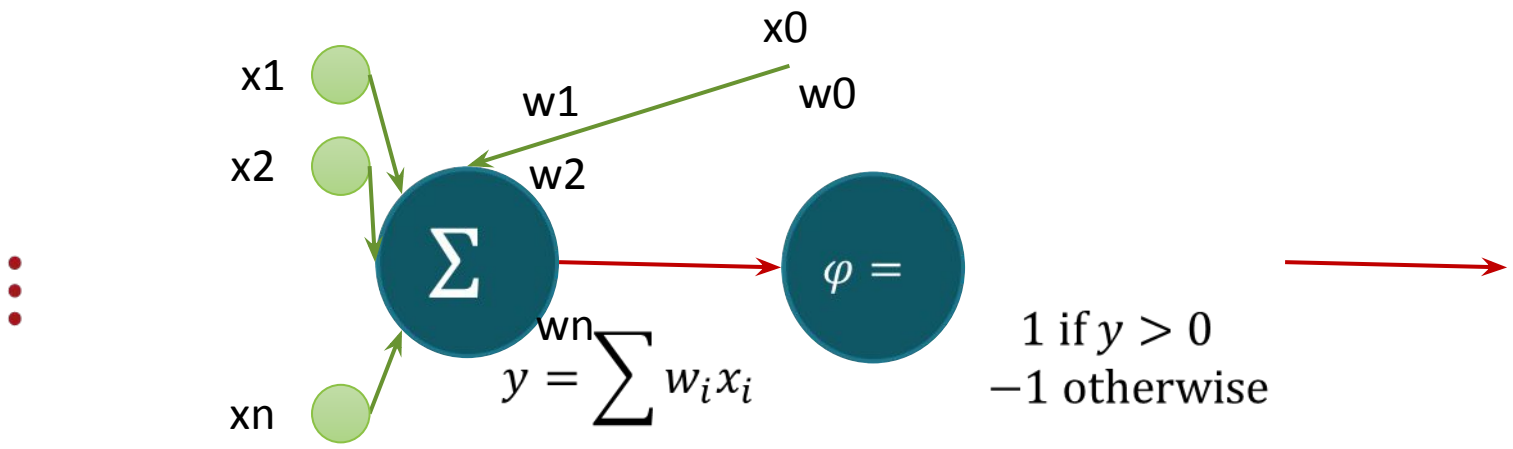
For Example

$$y = \text{sign}(\sum_{i=0}^{n} w_i x_i - t)$$

- The *n*-dimensional input vector **x** is mapped into variable y by means of the scalar product and a nonlinear function mapping

Perceptrons

- Basic unit in a neural network: Linear separator
  - N inputs, x1 ... xn
  - Weights for each input, w1 ... wn
  - A bias input x0 (constant) and associated weight w0
  - Weighted sum of inputs, $y = \sum_{i=0}^{n} w_i x_i$
  - A threshold function, i.e., 1 if y > 0, −1 if y <= 0



Perceptron

- It is based on a slightly different artificial neuron called a ***linear threshold unit* (LTU)**: the inputs and output are now numbers (instead of binary on/off values) and each input connection is associated with a weight.
  - Parts
    - N inputs, $x_1 \ldots x_n$
    - Weights for each input, $w_1 \ldots w_n$
    - A bias input x_0 (constant) and associated weight w_0
    - Weighted sum of inputs, $y = w_0x_0 + w_1x_1 + \ldots + w_nx_n$
    - A threshold function or activation function,
      - i.e 1 if $y > t$, -1 if $y <= t$

How is a Perceptron trained?

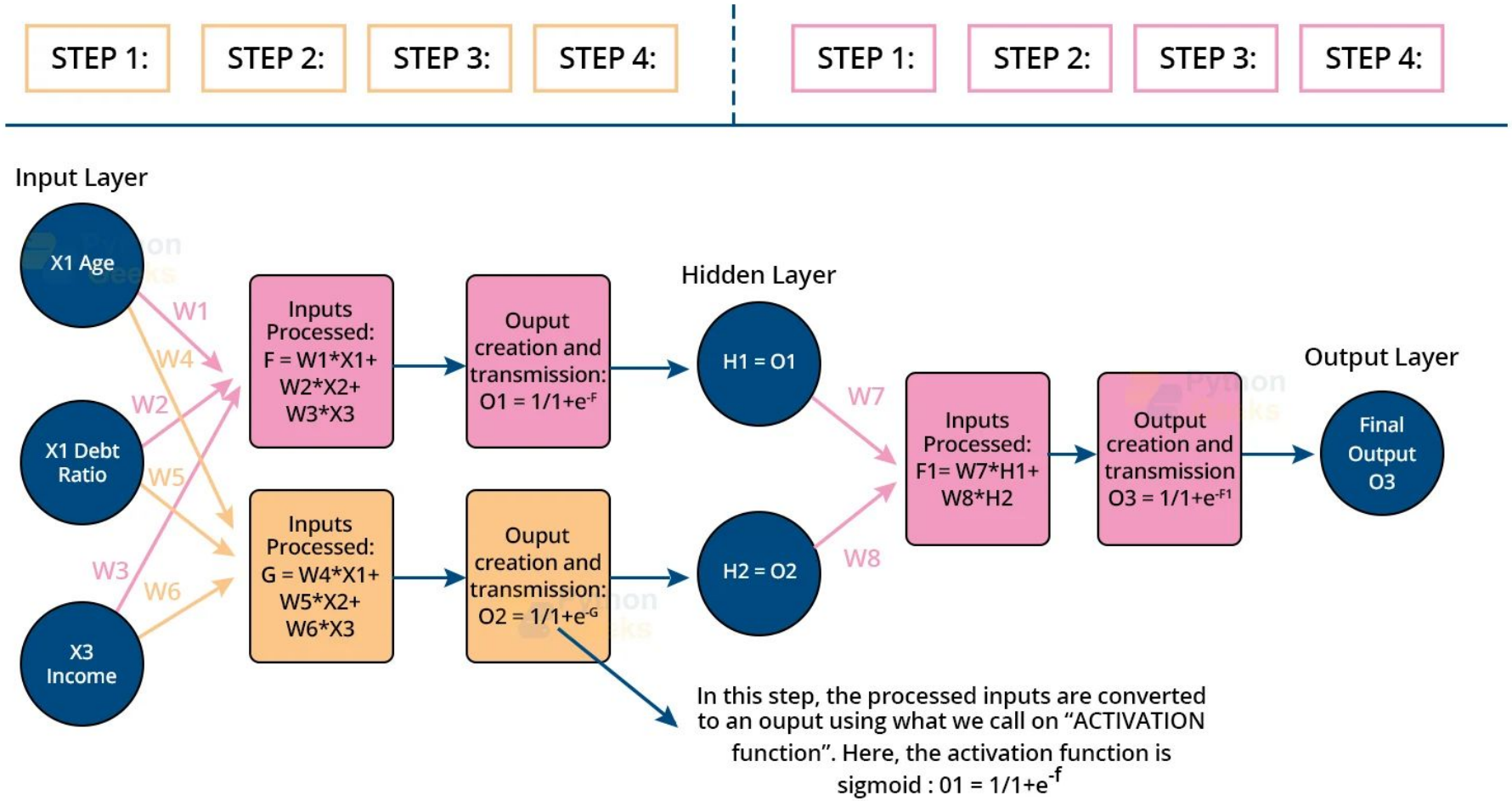
- "**Cells that fire together, wire together**". This rule later became known as Hebb's rule (or *Hebbian learning*).

- The connection weight between two neurons is increased whenever they have the same output.

- Perceptrons are trained using a variant of this rule that takes into account the error made by the network; it does not reinforce connections that lead to the wrong output.

- More specifically, the Perceptron is fed one training instance at a time, and for each instance it makes its predictions. For every output neuron that produced a wrong prediction, it reinforces the connection weights from the inputs that would have contributed to the correct prediction.

Training Rule



Training Rule

- ***How to learn weights automatically?***

- Here the precise learning problem is to determine a weight vector that causes the perceptron to produce the correct (+1 or -1) output for each of the given training examples.

- Following two algorithms are well known for solving this learning problem
  1. ***The Perceptron Training Rule***
  2. ***Gradient Descent and the Delta Rule***

- These two rules are important to ANNs because they provide the basis for learning networks of many units

Training Rule

- Initialize random weights
- ***Repeat***
  - Iteratively apply the perceptron to each training example
  - Modifying the perceptron weights whenever it misclassifies an example (using the *perceptron training rule)*.
- ***Until*** the perceptron classifies all training examples correctly.

- **Perceptron training rule is as follows**

$$w_i = w_i + \Delta w_i$$

$$\Delta w_i = \eta(y - \hat{y})x_i$$

Perceptron learning rule (weight update)

$$w_i = w_i + \Delta w_i$$

$$\Delta w_i = \eta(y - \hat{y})x_i$$

- $w_{i,\ j}$ is the connection weight between the ith input neuron and the jth output neuron.
- x_i is the ith input value of the current training instance.
- $\tilde{y}_j$ is the output of the jth output neuron for the current training instance.
- y_j is the target output of the jth output neuron for the current training instance.
- η is the learning rate (the amount that the weights are updated during training (0.1 to 1.0))

The Perceptron Training Rule

Convergence

- It can be proved that this update rule will converge towards successful weight values provided
  - Training data is linearly separable and
  - η is sufficiently small

To understand the convergence, consider the following cases:

- ***Case 1:*** The training example is correctly classified by the perceptron.
  - In this case, $(t - o)$ is zero, making $\triangle w_i$ = zero, so that no weights are updated.

The Perceptron Training Rule

- ***Case 2 (a):*** If $t = 1$, and $o = -1$,
  - In this case, to make the perceptron output a + 1 instead of - 1, the weights must be altered such that the value of ***w.x***. is **increased.**
    - For example, if $x_i > 0$ then

- ***Case 2 (b):***, if $t = -1$ and $o = 1$, then weights associated with positive x_i will be **decreased** rather than increased.

Inferences from Neural Network

- If the data is linearly separable and η is sufficiently small, it will converge to a hypothesis that classifies all training data correctly in a finite number of iterations
- The decision boundary of each output neuron is linear.
- Perceptrons are **incapable** of **learning complex patterns**.
- However, if the training instances are linearly separable, **Rosenblatt** demonstrated that this algorithm would converge to a solution.
- This is called the ***Perceptron convergence theorem***.
- Limitations of Perceptrons can be eliminated by stacking multiple Perceptrons.
- The resulting ANN is called a ***Multi-Layer Perceptron*** (MLP)

Gradient Descent

- Perceptron training rule may not converge if points are not linearly separable
- Gradient descent by changing the weights by the total error for all training points.
  - If the data is not linearly separable, then it will converge to the best fit

Gradient Descent With Delta Rule

- **Performance of Perceptron training rule**
  - Although the perceptron rule finds a successful weight vector when the training examples are linearly separable, it can fail to converge if the examples are not linearly separable.

- A second training rule, called the delta rule, is designed to overcome this difficulty.

- The key idea behind the delta rule is to use ***gradient descent*** to search the hypothesis space of possible weight vectors to find the weights that best fit the training examples

Gradient Descent

- To understand, consider simpler *linear unit*

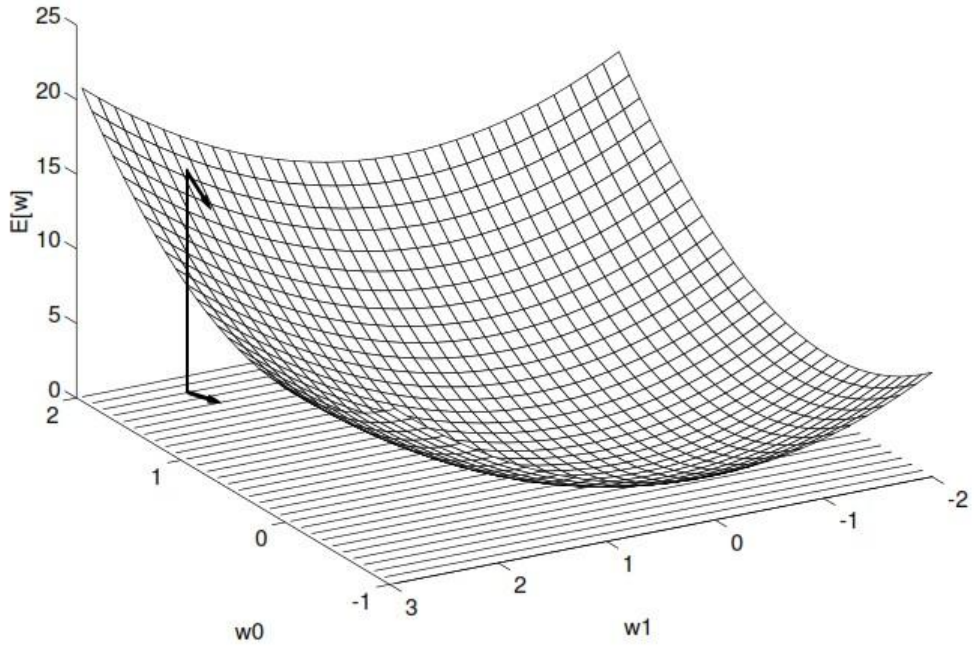
$$o = w_0 + w_1 x_1 + \cdots + w_n x_n$$

- *Let's learn w_i's that minimize squared error*

$$E = \frac{1}{2} \sum_j (y - \hat{y})^2$$

- *Where D is the set of training examples*

Gradient Descent



- Gradient $$\nabla E[\vec{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots \frac{\partial E}{\partial w_n}\right]$$
- Training rule $$\Delta \vec{w} = -\eta \nabla E[\vec{w}]$$
- i.e. $$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$ ---------> Eq(i)

Gradient Descent Algorithm

GRADIENT-DESCENT($training\_examples, \eta$)

Each training example is a pair of the form $\langle \vec{x}, t \rangle$*, where* $\vec{x}$ *is the vector of input values, and* t *is the target output value.* η *is the learning rate (e.g., .05).*

- Initialize each w_i to some small random value
- Until the termination condition is met, Do
  - Initialize each Δw_i to zero.
  - For each $\langle \vec{x}, t \rangle$ in *training_examples*, Do
    - Input the instance $\vec{x}$ to the unit and compute the output o
    - For each linear unit weight w_i, Do

$$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$$

  - For each linear unit weight w_i, Do

$$w_i \leftarrow w_i + \Delta w_i$$

Algorithm: Gradient Descent Algorithm for training a linear unit

Convergence

- Because the error surface contains only a single global minimum, this algorithm will converge to a weight vector with minimum error, regardless of whether the training examples are linearly separable, given a sufficiently small learning rate η is used.

