Building Innovative Systems

Topic: Neural Network
(Part-I)

ARTIFICIAL NEURAL NETWORKS

- They are extremely powerful computational devices (Turing equivalent, universal computers)
- Massive parallelism makes them very efficient
- They can learn and generalize from training data – so there is no need for enormous feats of programming
- They are particularly fault tolerant – this is equivalent to the "graceful degradation" found in biological systems
- They are very noise tolerant – so they can cope with situations where normal symbolic systems would have difficulty
- In principle, they can do anything a symbolic/logic system can do, and more.

Several key features of the processing elements of ANN are suggested by the properties of biological neurons

1. The processing element receives many signals.
2. Signals may be modified by a weight at the receiving end.
3. The processing element sums the weighted i/ps.
4. Under appropriate circumstances (sufficient i/p), the neuron transmits a single o/p.
5. The output from a particular neuron may go to many other neurons (the axon branches).

6. Information processing is local.

7. Memory is distributed:

   a) Long-term memory resides in the neurons' synapses or weights.

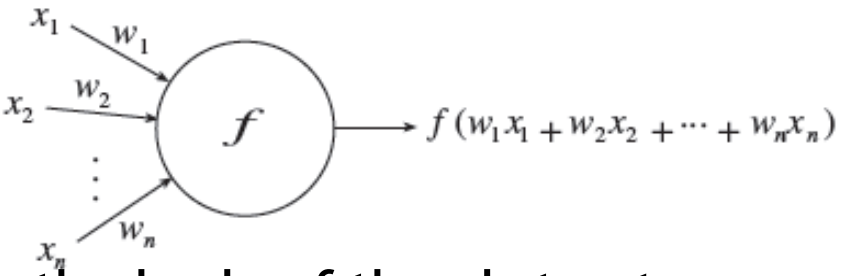
   b) Short-term memory corresponds to the signals sent by the neurons.

8. A synapse's strength may be modified by experience.

9. Neurotransmitters for synapses may be excitatory or inhibitory.

NEURONS

A NN consists of a large number o simple processing elements called neurons.



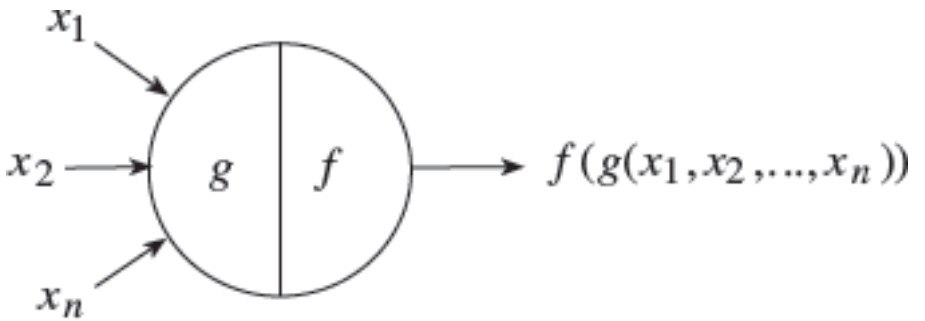
- The primitive function f computed in the body of the abstract neuron can be selected arbitrarily.
- Usually the input channels have an associated weight, which means that the incoming information x_i is multiplied by the corresponding weight w_i.
- The transmitted information is integrated at the neuron (usually just by adding the different signals) and the primitive function is then evaluated.

Activation Functions At The Computing Units

Normally very simple activation functions of one argument are used at the nodes. This means that the **incoming n arguments have to be reduced to a single numerical value.**

Therefore computing units are split into two functional parts:

- **an integration function g** that reduces the n arguments to a single value and
- the output or **activation function f** that produces the output of this node taking that single value as its argument.
- Usually the integration function g is the addition function.



Generic computing unit

CHARACTERIZED BY ITS:

1. Architecture

   Pattern of connections between the neurons

2. Training/Learning algorithm

   Methods of determining the weights on the connections

3. Activation function

TYPICAL ARCHITECTURES

The arrangement of

- **neurons into layers** &
- **the connection patterns within and between layers**

is called the **network architecture**.

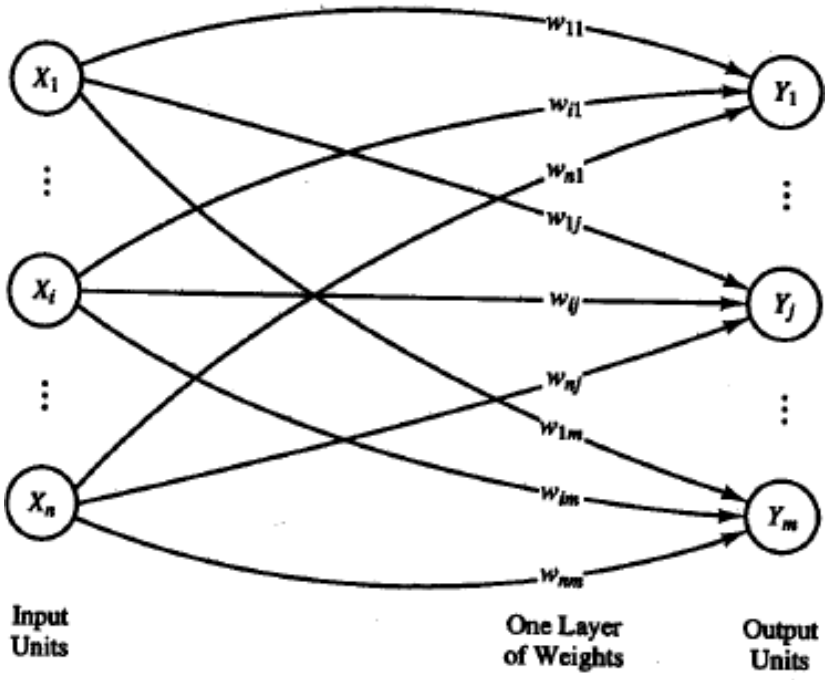
SINGLE LAYER /MULTI LAYER

- Neural nets are often classified as single layer or multilayer.
- The i/p units are not counted as a layer because they do not perform any computation.
- The no. of layers in the N N is the no. of layers of weighted inter-conneted links between slabs of neurons.
- Typically, neurons in the same layer behave in the same manner.
- To be more specific, in many neural networks, the neurons within a layer are either fully interconnected or not interconnected at all.

SINGLE LAYER NET

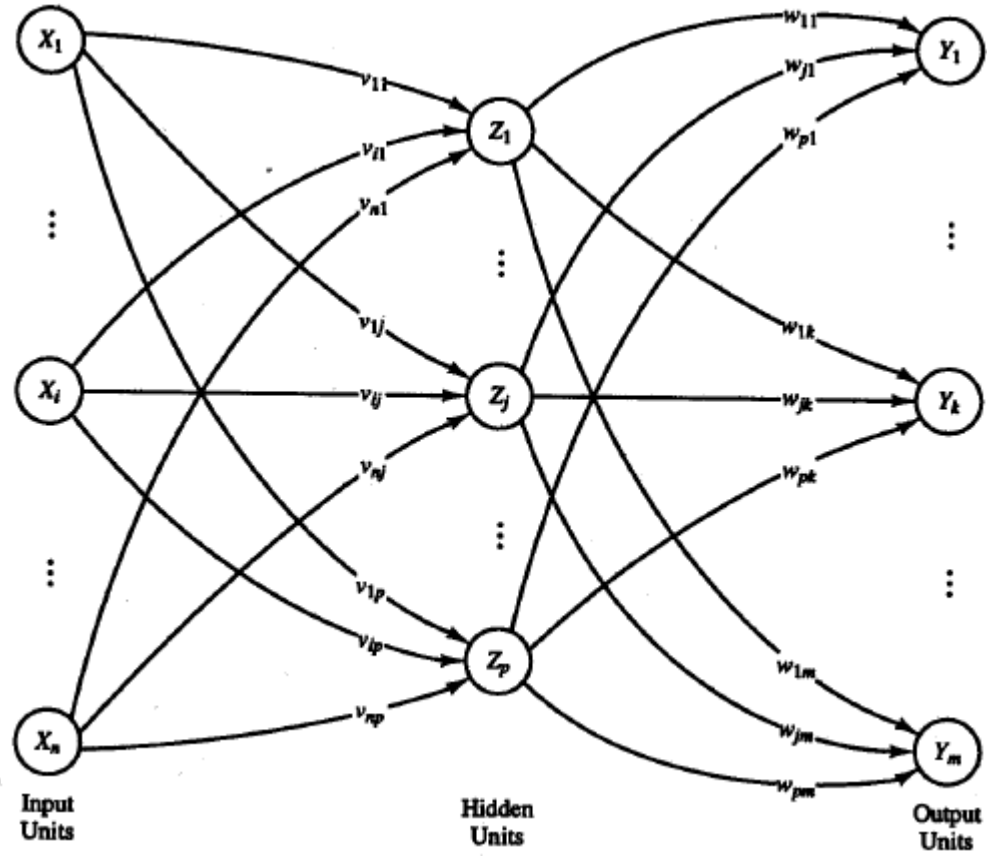
A single-layer net has one layer of connection weights.

The units can be distinguished as input units, which receive signals from the outside world, and output units, from which the response of the net can be read.



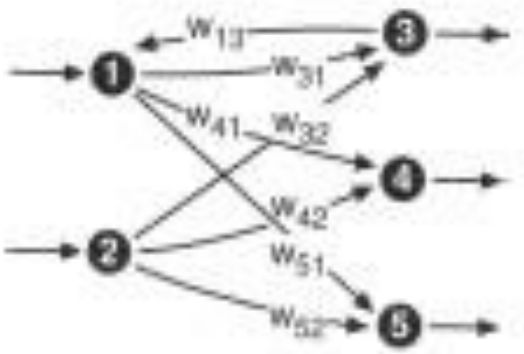
MULTI - LAYER NET

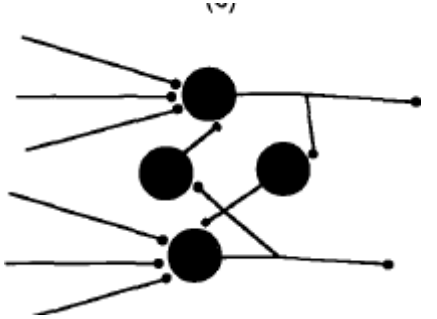
- More complicated mapping problems may require a multilayer network.
- A multilayer net is a net with one or more layers (or levels) of nodes (the so called **hidden units**) between the input units and the output units.



FEED FORWARD /FEEDBACK CONNECTIONS

- Local groups of neurons can be connected in either,
  - a *feedforward* architecture, in which the network has no loops, or
  - a *feedback* (recurrent) architecture, in which loops occur in the network because of feedback connections.





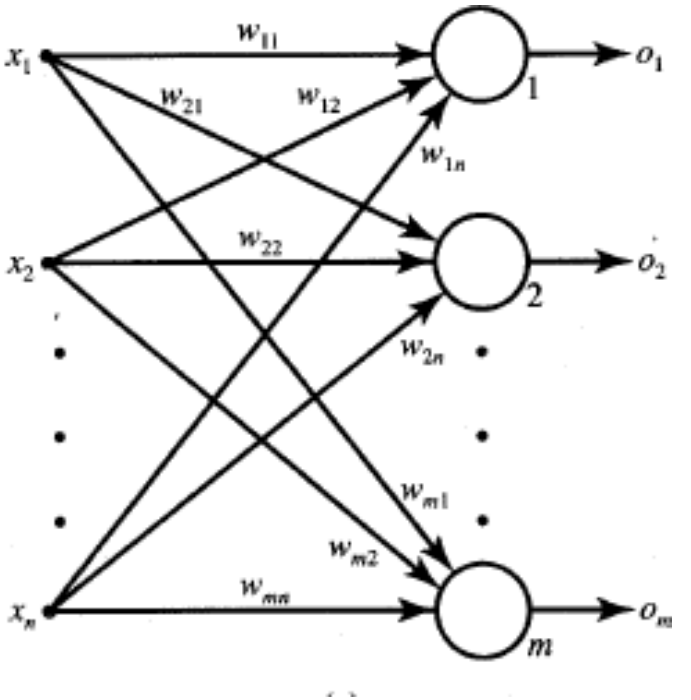
FEED FORWARD NN
CONVENTIONS FOLLOWED IN ZURADA

- Consider an elementary feedforward architecture of m neurons receiving n inputs.
- The network layer is described by the formula

$$\mathbf{o} = \Gamma[\mathbf{W}\mathbf{x}]$$

- Its output and input can be shown as vectors.

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



Weight w_{ij} *connects the* i^{th} *neuron with the* j^{th} *input. The* I and II subscripts denote the index of the destination and source nodes, respectively. The activation value for the i^{th} neuron is

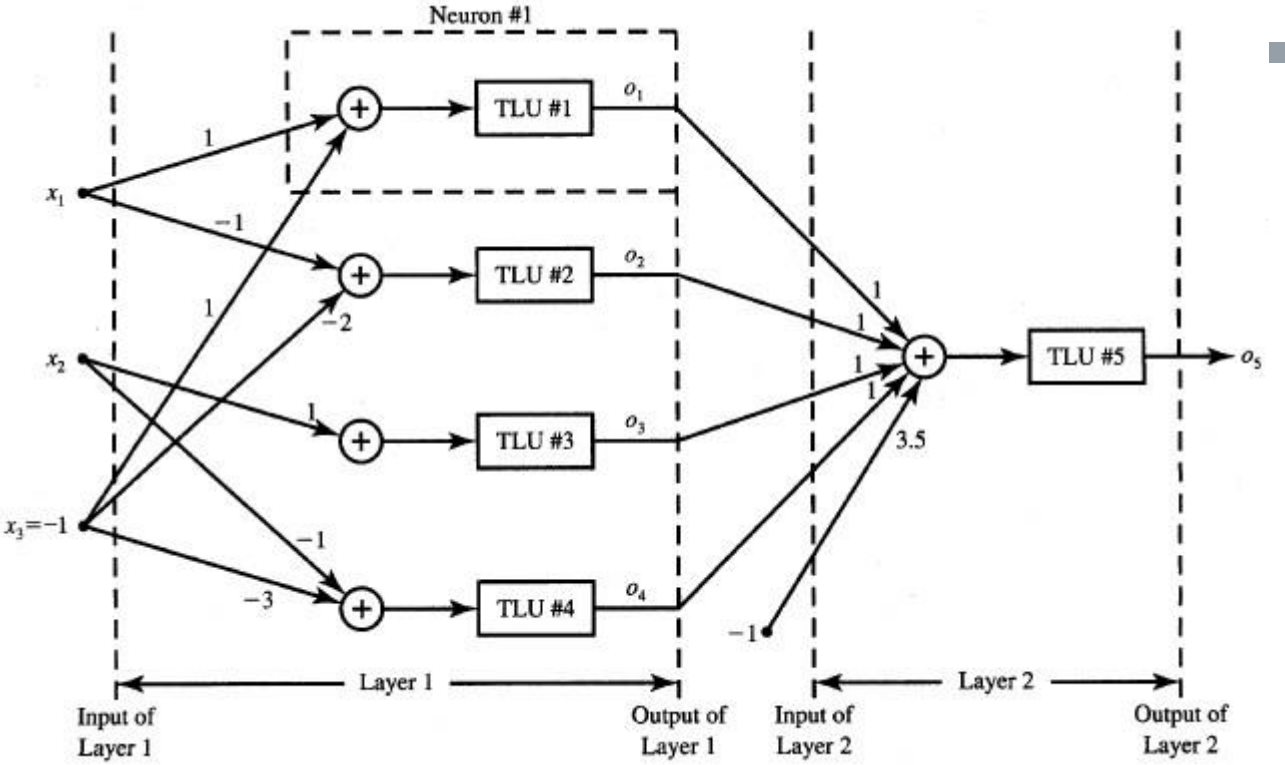
$$net_i = \sum_{j=1}^{n} w_{ij} x_j, \quad \text{for } i = 1, 2, \ldots, m$$

The nonlinear transformation involving the activation function $f(net_i)$, for i = 1, 2, . . . , m, completes the processing of x. The transformation, performed by each of the m neurons in the network is

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

where weight vector w_i contains weights leading toward the i^{th} node

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



Consider the two-layer feedforward network using neurons having the bipolar binary activation function

$$f(net) \triangleq \operatorname{sgn}(net) = \begin{cases} +1, & net > 0 \\ -1, & net < 0 \end{cases}$$

By inspection of the network diagram, obtain the output, input vectors, and the weight matrix for the first layer.

$$\mathbf{o} = [o_1 \quad o_2 \quad o_3 \quad o_4]^t$$
$$\mathbf{x} = [x_1 \quad x_2 \quad -1]^t$$

$$\mathbf{W}_1 = \begin{bmatrix} 1 & 0 & 1 \\ -1 & 0 & -2 \\ 0 & 1 & 0 \\ 0 & -1 & -3 \end{bmatrix}$$

For the second layer ?

$$\mathbf{o} = [o_5]$$
$$\mathbf{x} = [o_1 \quad o_2 \quad o_3 \quad o_4 \quad -1]^t$$

$$\mathbf{W}_2 = [1 \quad 1 \quad 1 \quad 1 \quad 3.5]$$

Compute the response of the first layer for bipolar binary activation

$$\mathbf{o} = [\operatorname{sgn}(x_1 - 1) \quad \operatorname{sgn}(-x_1 + 2) \quad \operatorname{sgn}(x_2) \quad \operatorname{sgn}(-x_2 + 3)]^t$$

The response of the second layer

$$o_5 = \operatorname{sgn}(o_1 + o_2 + o_3 + o_4 - 3.5)$$

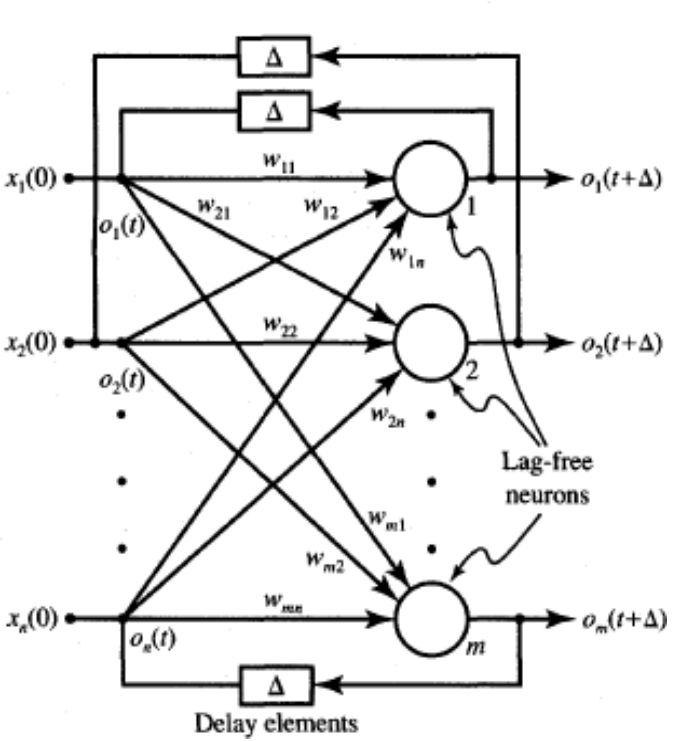
When will the fifth neuron fire ?

The fifth neuron responds + 1 if and only if $o_1 = o_2 = o_3 = o_4 = 1$.

RECURRENT NETWORKS

- It has atleast one feedback loop.
- Output o_i is controlled through outputs o_j, for $j = 1, 2, \ldots, m$.
- The present output, o(t), controls the output at the following instant, o(t + Δ).
- The time Δ elapsed between t and t + Δ is introduced by the delay elements in the feedback loop
- The mapping of o(t) into o(t + Δ) can now be written as

$$\mathbf{o}(t + \Delta) = \Gamma[\mathbf{W}\mathbf{o}(t)]$$



- The input x(t) is only needed to initialize this network so that o(0) = x(0).
- The input is then removed and the system remains autonomous for t > 0.

DISCRETE TIME RECURRENT NETWORK

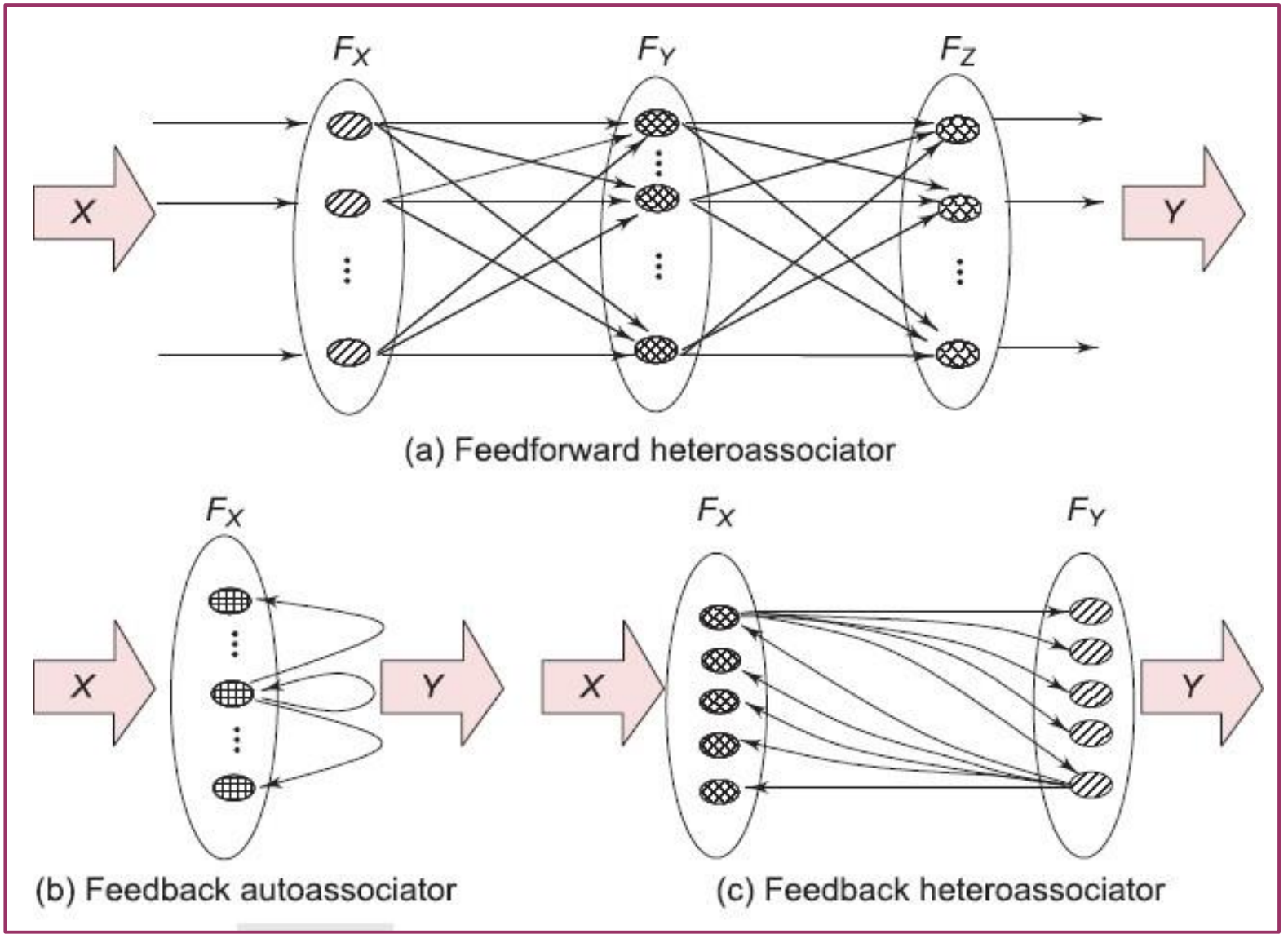
Time is considered as a discrete variable and the network performance is observed at discrete time instants $\Delta, 2\Delta, 3\Delta, \ldots,$

$$\mathbf{o}^{k+1} = \Gamma[\mathbf{W}\mathbf{o}^k], \quad \text{for } k = 1, 2, \ldots$$

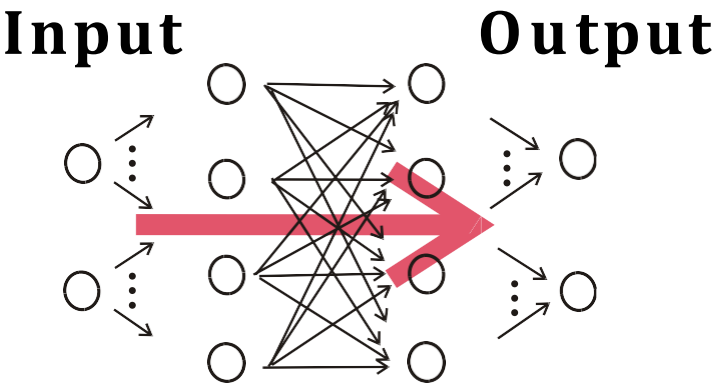
where k is the instant number.

The network is called recurrent since its response at the $(k + l)^{th}$ instant depends on the entire history of the network starting at k =0

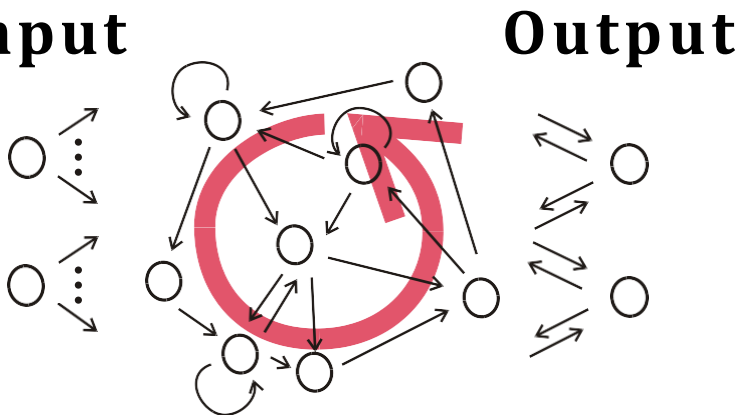
$$\mathbf{o}^1 = \Gamma[\mathbf{W}\mathbf{x}^0]$$
$$\mathbf{o}^2 = \Gamma\left[\mathbf{W}\Gamma[\mathbf{W}\mathbf{x}^0]\right]$$
$$\ldots$$
$$\mathbf{o}^{k+1} = \Gamma\left[\mathbf{W}\Gamma\left[\ldots\Gamma[\mathbf{W}\mathbf{x}^0]\ldots\right]\right]$$


F_X
F_Y
F_Z
X
Y
(a) Feedforward heteroassociator
F_X
X
Y
(b) Feedback autoassociator
F_X
F_Y
X
Y
(c) Feedback heteroassociator

FEED FORWARD VS. RECURRENT NN



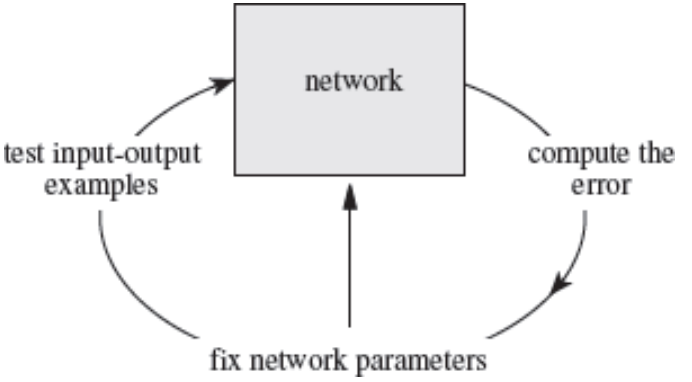
- connections only "from left to right", no connection cycle
- activation is fed forward from input to output through "hidden layers"
- no memory



- at least one connection cycle
- activation can "reverberate", persist even with no input
- system with memory

LEARNING ALGORITHMS FOR NN

- **A learning algorithm is an adaptive method by which a network of computing units self-organizes to implement the desired behavior.**
- This is done by presenting some examples of the desired input output mapping to the network.
  - A correction step is executed iteratively until the network learns to produce the desired response.
- The learning algorithm is a closed loop of presentation of examples and of corrections to the network parameters



LEARNING PROCESS

A NN could learn from many things but, the question is how to implement it. In principle, a neural network changes when its components are changing.

Theoretically, a neural network could learn by

1. developing new connections,
2. deleting existing connections,
3. changing connecting weights,
4. changing the threshold values of neurons,
5. varying one or more of the three neuron functions (activation function, propagation function and output function),
6. developing new neurons, or
7. deleting existing neurons (and so also the existing connections).

the most common procedure

CLASSES OF LEARNING ALGORITHMS

1. Supervised

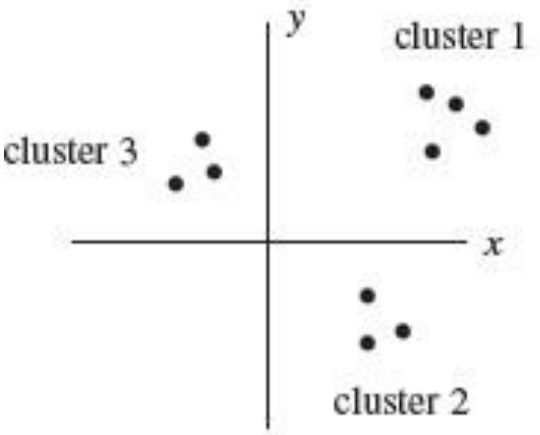
Supervised learning denotes a method in which some input vectors are collected and presented to the network. The output computed by the network is observed and the deviation from the expected answer is measured.

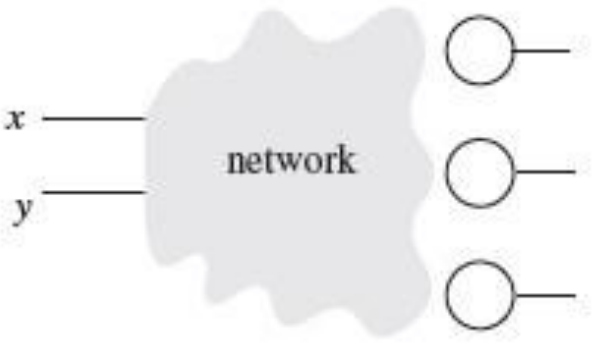
The weights are corrected according to the magnitude of the error in the way defined by the learning algorithm.

This kind of learning is also called learning with a teacher, since a control process knows the correct answer for the set of selected input vectors.

