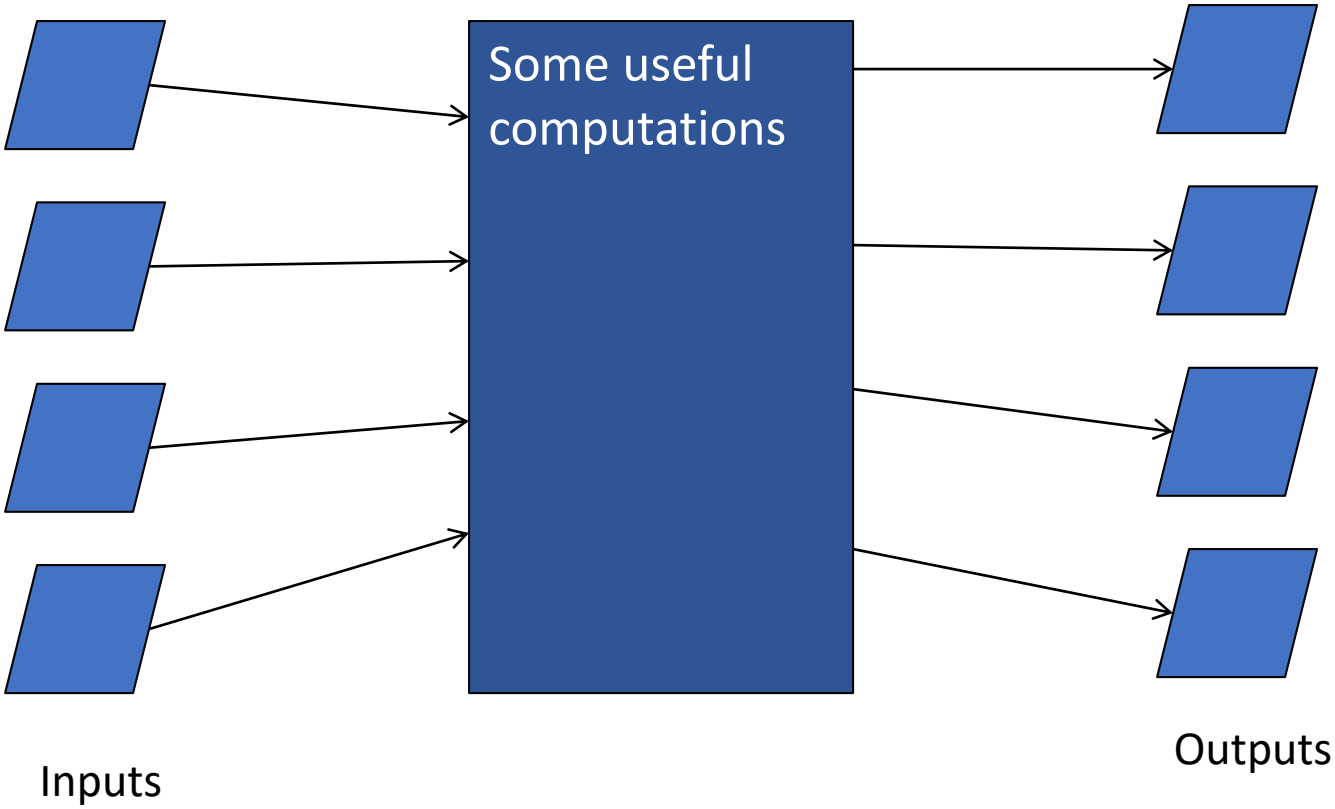
There is a fantastic existence proof that learning is possible, which is the bag of water and electricity (together with a few trace chemicals) sitting between your ears.

Stephen Marsland. “Machine learning: an algorithmic perspective”