Remarks On The Two Training Rules

The key difference between these algorithms is that

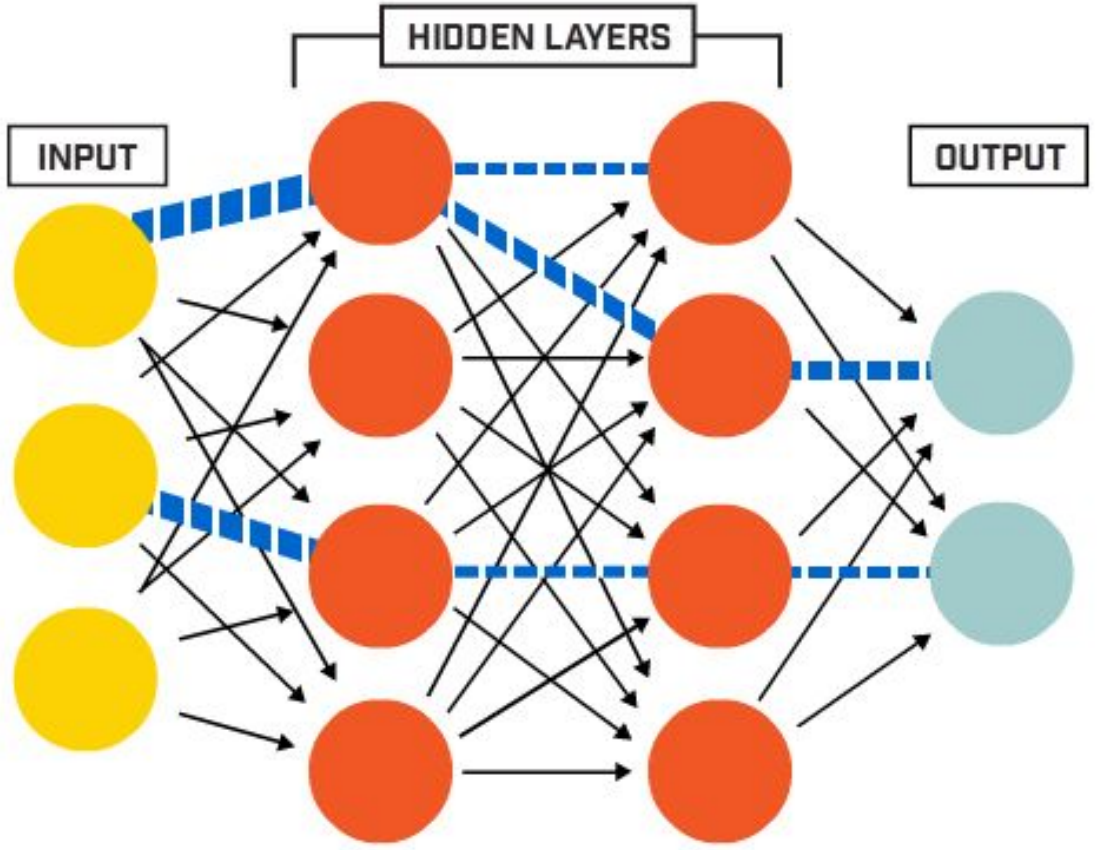
- The perceptron training rule updates weights based on the error in the thresholded perceptron output
- Whereas, the delta rule updates weights based on the error in the unthresholded linear combination of inputs.

- ***Convergence properties.***
  - The perceptron training rule converges after a finite number of iterations to a hypothesis that perfectly classifies the training data, provided the training examples are linearly separable.
  - The delta rule converges toward the minimum error hypothesis, possibly requiring unbounded time, but converges regardless of whether the training data are linearly separable.

ANN

- ANN possess a large number of processing elements called nodes/neurons which operate in parallel.
- Neurons are connected with others by connection link.
- Each link is associated with weights which contain information about the input signal.
- Each neuron has an internal state of its own which is a function of the inputs that neuron receives- Activation level

Activation Function



Activation Functions

1. **Identity Function**
   f(x)=x for all x
2. **Binary Step function**

$$f(x) = \begin{cases} 1\ if x \geq \theta \\ 0\ if x < \theta \end{cases}$$

3. **Bipolar Step function**

$$f(x) = \begin{cases} 1\ if x \geq \theta \\ -1\ if x < \theta \end{cases}$$

4. **Sigmoidal Functions:- Continuous functions**
5. **Ramp functions**

$$f(x) = \begin{cases} 1\ if x > 1 \\ x\ if\ 0 \leq x \leq 1 \\ 0\ if x < 0 \end{cases}$$

Activation Functions

- **The *hyperbolic tangent* function**

  $\tanh(z) = 2^{\sigma}(2^{z}) - 1$

- **The *sigmoid* function**

  $S(x) = 1/(1 + e^{-x})$

- **The ReLU function**

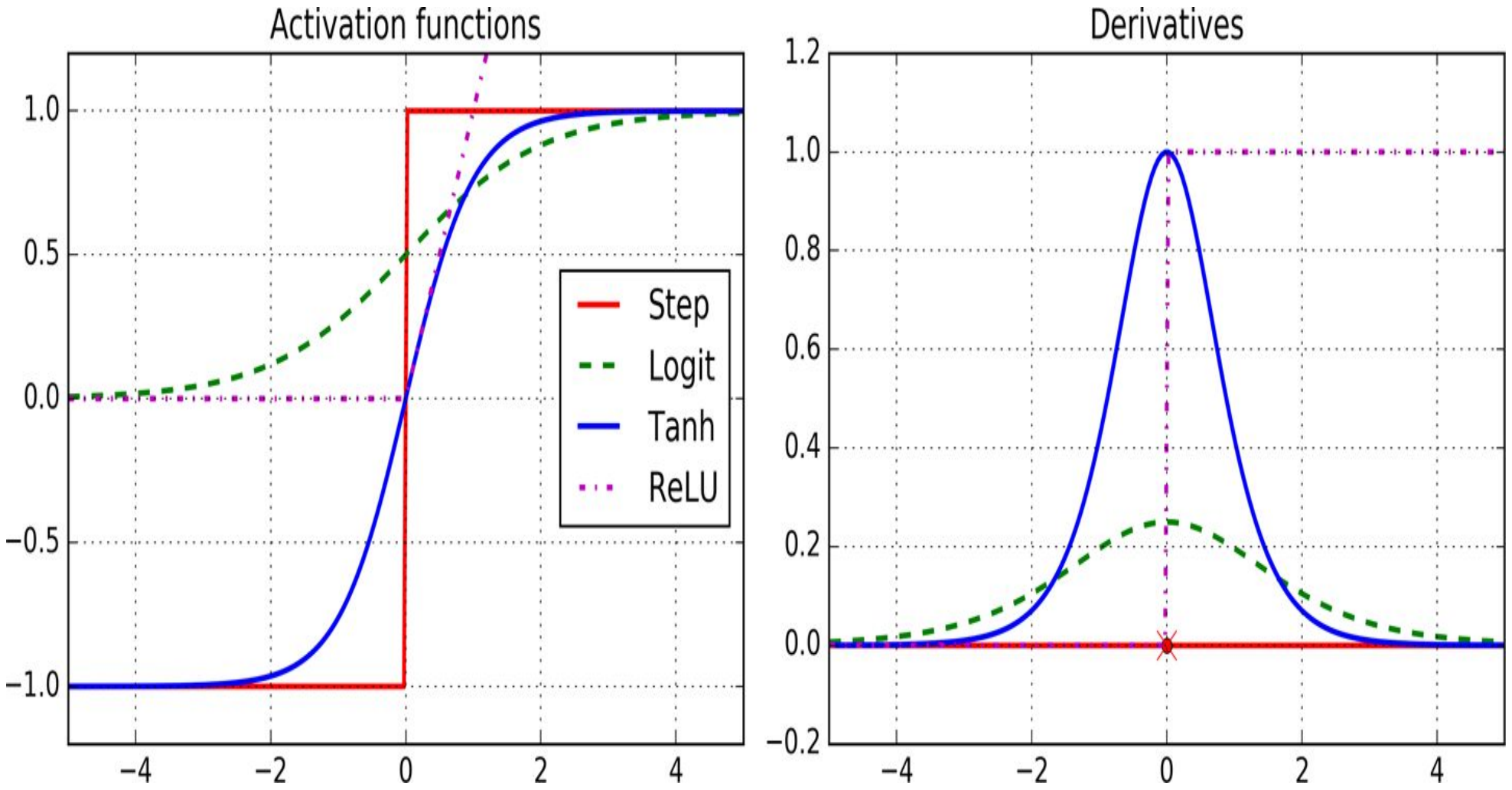


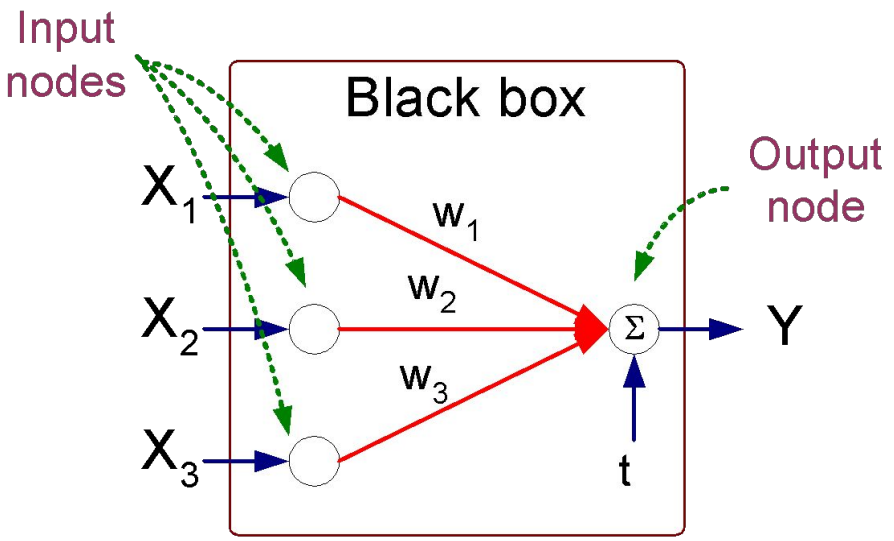
Fig. Activation functions and their derivatives

Artificial Neural Networks (ANN)

- Model is an assembly of inter-connected nodes and weighted links

- Output node sums up each of its input value according to the weights of its links

- Compare output node against some threshold t



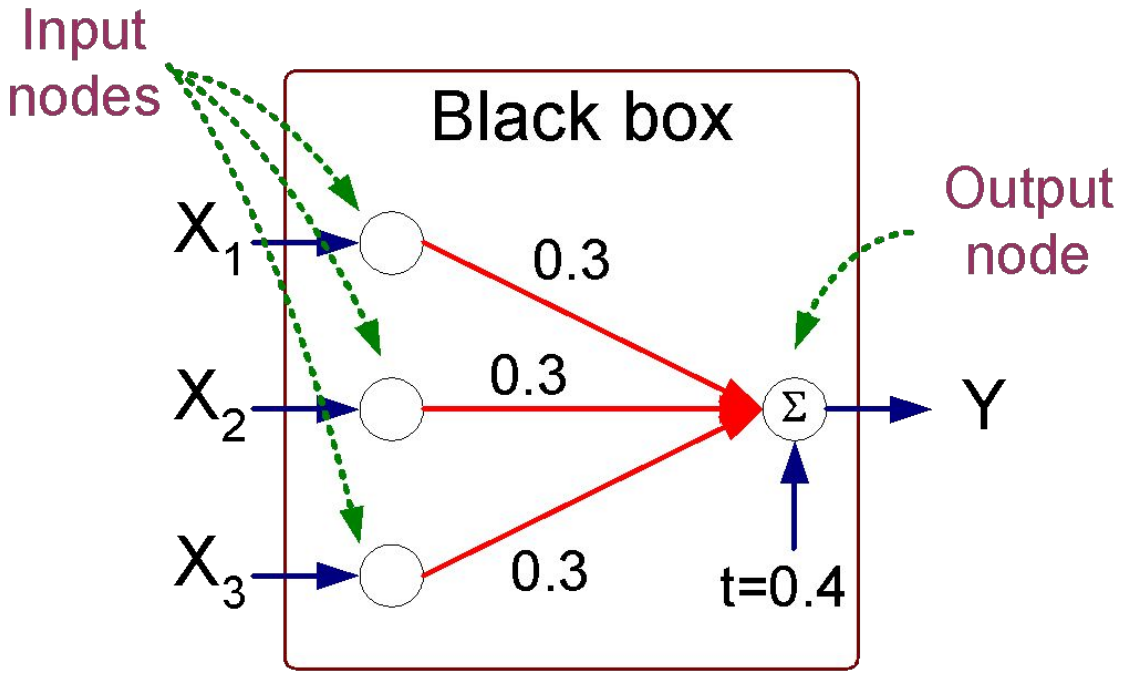
Perceptron Model

$$Y = I(\sum_i w_i x_i - t)$$ or

$$Y = sign(\sum_i w_i x_i - t)$$

Artificial Neural Networks (ANN)

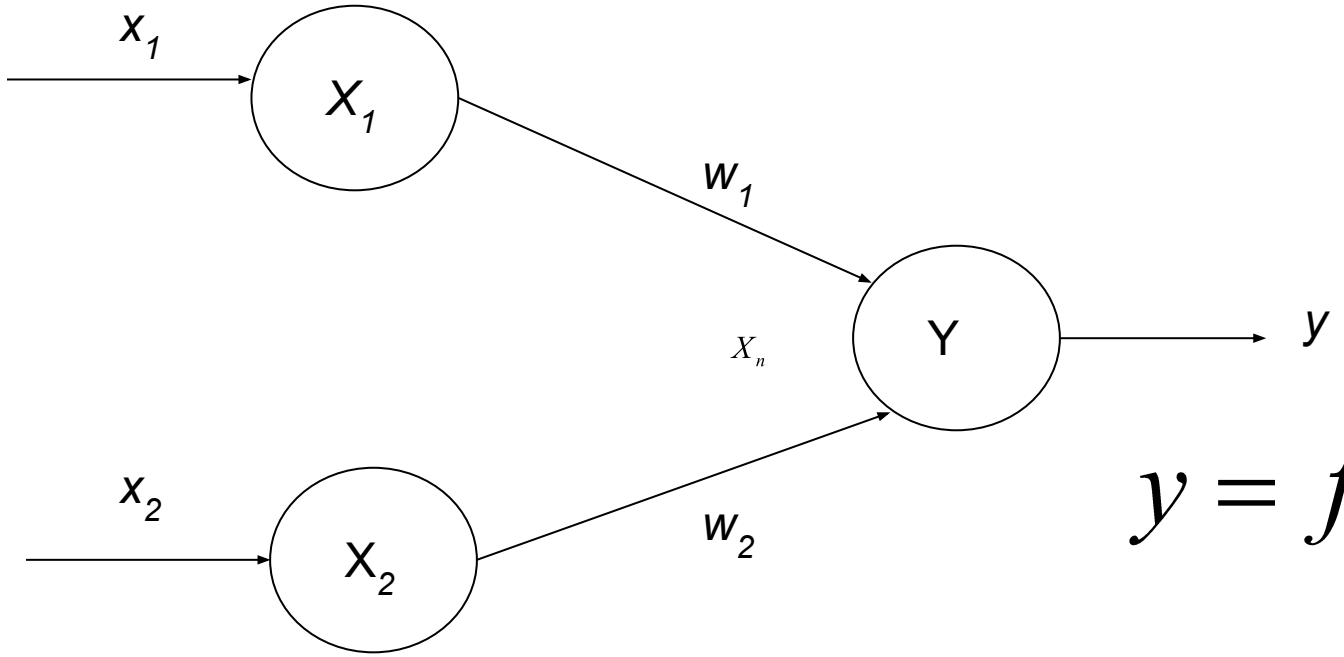
X_1	X_2	X_3	Y
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	0
0	1	0	0
0	1	1	1
0	0	0	0



$$Y = I(0.3X_1 + 0.3X_2 + 0.3X_3 - 0.4 > 0)$$

$$\text{where } I(z) = \begin{cases} 1 & \text{if } z \text{ is true} \\ 0 & \text{otherwise} \end{cases}$$