2. Unsupervised

- Unsupervised learning is used when, for a given input, the exact numerical output a network should produce is unknown.





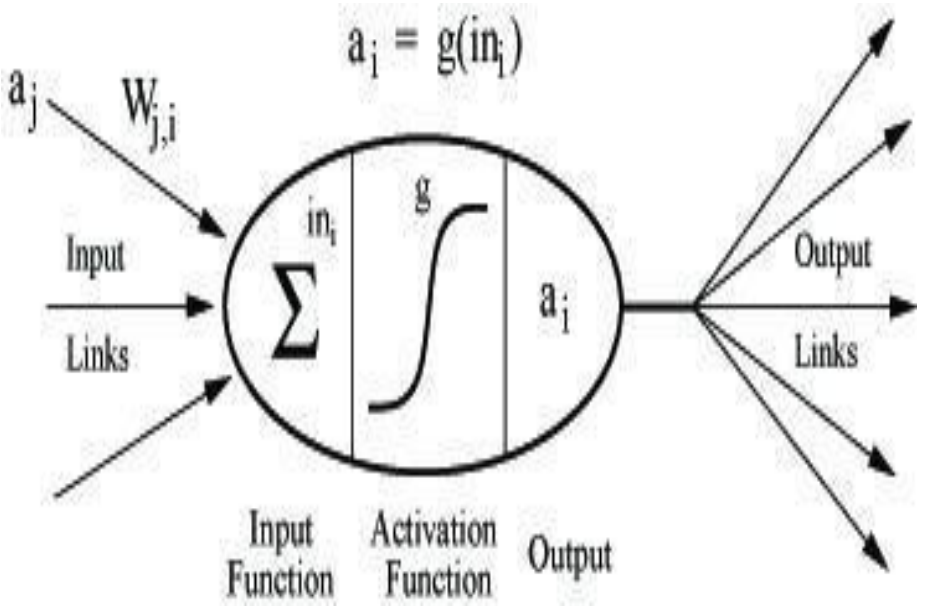
- In this case we do not know a priori which unit is going to specialize on which cluster. Generally we do not even know how many well-defined clusters are present. Since no “teacher” is available, the network must organize itself in order to be able to associate clusters with units.

In a nutshell

- Learning can be done in **supervised** or **unsupervised** training.
- **In supervised training, both the inputs and the outputs are provided.**
  - The network then processes the inputs and compares its resulting outputs against the desired outputs.
  - Errors are then calculated, causing the system to adjust the weights which control the network.
  - This process occurs over and over as the weights are continually tweaked.
- **In unsupervised training, the network is provided with inputs but not with desired outputs.**
  - The system itself must then decide what features it will use to group the input data.

- Supervised Learning
  - Recognizing hand-written digits, pattern recognition, regression.
  - Labeled examples
    (input , desired output)
  - Neural Network models: perceptron, feed-forward, radial basis function, support vector machine.
- Unsupervised Learning
  - Find similar groups of documents in the web, content addressable memory, clustering.
  - Unlabeled examples
    (different realizations of the input alone)
  - Neural Network models: self organizing maps, Hopfield networks.

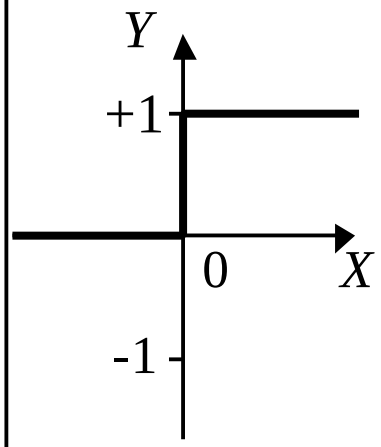
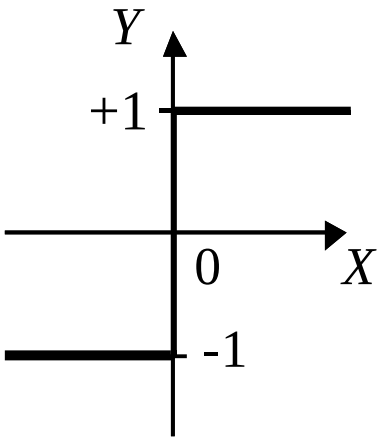
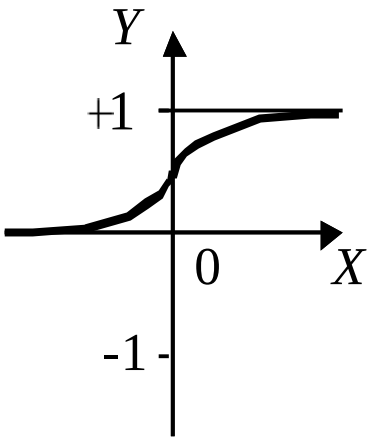
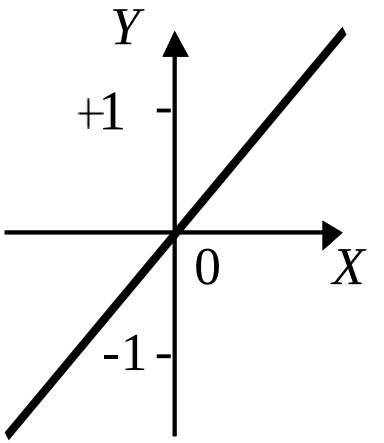
ACTIVATION FUNCTION



- Receives n inputs
- Multiplies each input by its weight
- Applies activation function to the sum of results
- Outputs result

- Usually, doesn't just use weighted sum directly
- Applies some function to weighted sum before use (e.g., as output)
- Call this the *activation function*

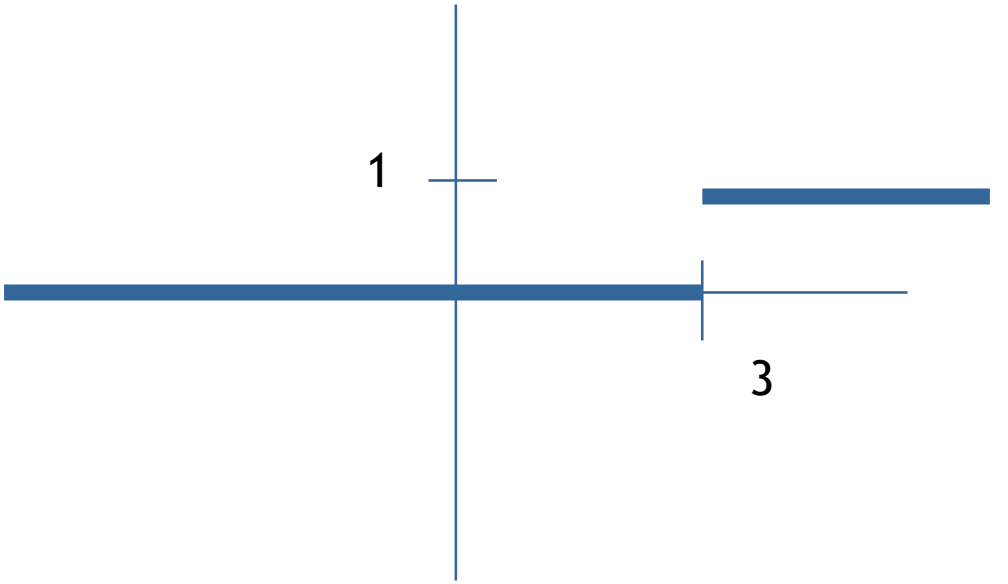
SOME COMMONLY USED ACTIVATION FUNCTIONS OF A NEURON

Step function	*Sign function*	*Sigmoid function*	*Linear function*
			
$Y^{step} = \begin{cases} 1, \text{ if } X \geq 0 \\ 0, \text{if } X < 0 \end{cases}$	$Y^{sign} = \begin{cases} +1, \text{if } X \geq 0 \\ -1, \text{if } X < 0 \end{cases}$	$Y^{sigmoid} = \frac{1}{1+e^{-X}}$	$Y^{linear} = X$

Step Function Example

- Let threshold, $\theta = 3$

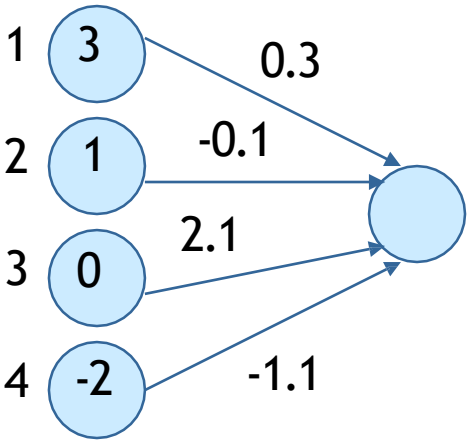
$$f(x) = \begin{cases} 1 \; if \;\; x \geq \theta \\ 0 \; if \;\; x < \theta \end{cases}$$



Input: (3, 1, 0, -2)

If Input:
(0, 10, 0, 0) ?

$$f(-1) = 0$$

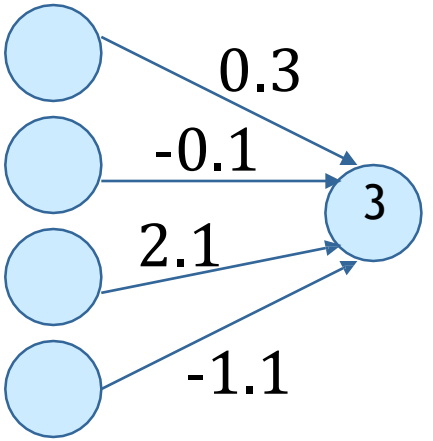


Network output after passing through step activation function???

$$f(3) = 1$$

Sigmoidal Example

Input: (3, 1, 0, -2)



steepness parameter

$$\sigma = 2$$

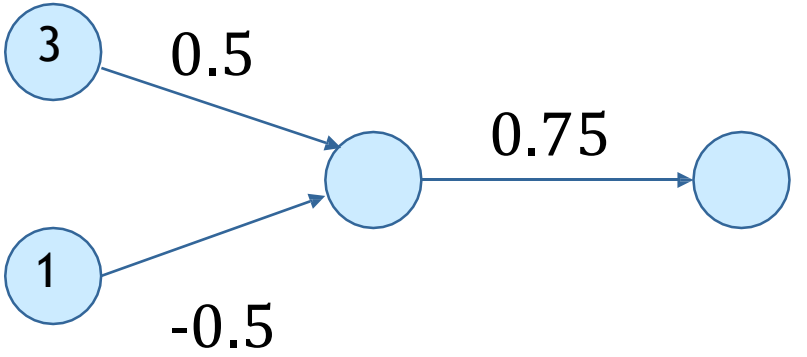
$$f(x) = \frac{1}{1+e^{-2x}}$$

Network output?

$$f(3) = \frac{1}{1+e^{-2x}} = .998$$

- A two weight layer, feed forward network
- Two inputs, one output, one hidden unit

Input: (3, 1)



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

$\sigma = ??$

$\sigma = 1$

1/e = 0.367

What is the output?

- Example solution
  - Activation of hidden unit

    f(0.5(3) + -0.5(1)) = f(1.5 – 0.5) = f(1) = 0.731
  - Output activation

    f(0.731(0.75)) = f(0.548) = .634

McCULLOCH- PITTS (A FEED-FORWARD NETWORK)

- It is one of the first of NN & very simple.
  - The nodes produce only binary results and the edges transmit exclusively ones or zeros.
  - A connection path is **excitatory** if the weight on the path is positive; otherwise it is **inhibitory**.
  - All excitatory connections into a particular neuron have the same weights.
    - However it may receive multiple inputs from the same source, so the excitatory weights are effectively positive integers.

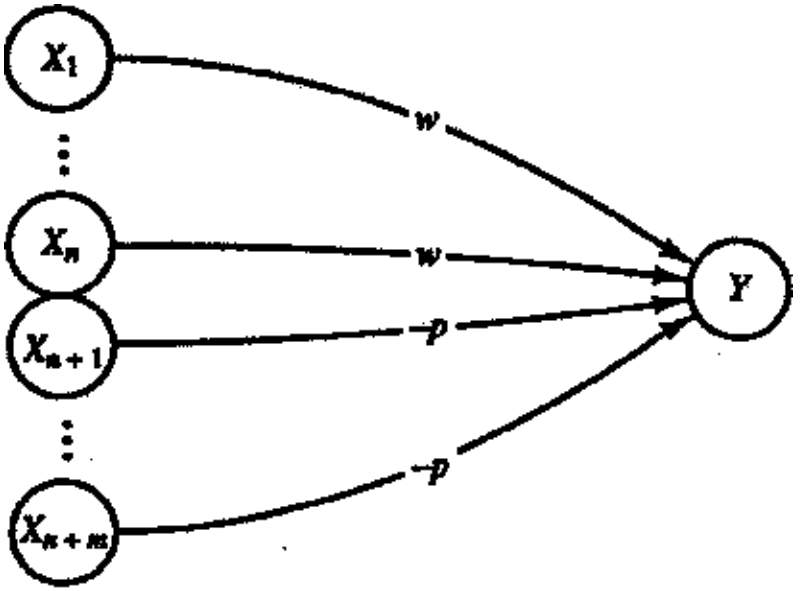
- Although all excitatory connections to a neuron have the same weights, but the weights coming into one unit need not be the same as coming into another unit.
- Each neuron has a fixed threshold such that if the net input to the neuron is greater than the threshold, the neuron fires.
- The threshold is set so that inhibition is absolute. That is, any nonzero inhibitory input will prevent the neuron from firing.
- It takes one time step for a signal to pass over one connection link.

The rule for evaluating the input to a McCulloch–Pitts unit is the following:

- Assume that a McCulloch–Pitts unit gets an input $x_1, x_2, \ldots, x_n$ through n excitatory edges and an input $y_1, y_2, \ldots, y_m$ through m inhibitory edges.
- If $m \geq 1$ and at least one of the signals $y_1, y_2, \ldots, y_m$ is 1, the unit is inhibited and the result of the computation is 0.
- Otherwise the total excitation $x = x_1 + x_2 + \cdots + x_n$ is computed and compared with the threshold θ of the unit (if $n = 0$ then $x = 0$). If $x \geq \theta$ the unit *fires* a 1, if $x < \theta$ the result of the computation is 0.

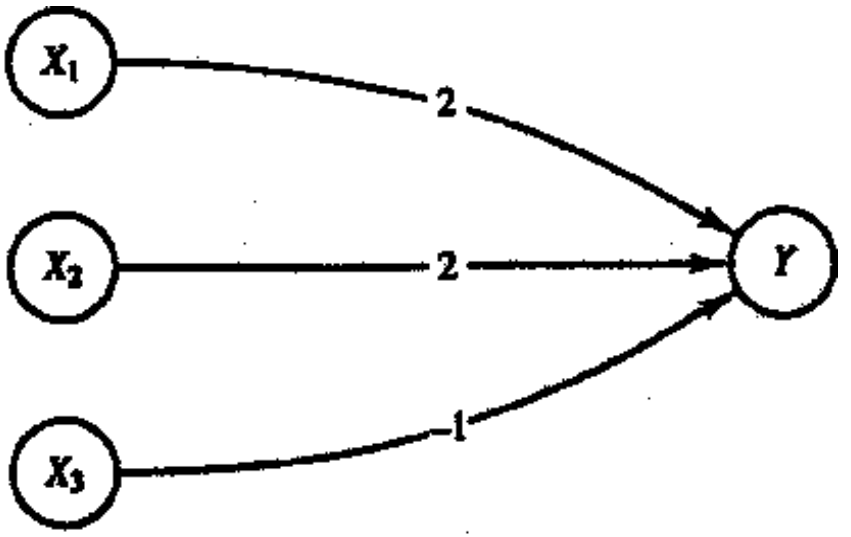
ARCHITECTURE

- In general, McCulloch-Pitts neuron Y can receive signals from any number of neurons.
- Each connection is either excitatory, with w > 0, or inhibitory with weight –p.



Architecture of a McCulloch-Pitts neuron *Y*.

$$f(y\_in) = \begin{cases} 1 & \text{if } y\_in \geq \theta \\ 0 & \text{if } y\_in < \theta \end{cases}$$



A simple McCulloch-Pitts neuron Y.

"The threshold is set so that inhibition is absolute. That is, any nonzero inhibitory input will prevent the neuron from firing."

What threshold value should we set?

The threshold for unit Y is 4

- Suppose there are n excitatory input links with weight w & m inhibitory links with weight –p.
- What should be the threshold value?
- The condition that inhibition is absolute requires that for the activation function satisfy the inequality:

$$\Theta > nw - p$$

- A neuron fires if it receives k or more excitatory inputs and no inhibitory inputs, **what is the relation between k & Θ?**

$$kw \geq \Theta > (k-1)w$$

SOME SIMPLE McCULLOCH-PITTS NEURONS

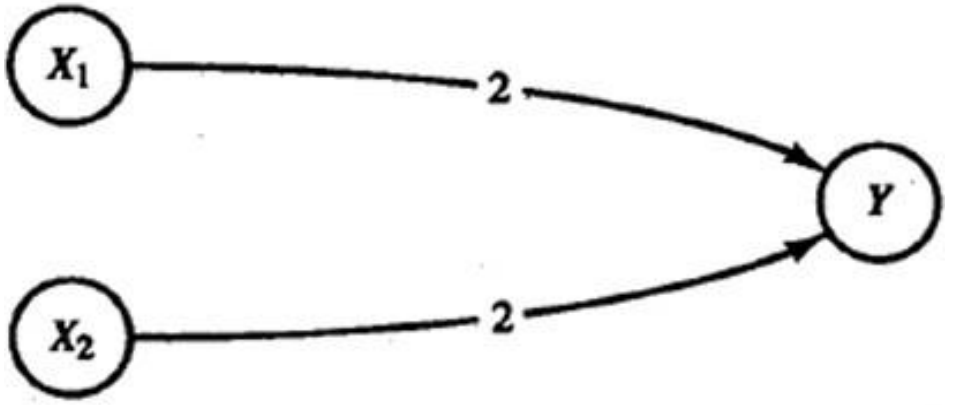
- The weights for a McCulloch-Pitts neuron are set, together with the threshold for the neuron's activation function, so that the neuron will perform a simple logic function.
- **Using these simple neurons as building blocks, we can model any function or phenomenon that can be represented as a logic function.**

EXAMPLES

- Train a McCulloch-Pitts neural network to perform the OR function.
- Train a McCulloch-Pitts neural network to perform the AND function.
- Train a McCulloch-Pitts neural network to perform the NOR & NOT function.
- Train a McCulloch-Pitts neural network to perform the AND NOT function.
- Train a McCulloch-Pitts neural network to perform the XOR function.

OR

x_1	x_2	→	y
1	1		1
1	0		1
0	1		1
0	0		0

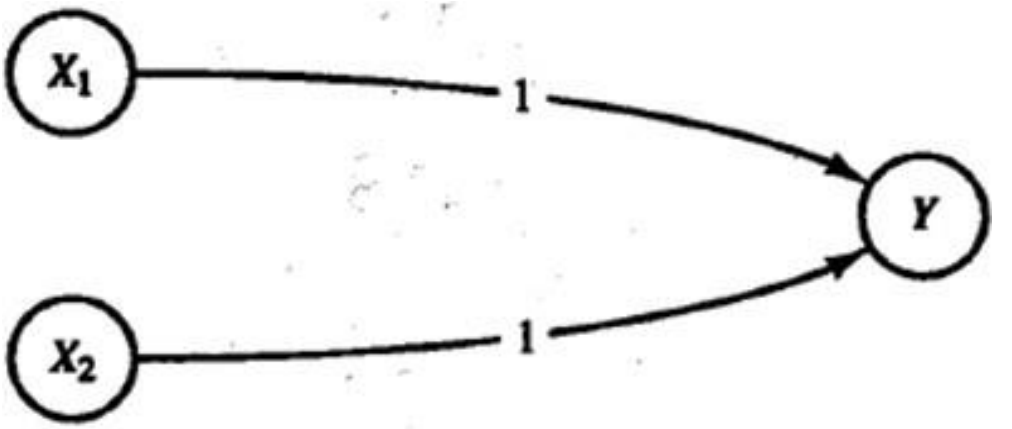


A McCulloch-Pitts neuron to perform the logical OR function.

Take threshold as 2

AND

x_1	x_2	$\rightarrow$	y
1	1		1
1	0		0
0	1		0
0	0		0

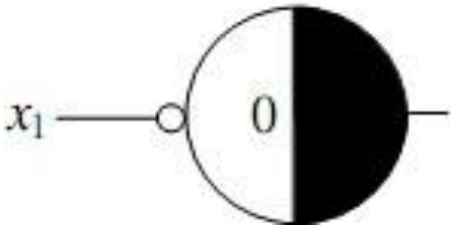


A McCulloch-Pitts neuron to perform the logical AND function.

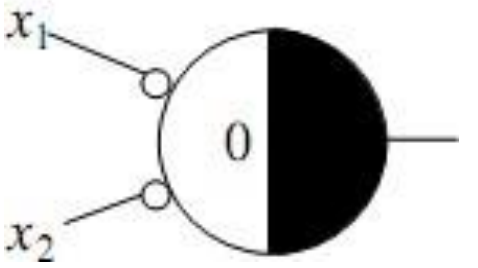
Take threshold as 2

NOT & NOR

NOT

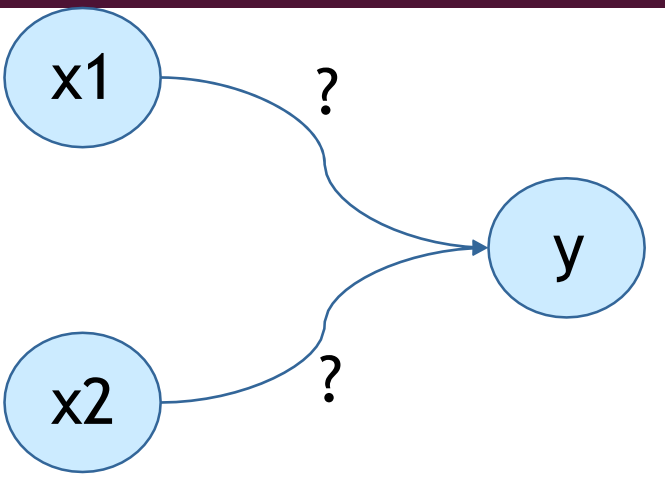


NOR



XOR

x_1	x_2	$\rightarrow$	y
1	1		0
1	0		1
0	1		1
0	0		0



- How long do we keep looking for a solution? We need to be able to calculate appropriate parameters rather than looking for solutions by trial and error.
- Each training pattern produces a linear inequality for the output in terms of the inputs and the network parameters. These can be used to compute the weights and thresholds.

FINDING THE WEIGHTS ANALYTICALLY

- We have two weights *w1* and *w2* and the threshold Θ, and for each training pattern we need to satisfy

$$out = \mathrm{sgn}(w_1 in_1 + w_2 in_2 - \theta)$$

in_1	in_2	out
0	0	0
0	1	1
1	0	1
1	1	0

So what inequations do we get?

- For the XOR network
  - Clearly the second and third inequalities are incompatible with the fourth, so there is in fact no solution.
  - We need more complex networks, e.g. that combine together many simple networks, or use different activation / thresholding /transfer functions.

in_1	in_2	out
0	0	0
0	1	1
1	0	1
1	1	0

$\Rightarrow$

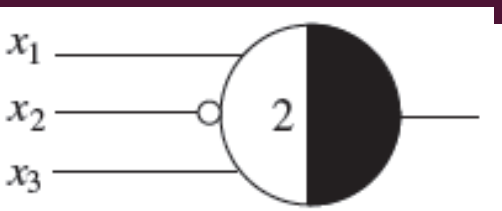
$$
\begin{aligned}
w_1\,0 + w_2\,0 - \theta &< 0 \\
w_1\,0 + w_2\,1 - \theta &\geq 0 \\
w_1\,1 + w_2\,0 - \theta &\geq 0 \\
w_1\,1 + w_2\,1 - \theta &< 0
\end{aligned}
$$

$\Rightarrow$

$$
\begin{aligned}
\theta &> 0 \\
w_2 &\geq \theta \\
w_1 &\geq \theta \\
w_1 + w_2 &< \theta
\end{aligned}
$$

MCCULLOCH–PITTS UNITS CAN BE USED AS BINARY DECODERS

Suppose F is a function with 3 arguments. Design McCulloch-Pitts unit for (1,0,1).

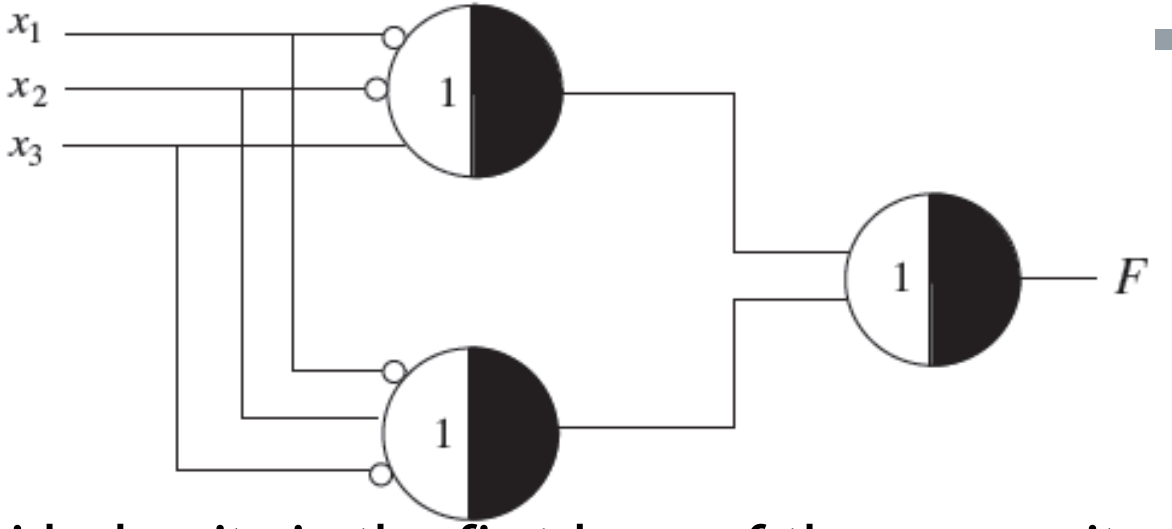


Decoder for the vector (1, 0, 1)

Assume that a function F of three arguments has been defined according to the following table. Design McCulloch-Pitts units for it.

input vectors	*F*
(0,0,1)	1
(0,1,0)	1
all others	0

To compute this function it is only necessary to decode all those vectors for which the function's value is 1.

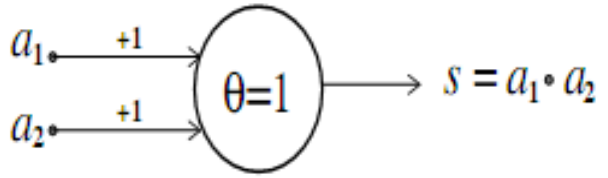


- The individual units in the first layer of the composite network are decoders.
- For each vector for which F is 1 a decoder is used. In our case we need just two decoders.
- Components of each vector which must be 0 are transmitted with inhibitory edges, components which must be 1 with excitatory ones.
- The threshold of each unit is equal to the number of bits equal to 1 that must be present in the desired input vector.
- The last unit to the right is a disjunction: if any one of the specified vectors can be decoded this unit fires a 1.