Perceptron

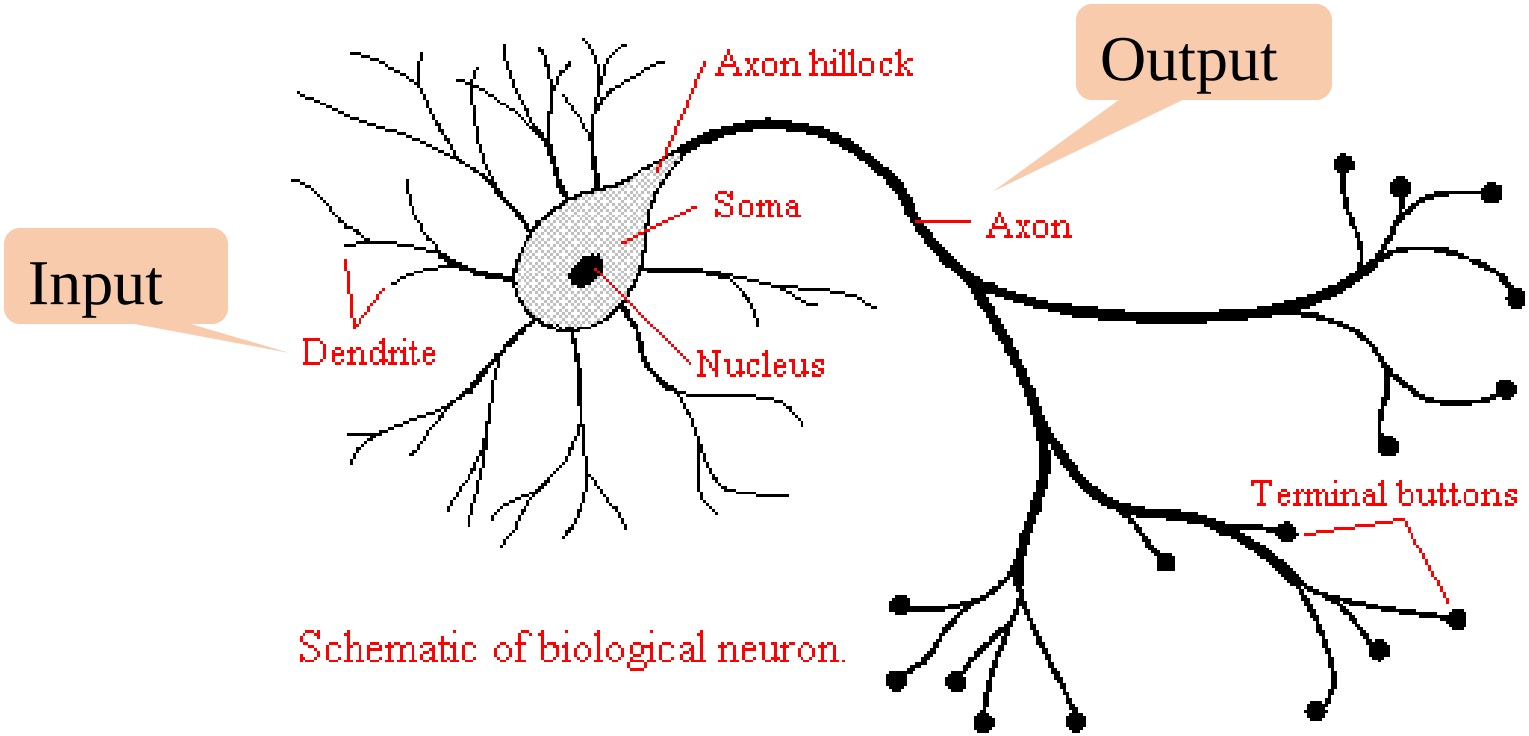
Lecture 09.01

How computer works



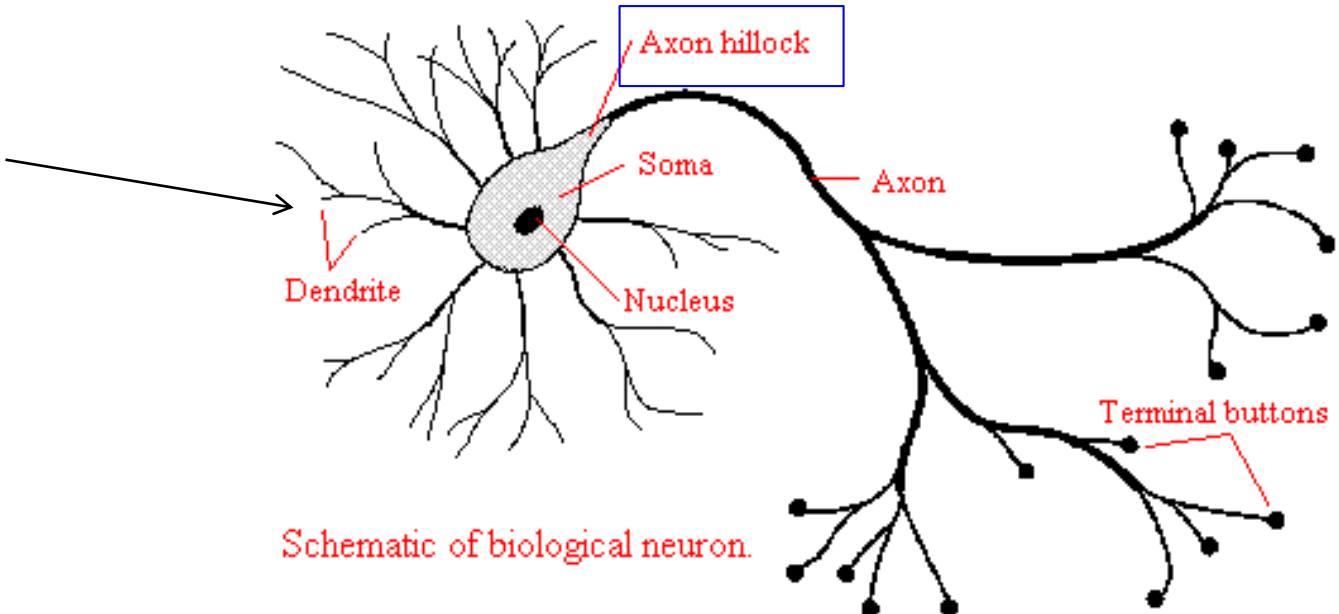
How brain works: neurons

Neuron is an electrically excitable cell that processes and transmits information by electrical and chemical signaling.