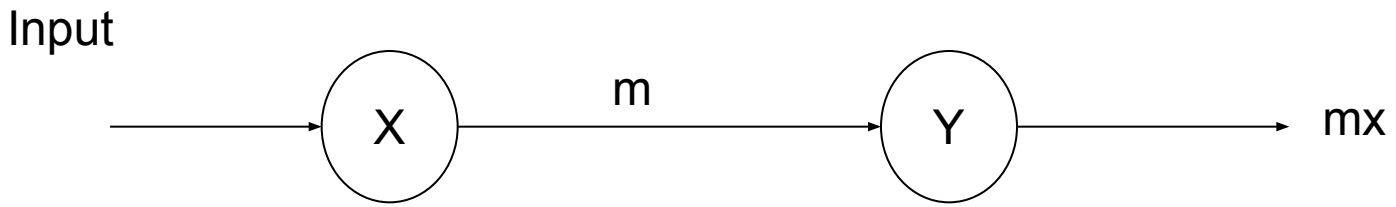
Artificial Neural Networks



$$y = f(y_{in})$$

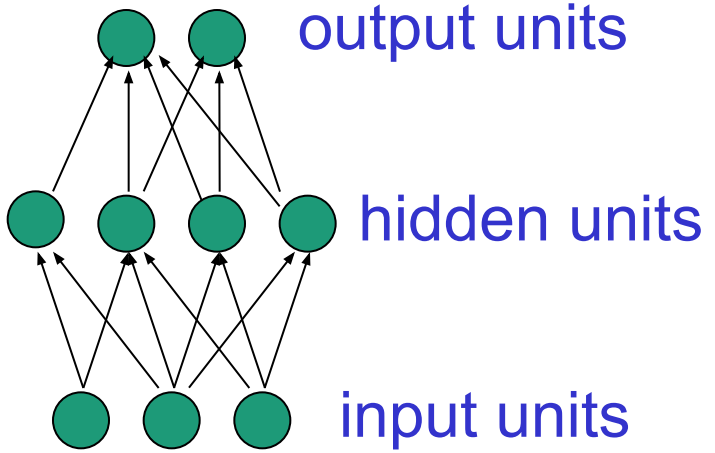
$$y_{in} = x_1 w_1 + x_2 w_2$$

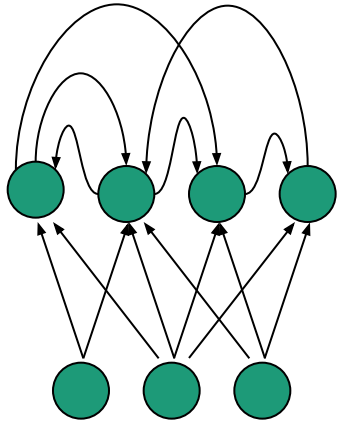
Neural Network of pure linear equation.



Types of connectivity

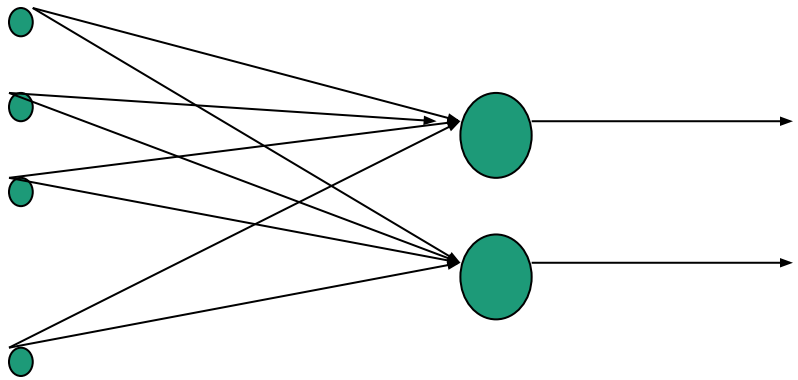
- Feedforward networks
  - These compute a series of transformations
  - Typically, the first layer is the input and the last layer is the output.
- Recurrent networks
  - These have directed cycles in their connection graph. They can have complicated dynamics.
  - More biologically realistic.





Different Network Topologies

- Single layer feed-forward networks
  - Input layer projecting into the output layer

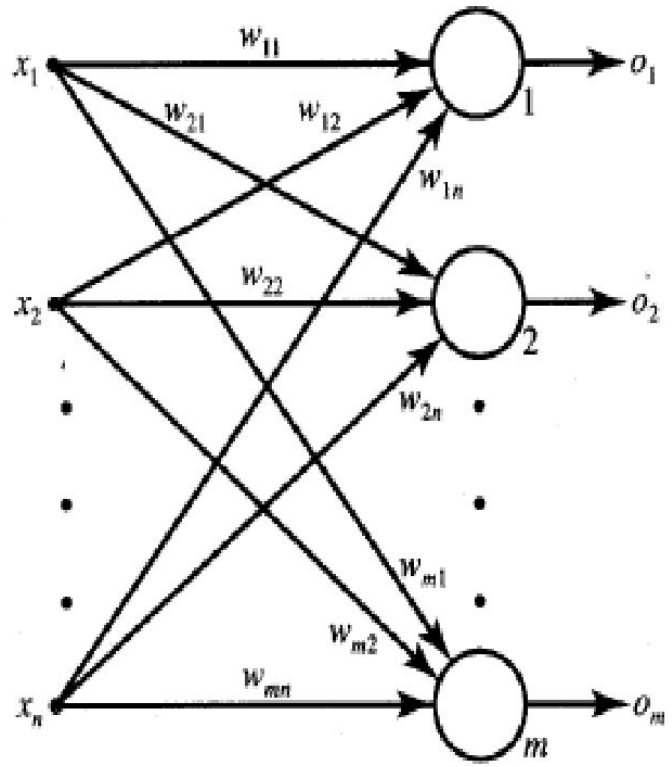


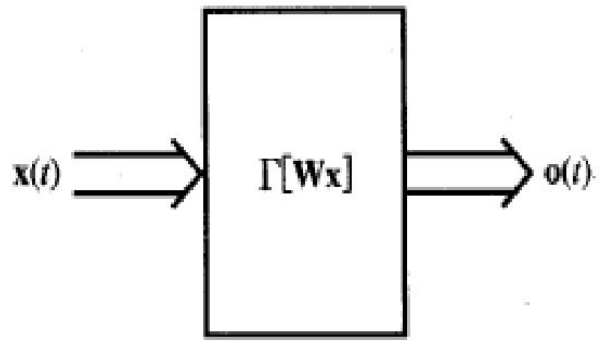
Single layer network

Input layer

Output layer

Single layer Feedforward Network





Feedforward Network

- Its output and input vectors are respectively

$$\mathbf{o} = [o_1 \quad o_2 \quad \cdots \quad o_m]^t$$
$$\mathbf{x} = [x_1 \quad x_2 \quad \cdots \quad x_n]^t$$

- Weight w_{ij} connects the *i'th* neuron with *j'th* input. Activation rule of ith neuron is

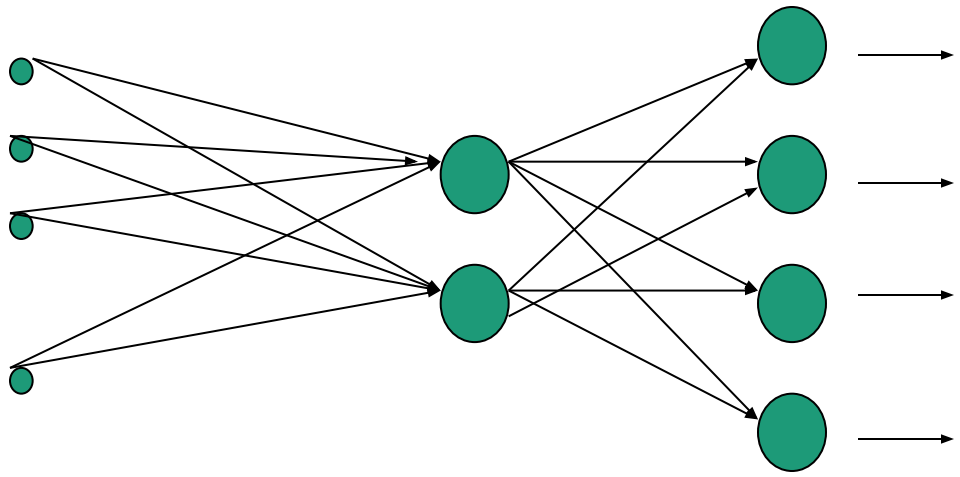
$$net_i = \sum_{j=1}^{n} w_{ij} x_j, \quad \text{for } i = 1, 2, \ldots, m$$

$$o_i = f(\mathbf{w}_i^t \mathbf{x}), \quad \text{for } i = 1, 2, \ldots, m$$

$$\mathbf{w}_i \triangleq [w_{i1} \quad w_{i2} \quad \cdots \quad w_{in}]^t$$

Different Network Topologies

- Multi-layer feed-forward networks
  - One or more hidden layers. Input projects only from previous layers onto a layer.



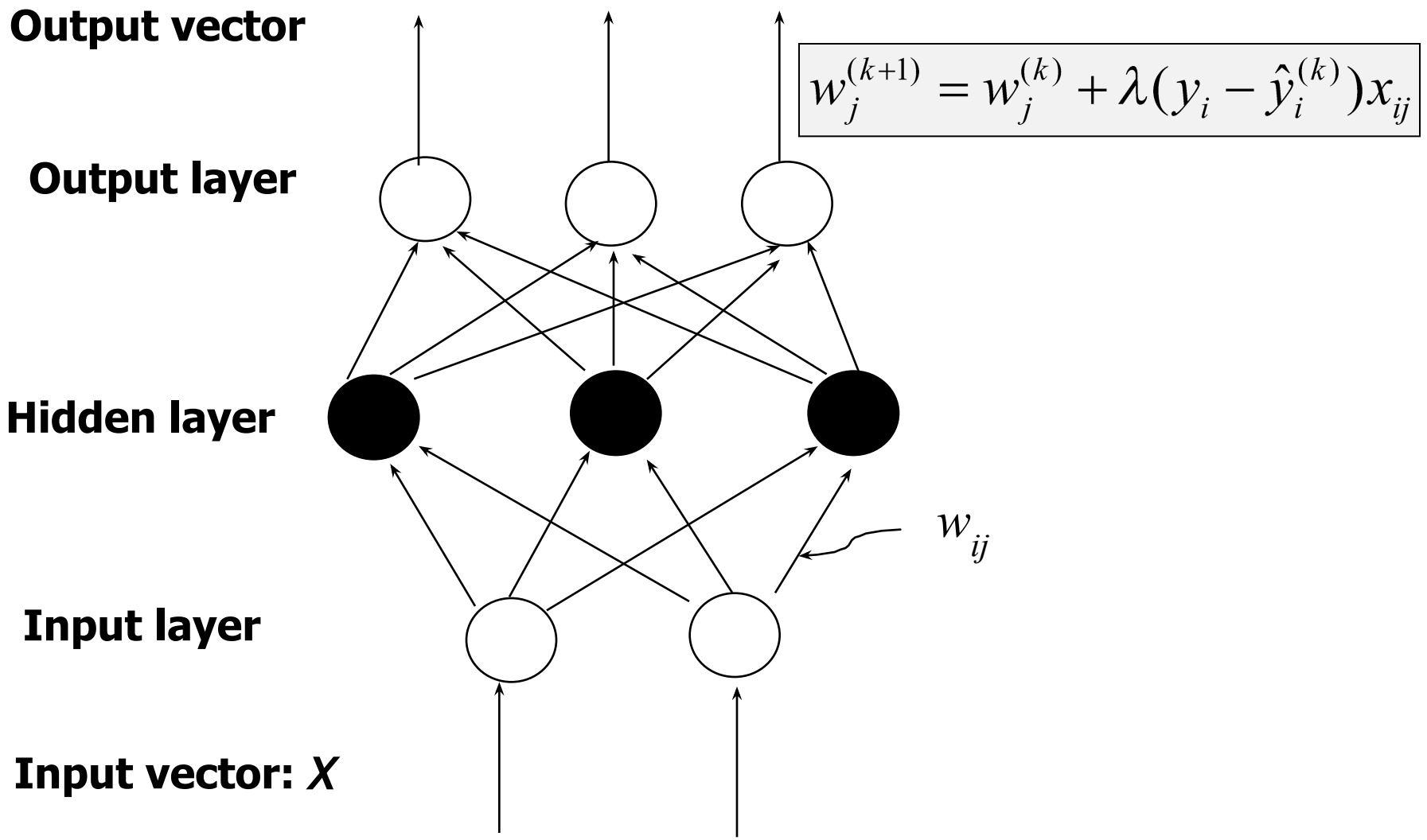
Input layer

Hidden layer

Output layer

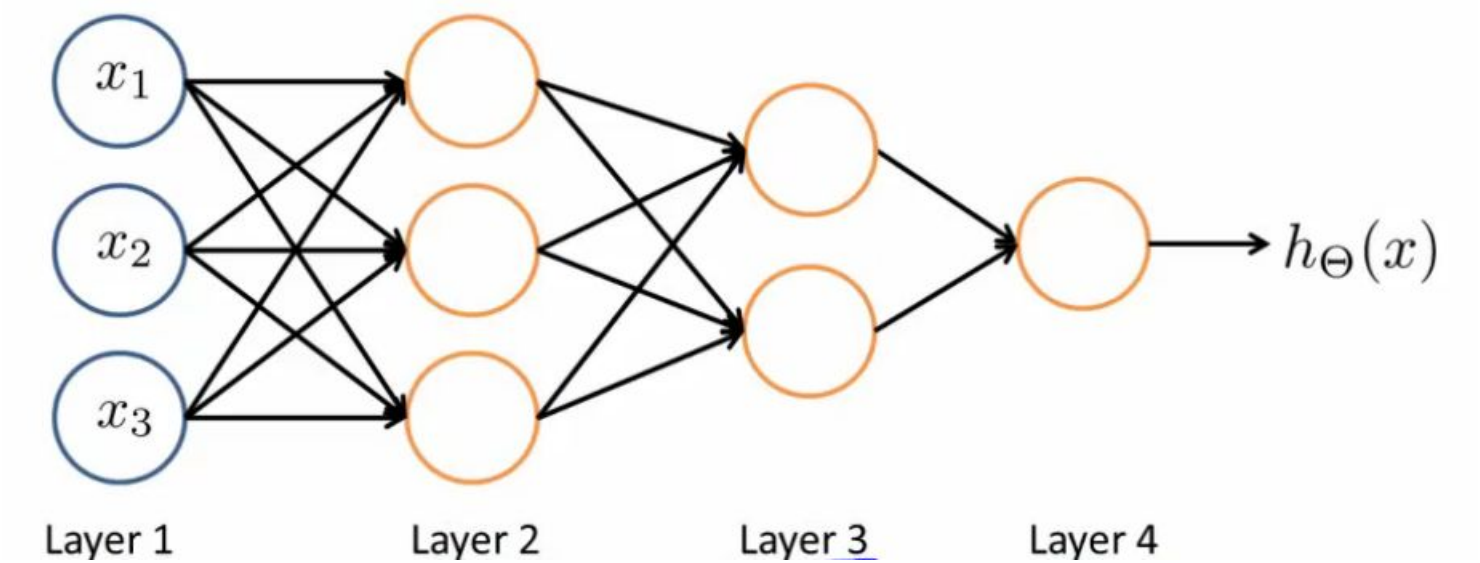
2-layer or
1-hidden layer
fully connected
network