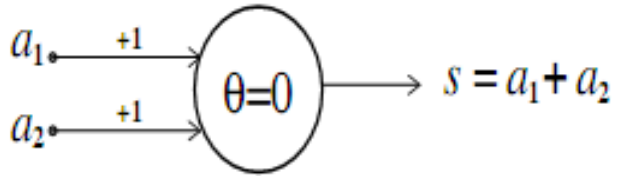
2. Design networks using M-P neurons to realize the following logic functions using **± 1 for the weights.**

a) $s(a_1, a_2, a_3) = a_1\, a_2\, a_3$
b) $s(a_1, a_2, a_3) = \sim a_1\, a_2 \sim a_3$
c) $s(a_1, a_2, a_3) = a_1\, a_3 + a_2\, a_3 + \sim a_1 \sim a_3$

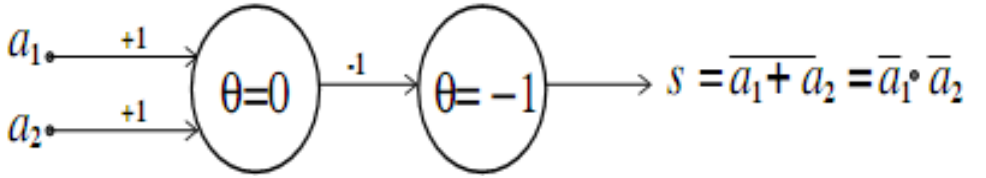
and

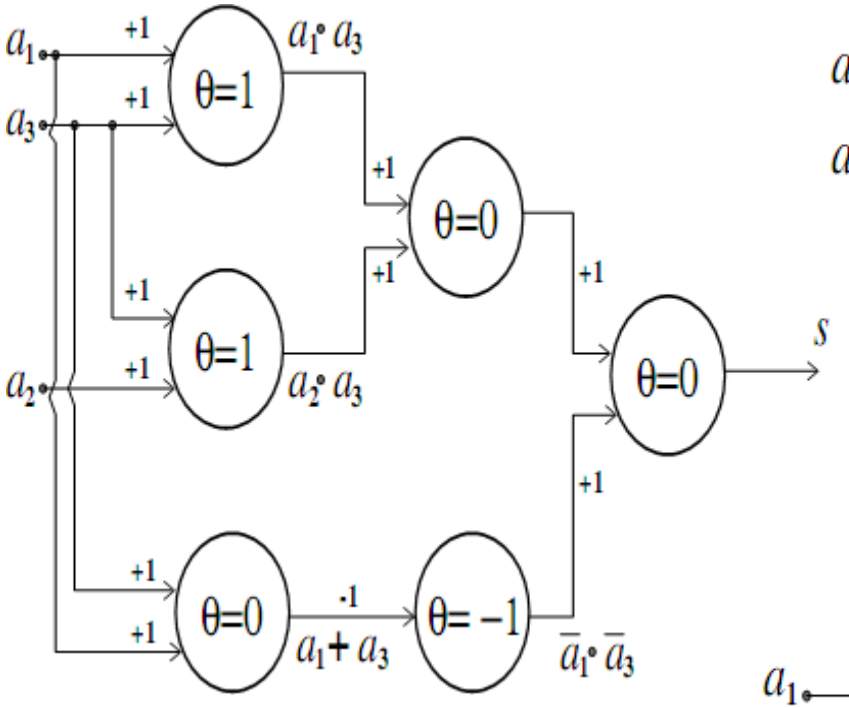


or

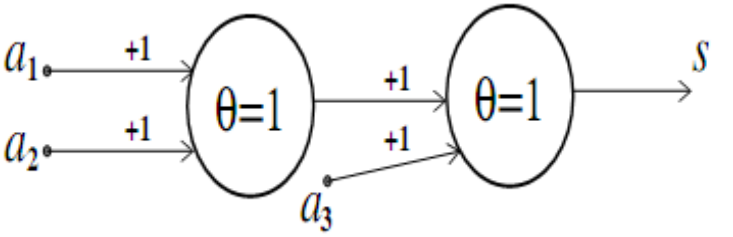


nor

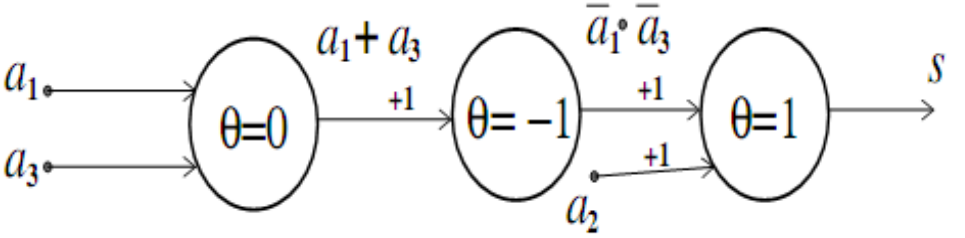



a_1
a_3
a_2
+1
θ=1
$a_1 \cdot a_3$
θ=0
$a_2 \cdot a_3$
θ=0
s
θ=0
-1
$a_1 + a_3$
θ= −1
$\bar{a}_1 \cdot \bar{a}_3$

(c)


a_1
a_2
a_3
+1
θ=1
θ=1
s

(a)

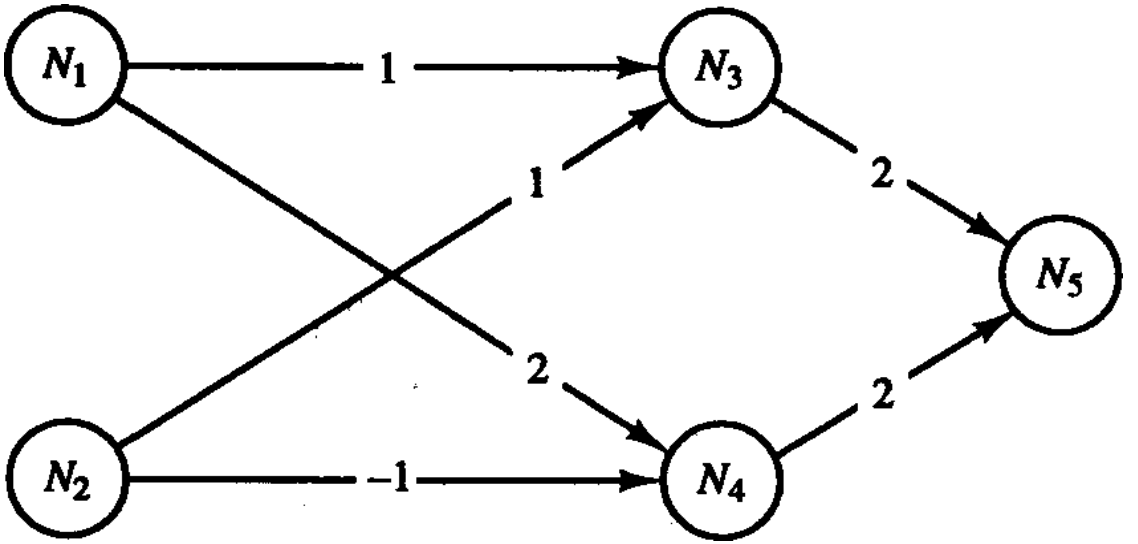

a_1
a_3
θ=0
$a_1 + a_3$
+1
θ= −1
$\bar{a}_1 \cdot \bar{a}_3$
+1
θ=1
s
+1
a_2

(b)

Consider the neural network of McCulloch-Pitts neurons shown in Figure 1.25. Each neuron (other than the input neurons, N_1 and N_2) has a threshold of 2.

a. Define the response of neuron N_5 at time t in terms of the activations of the input neurons, N_1 and N_2, at the appropriate time.

b. Show the activation of each neuron that results from an input signal of $N_1 = 1$, $N_2 = 0$ at $t = 0$.



RECURRENT NETWORKS

- Neural networks were designed on analogy with the brain.
- The brain's memory, however, works by association.
  - For example, we can recognize a familiar face even in an unfamiliar environment within 100-200 ms.
  - We can also recall a complete sensory experience, including sounds and scenes, when we hear only a few bars of music. The brain routinely associates one thing with another.

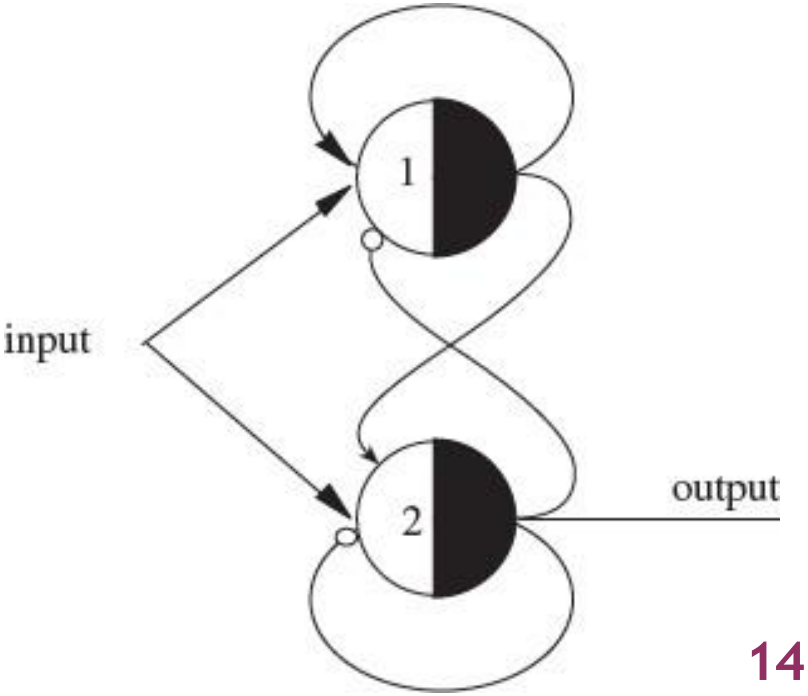
To emulate the human memory's associative characteristics we need a different type of network: a recurrent neural network.

A recurrent neural network has feedback loops from its outputs to its inputs.

The presence of such loops has a profound impact on the learning capability of the network.

- McCulloch–Pitts units can be used in recurrent networks by introducing a temporal factor in the computation.
- It is assumed that computation of the activation of each unit consumes a time unit.
  - If the input arrives at time t the result is produced at time t + 1.
- Care needs to be taken to coordinate the arrival of the input values at the nodes.
  - This could make the introduction of additional computing elements necessary, whose sole mission is to insert the necessary delays for the coordinated arrival of information.
- This is the same problem that any computer with clocked elements has to deal with.

Design a network that processes a sequence of bits, giving off one bit of output for every bit of input, but in such a way that any two consecutive ones are transformed into the sequence 10. E.g. The binary sequence 00110110 is transformed into the sequence 00100100.



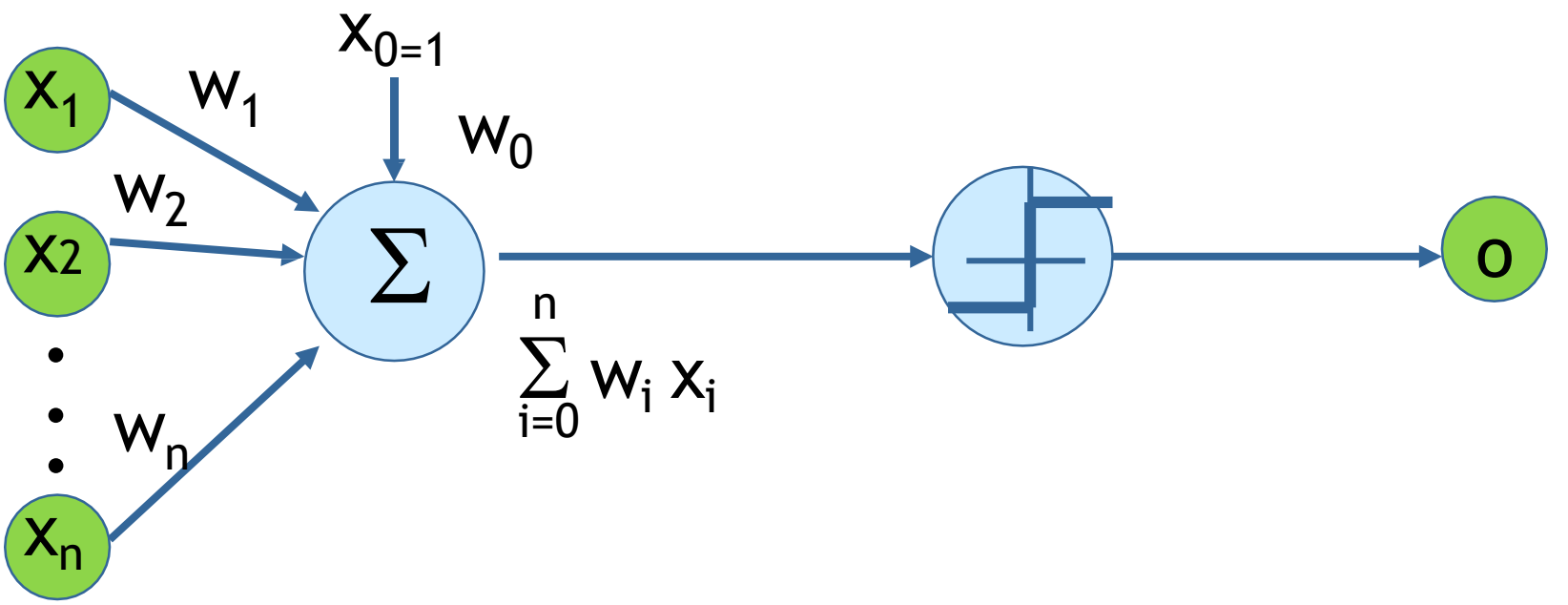
McCulloch–Pitts units can be used to build networks capable of computing any logical function and of simulating any finite automaton. However ...

- The computing units are too similar to conventional logic gates and the network must be completely specified before it can be used.
- There are no free parameters which could be adjusted to suit different problems.
- Learning can only be implemented by modifying the connection pattern of the network and the thresholds of the units, but this is necessarily more complex than just adjusting numerical parameters.

For that reason, we turn our attention to weighted networks and consider their most relevant properties.

PERCEPTRON

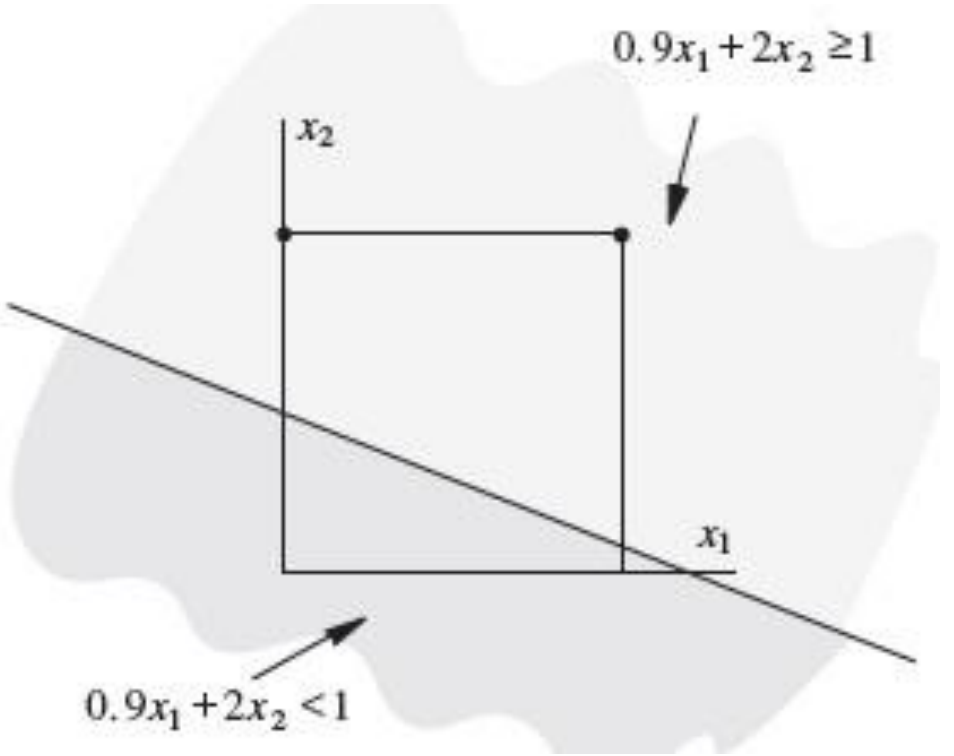
- In 1958 Frank Rosenblatt, an American psychologist, proposed the perceptron, a more general computational model than McCulloch–Pitts units.
- From a formal point of view, the only difference between McCulloch–Pitts elements and perceptrons is the presence of weights in the networks.
- Perceptrons are multilayer feed forward networks without recurrence and with fixed input and output layers.
- The perceptron has exactly two possible output values (e.g. {0,1} or {−1,1}). Thus, a binary threshold function is used as activation function, depending on the threshold value Θ of the neuron.



A simple perceptron is a computing unit with threshold which, when receiving the n real inputs x_1, x_2, . . . , x_n through edges with the associated weights w_1,w_2, . . .,w_n, outputs 1 if the inequality $\Sigma w_i x_i$, i = 1 – n, holds and 0 otherwise.

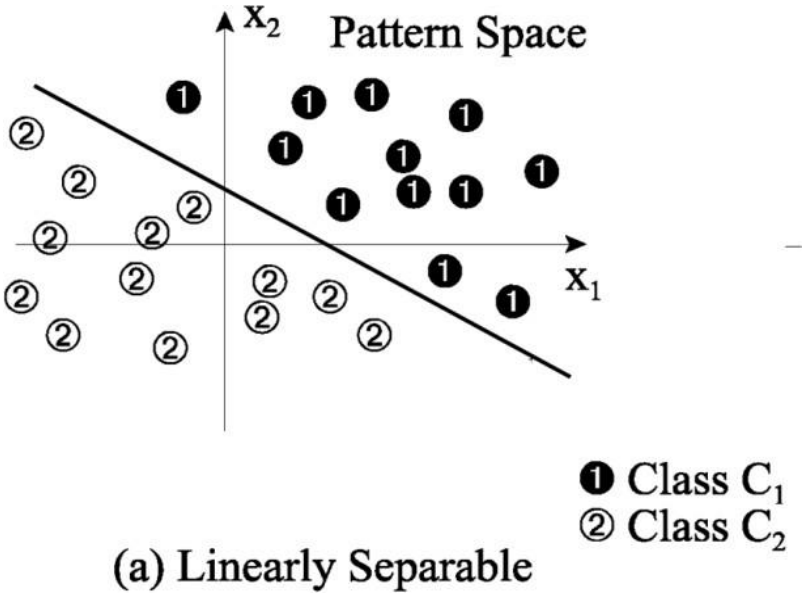
A perceptron separates the input space into two half-spaces.
For points belonging to one half-space the result of the computation is 0, for points belonging to the other it is 1.

A perceptron with threshold 1, at which two edges with weights 0.9 and 2.0 impinge, tests the condition $0.9x_1 + 2x_2 \geq 1$

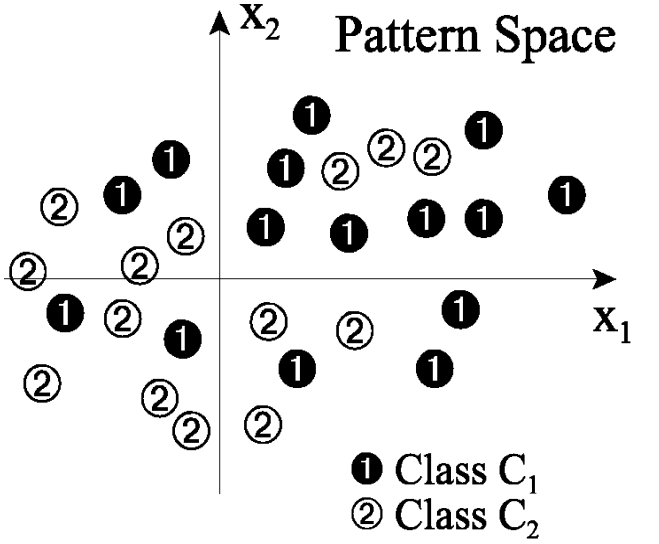


A perceptron can learn only examples that are called "linearly separable".

Linearly Separable Patterns



(a) Linearly Separable



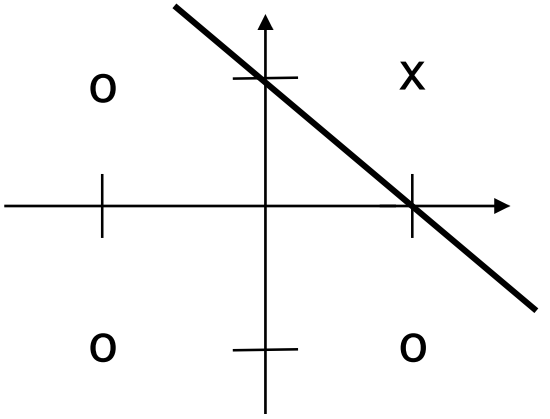
(b) Not Linearly Separable

Perceptrons can learn many boolean functions: AND, OR, NAND, NOR, but not XOR

Are AND & OR functions linearly separable?

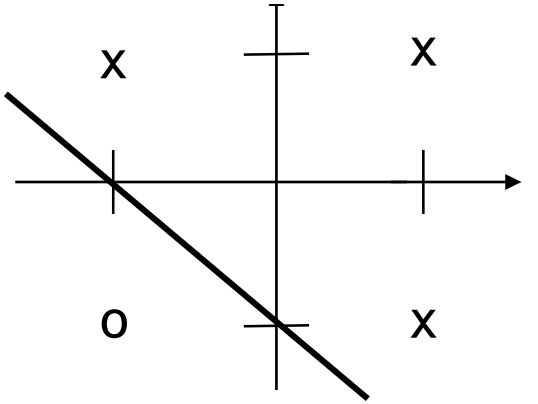
What about XOR?

x_1	x_2	→	y
1	1		1
1	0		0
0	1		0
0	0		0

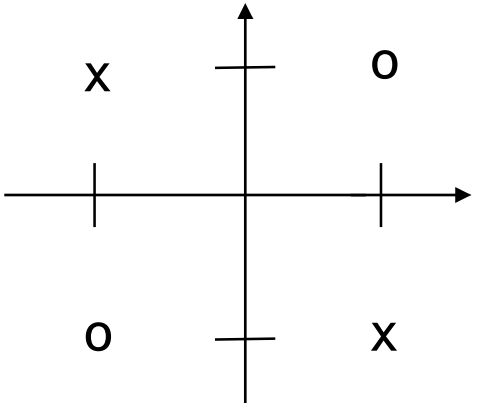


x: class I (y = 1)
o: class II (y = -1)

x_1	x_2	→	y
1	1		1
1	0		1
0	1		1
0	0		0



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



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

XOR

x_1	x_2	→	y
1	1		0
1	0		1
0	1		1
0	0		0

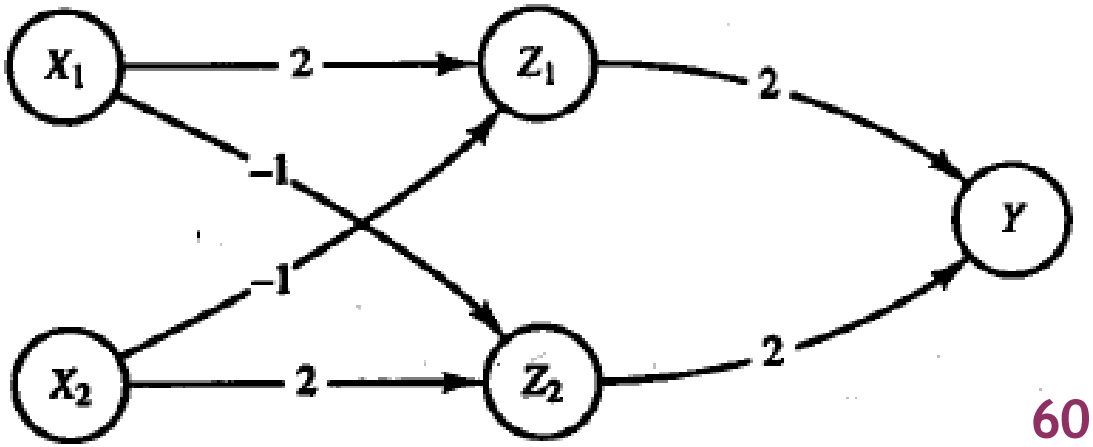
However, every boolean function can be represented with a perceptron network that has two levels of depth or more.

$$x_1 \text{ XOR } x_2 \leftrightarrow (x_1 \text{ AND NOT } x_2) \text{ OR } (x_2 \text{ AND NOT } x_1).$$

$$z_1 = x_1 \text{ AND NOT } x_2$$

$$y = z_1 \text{ OR } z_2.$$

$$z_2 = x_2 \text{ AND NOT } x_1.$$



PERCEPTRON LEARNING ALGORITHM

How does a perceptron learn the appropriate weights?

1. Assign random values to the weight vector
2. Apply the *weight update rule to every training example*
3. Are all training examples correctly classified?
   a. Yes. Quit
   b. No. Go back to Step 2.

There are two popular weight update rules.

i) The perceptron rule, and

ii) Delta rule

PERCEPTRON RULE

We start with an e.g.

- Consider the features:

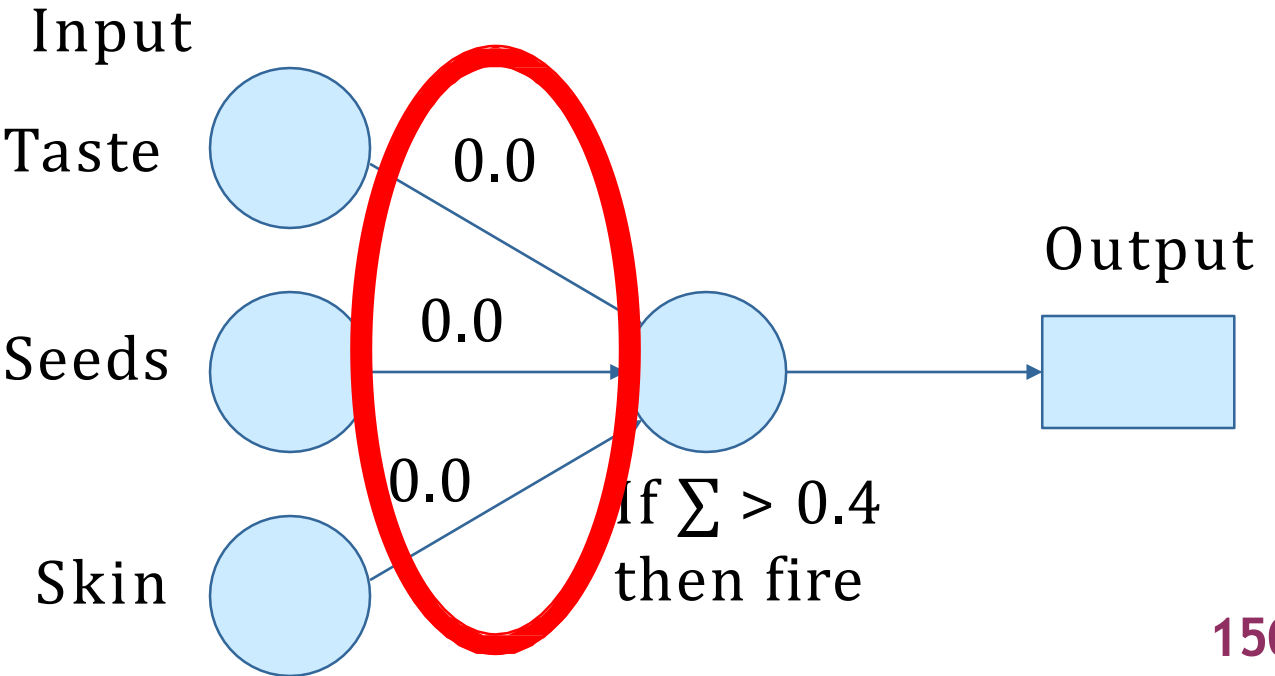
For output:

Good_Fruit = 1

Not_Good_Fruit = 0

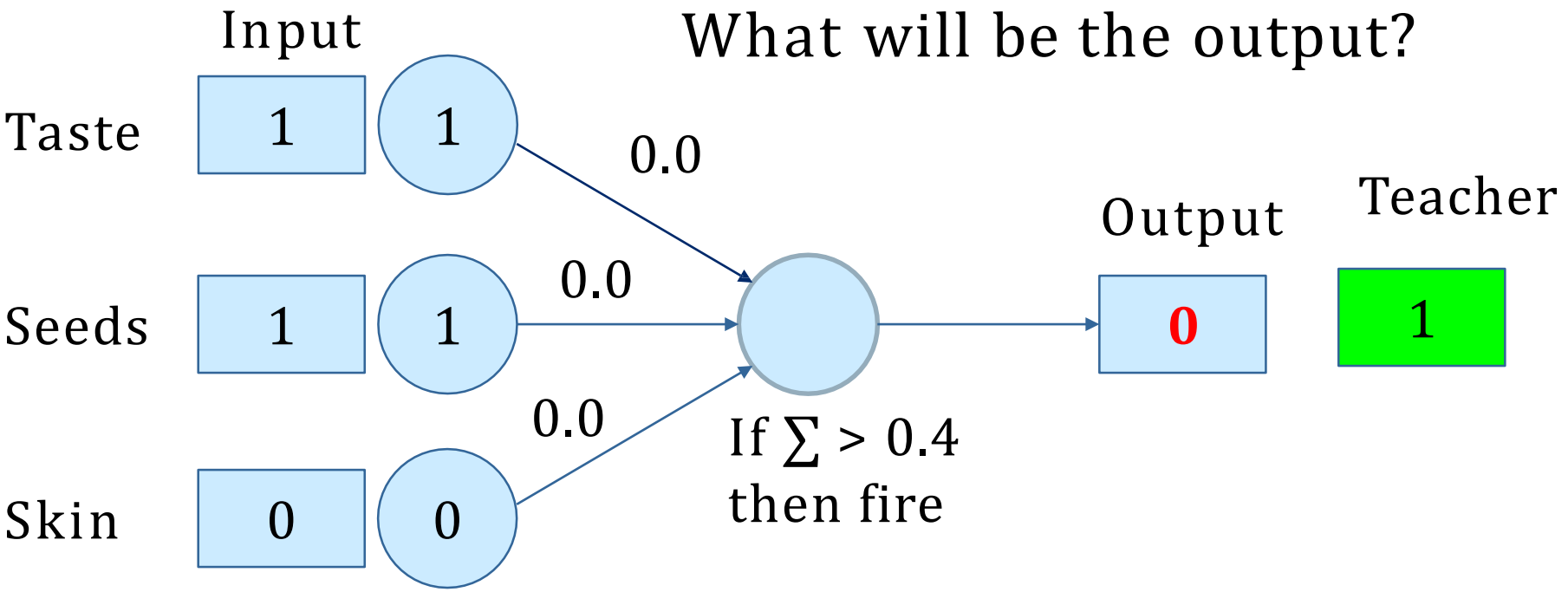
Taste	Sweet = 1, Not_Sweet = 0
Seeds	Edible = 1, Not_Edible = 0
Skin	Edible = 1, Not_Edible = 0

- Let's start with no knowledge:
- The weights are empty:



- To train the perceptron, we will show it with example and have it categorize each one.
- Since it's starting with no knowledge, it is going to make mistakes. When it makes a mistake, we are going to adjust the weights to make that mistake less likely in the future.
- When we adjust the weights, we're going to take relatively small steps to be sure we don't over-correct and create new problems.
- It's going to learn the category "good fruit" defined as anything that is sweet & either skin or seed is edible.
  - Good fruit = 1
  - Not good fruit = 0

Pomegranate is Good:



- Since we got it wrong, we know we need to change the weights.
  - Δw = learning rate x (overall teacher - overall output) x node output

- **The three parts of that are:**
  - **Learning rate:**
    We set that ourselves. It should be large enough that learning happens in a reasonable amount of time, but small enough that it doesn't go too fast.
    Let's take it as 0.25.
  - **(overall teacher - overall output):**
    The teacher knows the correct answer (e.g., that a pomegranate should be a good fruit). In this case, the teacher says 1, the output is 0, so (1 - 0) = 1.
  - **node output:**
    That's what came out of the node whose weight we're adjusting. For the first node, 1.

- $\Delta w = 0.25 \times 1 \times 1 = \mathbf{0.25}$
- Since it's a Δw, it's telling us how much to change the first weight. In this case, we're adding 0.25 to it.

ANALYSIS OF PERCEPTRON RULE

- **(overall teacher - overall output):**
  - **If we get the categorization right**, (overall teacher - overall output) will be zero.
  - In other words, if we get it right, we won't change any of the weights.
  - **If we get the categorization wrong,** (overall teacher - overall output) will either be -1 or +1.
    - If we said "yes" when the answer was "no," we're too high on the weights and we will get a (teacher - output) of -1 which will result in reducing the weights.
    - If we said "no" when the answer was "yes," we're too low on the weights and this will cause them to be increased.

- **Node output:**
  - If the node whose weight we're adjusting sent in a 0, then it didn't participate in making the decision. In that case, it shouldn't be adjusted. Multiplying by zero will make that happen.
  - If the node whose weight we're adjusting sent in a 1, then it did participate and we should change the weight (up or down as needed).