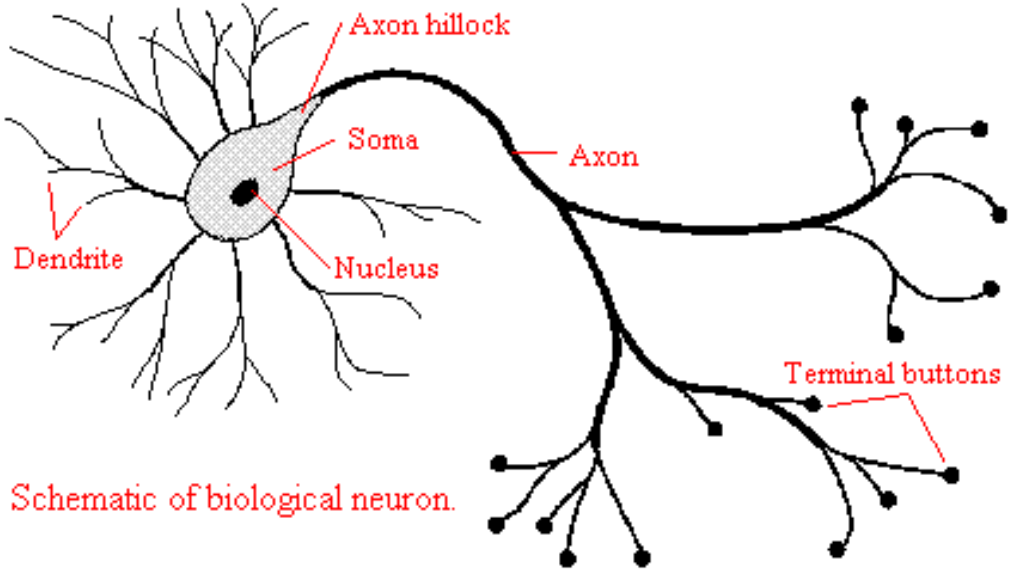
Neurons: signal summation

- Dendrite(s) receive an electric charge.
- The strengths of all the received charges are added together (spatial and temporal **summation**).
- The aggregate value is then passed to the soma (cell body) to axon hillock.



Neurons: activation threshold

- If the aggregate input is greater than the axon hillock's threshold value, then the neuron *fires*, and an output signal is transmitted down the axon.

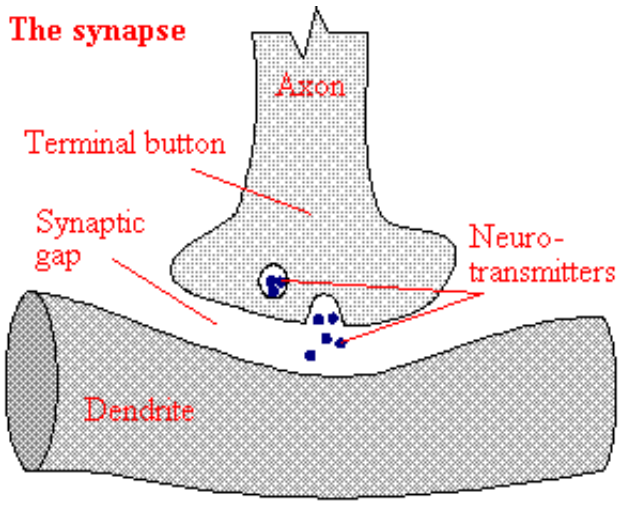


Neurons: the output signal is constant

- The strength of the output is constant, regardless of whether the input was just above the threshold, or a hundred times as great.
- This uniformity is critical in an analogue device such as a brain where small errors can snowball, and where error correction is more difficult.

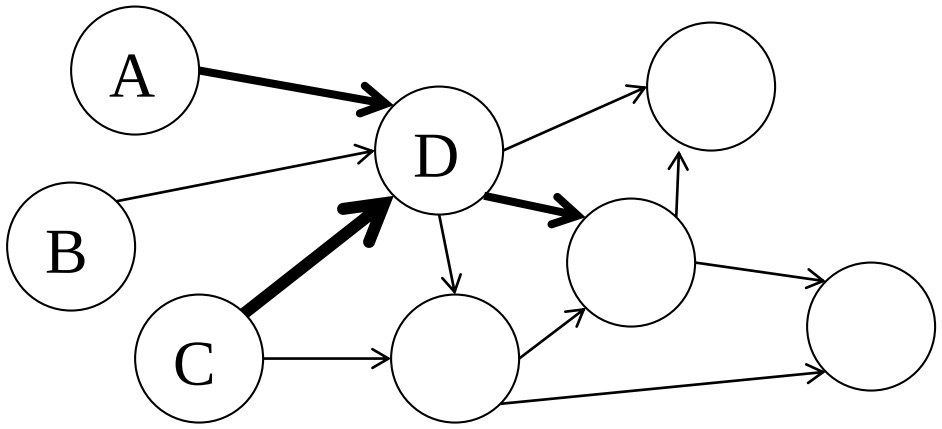
How real neurons communicate

- The signal is transmitted to other neurons through synapses.
- The physical and neurochemical characteristics of each synapse determine the strength and polarity of the new input signal.
- This is where the brain is the most flexible: neuroplasticity.