A Multi-Layer Feed-Forward Neural Network



Different Network Topologies

- Multi-layer feed-forward networks



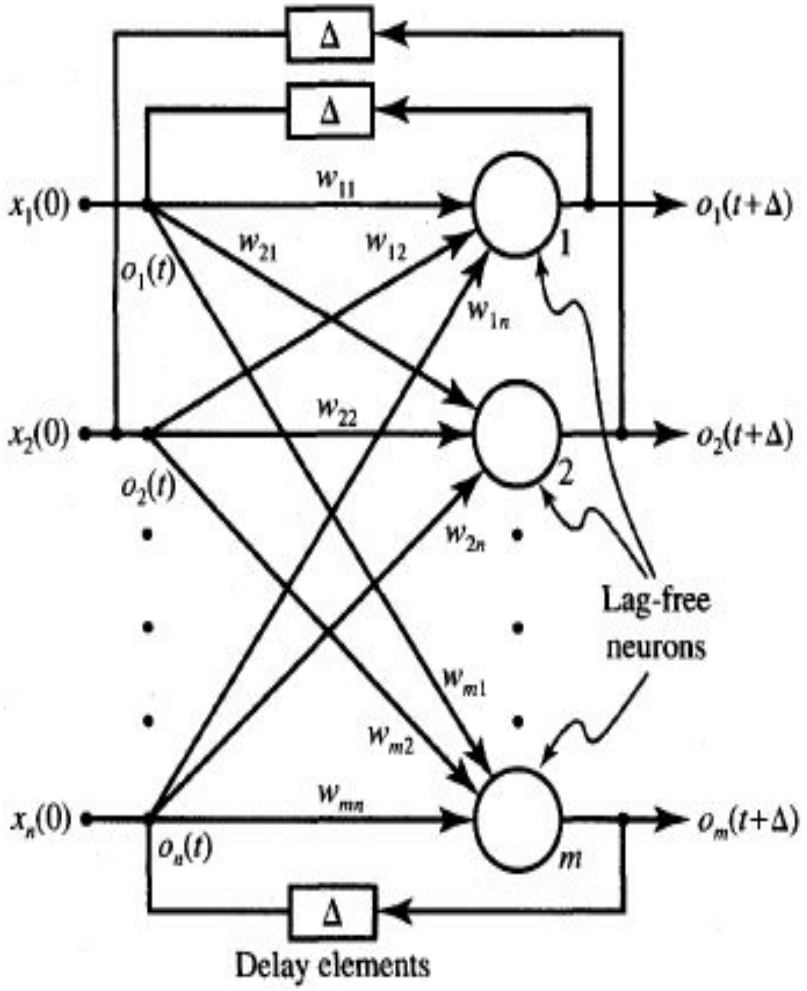
Input layer

Hidden layers

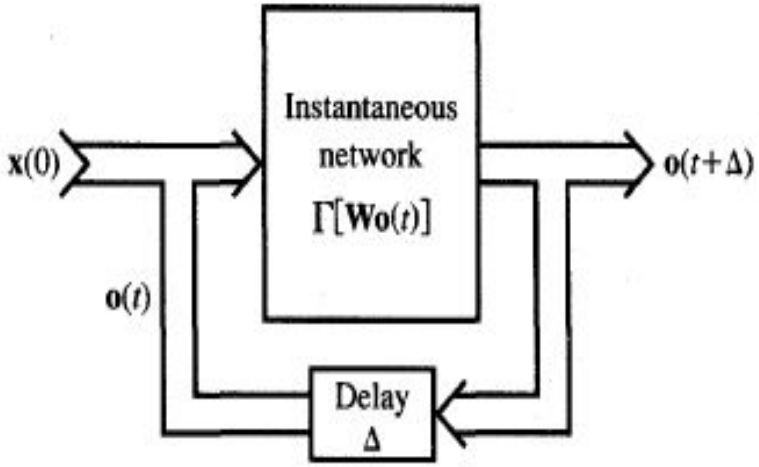
Output layer

Different Network Topologies

- Feedback network

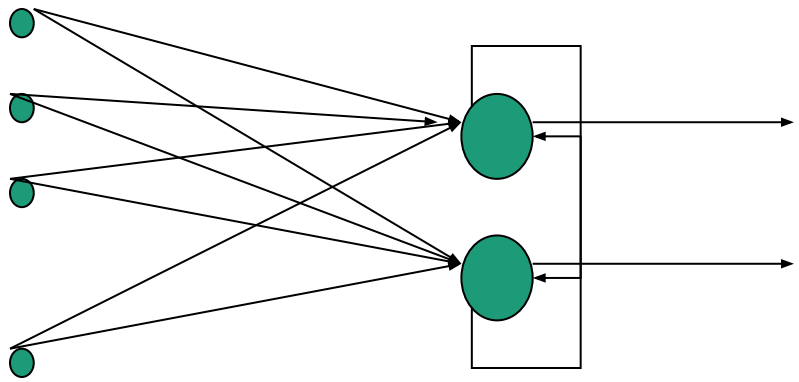


When outputs are directed back as inputs to same or preceding layer nodes it results in the formation of feedback networks



Different Network Topologies

- Recurrent networks
  - Feedback networks with closed loop are called **Recurrent Networks**. The response at the k+1'th instant depends on the entire history of the network starting at k=0.
  - A network with feedback, where some of its inputs are connected to some of its outputs (discrete time).

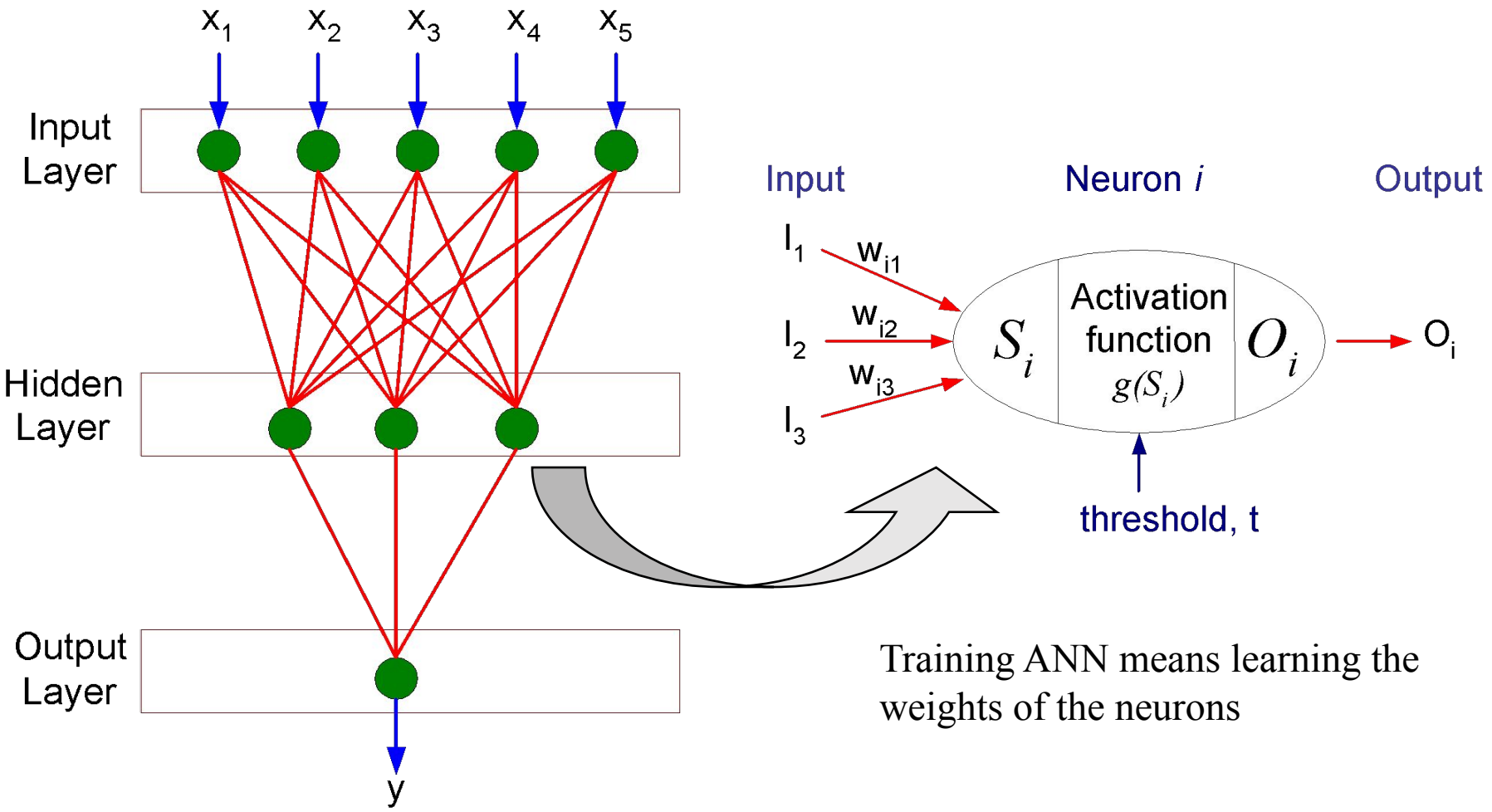


Recurrent network

Input layer

Output layer

General Structure of ANN



Training ANN means learning the weights of the neurons

Case Study

- A bank wants to assess whether to approve a loan application to a customer.
- So, we wants to predict whether a customer is likely to default on the loan or not ?

Framework ANN

1. Assign Random weights to all the linkages to start the algorithm
2. Using the inputs and the (Input ->Hidden node) linkages find the activation rate of Hidden Nodes
3. Using the activation rate of Hidden nodes and linkages to Output, find the activation rate of Output Nodes
4. Find the error rate at the output node and recalibrate all the linkages between Hidden Nodes and Output Nodes
5. Using the Weights and error found at Output node, cascade down the error to Hidden Nodes
6. Recalibrate the weights between hidden node and the input nodes
7. Repeat the process till the convergence criterion is met
8. Using the final linkage weights score the activation rate of the output nodes

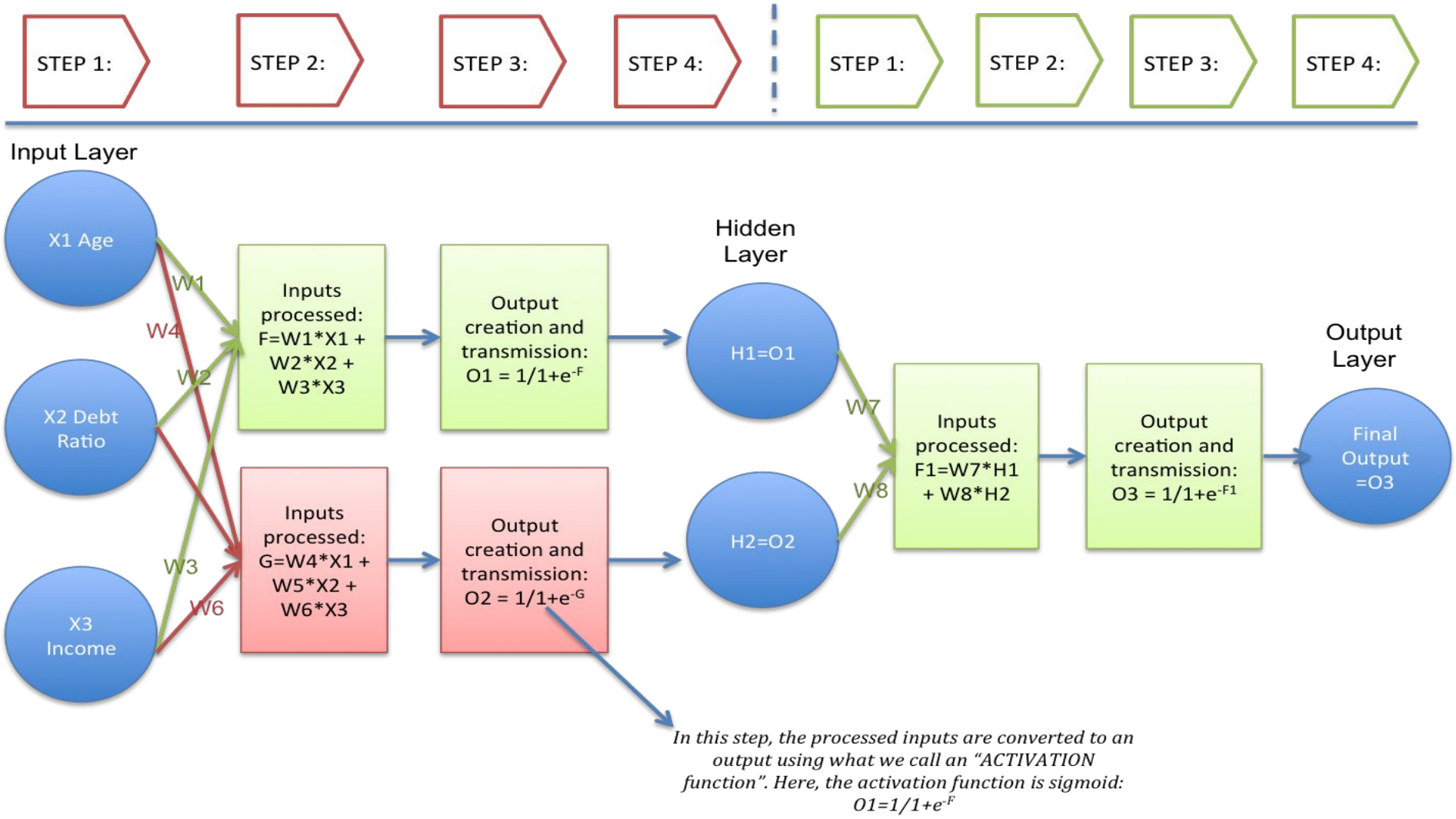
Bank Dataset

Customer ID	Customer Age	Debt Ratio (% of Income)	Monthly Income ($)	Loan Defaulter Yes:1 No:0 (Column W)	Default Prediction (Column X)
1	45	0.80	9120	1	0.76
2	40	0.12	2000	1	0.66
3	38	0.08	3042	0	0.34
4	25	0.03	3300	0	0.55
5	49	0.02	63588	0	0.15
6	74	0.37	3500	0	0.72

So, we have to predict Column W.

A prediction closer to 1 indicates that the customer has more chances to default.