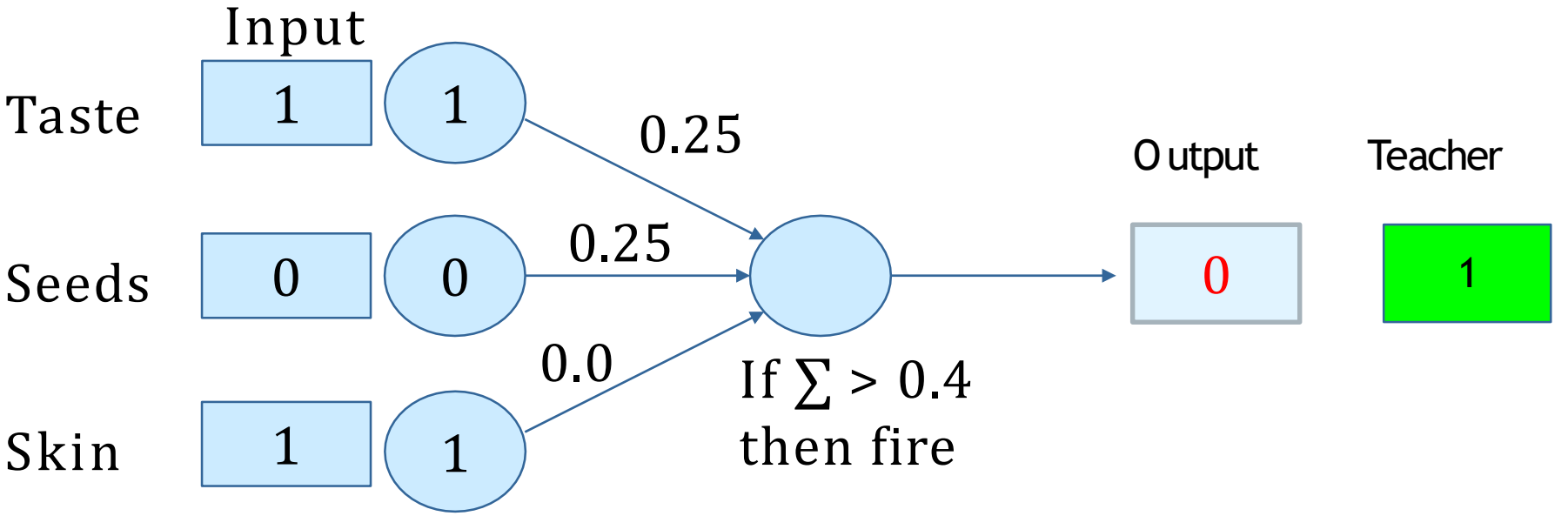
How do we change the weights for pomegranate?

Feature:	Learning rate:	(overall teacher – overall output)	Node output:	Δw
taste	0.25	1	1	+0.25
seeds	0.25	1	1	+0.25
skin	0.25	1	0	0

- To continue training, we show it the next example, adjust the weights…
- We will keep cycling through the examples until we go all the way through one time without making any changes to the weights. At that point, the concept is learned.

Pear is good:

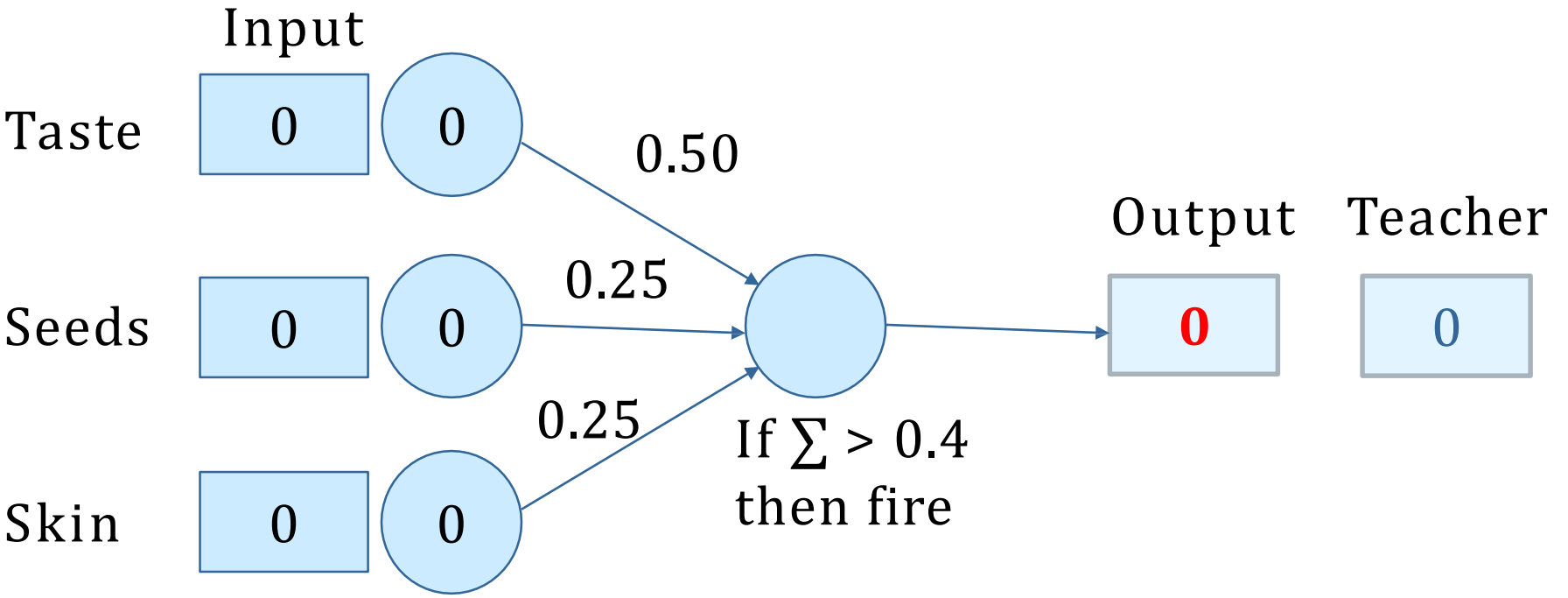
What will be the output?



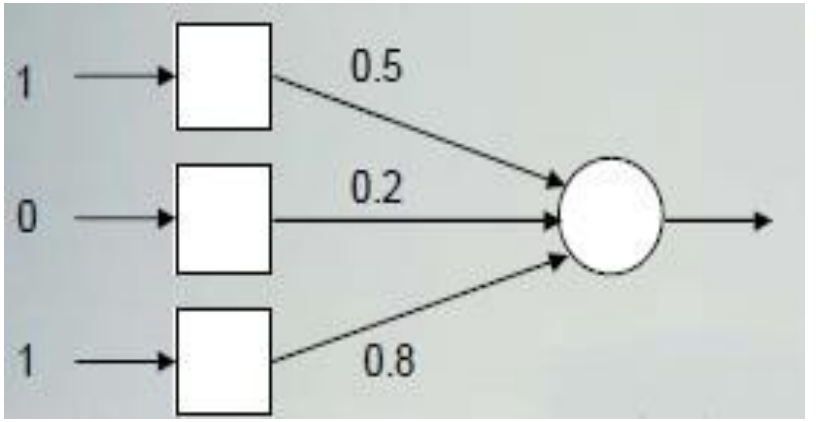
How do we change the weights for pear?

Feature:	Learning rate:	(overall teacher – overall o/p):	Node output:	Δw
taste	0.25	1	1	+0.25
seeds	0.25	1	0	0
skin	0.25	1	1	+0.25

Lemon is not good:



- Do we change the weights for lemon?
- Since (overall teacher - overall output)=0, there will be no change in the weights.



What will be the output if the threshold is 1.2?

- 1 * 0.5 + 0 * 0.2 + 1 * 0.8 =1.3
- Threshold = 1.2 & 1.3 > 1.2
- **So, o/p is 1**

Assume Output was supposed to be 0.
If $\alpha = 1$, (α is the learning rate)
What will be the new weights?

Assume $\alpha = 1$

$W_{1new} = 0.5 + 1*(0-1)*1 = -0.5$
$W_{2new} = 0.2 + 1*(0-1)*0 = 0.2$
$W_{3new} = 0.8 + 1*(0-1)*1 = -0.2$

Consider the following set of input training vectors & the initial weight vector.

$$x_1 = \begin{bmatrix} 1 \\ -2 \\ 0 \\ -1 \end{bmatrix} \quad x_2 = \begin{bmatrix} 0 \\ 1.5 \\ -0.5 \\ -1 \end{bmatrix} \quad x_3 = \begin{bmatrix} -1 \\ 1 \\ 0.5 \\ -1 \end{bmatrix} \quad w = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0.5 \end{bmatrix}$$

The learning constant **c = 0.1**
The teacher's responses for x_1, x_2, x_3 are
d_1 = -1, d_2 = -1, d_3 = 1.

Train the perceptron using Perceptron Learning rule. The activation function is sgn.

$$net_1 = [1 \quad -1 \quad 0 \quad 0.5] \begin{bmatrix} 1 \\ -2 \\ 0 \\ -1 \end{bmatrix} = 2.5$$ $O_1 = sgn(2.5) = 1$ & $d_1 = -1$

$\Delta wi = \eta\,(t\text{-}o)\,xi$ and $w1 = w + \Delta w1$

$$w1 = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0.5 \end{bmatrix} + 0.1(-1-1)\begin{bmatrix} 1 \\ -2 \\ 0 \\ -1 \end{bmatrix} = \begin{bmatrix} 0.8 \\ -0.6 \\ 0 \\ 0.7 \end{bmatrix}$$

$$net_2 = [0.8 \quad -0.6 \quad 0 \quad 0.7] \begin{bmatrix} 0 \\ 1.5 \\ -0.5 \\ -1 \end{bmatrix} = -1.6$$

$$net_3 = [0.8 \quad -0.6 \quad 0 \quad 0.7] \begin{bmatrix} -1 \\ 1 \\ 0.5 \\ -1 \end{bmatrix} = -2.1$$

$$w3 = \begin{bmatrix} 0.8 \\ -0.6 \\ 0 \\ 0.7 \end{bmatrix} + 0.1(1+1)\begin{bmatrix} -1 \\ 1 \\ 0.5 \\ -1 \end{bmatrix} = \begin{bmatrix} 0.6 \\ -0.4 \\ 0.1 \\ 0.5 \end{bmatrix}$$

Will correction be required?

No correction, since $o_2 = sgn(-1.6) = -1 = d_2$

Will correction be required?

Yes, since $o_3 = sgn(-2.1) = -1$ while $d_2 = 1$

PERCEPTRON TRAINING RULE

Strength:

☺ If the data is linearly separable and η is set to a sufficiently small value, it will converge to a hypothesis that classifies all training data correctly in a finite number of iterations

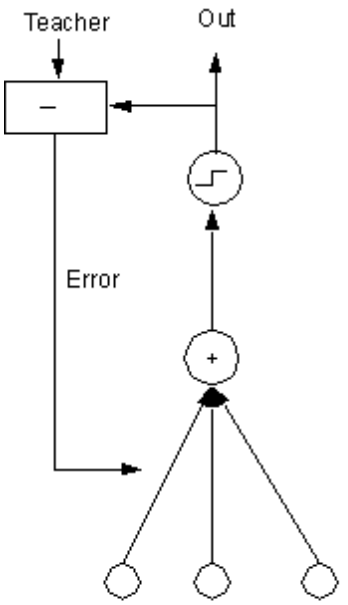
Weakness:

☹ If the data is not linearly separable, it will not converge

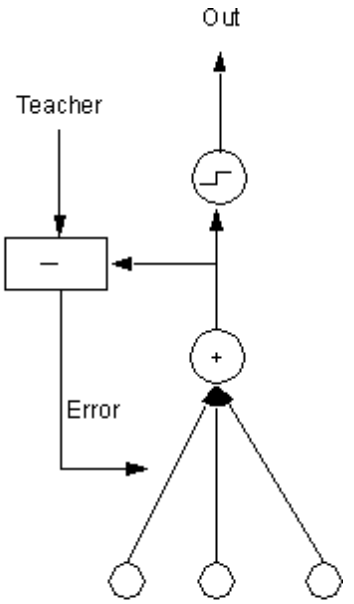
DELTA RULE

- Developed by Widrow & Hoff, aks Least Mean Square (LMS)
- 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.
- The **Delta rule**, is designed to overcome this difficulty.
- The key idea of delta rule: to **use gradient descent to search the space of possible weight vector to find the weights that best fit the training examples.**
- The delta learning rule is only valid for continuous activation functions and in the supervised training mode.

LINEAR UNITS



Perceptron Learning



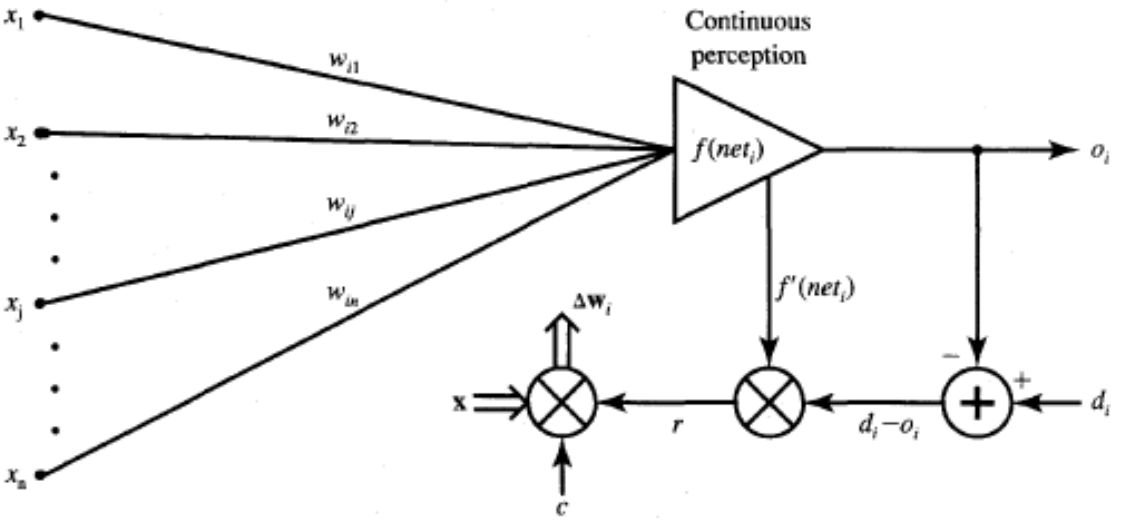
Delta Rule

- Linear units are like perceptrons, but the output is used directly (not thresholded to 1 or -1)
- A linear unit can be thought of as an unthresholded perceptron

The learning signal for this rule is called delta and is defined as follows

$$r \triangleq [d_i - f(\mathbf{w}_i^t \mathbf{x})] f'(\mathbf{w}_i^t \mathbf{x})$$

The term $f'(w_i x)$ is the derivative of the activation function f (net) computed for net = $w_i x$.



DERIVATION OF LEARNING RULE

- This learning rule can be derived from the condition of least squared error between o_i and d_i.
- Calculating the gradient vector with respect to w_i of the squared error defined as

$$E \triangleq \frac{1}{2}(d_i - o_i)^2$$

- This is equivalent to

$$E = \frac{1}{2}\left[d_i - f(\mathbf{w}_i^t\mathbf{x})\right]^2$$

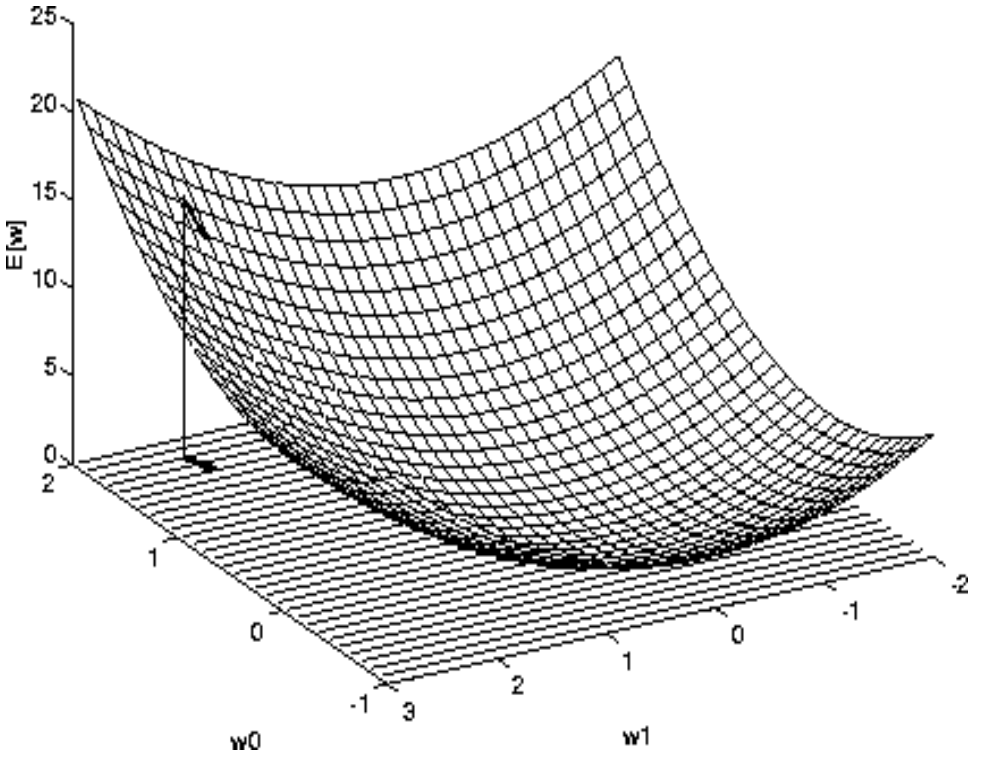
- We obtain the error gradient vector value as

$$\nabla E = -(d_i - o_i)f'(\mathbf{w}_i^t\mathbf{x})\mathbf{x}$$

BETWEEN THE LINES - HYPOTHESIS SPACE

- To understand the gradient descent algorithm, it is helpful to visualize the entire space of possible weight vectors and their associated E values, as illustrated on the next slide.
  - Here the axes w_0,w_1 represents possible values for the two weights of a simple linear unit. The w_0,w_1 plane represents the entire hypothesis space.
  - The vertical axis indicates the error E relative to some fixed set of training examples. The error surface shown in the figure summarizes the desirability of every weight vector in the hypothesis space.
- For linear units, this **error surface** must be **parabolic** with a single global minimum. And we desire a weight vector with this minimum.

The error surface



How can we calculate the direction of steepest descent along the error surface?

This direction can be found by computing the derivative of E w.r.t. each component of the vector w.

- The components of the gradient vector are

$$\frac{\partial E}{\partial w_{ij}} = -(d_i - o_i) f'(\mathbf{w}_i^t \mathbf{x}) x_j, \quad \text{for } j = 1, 2, \ldots, n$$

- Since the minimization of the error requires the weight changes to be in the negative gradient direction, we take

$$\Delta \mathbf{w}_i = -\eta \nabla E$$

where η is a positive constant.
So,

$$\Delta \mathbf{w}_i = \eta (d_i - o_i) f'(net_i) \mathbf{x}$$

- For the single weight, the adjustment becomes

$$\Delta w_{ij} = \eta (d_i - o_i) f'(\text{net}_i) x_j, \quad \text{for } j = 1, 2, \ldots, n$$

Note: The weight adjustment is computed based on minimization of the squared error.

FINDING $f'(net_i)$

Let us express f '(net) in terms of continuous perceptron output using the following bipolar continuous activation function

$$f(net) = \frac{2}{1+e^{(-net)}} - 1$$

f '(net) = ?

$$f'(net) = \frac{2e^{(-net)}}{\left[1+e^{(-net)}\right]^2}$$

Let $o = f(net)$. Then,

$$f'(net) = \frac{2e^{(-net)}}{\left[1+e^{(-net)}\right]^2} = \frac{1}{2}\left(1-o^2\right)$$

EXAMPLE

Consider the following set of input training vectors & the initial weight vector.

$$x_1 = \begin{bmatrix} 1 \\ -2 \\ 0 \\ -1 \end{bmatrix} \quad x_2 = \begin{bmatrix} 0 \\ 1.5 \\ -0.5 \\ -1 \end{bmatrix} \quad x_3 = \begin{bmatrix} -1 \\ 1 \\ 0.5 \\ -1 \end{bmatrix} \quad w = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0.5 \end{bmatrix}$$

The learning constant **c = 0.1**

The teacher's responses for x_1, x_2, x_3 are

d_1 = -1, d_2 = -1, d_3 = 1.

Train the perceptron using Delta rule.

$$f(x) = \frac{2}{1+e^{-x}} - 1$$

$$\text{net}_1 = [1 \quad -1 \quad 0 \quad 0.5] \begin{bmatrix} 1 \\ -2 \\ 0 \\ -1 \end{bmatrix} = 2.5$$

O_1 = ?

$$o_1 = \frac{2}{1 + e^{-2.5}} - 1 = 0.848$$

f'= ?

$$f' = \tfrac{1}{2}(1 - o^2) = 0.140$$

$$\Delta \mathbf{w}_i = \eta(d_i - o_i) f'(net_i)\mathbf{x}$$

$$w1 = \begin{bmatrix} 1 \\ -1 \\ 0 \\ 0.5 \end{bmatrix} + 0.1(-1 - 0.848)0.140 \begin{bmatrix} 1 \\ -2 \\ 0 \\ -1 \end{bmatrix} = \begin{bmatrix} 0.974 \\ -0.948 \\ 0 \\ 0.526 \end{bmatrix}$$

$net_2 = -1.948$ $o_2 = -0.75$ $f' = 0.218$

W2 = [0.974 -0.956 0.002 0.531]

$net_3 = -2.46$ $o_3 = -0.842$ $f' = 0.145$

W3 = [0.947 -0.929 0.016 0.505]

HOMEWORK

Determine the weights of a network with 4 input and 2 output units using

(a) Perceptron learning law and

(b) Delta learning law with **$f(x) = 1/(1 + e^{-x})$ for the following input output pairs:**

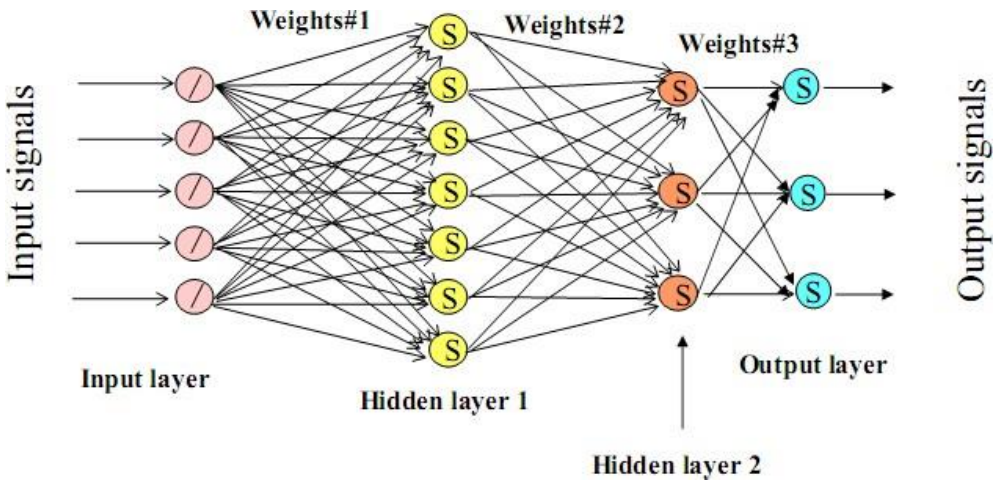
Input: [1100] [1001] [0011] [0110]

Output: [11] [10] [01] [00]

Take $f' = \frac{1}{2}(1 - o^2)$

MULTILAYER PERCEPTRON

- The single-layer networks suffer from the disadvantage that they are only able to solve linearly separable classification problems.
- The MLP is a hierarchical structure of several perceptrons, & overcomes the disadvantages of these single-layer networks.
- However the computational effort needed for finding the correct combination of weights increases substantially when more parameters and more complicated topologies are considered.

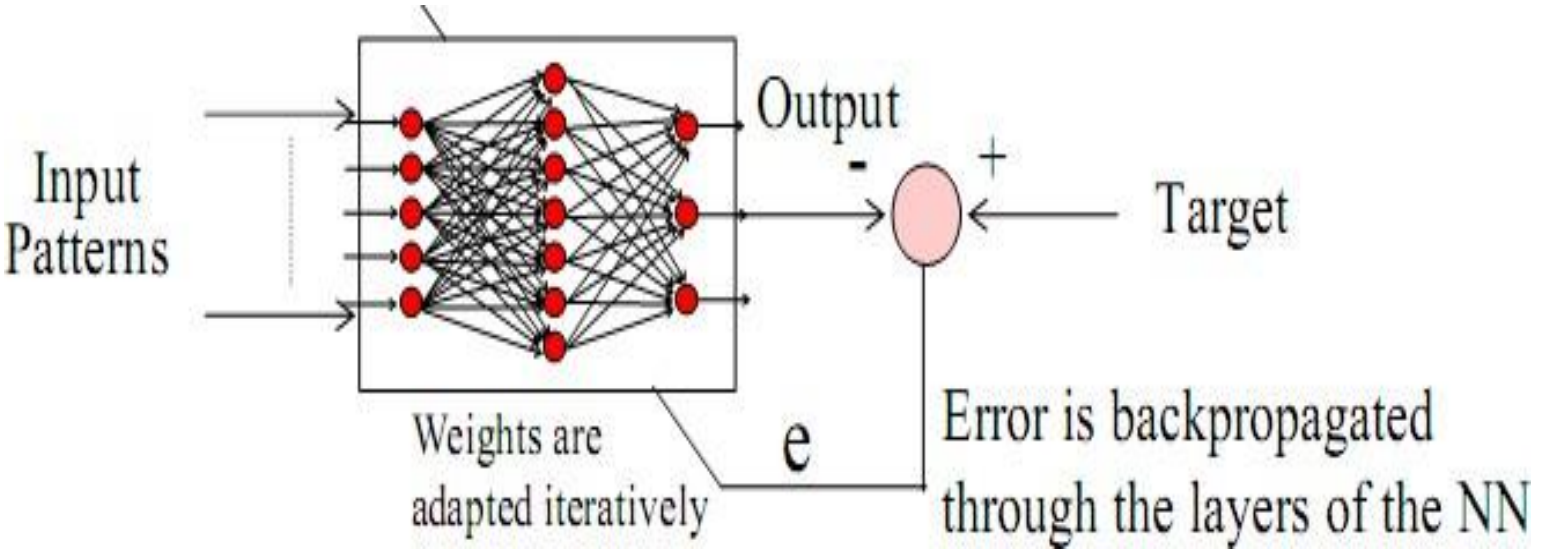


ARCHITECTURE OF MULTILAYER PERCEPTRON

- No connections within a layer
- No direct connections between input and output layers
- Fully connected between layers
- No. of output units need not equal no. of input units
- No. of hidden units per layer can be more or less than input or output units
- Each unit is a perceptron
- The training algorithm for MLP requires differentiable, continuous nonlinear activation functions. E.g the sigmoid, or logistic function: $a = \sigma(n) = 1/(1 + e^{-cn})$ where n is the sum of products from the weights w_i and the inputs x_i. c is a constant.

THE ERROR FUNCTION

- Consider a feed-forward network with n input and m output units. It can consist of any number of hidden units.
- We want to minimize the error function of the network, defined as $$E = \frac{1}{2}\sum_{i=1}^{p} \|\mathbf{o}_i - \mathbf{t}_i\|^2$$



BACK PROPAGATION ALGORITHM

- The BP algorithm is used to find a local minimum of the error function.
- The network is **initialized with randomly chosen weights.**
- **The gradient of the error function is computed and used to correct the initial weights.**
- E is a continuous and differentiable function of the weights $w_1, w_2, \ldots, w_l$ in the network.
- We can thus minimize E by using an iterative process of gradient descent, for which we need to calculate the gradient

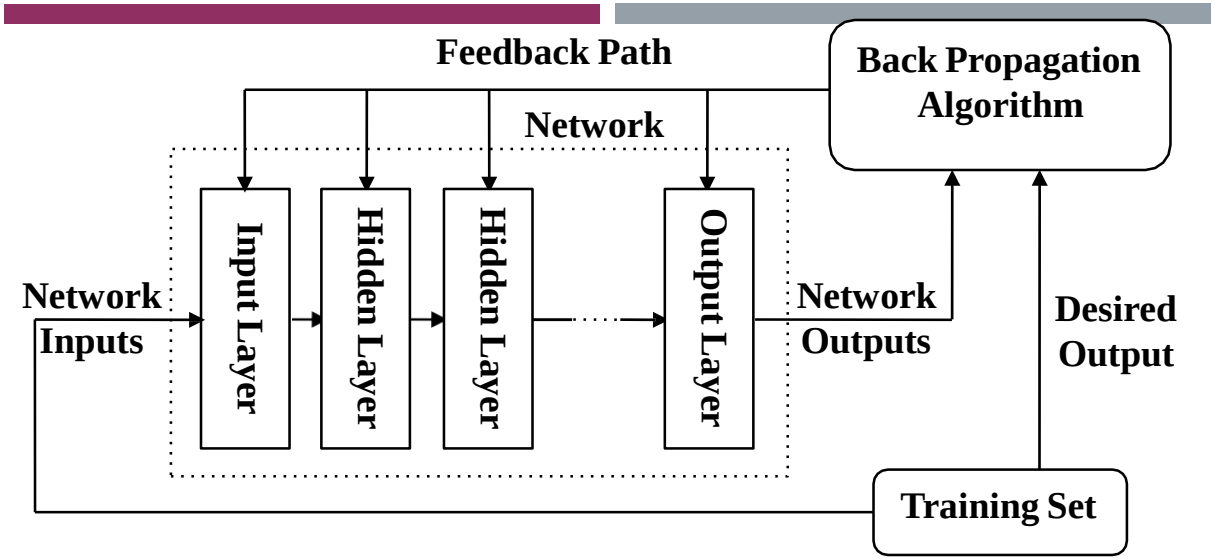
$$\nabla E = \left(\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_\ell}\right)$$

- Each weight is updated using the increment

$$\Delta w_i = -\gamma \frac{\partial E}{\partial w_i} \quad \text{for } i = 1, \ldots, \ell,$$

The back propagation algorithm includes two passes through the network:

- *forward pass* and
- *backward pass*.



How to find a method for efficiently calculating the gradient of a one-dimensional network function according to the weights of the network?