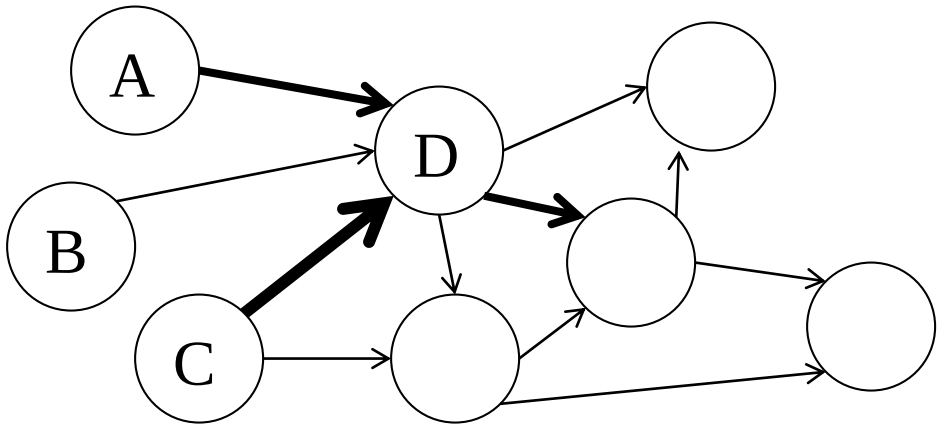
Modeling brain with networks

- The complicated biological phenomena may be modeled by a very simple model: nodes model neurons and edges model connections.
- The input nodes each have a weight that they contribute to the neuron, if the input is active. This corresponds to the strength of a synaptic connection.