ANN architecture for Bank dataset



Key points of architecture

1. The ANN architecture has an **input layer**, **hidden layer** (there can be more than 1) and the **output layer**, called as **MLP** (**Multi Layer Perceptron**)
2. The hidden layer can be seen as a "**distillation layer**" that distils some of the important patterns from the inputs and passes it onto the next layer. It makes the network faster and efficient by identifying only the important information from the inputs leaving out the redundant information.
3. The activation function serves two notable purposes:
   - It captures non-linear relationship between the inputs.
   - It helps convert the input into a more useful output.

1. In the above example, the activation function used is sigmoid:

$$O1 = 1 / (1 + \exp (- F))$$

Where $F = W1*X1 + W2*X2 + W3*X3$

Sigmoid activation function creates an output with values between 0 and 1.

There can be other activation functions like Tanh, softmax and RELU.

Similarly, the hidden layer leads to the final prediction at the output layer:

$$O3 = 1 / (1 + \exp (- F1))$$

Where $F\ 1= W7*H1 + W8*H2$

Here, the output value (O3) is between 0 and 1. A value closer to 1 (e.g. 0.75) indicates that there is a higher indication of customer defaulting.

The weights W are the importance associated with the inputs. If W1 is 0.56 and W2 is 0.92, then there is higher importance attached to X2: Debt Ratio than X1: Age, in predicting H1.

This is “feed-forward network”, input signals are flowing in only one direction.

A good model with high accuracy gives predictions that are very close to the actual values. So, in the table above, Column X values should be very close to Column W values. The error in prediction is the difference between column W and column X:

Customer ID	Customer Age	Debt Ratio (% of Income)	Monthly Income ($)	Loan Defaulter Yes:1 No:0 (Column W)	Default Prediction (Column X)	Prediction Error
1	45	0.80	9120	1	0.76	0.24
2	40	0.12	2000	1	0.66	0.34
3	38	0.08	3042	0	0.34	-0.34
4	25	0.03	3300	0	0.55	-0.55
5	49	0.02	63588	0	0.15	-0.15
6	74	0.37	3500	0	0.72	-0.72

8. The key to get a good model with accurate predictions is to find "**optimal values of W — weights**" that minimizes the prediction error. This is achieved by "**Back propagation algorithm**" and this makes ANN a learning algorithm because by learning from the errors, the model is improved.
9. The most common method of optimization algorithm is called "**gradient descent**", where, iteratively different values of W are used and prediction errors assessed. So, to get the optimal W, the values of W are changed in small amounts and the impact on prediction errors assessed. Finally, those values of W are chosen as optimal, where with further changes in W, errors are not reducing further.

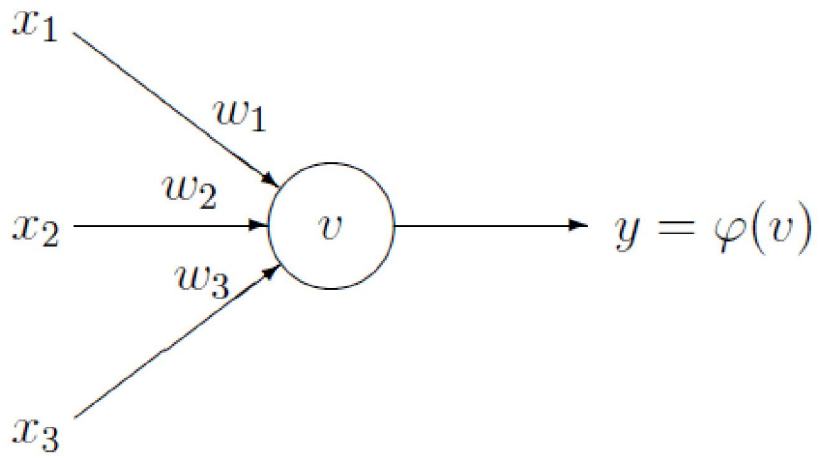
Advantages of neural Networks

1. ANNs have the ability to learn, model non-linear and complex relationships, which is really important because in real-life, many of the relationships between inputs and outputs are non-linear as well as complex.
2. ANNs can generalize — After learning from the initial inputs and their relationships, it can infer unseen relationships on unseen data as well, thus making the model generalize and predict on unseen data.

3. Unlike, many other prediction techniques, ANN does not impose any restrictions on the input variables (like how they should be distributed). Additionally, many studies have shown that ANNs can better model.
4. Data with high volatility and non-constant variance, given its ability to learn hidden relationships in the data without imposing any fixed relationships in the data. This is something very useful in financial time series forecasting (e.g. stock prices) where data volatility is very high.

Exercise

Q1: Below is a diagram if a single artificial neuron (unit):



- The node has three inputs x = (x1, x2, x3) that receive only binary signals (either 0 or 1). How many different input patterns this node can receive?
- What if the node had four inputs? Five? Can you give a formula that computes the number of binary input patterns for a given number of inputs?

Ans:

- For three inputs the number of combinations of 0 and 1 is 8:

x_1	0	1	0	1	0	1	0	1
x_2	0	0	1	1	0	0	1	1
x_3	0	0	0	0	1	1	1	1

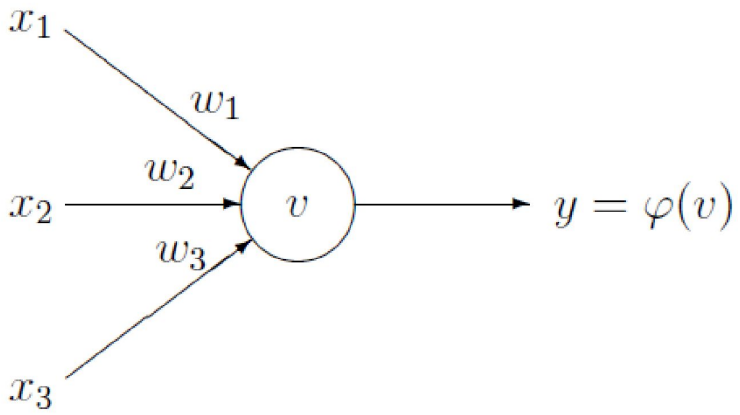
- For four inputs the number of combinations is 16:

x_1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1
x_2	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
x_3	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
x_4	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1

- You may check that for five inputs the number of combinations will be 32.
- Note that $8 = 2^3$, $16 = 2^4$ and $32 = 2^5$ (for three, four and five inputs).
- Thus, the formula for the number of binary input patterns is:

2^n , where n in the number of inputs

- **Q2: Consider the unit shown. Suppose that the weights corresponding to the three inputs have the following values:**



$$\begin{array}{lcr} w_1 & = & 2 \\ w_2 & = & -4 \\ w_3 & = & 1 \end{array}$$

and the activation of $\varphi(v) = \begin{cases} 1 & \text{if } v \geq 0 \\ 0 & \text{otherwise} \end{cases}$ the step-function:

Pattern	P_1	P_2	P_3	P_4
x_1	1	0	1	1
x_2	0	1	0	1
x_3	0	1	1	1

- Calculate what will be the output value y of the unit for each of the following input patterns:

Ans: To find the output value y for each pattern we have to:

Calculate the weighted sum:

$$v = \sum_i w_i \, x_i = w_1 \cdot x_1 + w_2 \cdot x_2 + w_3 \cdot x_3$$

Apply the activation function to v

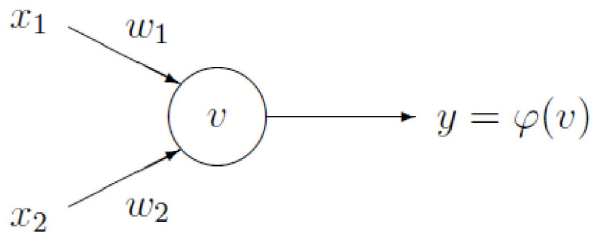
The calculations for each input pattern are:

$$\begin{aligned}
P_1: &\quad v = 2 \cdot 1 - 4 \cdot 0 + 1 \cdot 0 = 2\,, &&(2 > 0)\,, &&y = \varphi(2) = 1\\
P_2: &\quad v = 2 \cdot 0 - 4 \cdot 1 + 1 \cdot 1 = -3\,, &&(-3 < 0)\,, &&y = \varphi(-3) = 0\\
P_3: &\quad v = 2 \cdot 1 - 4 \cdot 0 + 1 \cdot 1 = 3\,, &&(3 > 0)\,, &&y = \varphi(3) = 1\\
P_4: &\quad v = 2 \cdot 1 - 4 \cdot 1 + 1 \cdot 1 = -1\,, &&(-1 < 0)\,, &&y = \varphi(-1) = 0
\end{aligned}$$

- **Q3: Logical operators (i.e. AND, OR) are the building blocks of any computational device. Logical functions return only two possible values, true or false, based on the truth or false values of their arguments.**

For example, operator AND returns true only when all its arguments are true, otherwise (if any of the arguments is false) it returns false. If we denote truth by 1 and false by 0, then logical function AND can be represented by the following table:

This function can be implemented by a single-unit with two inputs:



x_1 :	0	1	0	1
x_2 :	0	0	1	1
x_1 AND x_2 :	0	0	0	1

If the weights are w1 = 1 and w2 = 1 and the activation function is:

$$\varphi(v) = \begin{cases} 1 & \text{if } v \geq 2 \\ 0 & \text{otherwise} \end{cases}$$

Note that the threshold level is 2 (v >= 2).

a) Test how the neural AND function works.

b) Suggest how to change either the weights or the threshold level of this single-unit in order to implement the logical OR function

Ans:

a)

$$
\begin{aligned}
P_1: &\quad v = 1 \cdot 0 + 1 \cdot 0 = 0\ , \quad (0 < 2)\ , \quad y = \varphi(0) = 0 \\
P_2: &\quad v = 1 \cdot 1 + 1 \cdot 0 = 1\ , \quad (1 < 2)\ , \quad y = \varphi(1) = 0 \\
P_3: &\quad v = 1 \cdot 0 + 1 \cdot 1 = 1\ , \quad (1 < 2)\ , \quad y = \varphi(1) = 0 \\
P_4: &\quad v = 1 \cdot 1 + 1 \cdot 1 = 2\ , \quad (2 = 2)\ , \quad y = \varphi(2) = 1
\end{aligned}
$$

b) One solution is to increase the weights of the unit: w1 = 2 and w2 = 2. Alternatively, we could reduce the threshold to 1:

$$
\begin{aligned}
P_1: &\quad v = 2 \cdot 0 + 2 \cdot 0 = 0\ , \quad (0 < 2)\ , \quad y = \varphi(0) = 0 \\
P_2: &\quad v = 2 \cdot 1 + 2 \cdot 0 = 2\ , \quad (2 = 2)\ , \quad y = \varphi(2) = 1 \\
P_3: &\quad v = 2 \cdot 0 + 2 \cdot 1 = 2\ , \quad (2 = 2)\ , \quad y = \varphi(2) = 1 \\
P_4: &\quad v = 2 \cdot 1 + 2 \cdot 1 = 4\ , \quad (4 > 2)\ , \quad y = \varphi(4) = 1
\end{aligned}
$$

$$
\varphi(v) = \begin{cases} 1 & \text{if } v \geq 1 \\ 0 & \text{otherwise} \end{cases}
$$

Decision Surface Of A Perceptron

- A perceptron represents a hyperplane decision surface in the *n*- dimensional space of instances (i.e., points).

- The perceptron outputs a 1 for instances lying on one side of the hyperplane and outputs a -1 for instances lying on the other side

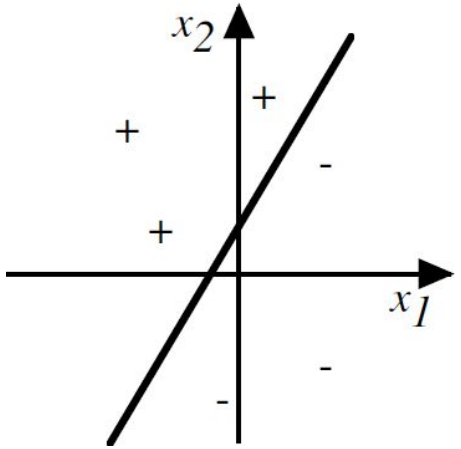


Fig: The decision surface represented by a two-input perceptron.

- The instances in the figure are ***linearly separable.***

Decision Surface Of A Perceptron

- Some sets of positive and negative examples cannot be separated by any hyperplane. Such examples are known as ***linearly non- separable.***

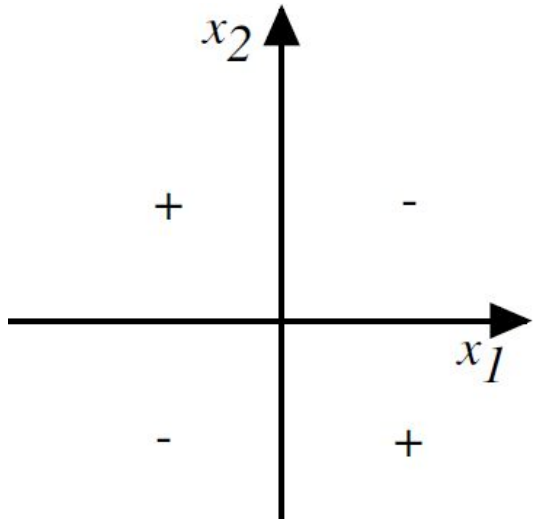
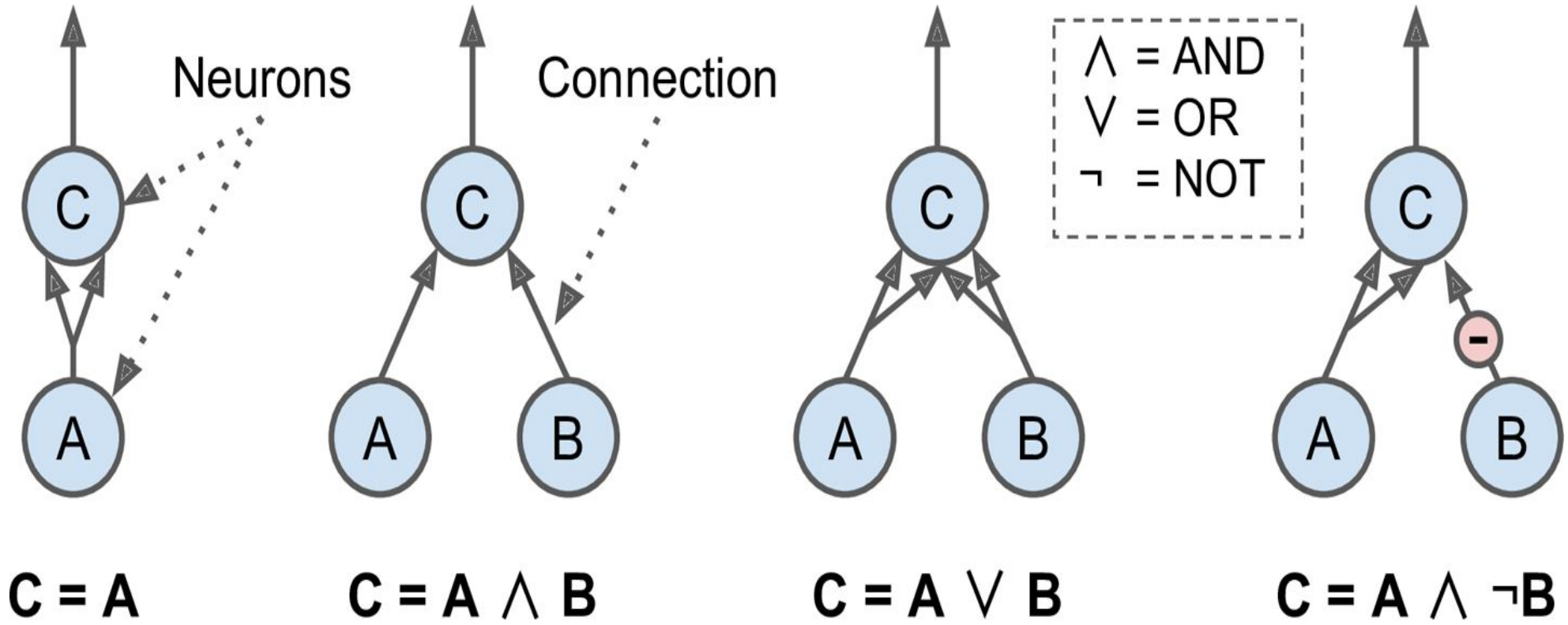


Fig: A set of training examples that is not linearly separable (i.e., that cannot be correctly classified by any straight line).