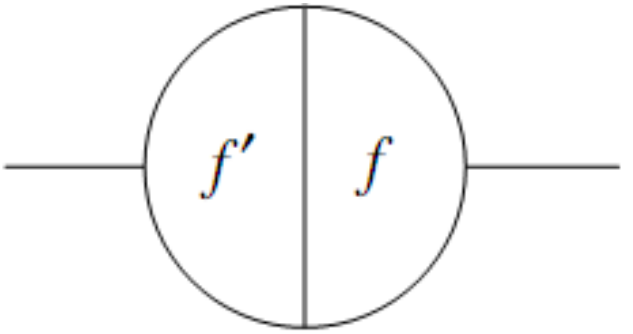
Network is equivalent to a complex chain of function compositions

Nodes of the network are given a composite structure

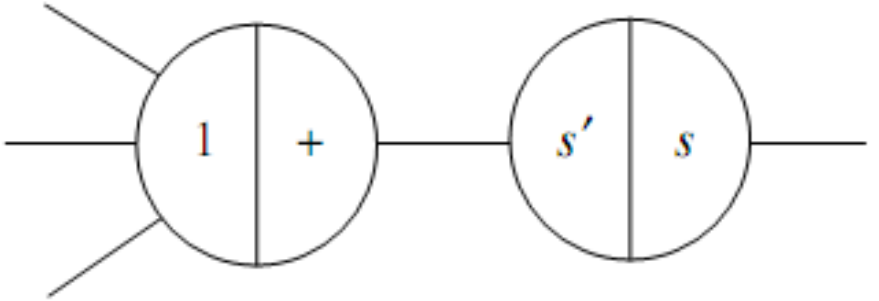
B-DIAGRAM

Each node now consists of a left and a right side

- The right side computes the primitive function associated with the node,
- The left side computes the derivative of this primitive function for the same input.



. The two sides of a computing unit



Separation of integration and activation function

The integration function can be separated from the activation function by splitting each node into two parts.

- The first node computes the sum of the incoming inputs,
- The second one the activation function s.

- The derivative of s is s' and
- the partial derivative of the sum of n arguments with respect to any one of them is just 1.

This separation simplifies the discussion, as we only have to think of a single function which is being computed at each node and not of two.

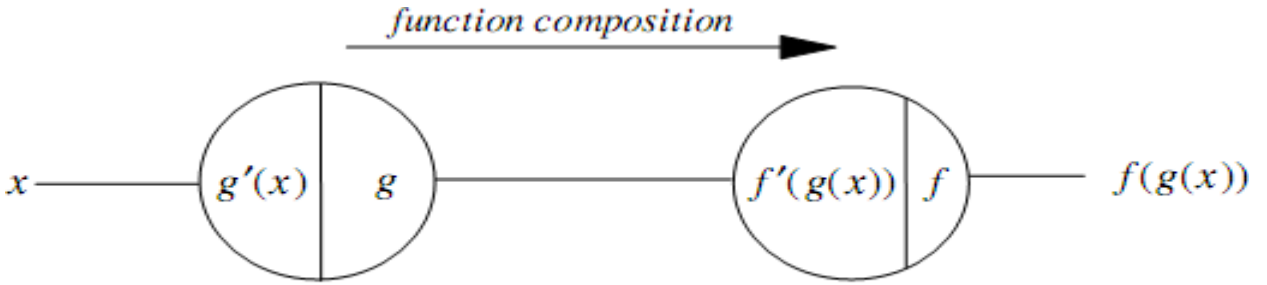
THE TWO PHASES OF THE LEARNING ALGORITHM

1. The Feed-forward step

- A training input pattern is presented to the network input layer. The network propagates the input pattern from layer to layer until the output pattern is generated by the output layer.
- Information comes from the left and each unit evaluates its primitive function f in its right side as well as the derivative f ’ in its left side.
- Both results are stored in the unit, but only the result from the right side is transmitted to the units connected to the right.

In the feed-forward step, **incoming information into a unit is used as the argument for the evaluation of the node's primitive function and its derivative.** In this step the network computes the composition of the functions f and g. The correct result of the function composition has been produced at the output unit and each unit has stored some information on its left side.

Network for the composition of two functions

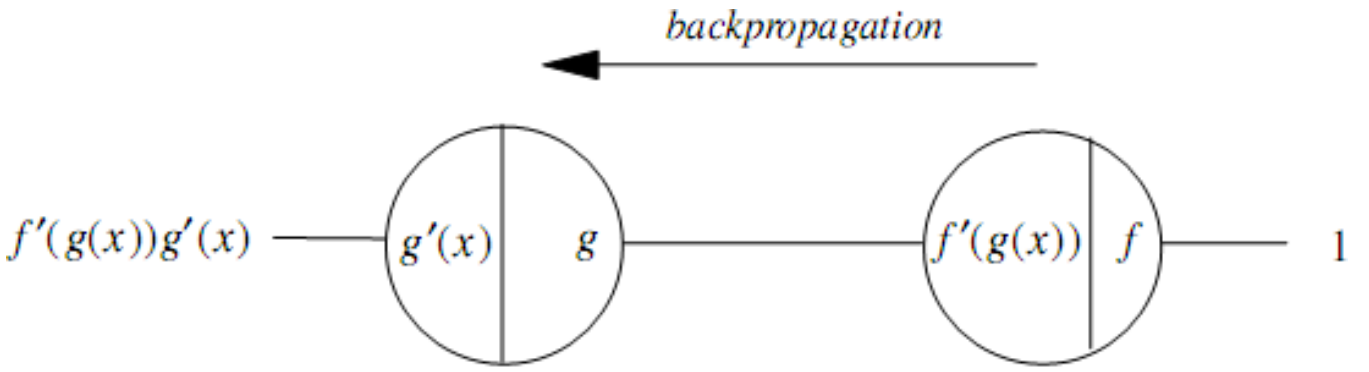


Result of the feed-forward step

2. The Backpropagation step

- If this pattern is different from the desired output, an error is calculated and then propagated backwards through the network from the output layer to the input layer.
- The stored results are now used.
- The weights are modified as the error is propagated.

The backpropagation step provides an implementation of the chain rule. Any sequence of function compositions can be evaluated in this way and its derivative can be obtained in the backpropagation step. We can think of the network as being used backwards with the input 1, whereby at each node the product with the value stored in the left side is computed.



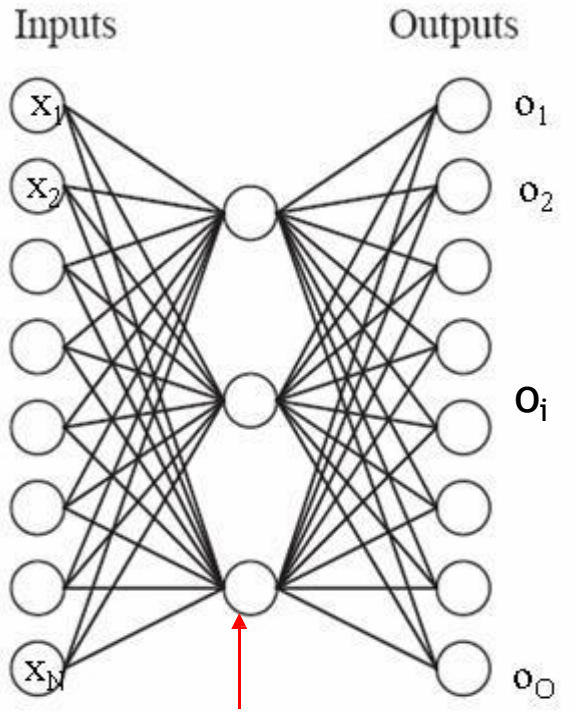
Result of the backpropagation step

TWO KINDS OF SIGNALS PASS THROUGH THESE NETWORKS:

-*function signals*: the input examples propagated through the hidden units and processed by their transfer functions emerge as outputs;

- *error signals*: the errors at the output nodes are propagated backward layer-by-layer through the network so that each node returns its error back to the nodes in the previous hidden layer.

Learning Algorithms for MLP



How to compute the errors for the hidden units?

Goal: minimize sum squared errors

$Err_1=y_1-o_1$

$Err_2=y_2-o_2$

$$E = \frac{1}{2}\sum_i (y_i - o_i)^2$$

$Err_i=y_i-o_i$

$$o_i = g\left(\sum_h w_{h,i}\, g\left(\sum_j w_{j,h}\, x_j\right)\right)$$

parameterized function of inputs: weights are the parameters of the function.

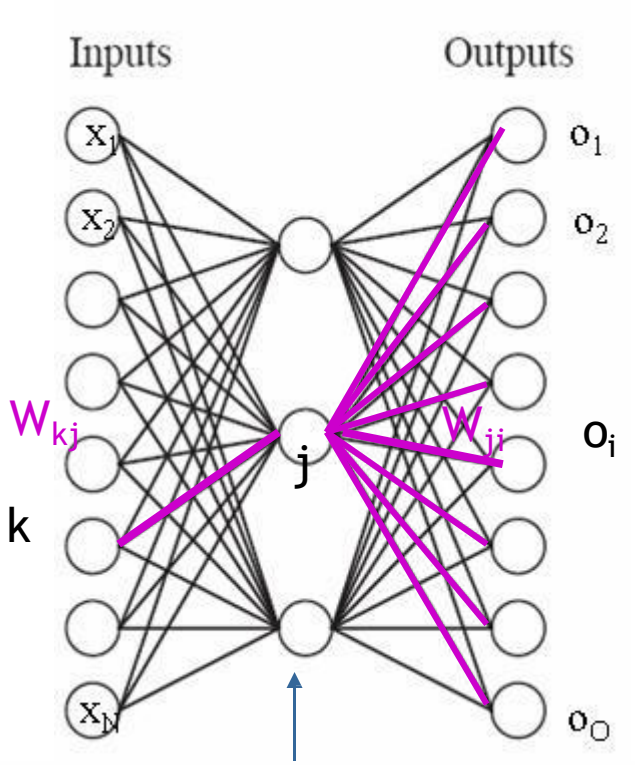
$Err_o=y_o-o_o$

Clear error at the output layer

We can **back-propagate** the error from the output layer to the hidden layers.

The back-propagation process emerges directly from a derivation of the overall error gradient.

Back Propagation Learning Algorithm for MLP



Perceptron update:

$Err_i = y_i - o_i$

Output layer weight update (similar to perceptron)

Hidden layer: back-propagate the error from the output layer:

Hidden node j is "responsible" for some fraction of the error i in each of the output nodes to which it connects

→ depending on the strength of the connection between the hidden node and the output node i.

THE BP ALGORITHM – WHAT IT ACTUALLY DOES!!!

- Like perceptron learning, BP attempts to reduce the errors between the output of the network and the desired result.
- However, assigning blame for errors to hidden nodes, is not so straightforward. The error of the output nodes must be propagated back through the hidden nodes.
- The contribution that a hidden node makes to an output node is related to the strength of the weight on the link between the two nodes and the level of activation of the hidden node when the output node was given the wrong level of activation.
- This can be used to estimate the error value for a hidden node in the penultimate layer, and that can, in turn, be used in making error estimates for earlier layers.

WEIGHT CHANGE EQUATION

The basic algorithm can be summed up in the following equation (the *delta rule*) for the change to the weight w_{ij} from node i to node j:

Weight change		learning rate		local gradient		input signal to node j
Δw_{ij}	=	η	×	δ_j	×	y_i

The local gradient δ_j is defined as follows:

- **Node *j* is an output node**

  δ_j is the product of f'(net_j) and the error signal e_j, where f(_) is the logistic function and net_j is the total input to node j (i.e. $\Sigma_i w_{ij} y_i$), and e_j is the error signal for node j (i.e. the difference between the desired output and the actual output);

- **Node *j* is a hidden node**

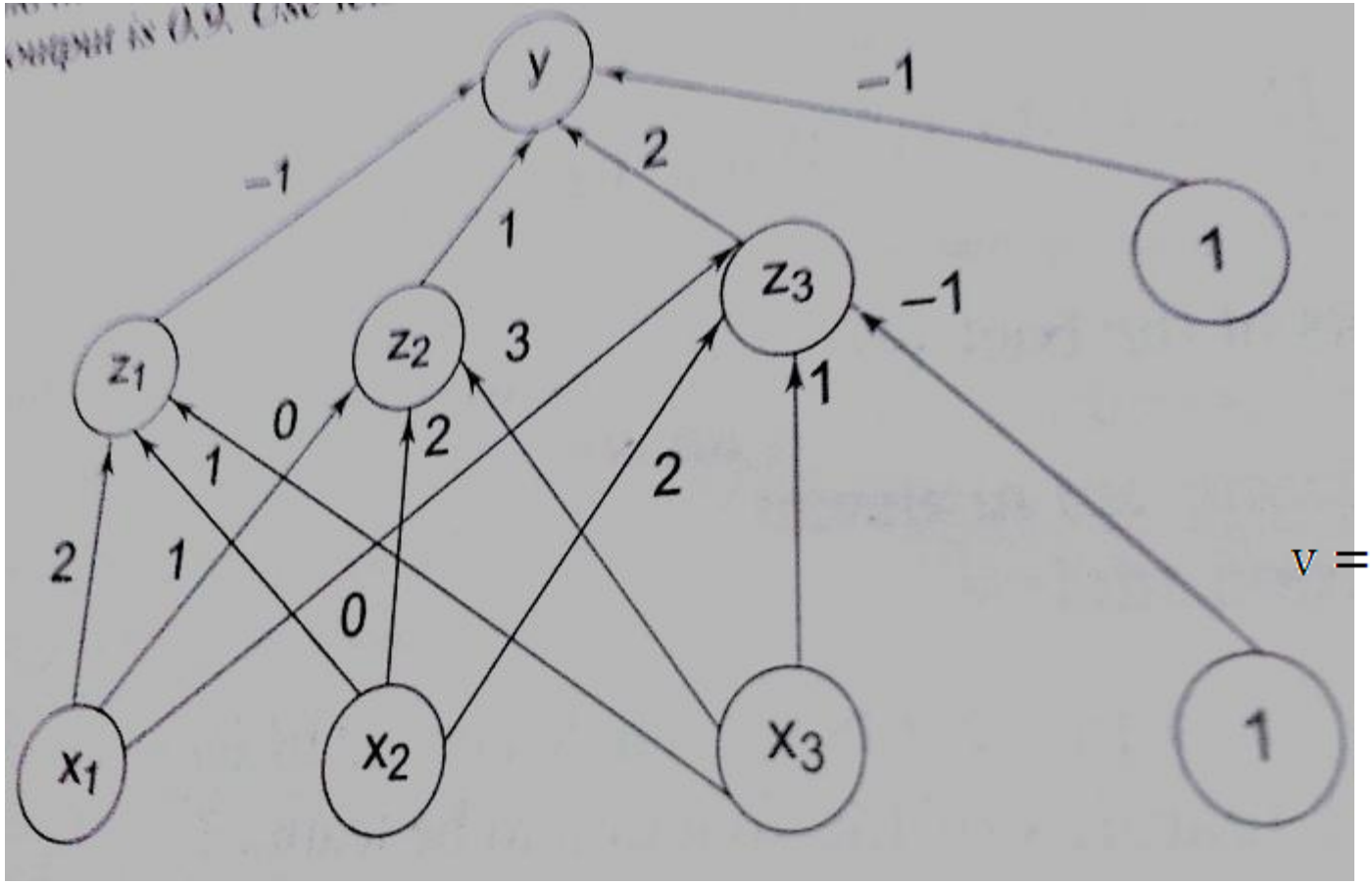
  δ_j is the product of f'(net_j) and the weighted sum of the δ's computed for the nodes in the next hidden or output layer that are connected to node j.

STOPPING CRITERION

Two commonly used stopping criteria are:

- stop after a certain number of runs through all the training data (each run through all the training data is called an *epoch*);
- stop when the total sum-squared error reaches some low level. By total sum-squared error we mean $\Sigma_p \Sigma_i e_i^2$ where p ranges over all of the training patterns and i ranges over all of the output units.

Find the new weights when the following network is presented the input pattern [0.6 0.8 0]. The target output is 0.9. Use learning rate α = 0.3 & binary sigmoid activation function.



Let v be the weight matrix of I layer & w of the II layer

$$v = \begin{bmatrix} & Z1 & Z2 & Z3 \\ X1 & 2 & 1 & 0 \\ X2 & 1 & 2 & 2 \\ X3 & 0 & 3 & 1 \end{bmatrix}$$

Feed Forward Stage

Step 1 Find the inputs at each of the hidden units.

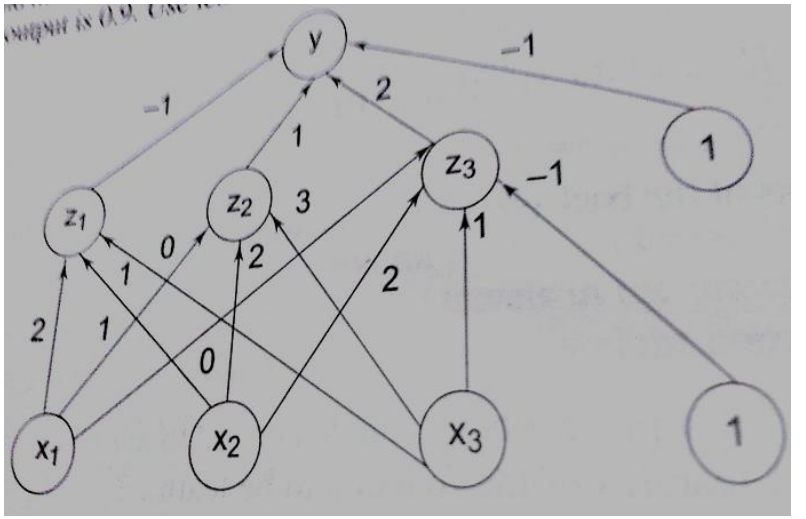
$net_{z1} = 0 + 0.6 \times 2 + 0.8 \times 1 + 0 \times 0 = 2$

So, we get

$net_{z1} = 2$

$net_{z2} = 2.2$

$net_{z3} = 0.6$ (since bias = -1)



Step 2 **Find the output of each of the hidden unit.**

$$z_1 = f(Z_{in1}) = \frac{1}{1 + e^{-2}} = 0.8808$$

So, we get

$o_{z1} = 0.8808$ $o_{z2} = 0.9002$ $o_{z3} = 0.646$

Step 3 **Find the input to output unit Y.**

$net_y = -1 + 0.8808 \times -1 + 0.9002 \times 1 + 0.646 \times 2 = 0.3114$

Step 4 **Find the output of the output unit.**

$o_y = 0.5772$

Back propagation of Error

Step 5 Find the gradient at the output unit Y.

$$\delta_1 = (t_1 - o_y)\, f'(net_y)$$

We know that for a binary sigmoid function

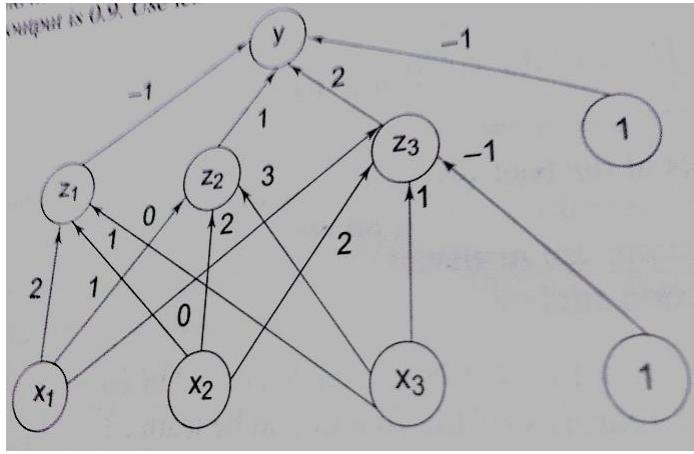
$$f'(x) = f(x)(1 - f(x))$$

So,

$$f'(net_y) = 0.5772\ (1 - 0.5772) = 0.244$$

$$\delta_1 = (0.9 - 0.5772)\ 0.244$$

$$\delta_1 = 0.0788$$



Step 6 Find the gradient at the hidden units.

Remember: If node j is a hidden node, then δ_j is the product of f'(net_j) and the weighted sum of the δ's computed for the nodes in the next hidden or output layer that are connected to node j.

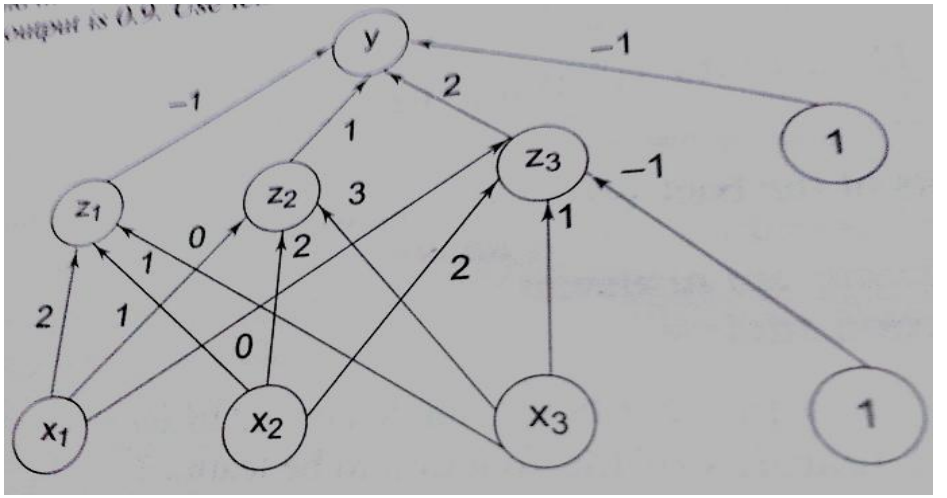
$$\delta_{z1} = \delta_1 \, w_{11} \, f'(net_{z1})$$

$$\delta_{z1} = 0.0788 \times -1 \times 0.8808 \times (1 - 0.8808)$$

$$\delta_{z1} = -0.0083$$

$$\delta_{z2} = 0.0071$$

$$\delta_{z3} = 0.0361$$



Weight Updation

Step 7 Weight updation at the hidden units.

Weight change = learning rate × local gradient × input signal to node j

$$\Delta w_{ij} = \eta \times \delta_j \times y_i$$

$\Delta w_{11} = \alpha\, \delta_1\, z_1 = 0.3 \times 0.0788 \times 0.8808 = 0.0208$

$\Delta w_{21} = \alpha\, \delta_1\, z_2 = 0.3 \times 0.0788 \times 0.9002 = 0.0212$

$\Delta w_{31} = \alpha\, \delta_1\, z_3 = 0.3 \times 0.0788 \times 0.6460 = 0.0153$

$\Delta v_{11} = \alpha\, \delta_{z1}\, x_1 = 0.3 \times -0.0083 \times 0.6 = -0.0015$

$\Delta v_{12} = \alpha\, \delta_{z2}\, x_1 = 0.3 \times 0.0071 \times 0.6 = 0.0013$

$\Delta v_{13} = \alpha\, \delta_{z3}\, x_1 = 0.3 \times 0.0361 \times 0.6 = 0.0065$

$\Delta v_{21} = \alpha\, \delta_{z1}\, x_2 = 0.3 \times -0.0083 \times 0.8 = -0.002$

$\Delta v_{22} = \alpha\, \delta_{z2}\, x_2 = 0.3 \times 0.0071 \times 0.8 = 0.0017$

$\Delta v_{23} = \alpha\, \delta_{z3}\, x_2 = 0.3 \times 0.0361 \times 0.8 = 0.0087$

$\Delta v_{31} = \alpha\, \delta_{z1}\, x_3 = 0.3 \times -0.0083 \times 0.0 = 0.0$

$\Delta v_{32} = \alpha\, \delta_{z2}\, x_3 = 0.3 \times 0.0071 \times 0.0 = 0.0$

$\Delta v_{33} = \alpha\, \delta_{z3}\, x_3 = 0.3 \times 0.0361 \times 0.0 = 0.0$

w_{11} (new)= w_{11}(old) + Δw_{11} = -1 + 0.0208 = - 0.9792

w_{21} (new)= 1.0212

w_{31} (new)= 2.0153

v_{11}(new) = v_{11}(old) + Δv_{11} = 2 - 0.0015 = 1.9985

v_{12}(new) = 1.0013

v_{13}(new) = 0.0065

v_{21} (new)= 0.998

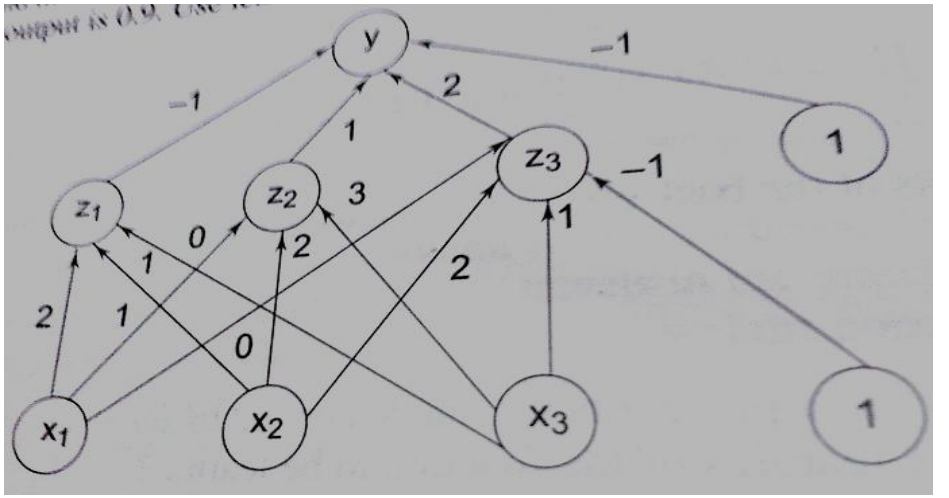
v_{22} (new)= 2.0017

v_{23} (new)= 2.0087

v_{31} (new)= 0

v_{32} (new)= 3

v_{33} (new)= 1



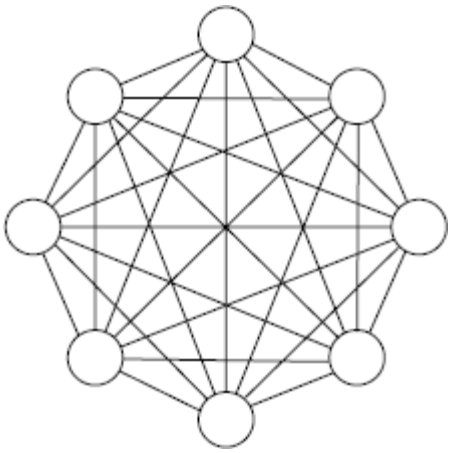
RECURRENT NEURAL NETWORKS

Depending on the density of feedback connections:

- **Total recurrent networks** (Hopfield model)
- **Partial recurrent networks**
  - With contextual units (Elman model, Jordan model)
  - Cellular networks (Chua model)

Hopfield Network

- It consists of a set K *of completely linked neurons* with binary activation, with the weights being symmetric b/w the individual neurons & without any neuron being *directly connected to* itself.
- The *state of* K *neurons with two possible states* $\in \{-1, 1\}$ *can* be described by a string $x \in \{-1,1\}^K$.
- The i/p of a Hopfield n/w is binary string $x \in \{-1, 1\}^K$ *that initializes the state of the n/w. After* the convergence of the n/w, the o/p is the binary string $y \in \{-1, 1\}^K$ *generated* from the new n/w state.
- All neurons are both i/p & o/p neurons.
- This network is capable of associating its i/p with one of the patterns stored in network's memory



- Neuron is characterized by its state s_i
- The o/p of the neuron is the function of the neuron's state: $y_i=f(s_i)$
- The applied function f is soft limiter which effectively limits the output to the [-1,1] range
- Neuron initialization
  - When an i/p vector **x** arrives to the n/w, the state of i^{th} neuron, i=1,…,N is initialized by the value of the i^{th} input: $s_i=x_i$
- Subsequently, while there is any change

$$s_i = \sum_{j \neq i} w_{i,j} y_j$$

$$y_i = f(s_i)$$

- Output of the network is vector $\mathbf{y}=[y_1 \ldots y_n]$ consisting of neuron outputs when the network stabilizes

UPDATING THE HOPFIELD NETWORK

Updates in the Hopfield network can be performed in two different ways:

- **Asynchronous:**

Only one unit is updated at a time. This unit can be picked at random, or a pre-defined order can be imposed from the very beginning.

- **Synchronous:**

All units are updated at the same time. This requires a central clock to the system in order to maintain synchronization. This method is less realistic, since biological or physical systems lack a global clock that keeps track of time.

The neuron i modifies its state s_i according to the deterministic rule

$$x_i(t+1) = \mathrm{sgn}\left(\sum_{j=1}^{n} w_{ij} x_j(t) - \theta_i\right)$$

- Individual units preserve their own states until they are selected for an update (incase of asynchronous update).

EXAMPLE

Let

$$v(0) = \begin{bmatrix} 1 \\ 1 \\ -1 \\ -1 \end{bmatrix}; \quad W = \begin{bmatrix} 0 & 1 & 1 & -1 \\ 1 & 0 & 1 & -1 \\ 1 & 1 & 0 & -1 \\ -1 & -1 & -1 & 0 \end{bmatrix}; \quad T \equiv 0$$

What will be new states of the neurons at time 1?

$$v(1) = \text{sgn}[Wv(0)] = \text{sgn}\{\begin{bmatrix} 1 \\ 1 \\ 3 \\ -1 \end{bmatrix}\} = \begin{bmatrix} 1 \\ 1 \\ 1 \\ -1 \end{bmatrix}$$

CONVERGENCE

- The subsequent computation of the network will occur until the network does not stabilize
- The network will stabilize when all the states of the neurons stay the same
- IMPORTANT PROPERTY:
  - Hopfield's network will ALWAYS stabilize after finite time if asynchronous updates are done.