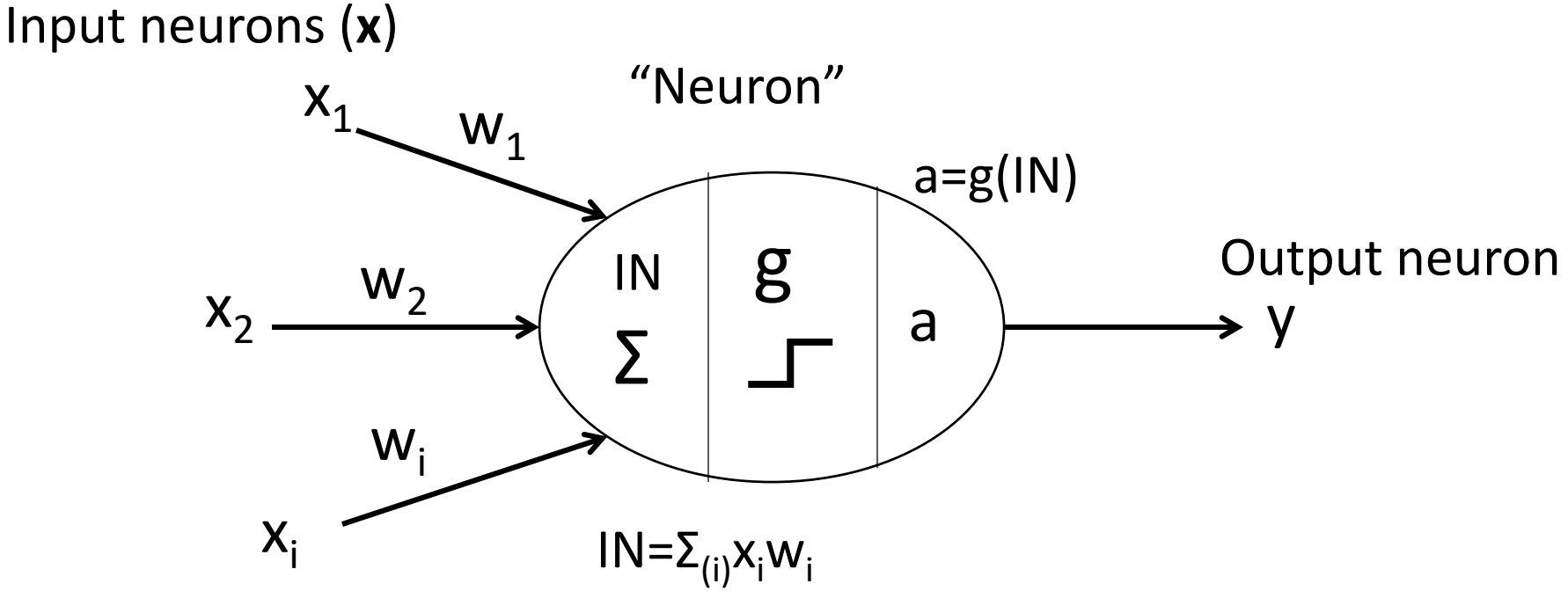


Modeling brain with networks

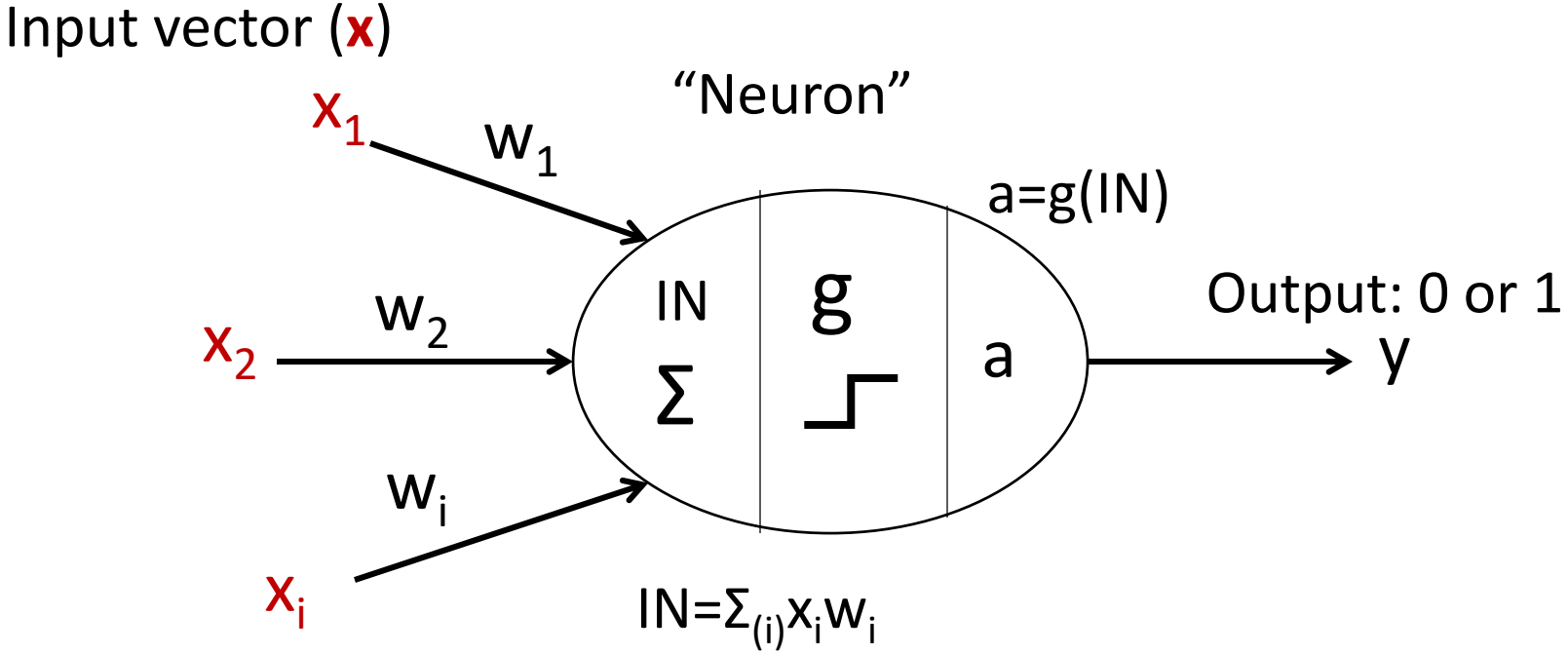
- Node takes input and triggers other nodes through connections
- Node D needs to think if it wants to propagate the signal
- The decision is made from the output of threshold function (0 or 1)



Mathematical model of a neuron (McCulloch and Pitt, 1943)