Logical Computations with Neurons



Representational Power Of Perceptron

- A single perceptron can be used to represent many Boolean functions such as AND, OR, NAND and NOR.

- In the following we see some of the examples

AND Boolean Function

- Assuming +1 denotes the positive class and -1 denotes the negative class in the perceptron output.

- The input and the output for AND function implementation are shown in the following table

x_1 , x_2 (Input)	AND	Perceptron output for AND Operation
0 , 0	0	-1
0 , 1	0	-1
1 , 0	0	-1
1 , 1	1	1

- *Linear function in this case for perceptron is*

$$w_0 + w_1 * x_1 + w_2 * x_2$$

- ***What could be the possible weights?***

AND Boolean Function

- ***What could be the possible weights?***
- ***The Ans:*** *One possible weight vector is as follows*

$$w_0 = -0.8,\ w_1 = 0.5,\ w_2 = 0.5$$

x_1 , x_2 (Input)	$w_0 + w_1 * x_1 + w_2 * x_2$	Perceptron output
0 , 0	-0.8	-1
0 , 1	-0.3	-1
1 , 0	-0.3	-1
1 , 1	0.2	1

OR Boolean Function

- On the same lines we can implement OR function

- The input and the output for OR function implementation are shown in the following table

x_1 , x_2 (Input)	OR	Perceptron output for OR operation
0 , 0	0	-1
0 , 1	1	1
1 , 0	1	1
1 , 1	1	1

- *Linear function in this case for perceptron is*

$$w_0 + w_1 * x_1 + w_2 * x_2$$

- ***What could be the possible weights?***

OR Boolean Function

- ***What could be the possible weights?***
- ***The Ans:*** *One possible weight vector is as follows*

$$w_0 = -0.3,\ w_1 = 0.5,\ w_2 = 0.5$$

x_1, x_2 (Input)	$w_0 + w_1 * x_1 + w_2 * x_2$	Perceptron output
0 , 0	-0.3	-1
0 , 1	0.2	1
1 , 0	0.2	1
1 , 1	0.7	1

Representational Power Of Perceptron

- Although perceptrons can represent all of the primitive Boolean functions such as AND, OR, NAND and NOR, some Boolean functions cannot be represented by a single perceptron

- E.g: XOR
  - whose value is 1 if and only if x_1 and x_2 are not equal.
- The set of linearly nonseparable training examples shown in Figure 2 corresponds to this XOR function

Multi-layer Neural Network

Limitations of Perceptrons

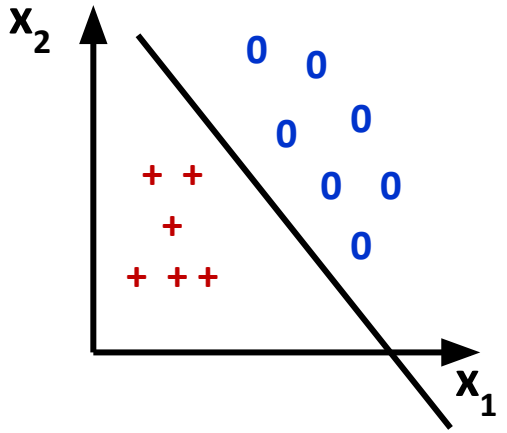
- Perceptrons have a *monotonicity* property:

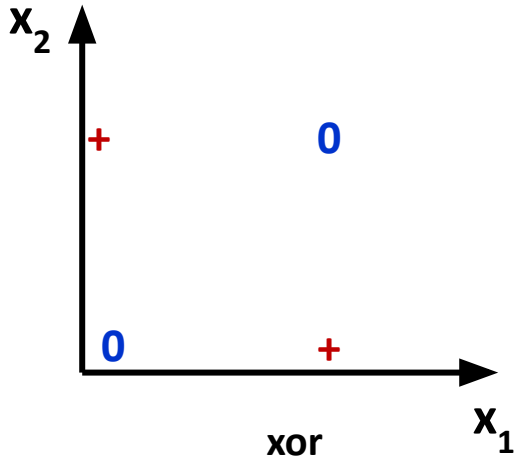
  If a link has positive weight, activation can only increase as the corresponding input value increases (*irrespective* of other input values)

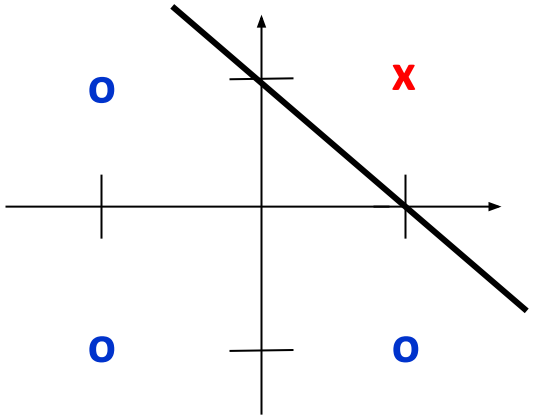
- Can't represent functions where input *interactions* can cancel one another's effect (e.g. XOR)
- Can represent only linearly separable functions

Single layer Perceptron

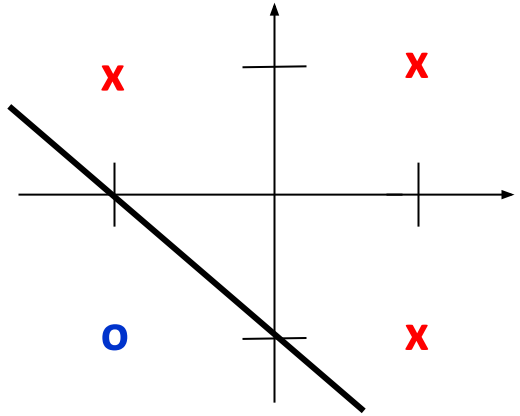
- Single layer perceptrons learn linear decision boundaries





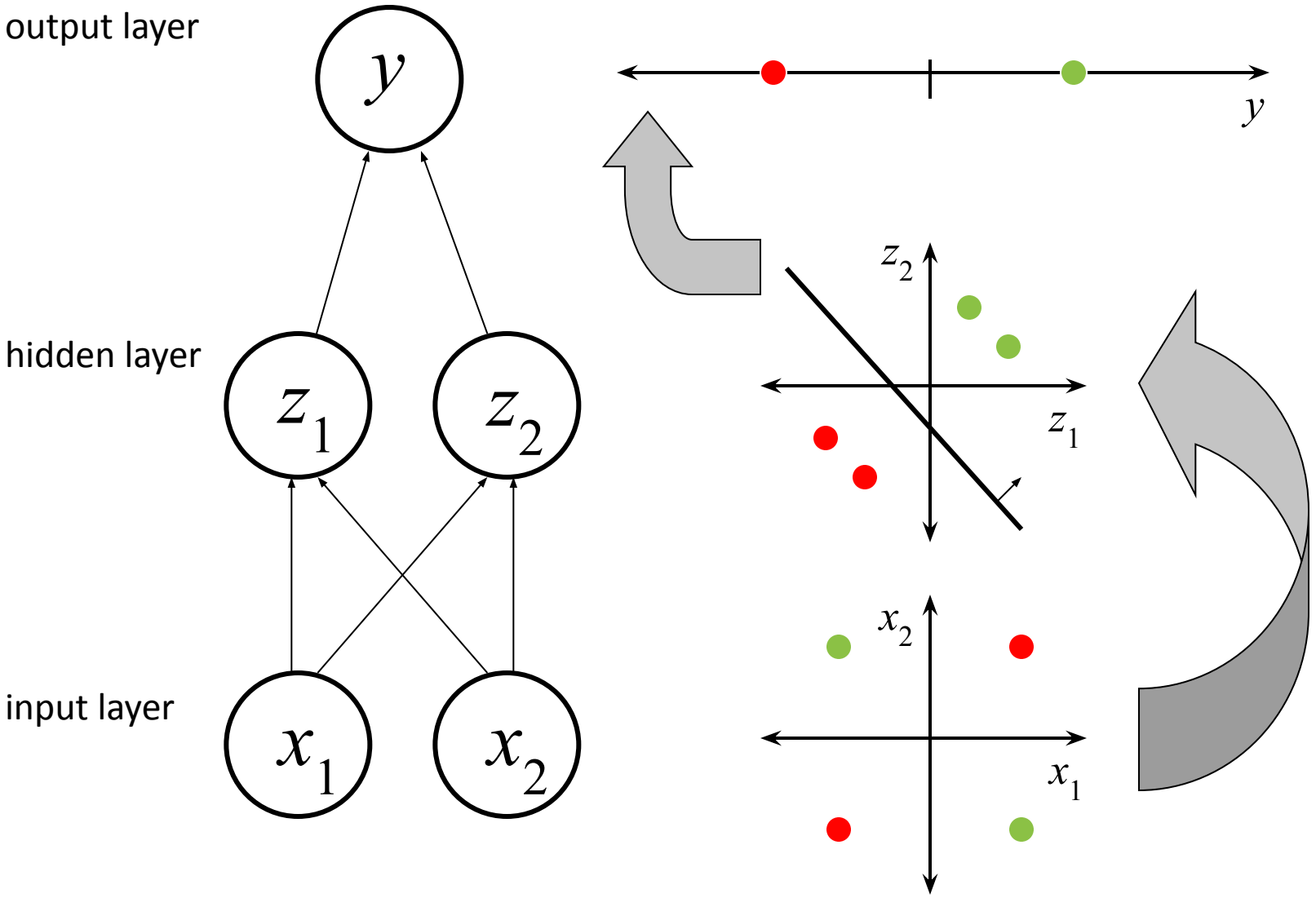


x: class I (y = 1)
o: class II (y = -1)



x: class I (y = 1)
o: class II (y = -1)

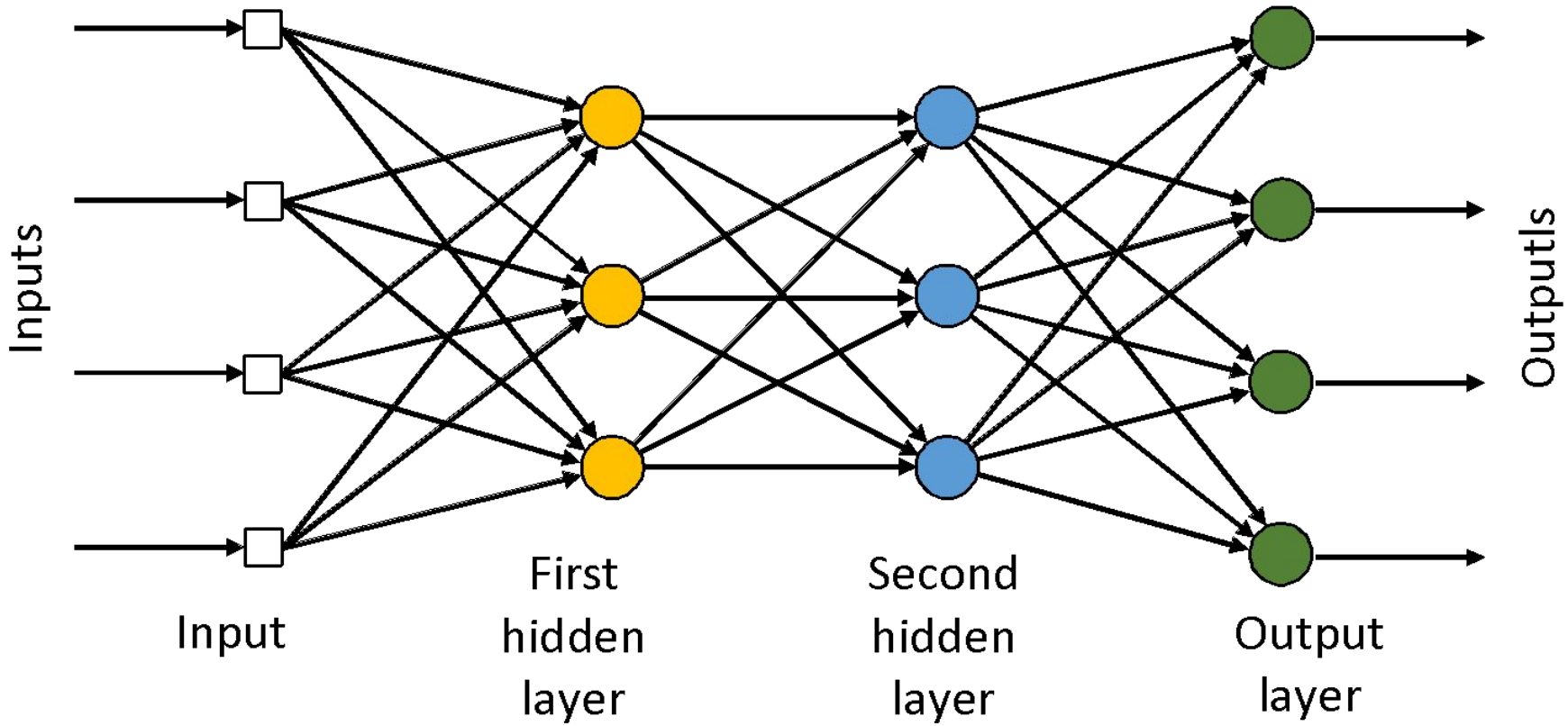
A solution: multiple layers



Power/Expressiveness of Multilayer Networks

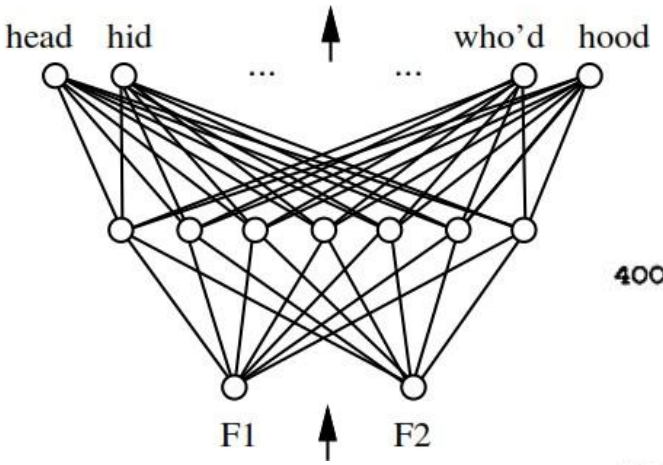
- Can represent interactions among inputs
- Two layer networks can represent any Boolean function, and continuous functions (within a tolerance) as long as the number of hidden units is sufficient and appropriate activation functions used
- Learning algorithms exist, but weaker guarantees than perceptron learning algorithms