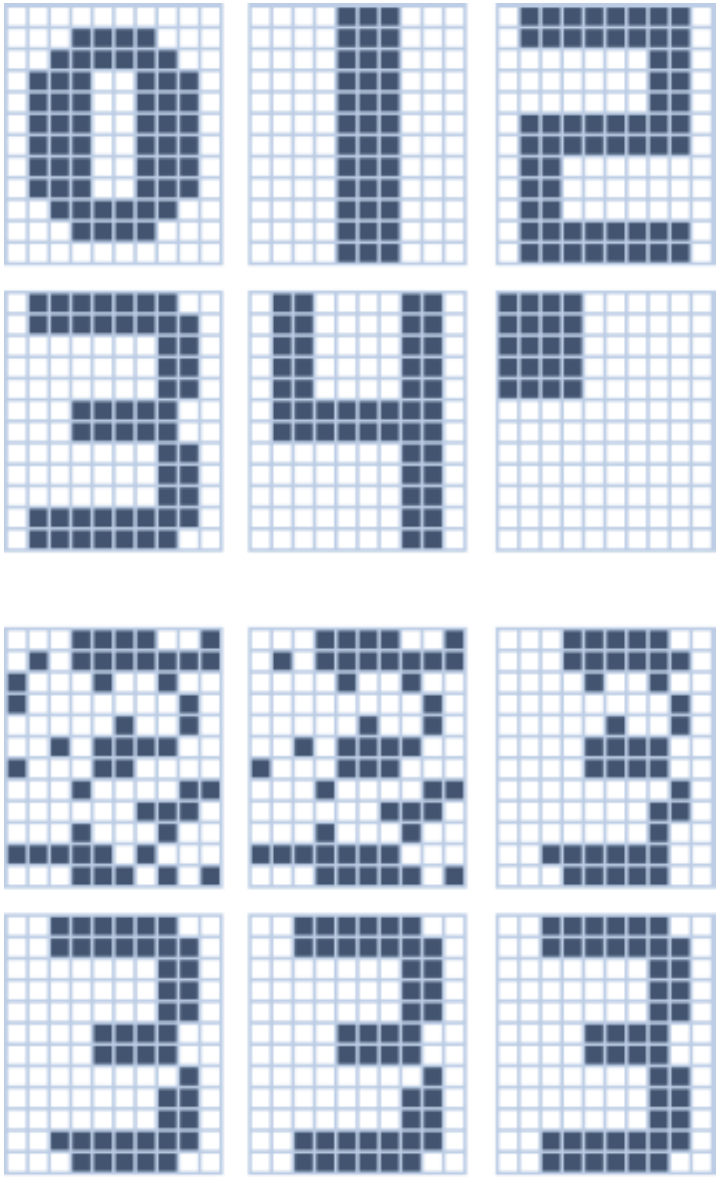


Illustration of the convergence of a Hopfield network.

- Each of the pictures has 10 × 12 = 120 binary pixels.
- In the Hopfield network each pixel corresponds to one neuron.
- The upper illustration shows the training samples
- The lower shows the convergence of a heavily noisy 3 to the corresponding training sample.

MAIN PROBLEMS WITH HOPFIELD NETWORK

- **What is "close"**
  - The output associated to input is one of stored vectors "closest" to the input
  - However, the notion of "closeness" is hard encoded in the weight matrix and we cannot have influence on it
- **Spurious states**
  - Assume that we memorize M different patterns into a Hopfield network
  - The network may have more than M stable states
  - Hence the output may be NONE of the vectors that are memorized in the network
  - In other words: among the offered M choices, we could not decide

Hopfield Network Algorithm

1. Assign connection weights

$$w_{ij} = \begin{cases} \frac{1}{N}\sum_{s=0}^{M-1} x_i^s x_j^s & i \neq j \\ 0 & i = j, \end{cases}$$

Hebbian Learning

where w_{ij} is the connection weight between node i and node j, and x_i^s is element i of the exemplar pattern s, and is either $+1$ or -1.
There are M patterns, from 0 to $M-1$, in total.

2. Initialise with unknown pattern

$$\mu_i(0) = x_i \qquad 0 \leq i \leq N-1$$

Probe pattern

where $\mu_i(t)$ is the output of node i at time t.

3. Iterate until convergence

$$\mu_i(t+1) = f_h\left[\sum_{i=0}^{N-1} w_{ij}\mu_j(t)\right] \qquad 0 \leq j \leq N-1$$

Dynamical evolution

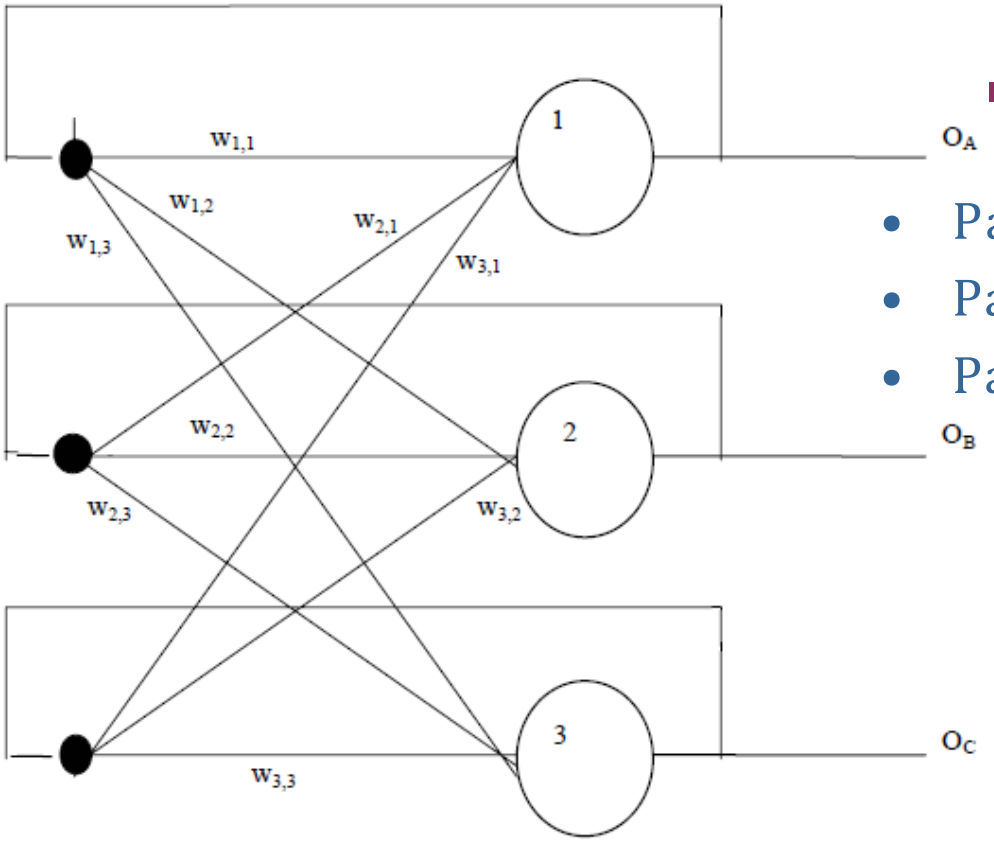
The function f_h is the hard-limiting non-linearity, the step function,
Repeat the iteration until the outputs from the node remain unchanged.

TRAINING – ONE SHOT METHOD

The method consists of a single calculation for each weight (so the whole network can be trained in "one pass").
The inputs are –1 and +1 (the neuron threshold is zero).

- Lets train this network for following patterns
- Pattern 1:-  ie $O_{a(1)}$=-1,$O_{b(1)}$=-1,$O_{c(1)}$=1
- Pattern 2:-  ie $O_{a(2)}$=1, $O_{b(2)}$=-1, $O_{c(3)}$=-1
- Pattern 3:-  ie $O_{a(3)}$=-1,$O_{b(3)}$=1, $O_{c(3)}$=1

How will the Hopfield network look?



- Pattern 1: $O_{a(1)}=-1, O_{b(1)}=-1, O_{c(1)}=1$
- Pattern 2: $O_{a(2)}=1, O_{b(2)}=-1, O_{c(3)}=-1$
- Pattern 3: $O_{a(3)}=-1, O_{b(3)}=1, O_{c(3)}=1$

$\frac{1}{3}$ *to be taken care of* !!

w1,1 = 0

w1,2 = OA(1) × OB(1) + OA(2) × OB(2) + OA(3) × OB(3) = (-1) × (-1) + 1 × (-1) + (-1) × 1 = -1

w1,3 = OA(1) × OC(1) + OA(2) × OC(2) + OA(3) × OC(3) = (-1) × 1 + 1 × (-1) + (-1) × 1 = -3

w2,2 = 0

w2,1 = OB(1) × OA(1) + OB(2) × OA(2) + OB(3) × OA(3) = (-1) × (-1) + (-1) × 1 + 1 × (-1) = -1

w2,3 = OB(1) × OC(1) + OB(2) × OC(2) + OB(3) × OC(3) = (-1) × 1 + (-1) × (-1) + 1 × 1 = 1

w3,3 = 0

w3,1 = OC(1) × OA(1) + OC(2) × OA(2) + OC(3) × OA(3) = 1 × (-1) + (-1) × 1 + 1 × (-1) = -3

w3,2 = OC(1) × OB(1) + OC(2) × OB(2) + OC(3) × OB(3) = 1 × (-1) + (-1) × (-1) + 1 × 1 = 1

The following can be proven….

- If the number M of memorized N-dimensional vectors is smaller than N/4ln(N)
- Then we can set the weights of the network as:

Used to simplify mathematics

$$\mathbf{W} = \frac{1}{N}\left(\sum_{m=1}^{M} \mathbf{x}_m^* \cdot \mathbf{x}_m^{*T} - MI\right)$$

Why this?

  - W is a symmetric matrix with **zeros on main diagonal**
- Such that the vectors $\mathbf{x}_m^*$ correspond to the stable states of the network

Verify for the previous Eg.

- Pattern 1: $[-1\ -1\ 1]^T$
- Pattern 2: $[1\ -1\ -1]^T$
- Pattern 3: $[-1\ 1\ 1]^T$

$$\mathbf{W} = \frac{1}{N}\left(\sum_{m=1}^{M} \mathbf{x}_m^* \cdot \mathbf{x}_m^{*T} - MI\right)$$

$$\begin{bmatrix} -1 \\ -1 \\ 1 \end{bmatrix} \begin{bmatrix} -1 & -1 & 1 \end{bmatrix} + \begin{bmatrix} -1 \\ 1 \\ -1 \end{bmatrix} \begin{bmatrix} 1 & -1 & -1 \end{bmatrix} + \begin{bmatrix} 1 \\ -1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & -1 & 1 \end{bmatrix} - 3 \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} 3 & -1 & -3 \\ -1 & 3 & 1 \\ -3 & 1 & 3 \end{bmatrix} - \begin{bmatrix} 3 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 3 \end{bmatrix} = \begin{bmatrix} 0 & -1 & -3 \\ -1 & 0 & 1 \\ -3 & 1 & 0 \end{bmatrix}$$

$\frac{1}{3}$ *to be taken care of* !!

WHY HOPFIELD NETWORK IS IMPORTANT?

- **Assume that we want to memorize M different N-dimensional vectors**
- **What does it mean "to memorize"?**
  - If vector x_m^* is on the input of the Hopfield's network - the same vector x_m^* will be on its output
  - If a vector "close" to vector x_m^* is on the input of the Hopfield's network - the vector x_m^* will be on its output

The Hopfield network memorizes by embedding knowledge into its weights

Consider a vector (1 0 1 1) to be stored in a net. Test a discrete Hopfield net with mistakes in the first & fourth components of the stored vector.

Step 1: Convert binary input to bipolar input.

(1 -1 1 1)

Step 2: Calculate the weights.

$$W = \begin{bmatrix} 1 \\ -1 \\ 1 \\ 1 \end{bmatrix} \begin{bmatrix} 1 & -1 & 1 & 1 \end{bmatrix} - \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$W = \begin{bmatrix} 0 & -1 & 1 & 1 \\ -1 & 0 & -1 & -1 \\ 1 & -1 & 0 & 1 \\ 1 & -1 & 1 & 0 \end{bmatrix}$$

Step 3: The input vector is

$$y = (-1\ -1\ 1\ -1).$$

Step 4: Choose unit 1 to update its activation.

$$s_i = \Sigma\, y_j w_{ij} = 1$$

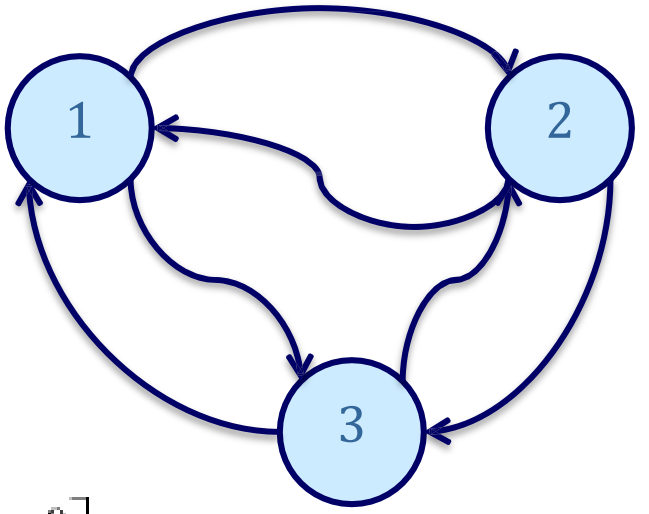
Step 5: $y = (1\ -1\ 1\ -1)$.

Choose unit 4 to update its activation

Continue in this manner

In how many possible states can this network be?

Design a network for the patterns (1, -1, 1) & (-1, 1, -1).



$$\mathbf{W} = \frac{1}{3}\begin{bmatrix} +1 \\ -1 \\ +1 \end{bmatrix}[+1, -1, +1] + \frac{1}{3}\begin{bmatrix} -1 \\ +1 \\ -1 \end{bmatrix}[-1, +1, -1] - \frac{2}{3}\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \frac{1}{3}\begin{bmatrix} 0 & -2 & +2 \\ -2 & 0 & -2 \\ +2 & -2 & 0 \end{bmatrix}$$

1/no. of neurons
Used to normalize the weights

Verify that these two are the stable states & the remaining 6 are unstable.

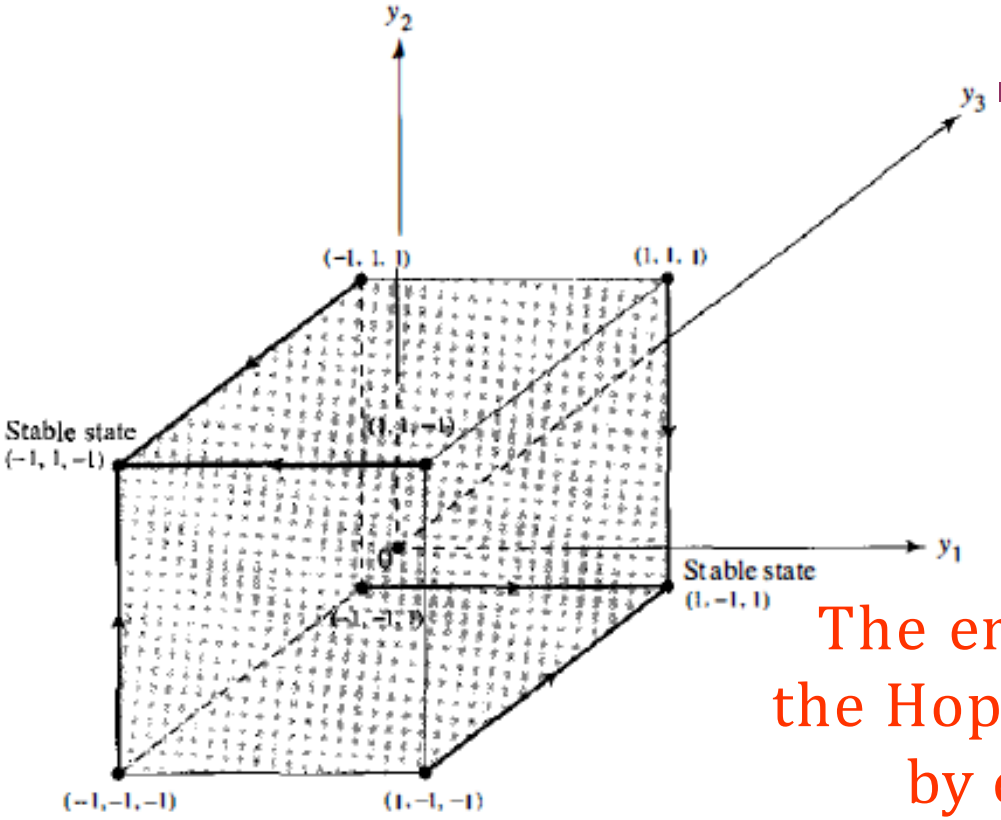


Diagram depicting the two stable states and flow of the network.

The error correcting capability of the Hopfield network is readily seen by examining the flow map

1. If the probe vector applied to the network equals (- 1 , - 1 , 1), (1 , 1 , 1) or (1 , - 1 , - 1), the resulting output is the fundamental memory (1 , - 1 , 1). Each of these values of the probe represents a single error, compared to the stored pattern.
2. If the probe vector equals (1, 1, -1). (- 1 , -1, - 1), or (- 1 , 1 , 1) the resulting network output is the fundamental memory (- 1 , 1, -1). Here again, each of these values of the probe represents a single error, compared to the stored pattern.

Building Innovative Systems

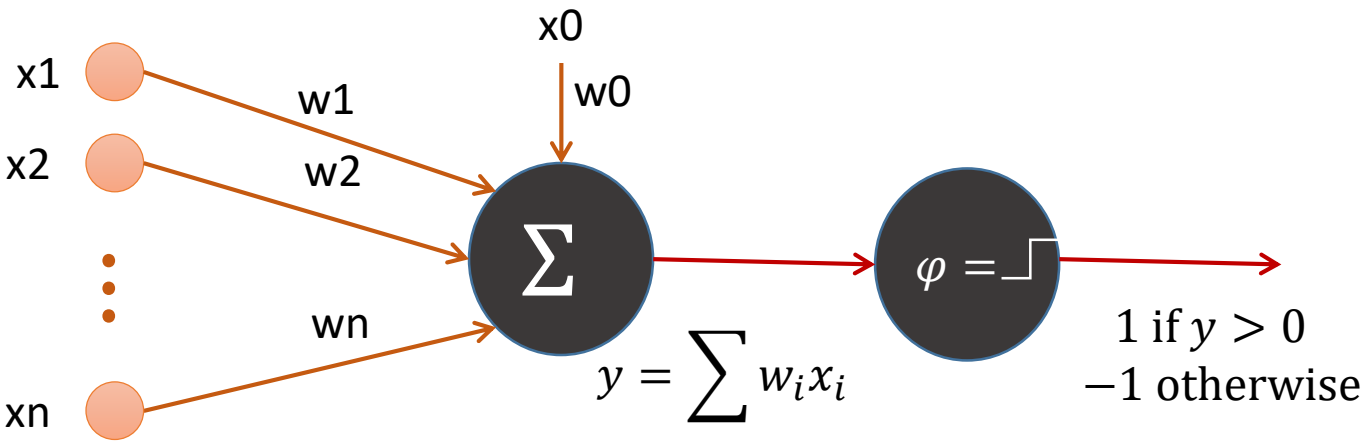
Topic: Gradient Descendent (Part-II)

Introduction

- Inspired by the human brain.
- Some NNs are models of biological neural networks
- Human brain contains a massively interconnected net of 10^{10}-10^{11} (10 billion) neurons (cortical cells)
  - Massive parallelism – large number of simple processing units
  - Connectionism – highly interconnected
  - Associative distributed memory
    - Pattern and strength of synaptic connections

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 training rule

Updates perceptron weights for a training ex as follows:

$$w_i = w_i + \Delta w_i$$

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

- 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

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

Linear neurons

- The neuron has a real-valued output which is a weighted sum of its inputs

$$\hat{y} = \sum_i w_i x_i = \mathbf{w}^T \mathbf{x}$$

- Define the error as the squared residuals summed over all training cases:

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

- Differentiate to get error derivatives for weights

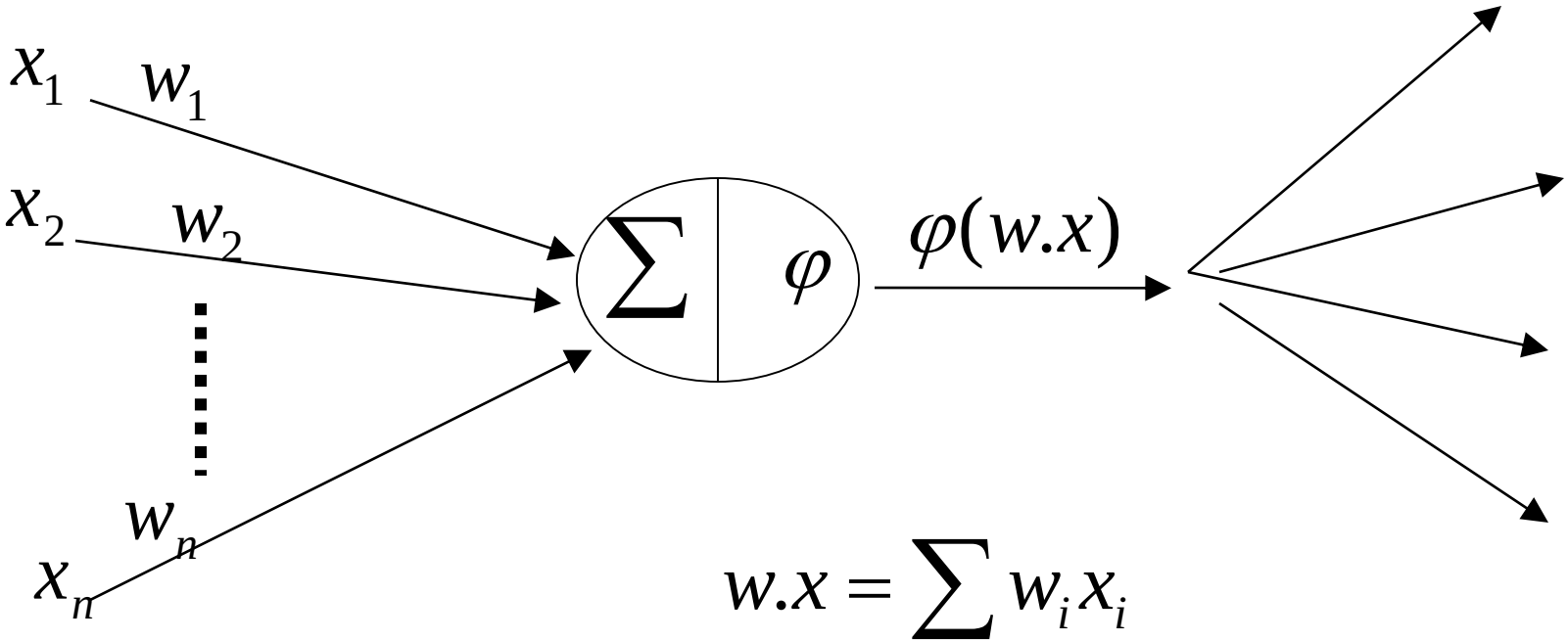
$$\frac{\partial E}{\partial w_i} = \frac{1}{2} \sum_{j=1..m} \frac{\partial \hat{y}_j}{\partial w_i} \frac{\partial E_j}{\partial \hat{y}_j} = - \sum_{j=1..m} x_{i,j} (y_j - \hat{y}_j)$$

- The batch delta rule changes the weights in proportion to their error derivatives summed over all training cases

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Computation at Units

- Compute a 0-1 or a *graded* function of the weighted sum of the inputs
- $\varphi()$ is the *activation* function



$$w.x = \sum w_i x_i$$

Neuron Model: Logistic Unit

$$\varphi(z) = \frac{1}{1+e^{-z}} = \frac{1}{1+e^{-w.x}}$$

$$\phi'(z) = \varphi(z)\big(1-\varphi(z)\big)$$

$$E = \frac{1}{2}\sum_d (y-\hat{y})^2 = \frac{1}{2}\sum_d (y-\varphi(w.x_d))^2$$

$$\frac{\partial E}{\partial w_i} = \sum_d \frac{1}{2}\frac{\partial E_d}{\partial \widehat{y_d}}\frac{\partial \widehat{y_d}}{\partial w_i}$$

$$= \sum_d (y_d - \widehat{y_d})\frac{\partial y}{\partial w_i}(y_d - \varphi(w.x_d))$$

$$= -\sum_d (y_d - \widehat{y_d})\,\varphi'(w.x_d)\,x_{i,d}$$

$$= -\sum_d (y_d - \widehat{y_d})\widehat{y_d}\,(1-\widehat{y_d})x_{i,d}$$

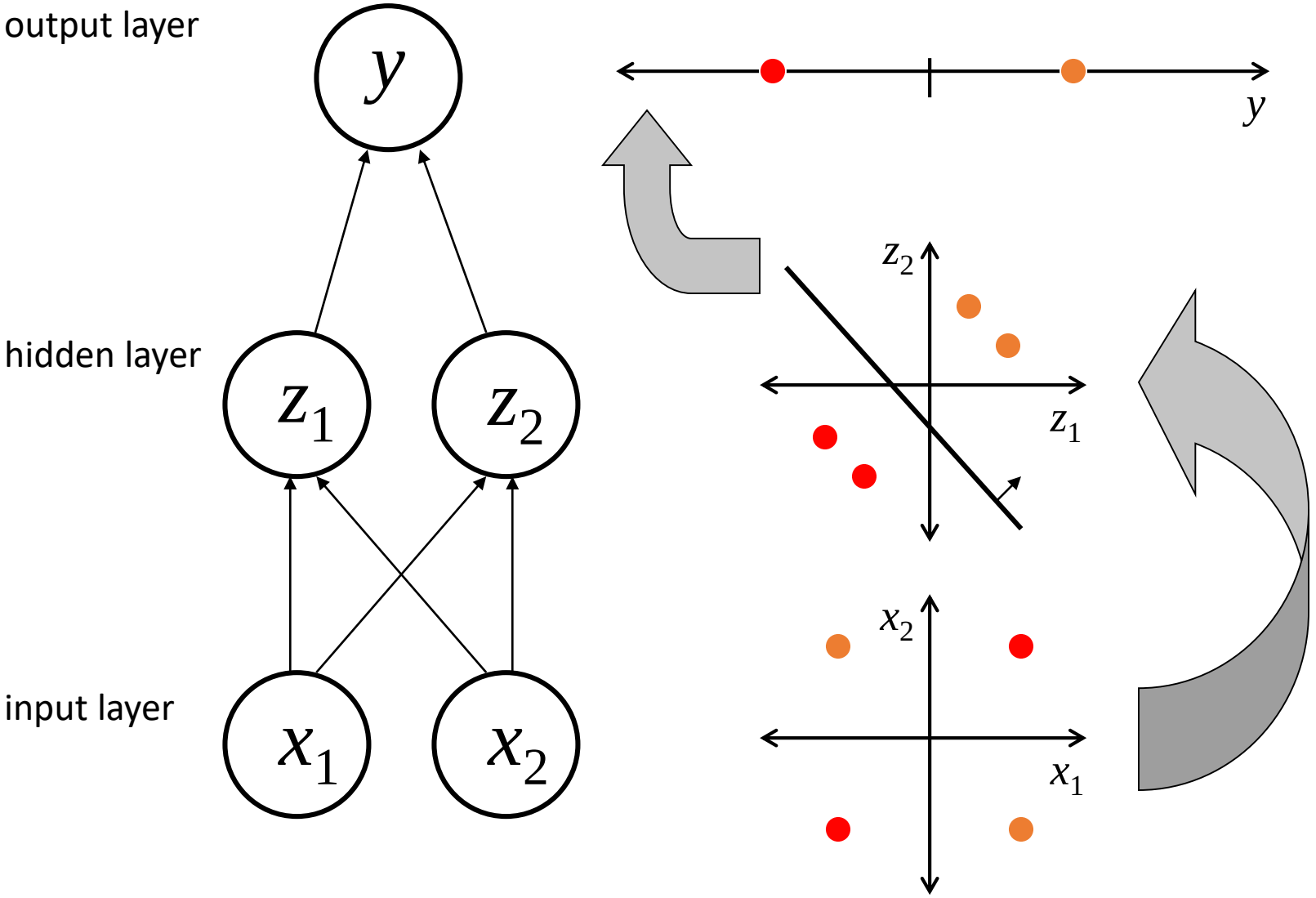
Training Rule: $\Delta w_i = \eta \sum_d (y_d - \widehat{y_d})\widehat{y_d}\,(1-\widehat{y_d})x_{i,d}$

Limitations of Perceptrons

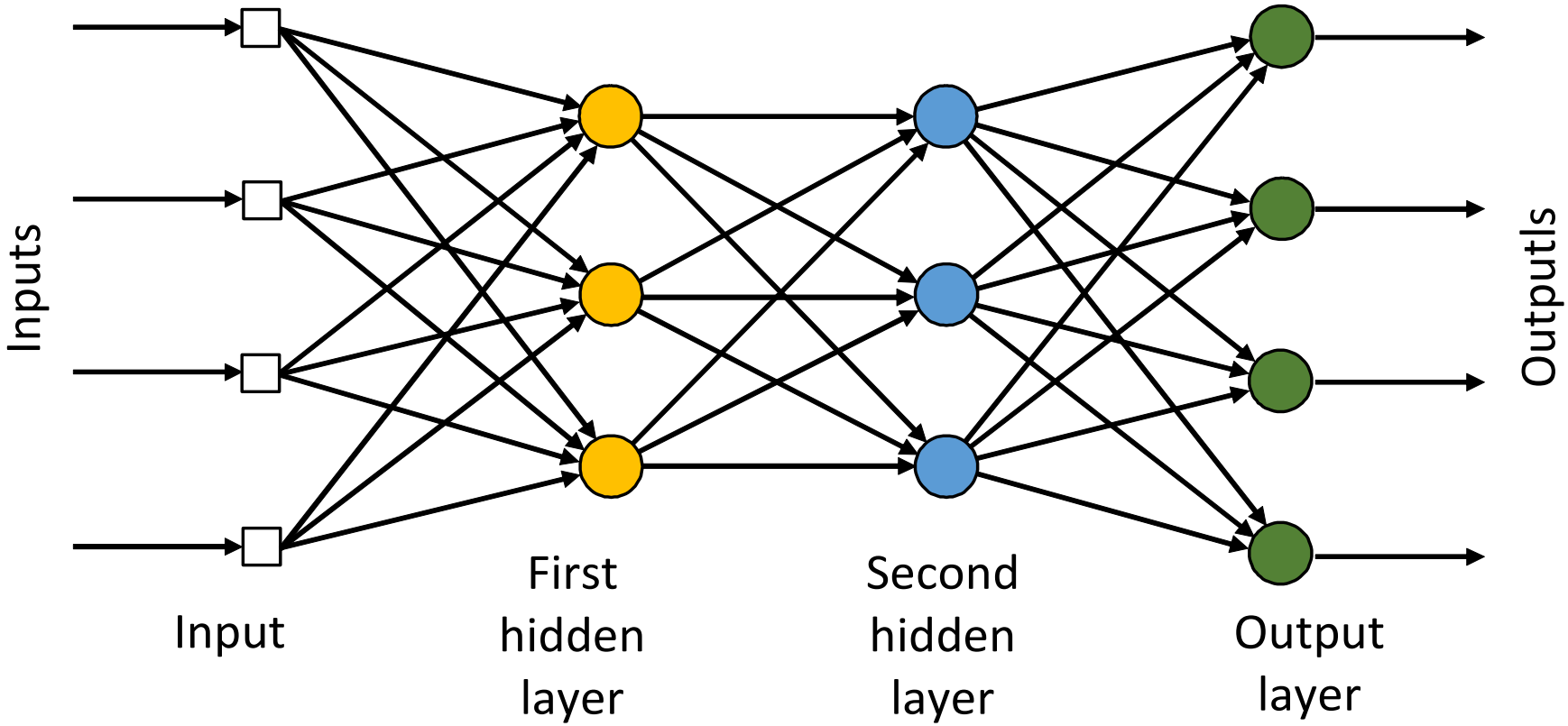
- Perceptrons have a *monotinicity* property:

  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

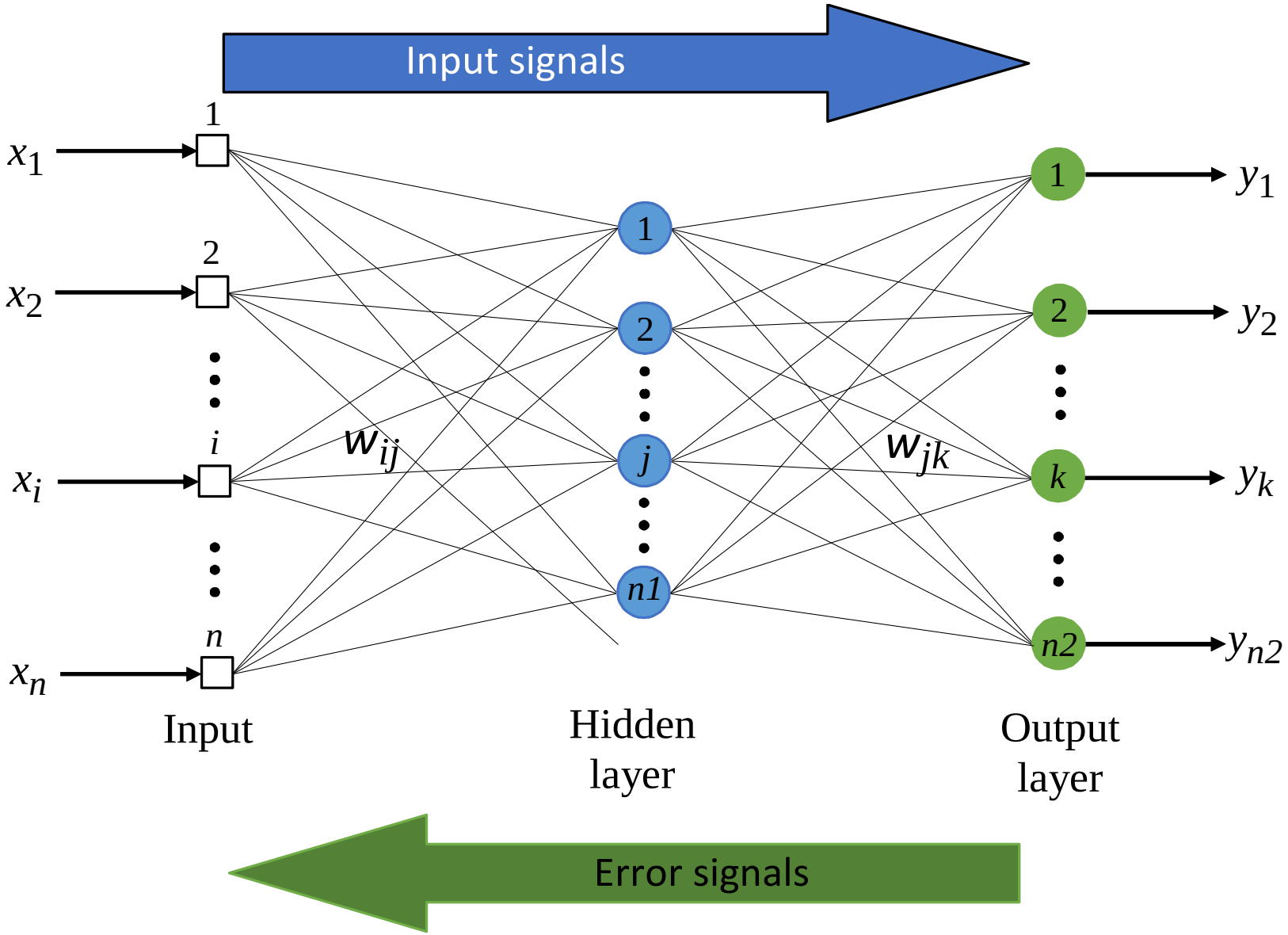
A solution: multiple layers



Multilayer Network

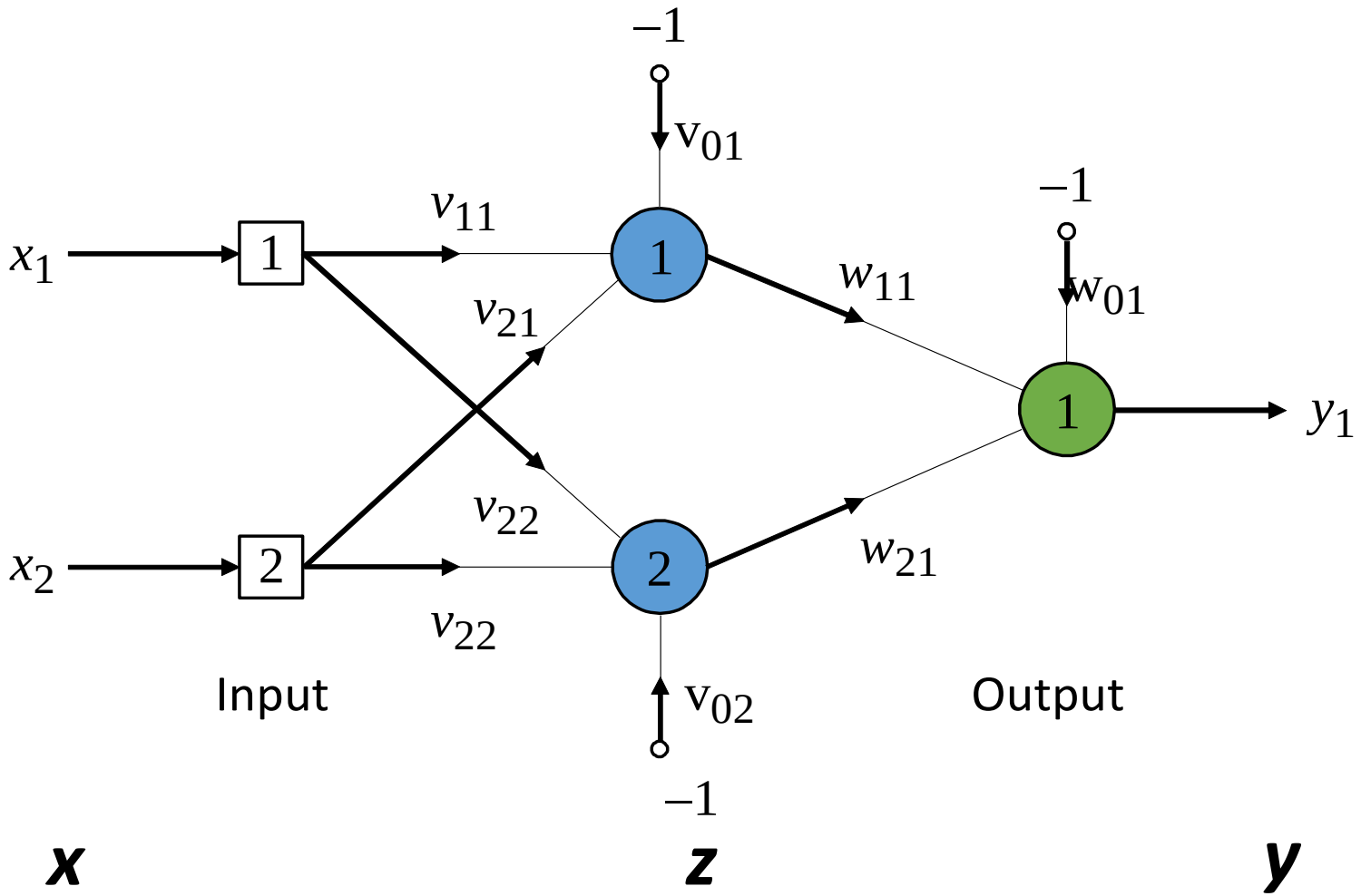


Two-layer back-propagation neural network



The back-propagation training algorithm

- Step 1: Initialisation
  Set all the weights and threshold levels of the network to random numbers uniformly distributed inside a small range



Backprop

- Initialization
  - Set all the weights and threshold levels of the network to random numbers uniformly distributed inside a small range
- Forward computing:
  - Apply an input vector ***x*** to input units
  - Compute activation/output vector ***z*** on hidden layer

    $z_j = \varphi(\sum_i v_{ij} x_i)$
  - Compute the output vector ***y*** on output layer

    $y_k = \varphi(\sum_j w_{jk} z_j)$

    y is the result of the computation.

Learning for BP Nets

- Update of weights in W (between output and hidden layers):
  - delta rule
- Not applicable to updating V (between input and hidden)
  - don’t know the target values for hidden units z1, Z2, … ,ZP
- Solution: Propagate errors at output units to hidden units to drive the update of weights in V (again by delta rule) (error BACKPROPAGATION learning)
- Error backpropagation can be continued downward if the net has more than one hidden layer.
- How to compute errors on hidden units?

Derivation

- For one output neuron, the error function is

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

- 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:

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

Derivation

- Finding the derivative of the error:

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

$$\frac{\partial net_j}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\left(\sum_{k=1}^{n} w_{kj} o_k\right) = o_i$$

$$\frac{\partial o_j}{\partial net_j} = \frac{\partial}{\partial net_j}\varphi\left(net_j\right) = \varphi\left(net_j\right)\left(1 - \varphi\left(net_j\right)\right)$$

Consider E as as a function of the inputs of all neurons $Z = \{z_1, z_2, \dots\}$ receiving input from neuron j,

$$\frac{\partial E\left(o_j\right)}{\partial o_j} = \frac{\partial E\left(net_{z_1}, net_{z_2}, \dots\right)}{\partial o_j}$$

taking the total derivative with respect to o_j, a recursive expression for the derivative is obtained:

$$\frac{\partial E}{\partial o_j} = \sum_l \left(\frac{\partial E}{\partial net_{z_l}}\frac{\partial net_{z_l}}{\partial o_j}\right) = \sum_l \left(\frac{\partial E}{\partial o_l}\frac{\partial o_l}{\partial net_{z_l}} w_{jz_l}\right)$$

$$\frac{\partial E}{\partial o_j} = \sum_l \left(\frac{\partial E}{\partial net_{z_l}} \frac{\partial net_{z_l}}{\partial o_j} \right) = \sum_l \left(\frac{\partial E}{\partial o_l} \frac{\partial o_l}{\partial net_{z_l}} w_{jz_l} \right)$$

- Therefore, the derivative with respect to o_j can be calculated if all the derivatives with respect to the outputs o_{z_l} of the next layer – the one closer to the output neuron – are known.
- Putting it all together:

$$\frac{\partial E}{\partial w_{ij}} = \delta_j o_i$$

With

$$\delta_j = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} = \begin{cases} (o_j - t_j) o_j (1 - o_j) \text{ if } j \text{ is an output neuron} \\ \left(\sum_z \delta_{z_l} w_{jl} \right) o_j (1 - o_j) \text{ if } j \text{ is an inner neuron} \end{cases}$$

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

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

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

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

where

$$\Delta w_{i,j} = \eta \delta_j x_{i,j}$$

x_d = input

y_d = target output

o_d = observed unit output

w_{ij} = wt from i to j

Backpropagation

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

Too slow to converge
Gets stuck in local minima

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

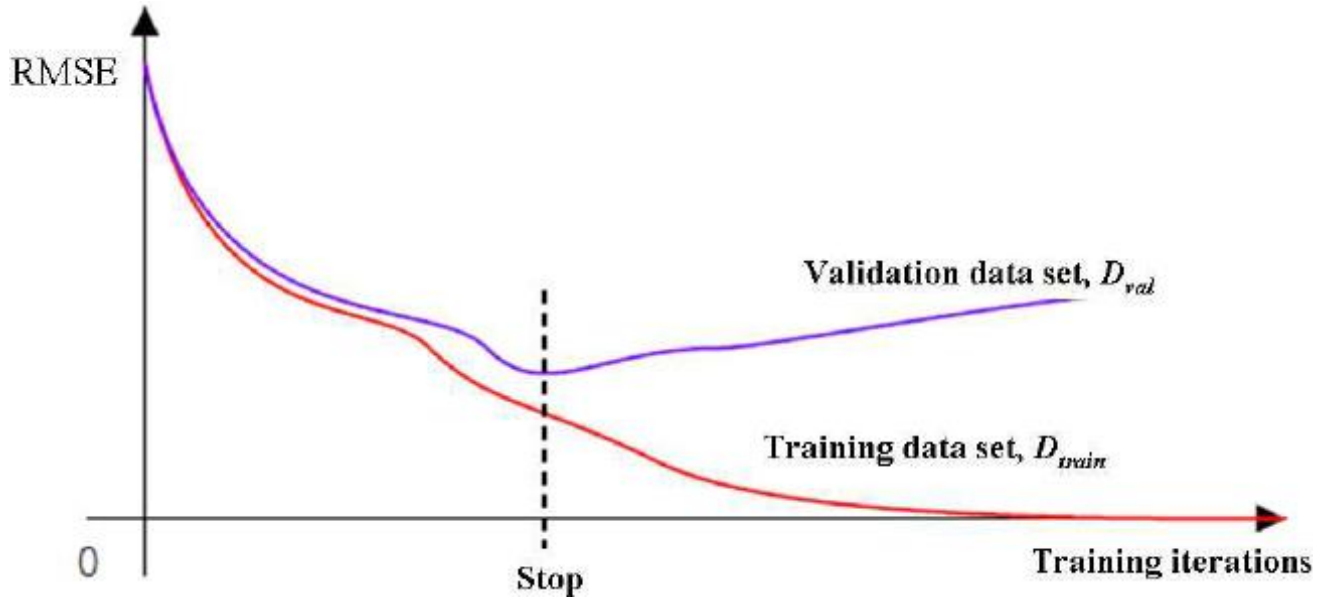
Learning in *epochs*
Stopping

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

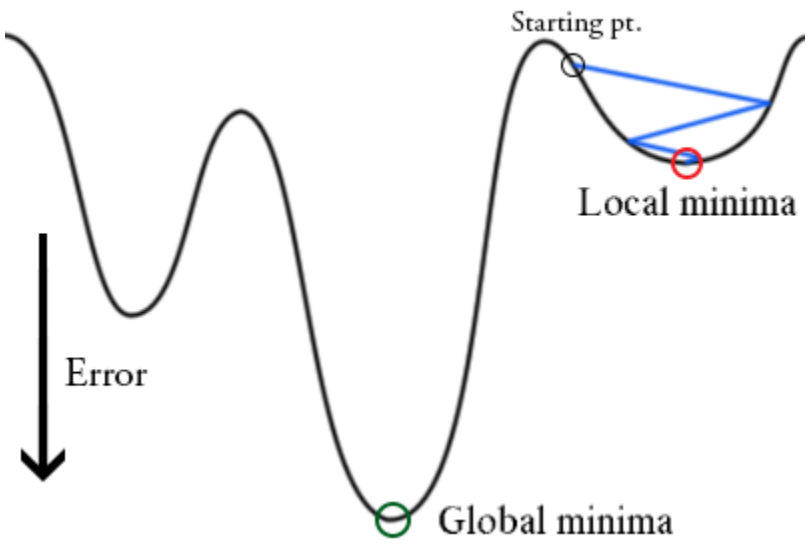
Stopping

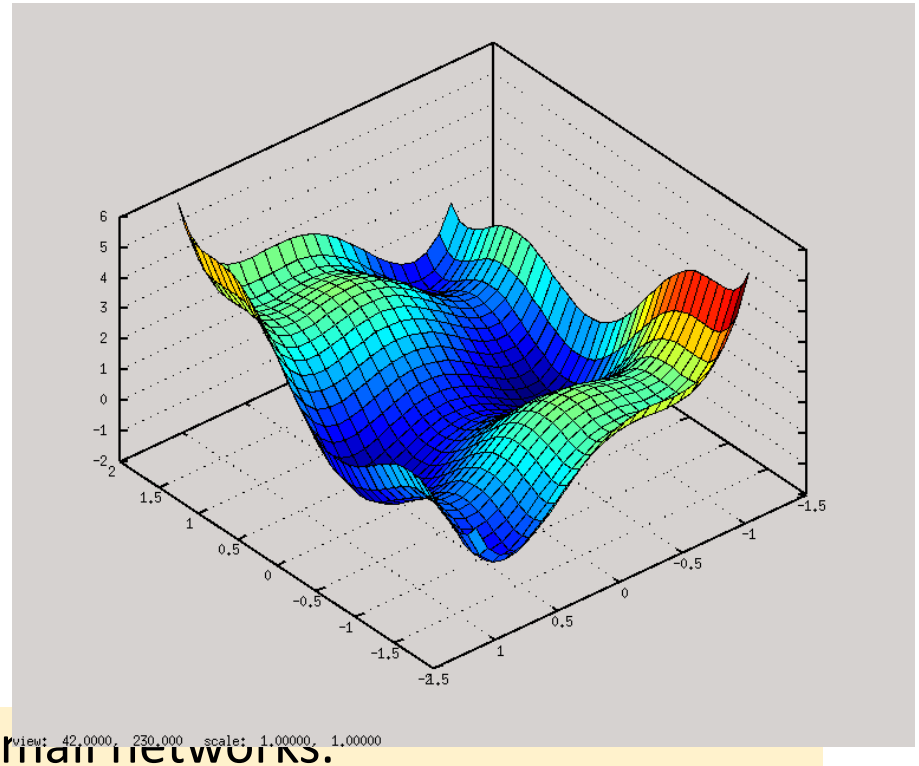
1. Fixed maximum number of epochs: most naïve
2. Keep track of the training and validation error curves.

Overfitting in ANNs



Local Minima





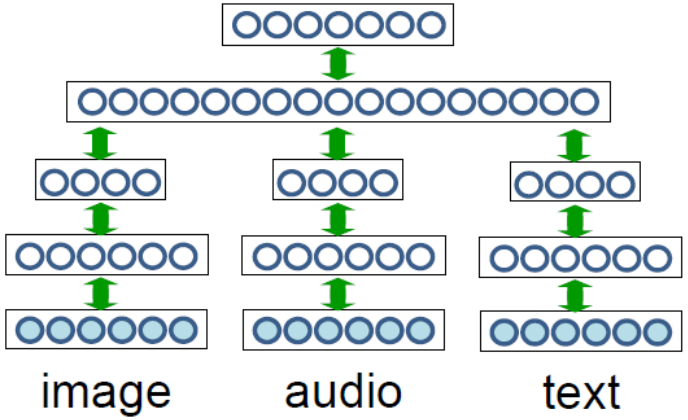
- NN can get stuck in local minima for small networks.
- For most large networks (many weights) local minima rarely occurs.
- It is unlikely that you are in a minima in every dimension simultaneously.

ANN

- Highly expressive non-linear functions
- Highly parallel network of logistic function units
- Minimizes sum of squared training errors
- Can add a regularization term (weight squared)
- Local minima
- Overfitting

Deep Learning

- Breakthrough results in
  - Image classification
  - Speech Recognition
  - Machine Translation
  - Multi-modal learning



Deep Neural Network

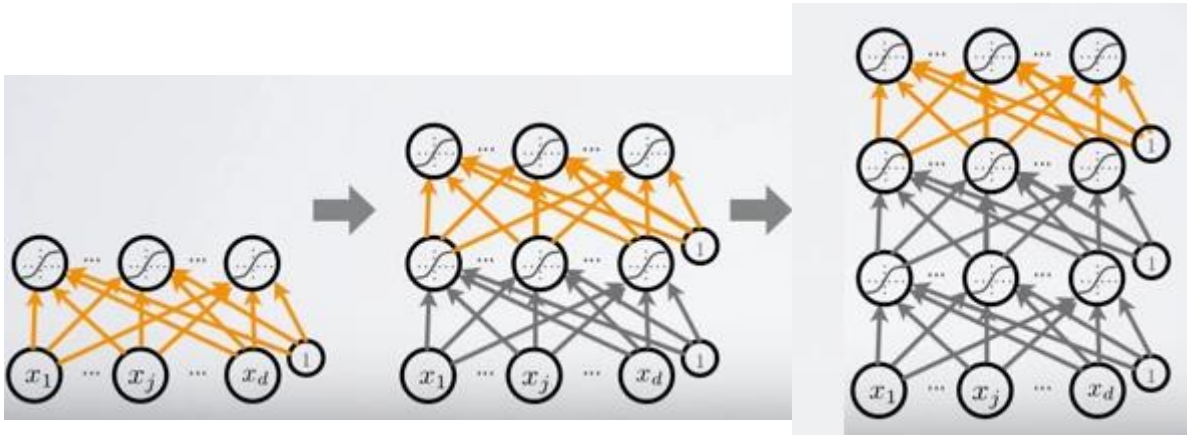
- Problem: training networks with many hidden layers doesn’t work very well
- Local minima, very slow training if initialize with zero weights.
- Diffusion of gradient.

Hierarchical Representation

- Hierarchical Representation help represent complex functions.
- NLP: character ->word -> Chunk -> Clause -> Sentence
- Image: pixel > edge -> texton -> motif -> part -> object
- Deep Learning: learning a hierarchy of internal representations
- Learned internal representation at the hidden layers (trainable feature extractor)
- Feature learning

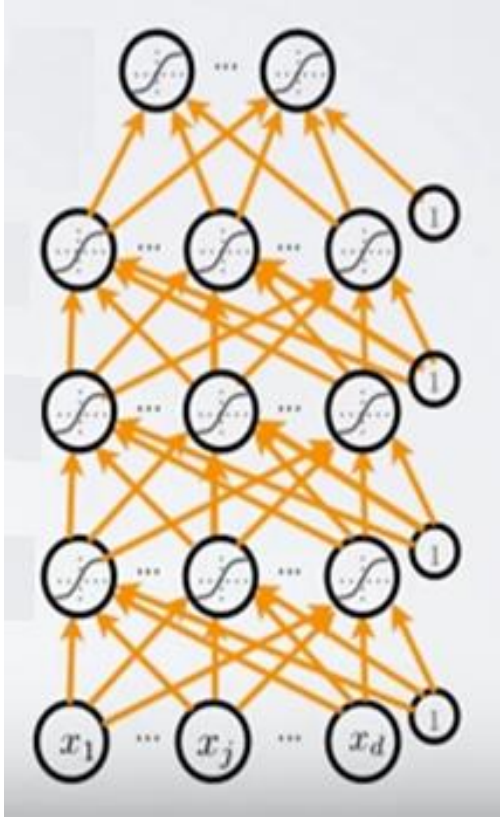
Unsupervised Pre-training

- We will use greedy, layer wise pre-training
  - Train one layer at a time
  - Fix the parameters of previous hidden layers
  - Previous layers viewed as feature extraction
- find hidden unit features that are more common in training input than in random inputs



Tuning the Classifier

- After pre-training of the layers
  - Add output layer
  - Train the whole network using supervised learning (Back propagation)



Deep neural network

- Feed forward NN
- Stacked Autoencoders (multilayer neural net with target output = input)
- Stacked restricted Boltzmann machine
- Convolutional Neural Network

A Deep Architecture: Multi-Layer Perceptron

Output Layer

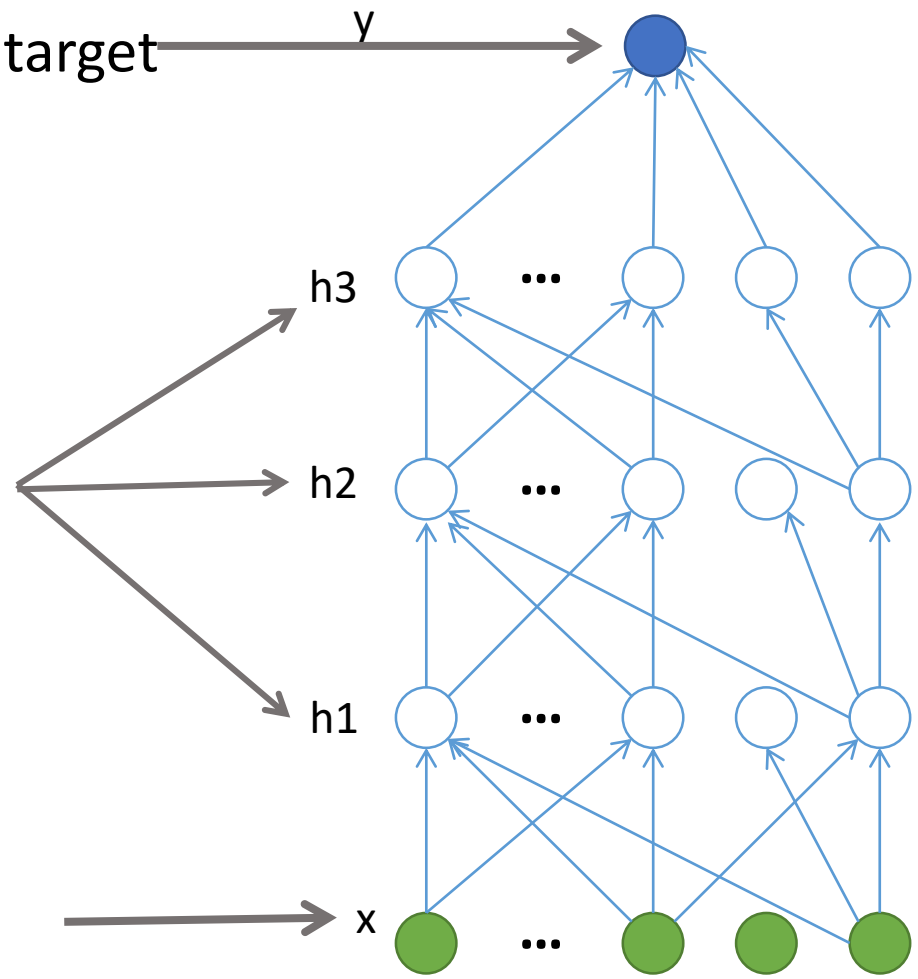
Here predicting a supervised target

Hidden layers

These learn more abstract representations as you head up

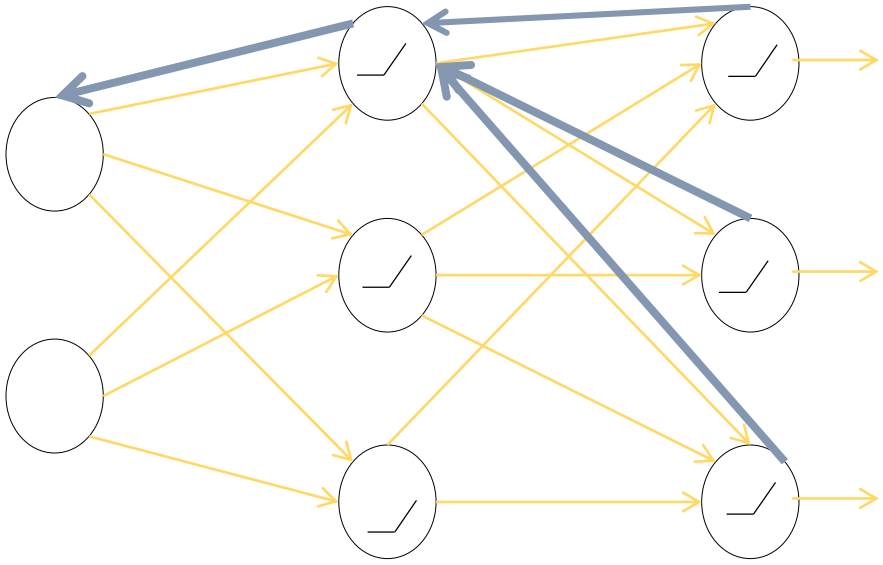
Input layer

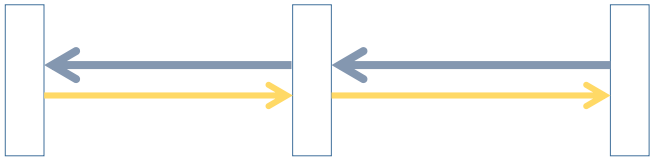
Raw sensory inputs



A Neural Network

- Training : Back Propagation of Error
  - Calculate total error at the top
  - Calculate contributions to error at each step going backwards
  - The weights are modified as the error is propagated