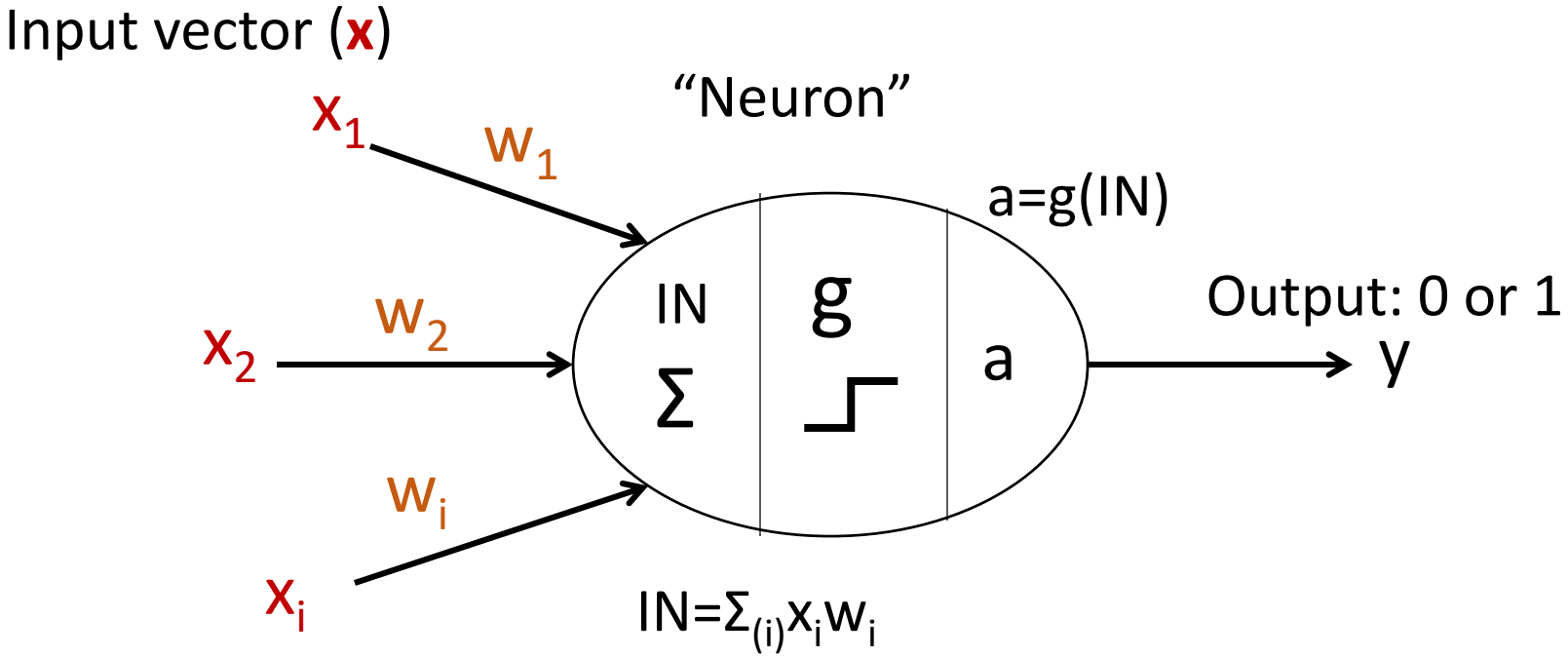


Terminology



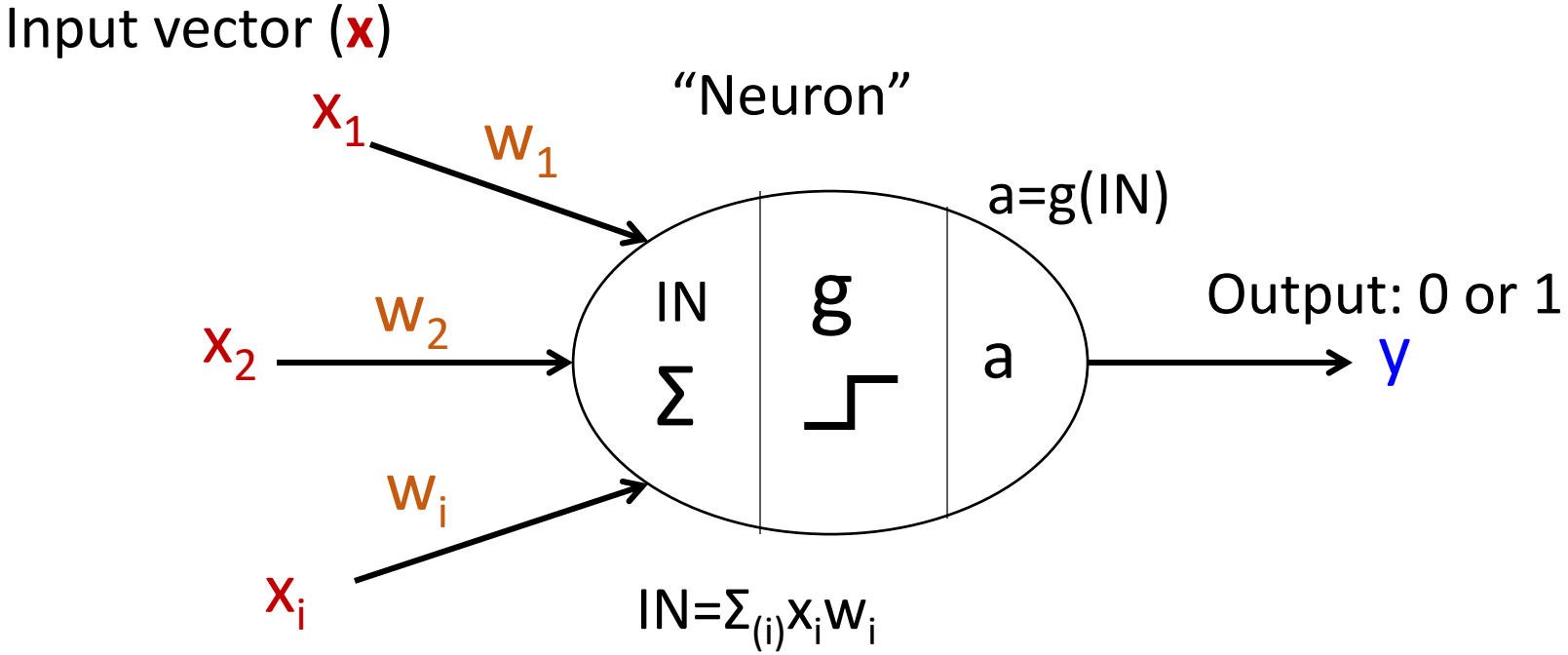
- An input vector **x** is the data given as one input to the processing "neuron" (corresponds to afferent neurons that transmit information to the brain).