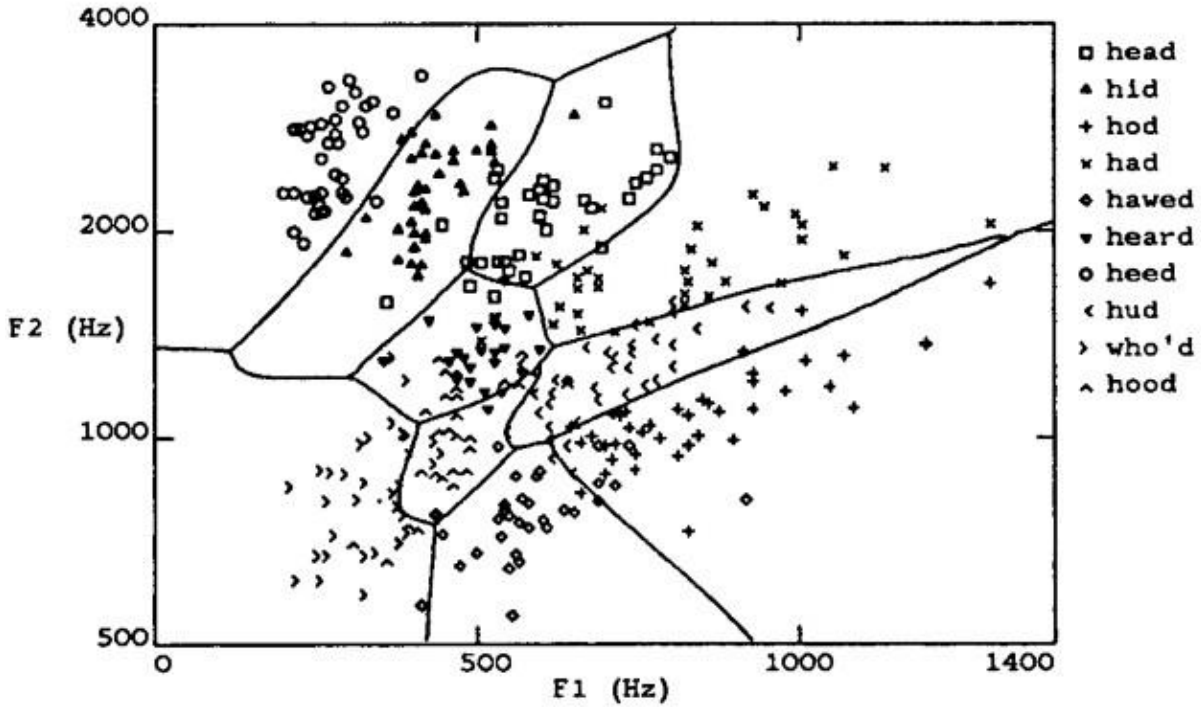
Multilayer Network



Multilayer Network

- ***Example:***
  - Multilayer network and its non-linear decision surface.





Sigmoid unit

- ***What type of unit shall we use as the basis for constructing multilayer networks?***
  - *Can we use Linear unit?*
  - *Can we use perceptron?*

Sigmoid unit

- ***What type of unit shall we use as the basis for constructing multilayer networks?***
  - *Can we use Linear unit?*
  - *Can we use perceptron?*
- *The answer to both the questions is no.*

- The reason is we need a unit whose output is a nonlinear function of its inputs, but whose output is also a differentiable function of its inputs.

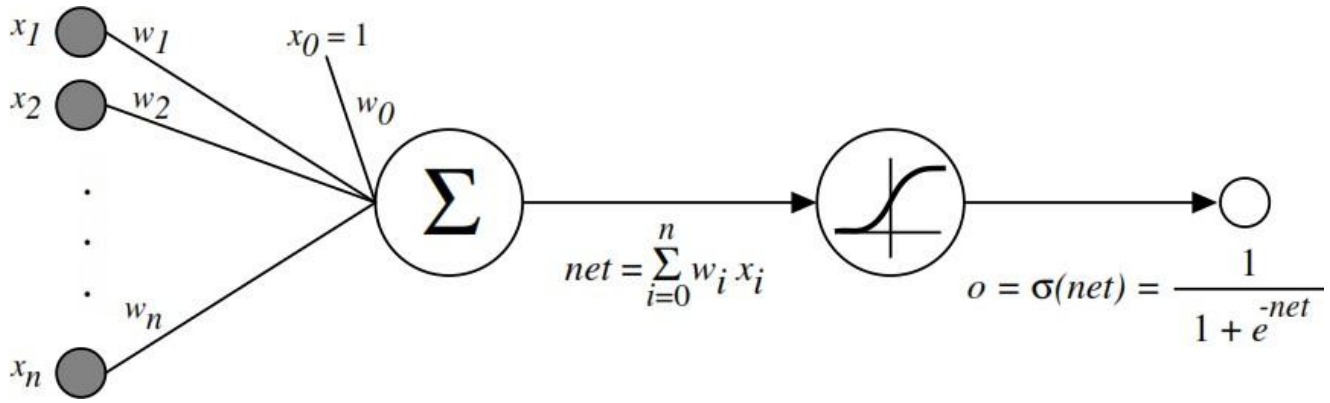
Sigmoid unit

- ***What type of unit shall we use as the basis for constructing multilayer networks?***
  - *Can we use Linear unit?*
  - *Can we use perceptron?*
- *The answer to both the questions is no.*

- The reason is we need a unit whose output is a nonlinear function of its inputs, but whose output is also a differentiable function of its inputs.

- One solution is the ***sigmoid unit***- a unit very much like a perceptron, but based on a smoothed, differentiable threshold function.

Sigmoid unit



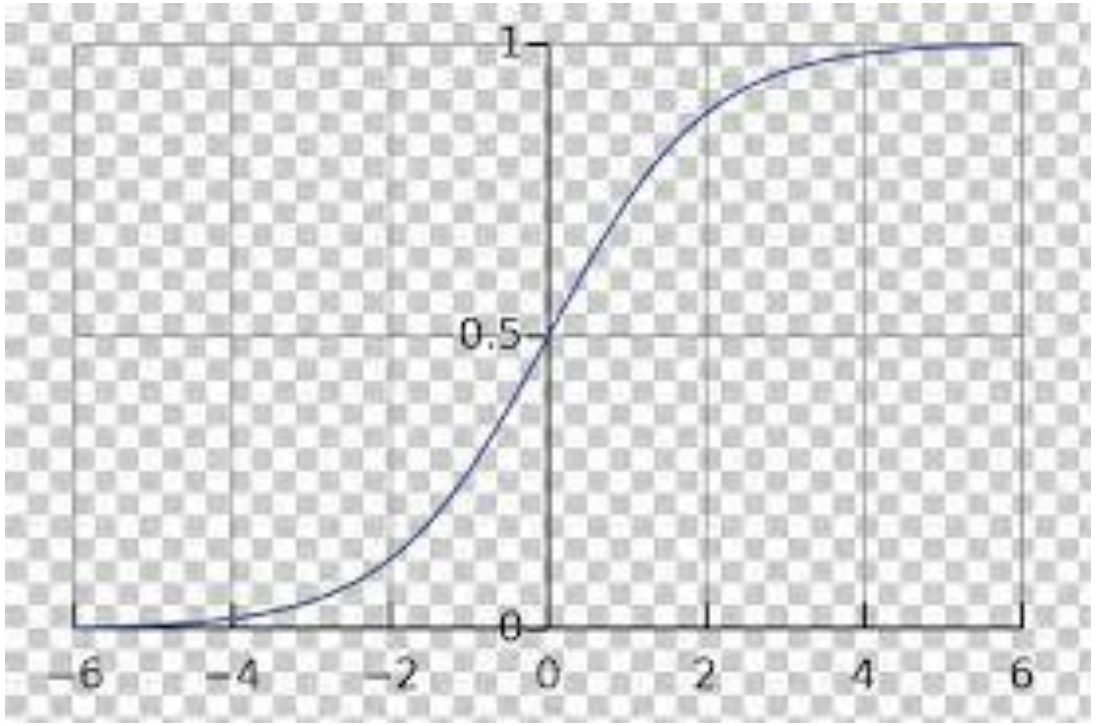
- $\sigma(z)$ is a ***sigmoid function***
  - $\sigma(z) = \frac{1}{1+e^{-z}}$

Sigmoid function

- Sigmoid function

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

- as $z \to \infty$, $\sigma(z) = 1$
- as $z \to -\infty$, $\sigma(z) = 0$
- at $z = 0$, $\sigma(z) = 0.5$



Sigmoid unit

- Nice properties of sigmoid function

$$\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$$

- We can derive gradient descent rules to train
  - *One sigmoid unit*
  - *Multilayer networks of sigmoid units Backpropagation*

Multilayer Networks

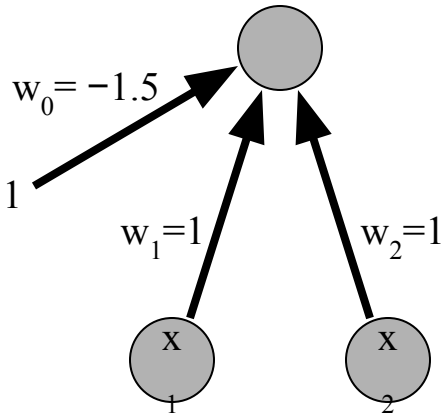
- **Example: X-OR**
  - As the classes are not linearly separable, a single perceptron would not be sufficient to represent this operation
  - ***What's the solution?***
  - ***Ans:*** Add extra layer

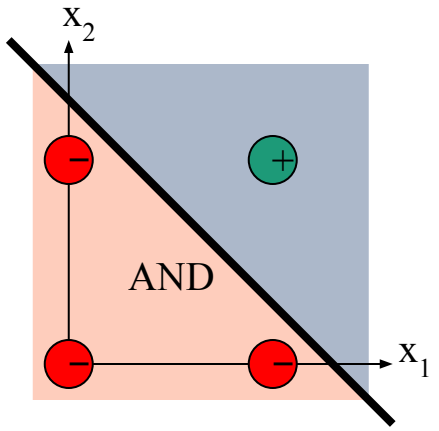
Multilayer Networks

- Adding extra layers
  - XOR(x_1,x_2) = OR(OR(x_1,x_2), AND(x_1,x_2))

Boolean AND

input x1	input x2	ouput
0	0	0
0	1	0
1	0	0
1	1	1



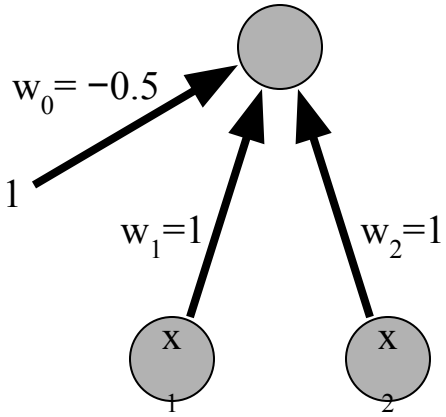


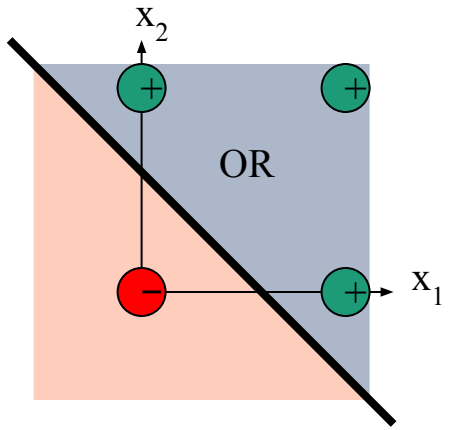
Imagine two inputs (x1, x2) and one output (y). You could train a perceptron-like node with weights (w1, w2) and a bias (b) such that:

- If w1x1 + w2x2 + b > threshold, then y = 1 (AND is true).
- If w1x1 + w2x2 + b <= threshold, then y = 0 (AND is false).

Boolean OR

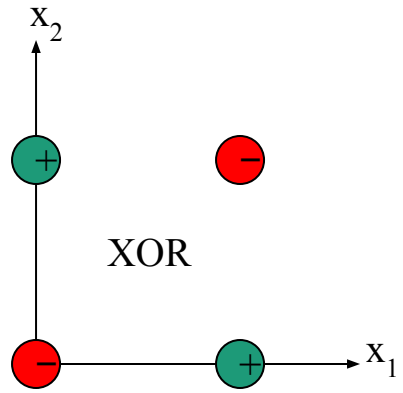
input x1	input x2	ouput
0	0	0
0	1	1
1	0	1
1	1	1