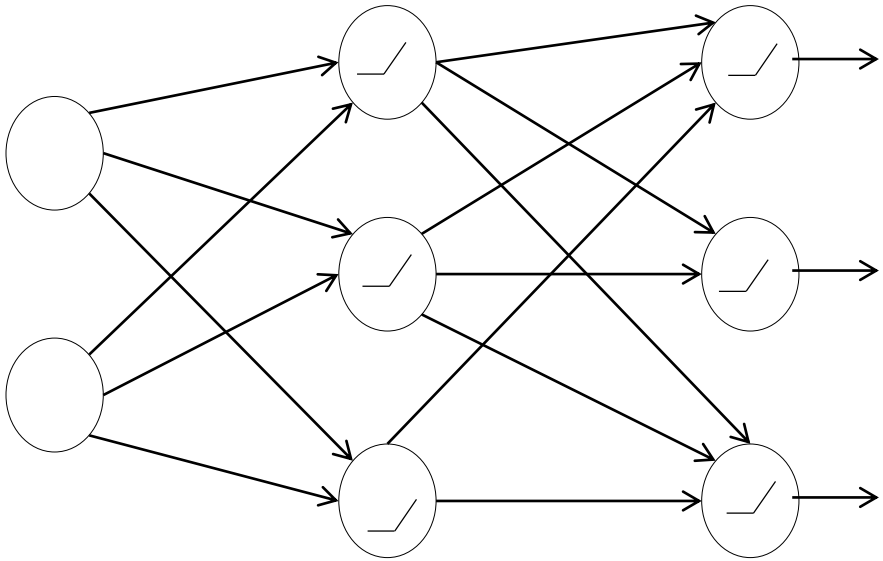


Training Deep Networks

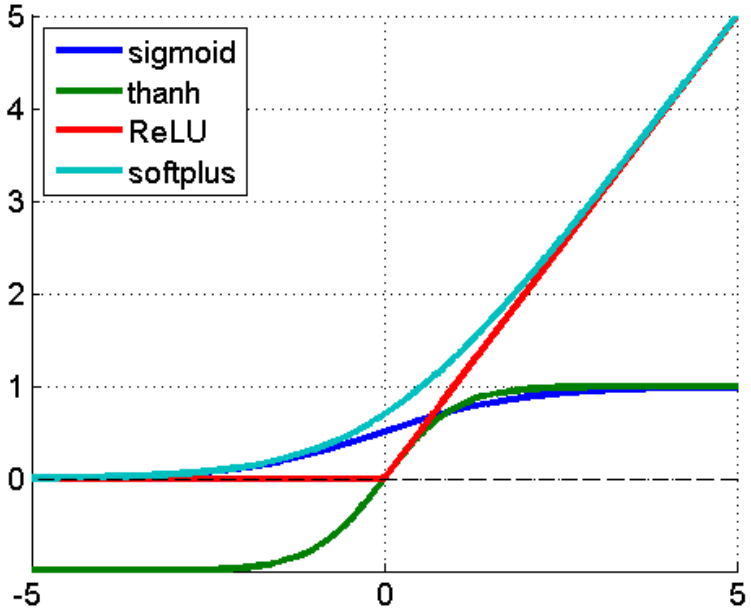
- Difficulties of supervised training of deep networks
  1. Early layers of MLP do not get trained well
     - Diffusion of Gradient – error attenuates as it propagates to earlier layers
     - Leads to very slow training
     - the error to earlier layers drops quickly as the top layers "mostly" solve the task
  2. Often not enough labeled data available while there may be lots of unlabeled data
  3. Deep networks tend to have more local minima problems than shallow networks during supervised training

Training of neural networks

- Forward Propagation :
  - Sum inputs, produce activation
  - feed-forward



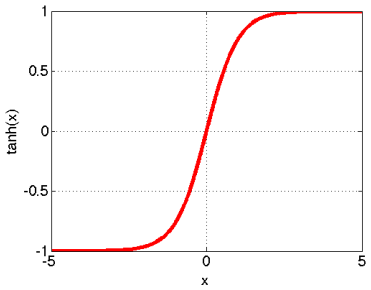
Activation Functions examples



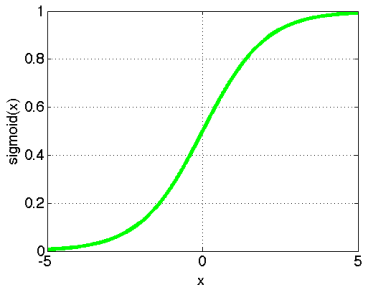
Activation Functions

Non-linearity

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



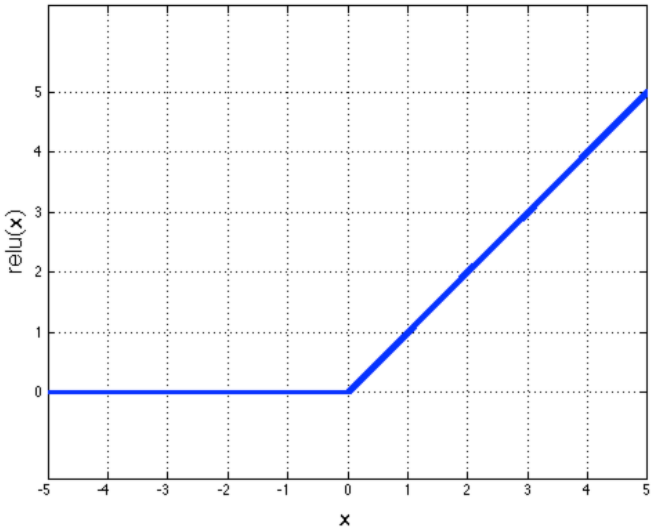
- $\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$



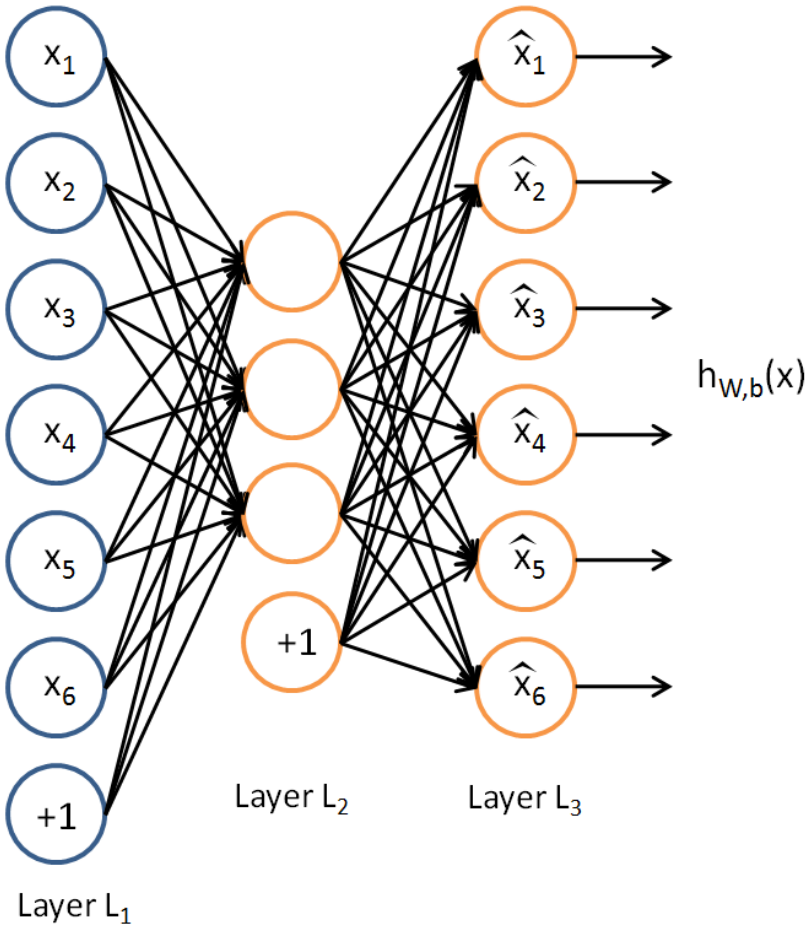
- Rectified linear
  relu(x) = max(0,x)

- Simplifies backprop
- Makes learning faster
- Make feature sparse

→ Preferred option



Autoencoder



Unlabeled training examples set $\{x^{(1)}, x^{(2)}, x^{(3)} \ldots\}$, $x^{(i)} \in \mathbb{R}^n$

Set the target values to be equal to the inputs. $y^{(i)} = x^{(i)}$

Network is trained to output the input (learn identify function).

$$h_{w,b}(x) \approx x$$

Solution may be trivial!

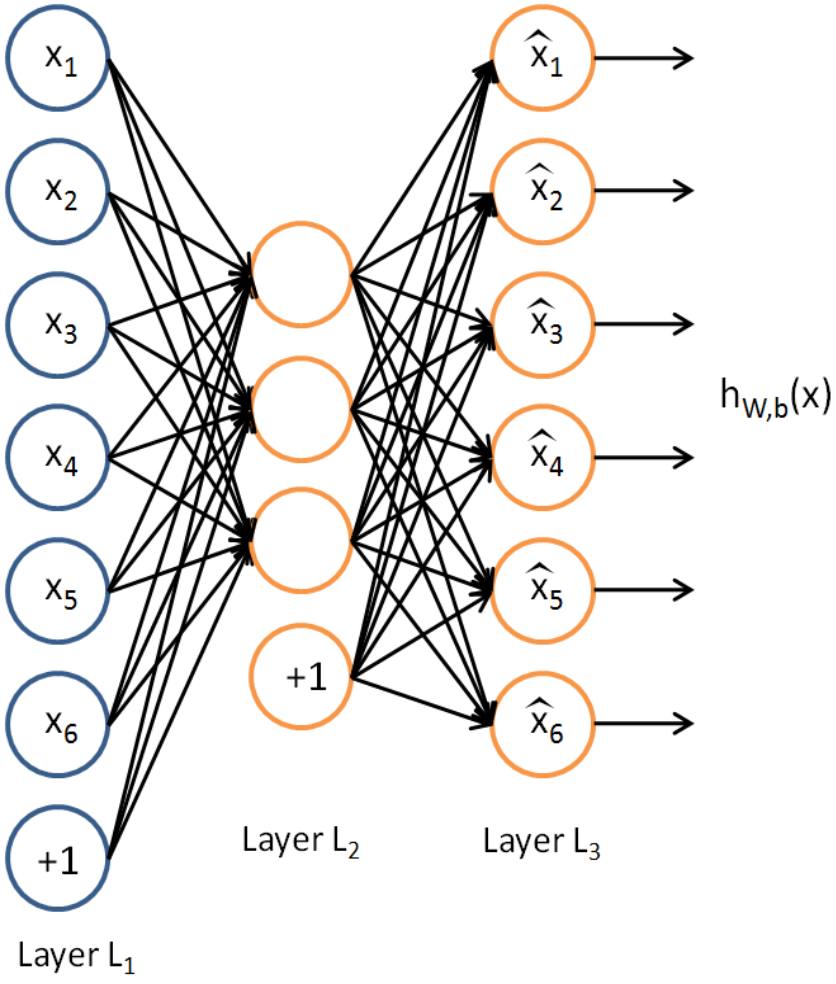
Autoencoders and sparsity

1. Place constraints on the network, like limiting the number of hidden units, to discover interesting structure about the data.
2. Impose sparsity constraint.

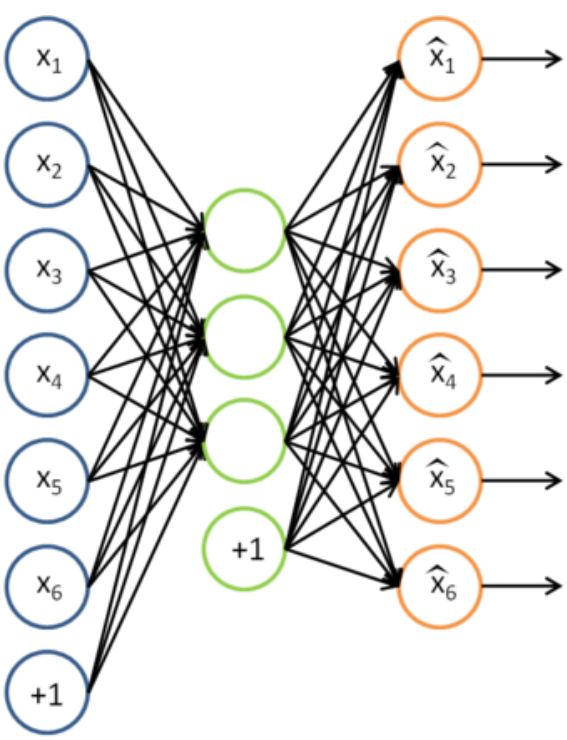
a neuron is “active” if its output value is close to 1

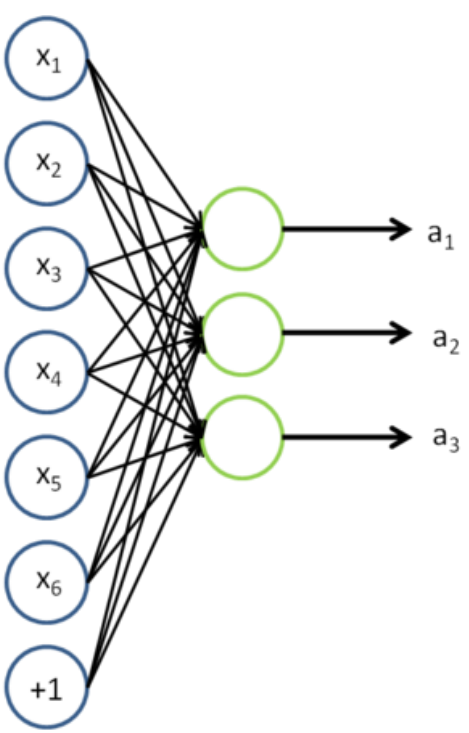
It is “inactive” if its output value is close to 0.

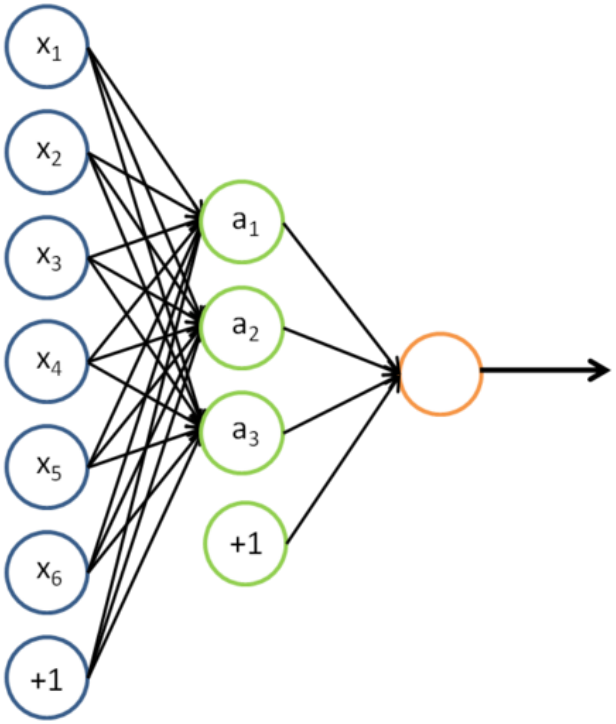
constrain the neurons to be inactive most of the time.



Auto-Encoders

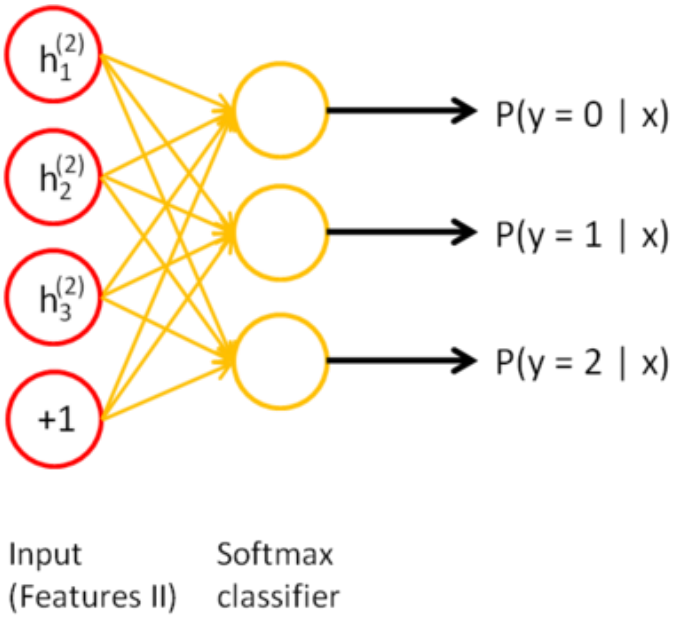




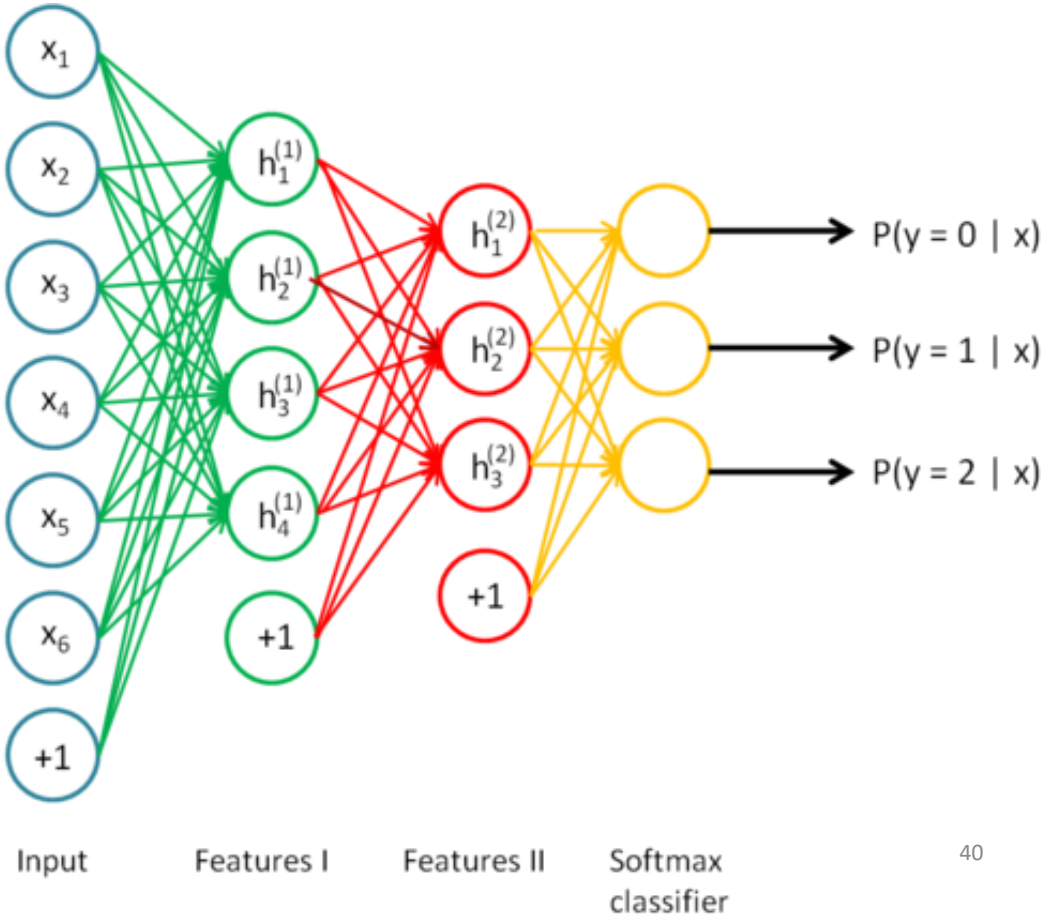


Stacked Auto-Encoders

- Do supervised training on the last layer using final features
- Then do supervised training on the entire network to fine- tune all weights



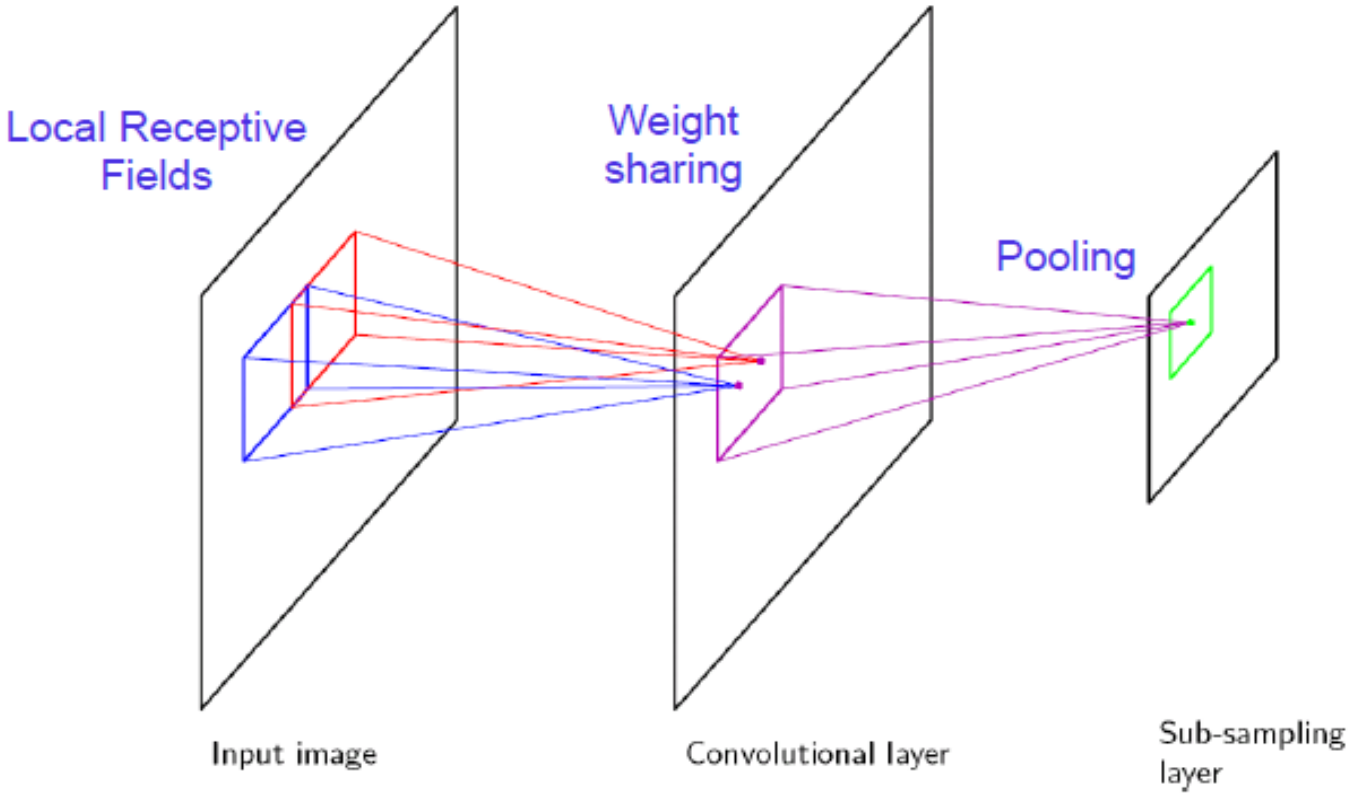
$$y_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$



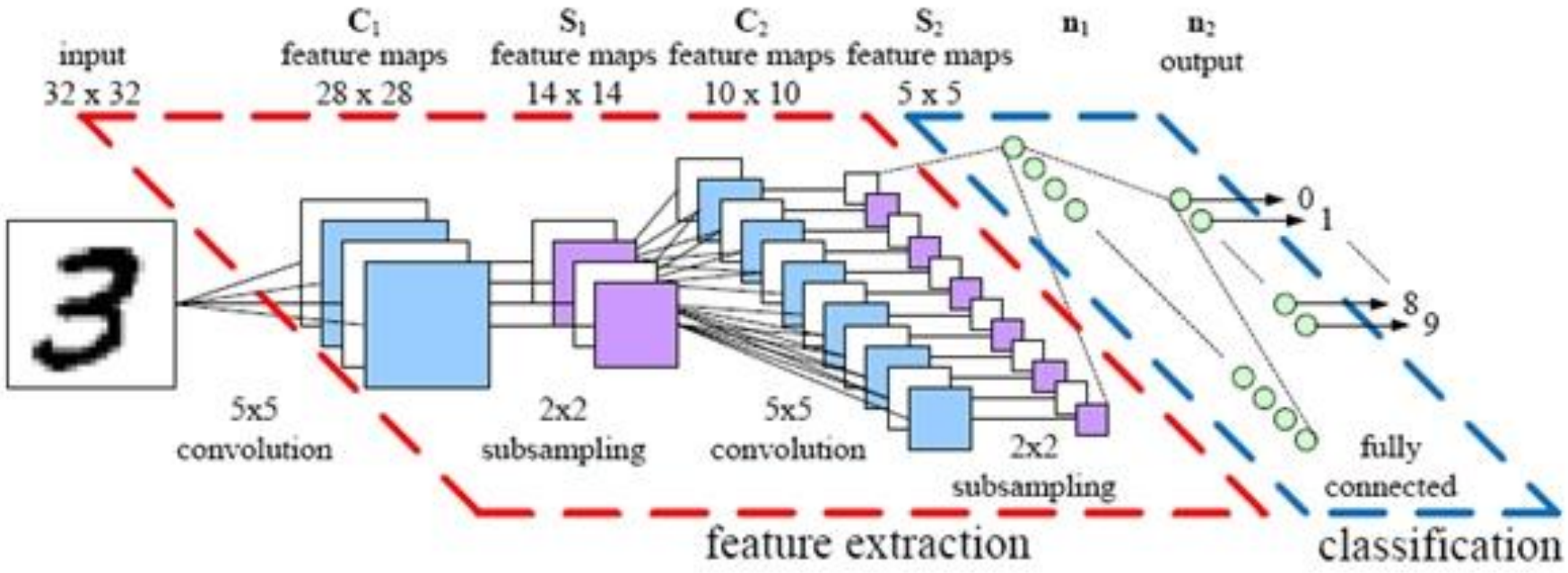
Convolutional Neural netwoks

- A CNN consists of a number of convolutional and subsampling layers.
- Input to a convolutional layer is a m x m x r image where m x m is the height and width of the image and r is the number of channels, e.g. an RGB image has r=3
- Convolutional layer will have k filters (or kernels)
- size n x n x q
- n is smaller than the dimension of the image and,
- q can either be the same as the number of channels r or smaller and may vary for each kernel

Convolutional Neural Networks

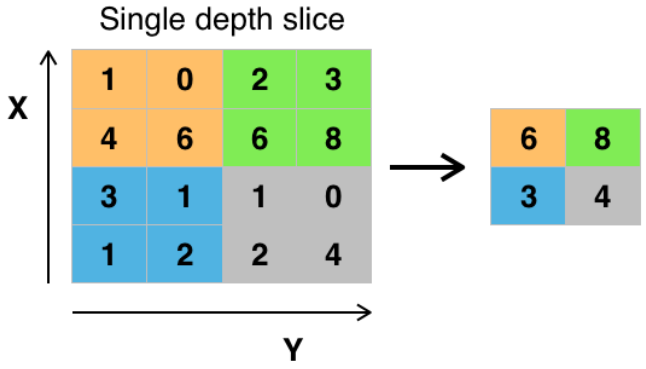


Convolutional layers consist of a rectangular grid of neurons
Each neuron takes inputs from a rectangular section of the previous layer
the weights for this rectangular section are the same for each neuron in the convolutional layer.



Pooling: Using features obtained after Convolution for Classification

The pooling layer takes small rectangular blocks from the convolutional layer and subsamples it to produce a single output from that block : max, average, etc.

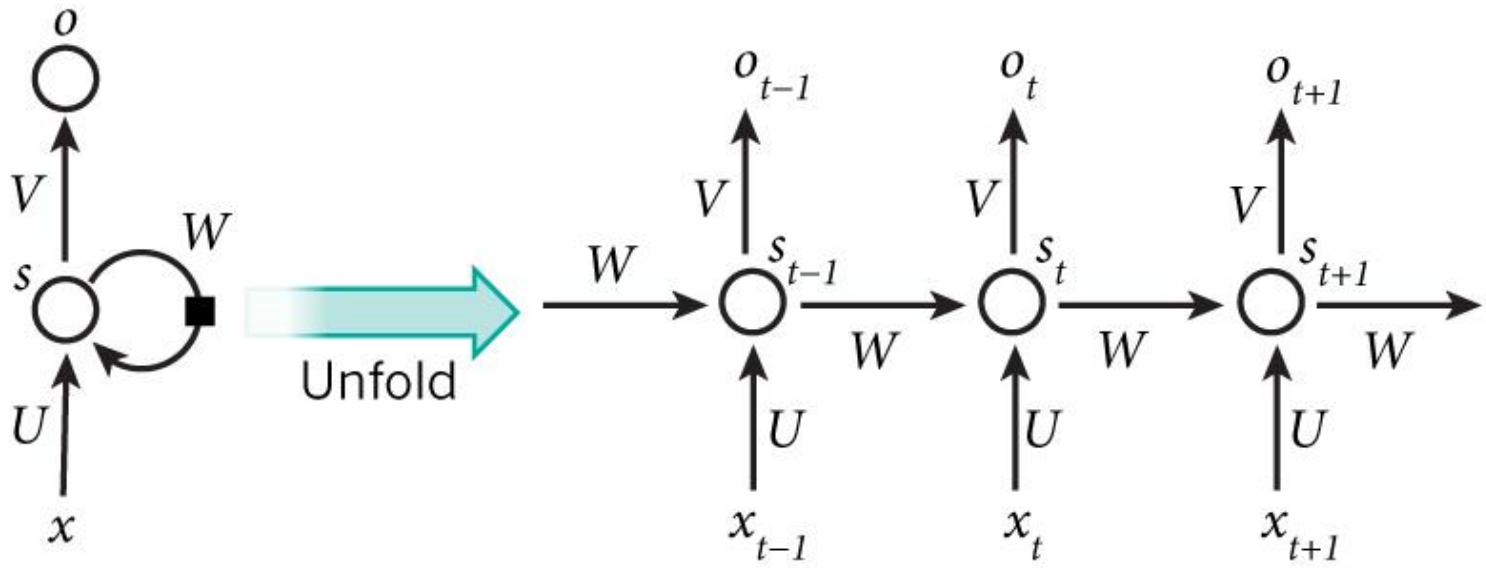


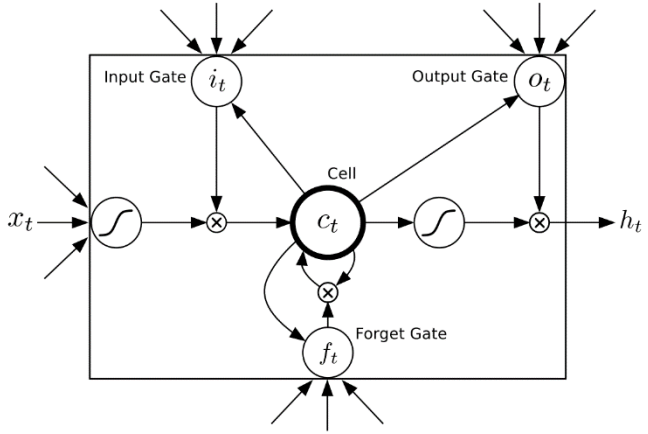
CNN properties

- CNN takes advantage of the sub-structure of the input
- Achieved with local connections and tied weights followed by some form of pooling which results in translation invariant features.

- CNN are easier to train and have many fewer parameters than fully connected networks with the same number of hidden units.

Recurrent Neural Network (RNN)





Thank You