Terminology



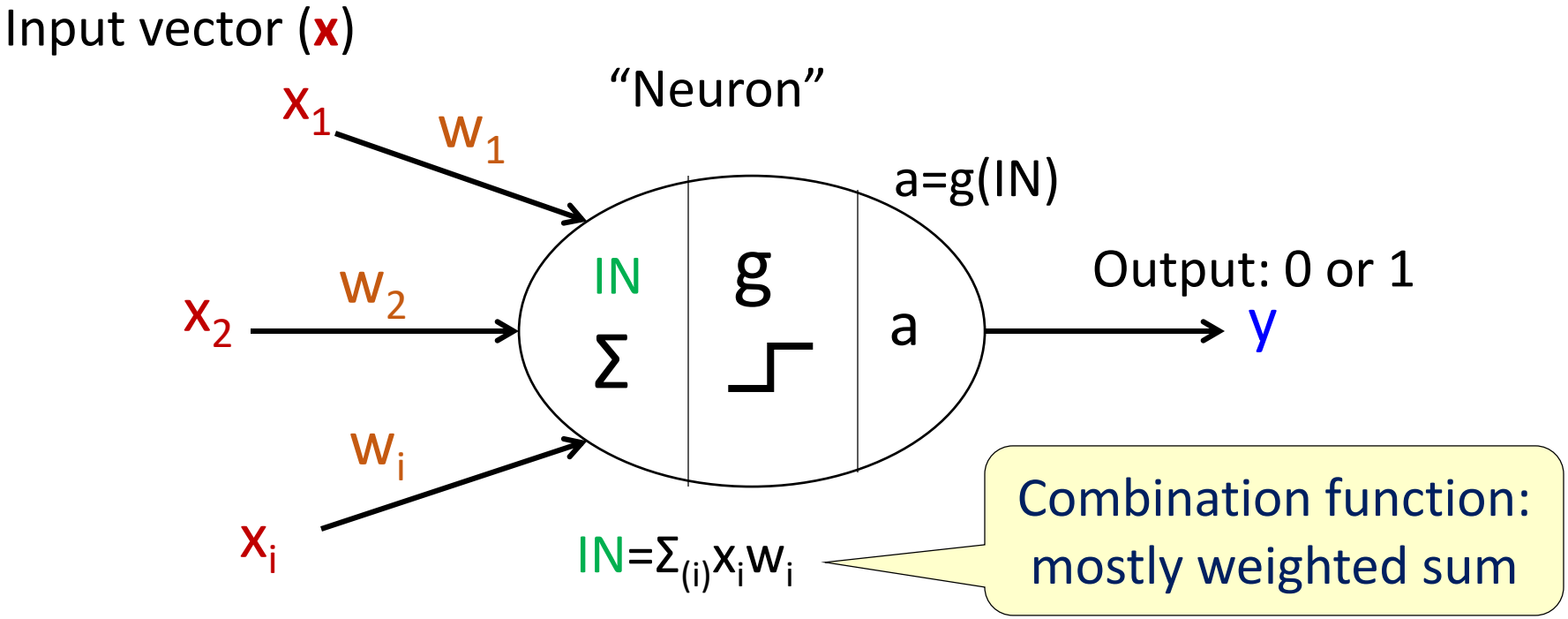
- Weights w_i , are the weighted connections between input neurons and the processing neuron (these weights are analogous to the strength of synaptic connections in the brain).
- They are arranged into a matrix **W**.

Terminology



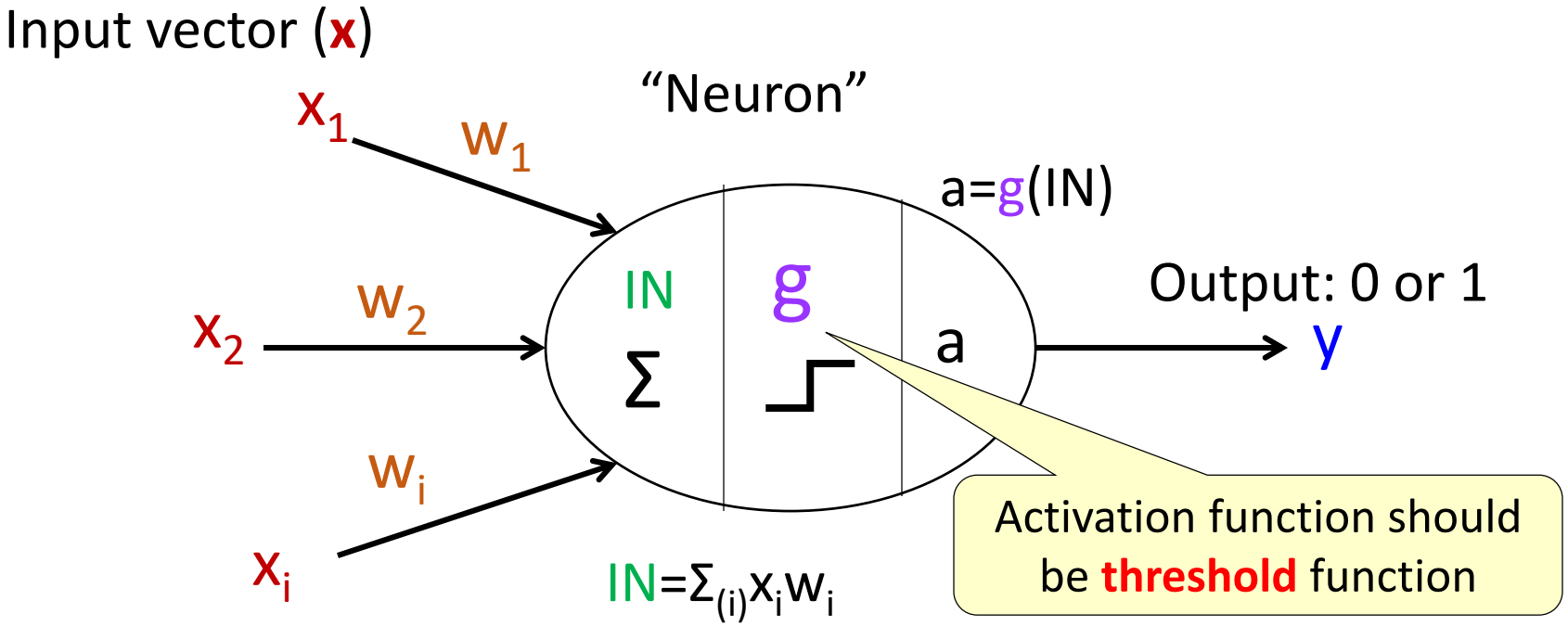
- The output **y**, shows the resulting action of processing neuron: neuron fires(1) or not(0).
- We can write **y(x,W)** to remind that the output depends on the inputs to the algorithm and the current set of weights of the network.