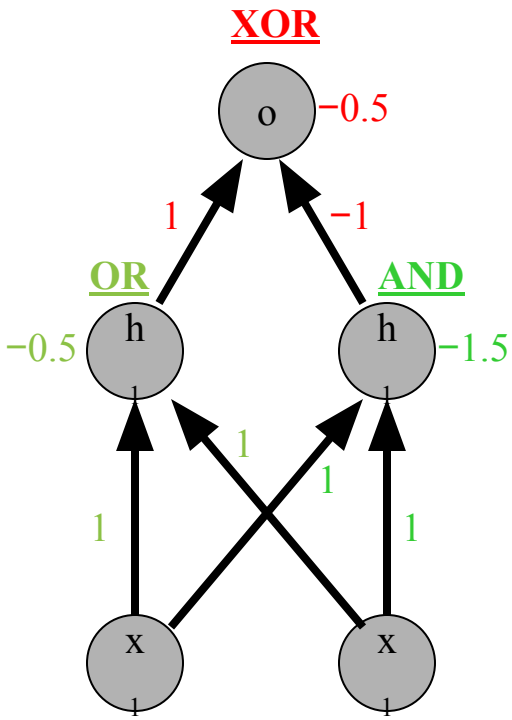
Boolean XOR

input x1	input x2	ouput
0	0	0
0	1	1
1	0	1
1	1	0



Boolean XOR

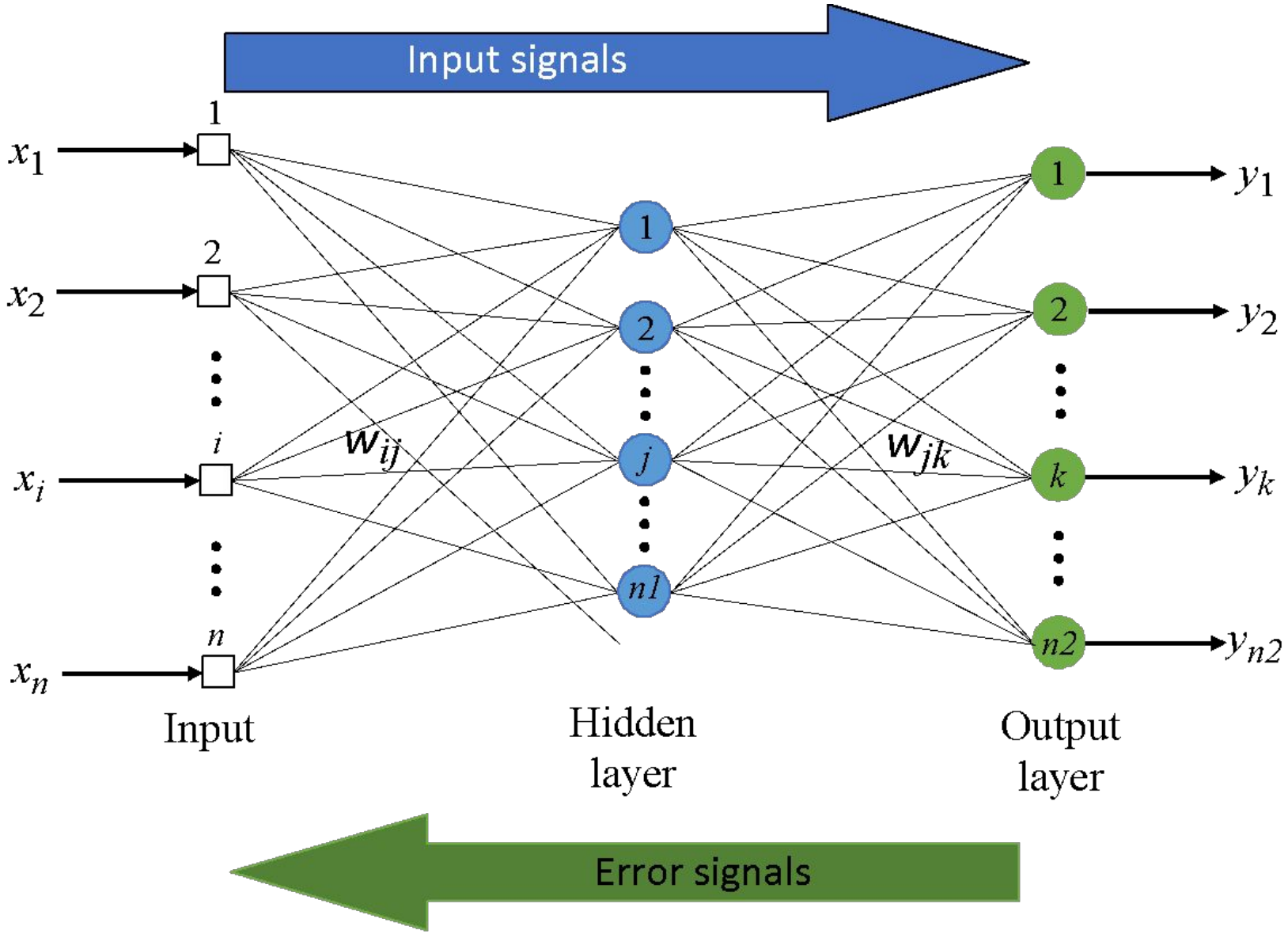
input x1	input x2	ouput
0	0	0
0	1	1
1	0	1
1	1	0



Representation Capability of NNs

- Single layer nets have limited representation power (linear separability problem). Multi-layer nets (or nets with non-linear hidden units) may overcome linear inseparability problem.
- Every Boolean function can be represented by a network with a single hidden layer.
- Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers.

Two-layer back-propagation neural network



Derivation

- For one output neuron, the error function is

$$E = \frac{1}{2}(y - o)^2$$

- For each unit j, the output o_j is defined as

$$o_j = \varphi(net_j) = \varphi\left(\sum_{k=1}^{n} w_{kj} o_k\right)$$

The input net_j to a neuron is the weighted sum of outputs o_k of previous n neurons.

- Finding the derivative of the error:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ij}}$$

- To update the weight w_{ij} using gradient descent, one must choose a learning rate η.

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}}$$

The Backpropagation Algorithm

Introduction

- The Backpropagation algorithm learns the weights for a multilayer network
- It employs gradient descent to attempt to minimize the squared error
- As we are considering networks with multiple output units rather than single units, we redefine E to sum the errors over all of the network output units.

$$E(\vec{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2$$

  - where *outputs* is the set of output units in the network
  - t_{kd} and o_{kd} are the target and output values associated with the kth output unit and training example d.

Introduction

- The learning problem faced by BACKPROPAGATION is to search a large hypothesis space defined by all possible weight values for all the units in the network.

- The situation can be visualized in terms of an error surface similar to that for linear units.

The Backpropagation Algorithm

- ***Notations***

  - x_{ij} denotes the input from node i to unit j, and w_{ij} denotes the corresponding weight.

  - δ_n, denotes the error term associated with unit n. It plays a role analogous to the quantity (t - o) as in delta training rule but it is slightly more complex here as we shall see.

The Backpropagation Algorithm

- Backpropagation(training_examples, η, n_i, n_o, n_{hidden})
  - Each training example is a pair of the form $\langle \vec{x}, \vec{t} \rangle$ where $\vec{x}$ is the vector of network input values, and $\vec{t}$ is the vector of target network output values.
  - η is the learning rate (e.g., .05).
  - n_i, is the number of network inputs, n_{hidden} is the number of units in the hidden layer, and n_o, the number of output units.
  - The input from unit *i* into unit *j* is denoted by x_{ij}, and the weight from unit *i* to unit *j* is denoted w_{ij}.
  - Create a feed-forward network with n_i inputs, n_{hidden} hidden units, and n_o output units.
  - Initialize all network weights to small random numbers (e.g., between -.05 and .05).

Backpropagation Algorithm

- Initialize all weights to small random numbers.
Until satisfied, do
  - For each training example, do
    - Input the training example to the network and compute the network outputs
    - For each output unit k

$$\delta_k \leftarrow o_k(1 - o_k)(y_k - o_k)$$

    - For each hidden unit h

$$\delta_h \leftarrow o_h(1 - o_h) \sum_{k \in outputs} w_{h,k}, \delta_k,$$

    - Update each network weight w_i, j

$$w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j}$$

x_d = input

y_d = target output

o_d = observed unit output

w_{ij} = wt from i to j

Termination condition

- The weight-update loop in BACKPROPAGATION may be iterated thousands of times in a typical application.

- ***Termination Criteria***
- A variety of termination conditions can be used to halt the procedure.
  - One may choose to halt after a fixed number of iterations, or
  - Once the error on the training examples falls below some threshold, or
  - Once the error on a separate validation set of examples meets some criterion.

- The choice of termination criterion is an important one, because too few iterations can fail to reduce error sufficiently, and too many can lead to overfitting the training data.

Remarks

- Gradient descent over entire network weight vector
- Can be generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum
- May include weight momentum α

$$\Delta w_{i,j}(n) = \eta \delta_j x_{i,j} + \alpha \Delta w_{i,j}(n-1)$$

- Training may be slow.
- Using network after training is very fast

Training practices: batch vs. stochastic vs. mini-batch gradient descent

- **Batch gradient descent**:
  1. Calculate outputs for the entire dataset
  2. Accumulate the errors, back-propagate and update

Too slow to converge
Gets stuck in local minima

- **Stochastic/online gradient descent**:
  1. Feed forward a training example
  2. Back-propagate the error and update the parameters

Converges to the solution faster
Often helps get the system out of local minima

- **Mini-batch gradient descent**:

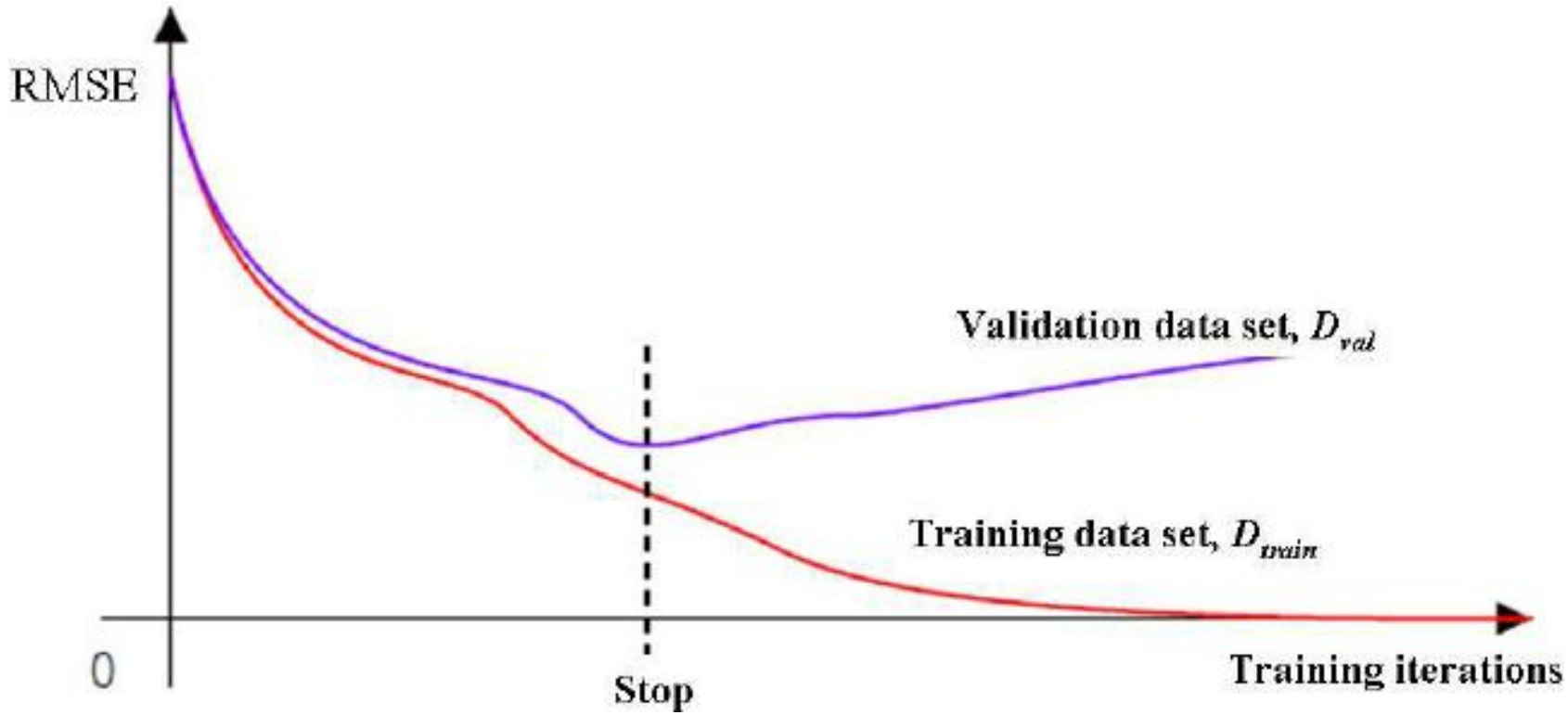
Learning in epochs Stopping

- Train the Neural Network on the entire training set over and over again
- Each such episode of training is called an “epoch”

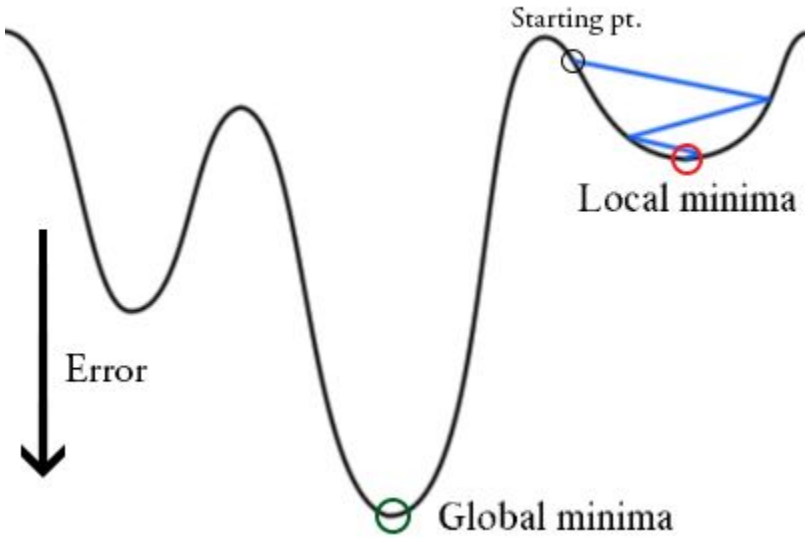
Stopping

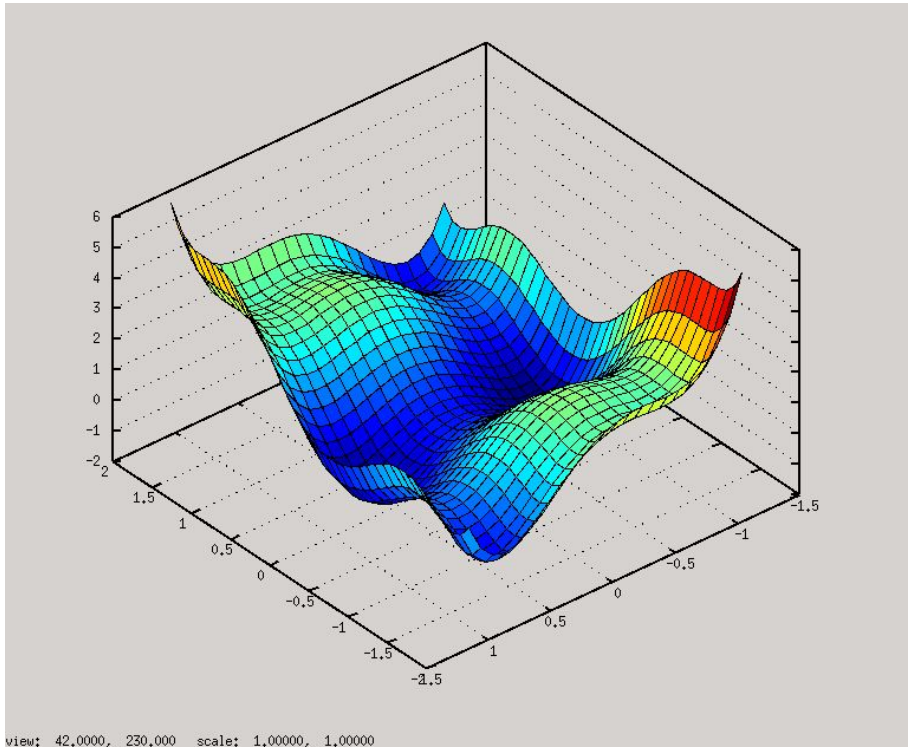
1. Fixed maximum number of epochs: most naïve
2. Keep track of the training and validation error curves.

Overfitting in ANNs



Local Minima





- NN can get stuck in local minima for small networks.
- For most large networks (many weights) local minima rarely occurs.
- It is unlikely that you are in a minima in every dimension simultaneously.