Terminology



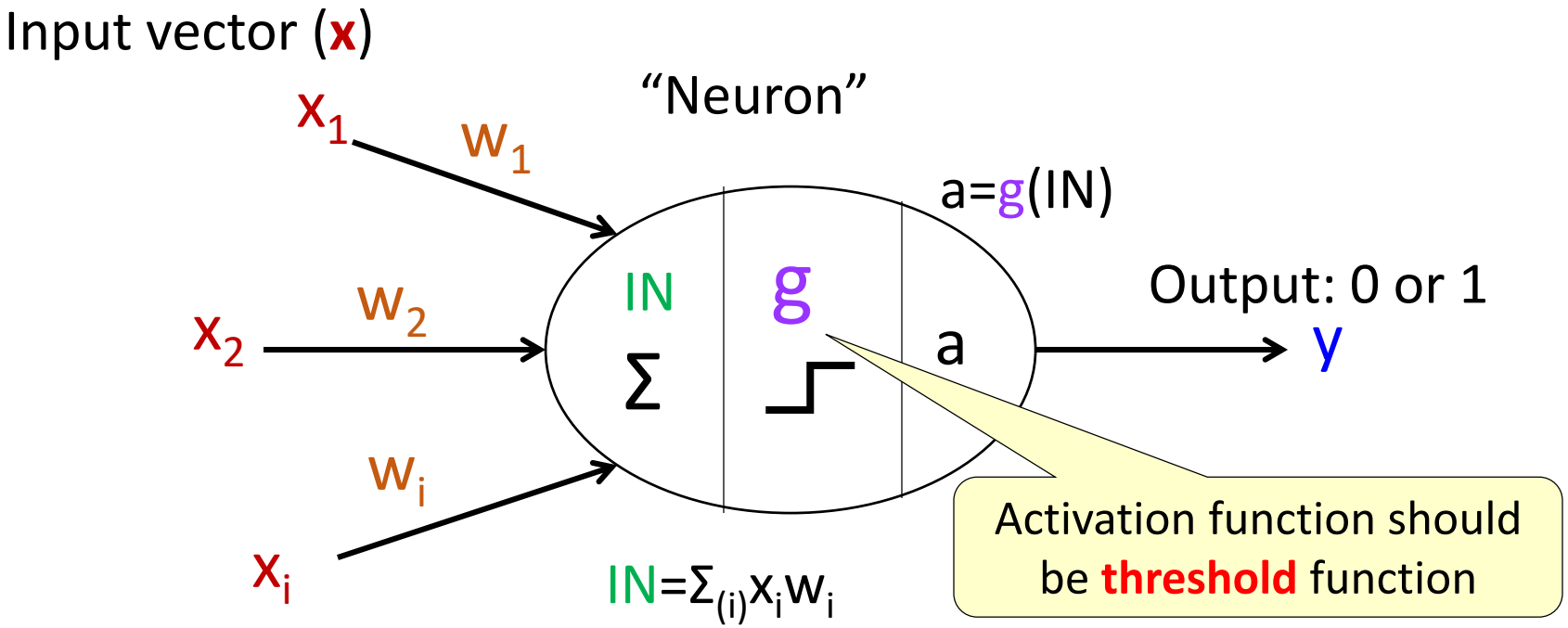
- The summation function IN sums all the signals from the input vector multiplied by weights, and feeds the result into activation function *g*.

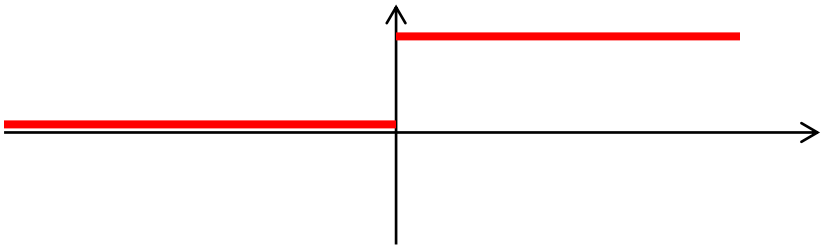
Terminology



- The activation function $g(\cdot)$ is a mathematical function that describes the firing of the neuron as a response to the weighted inputs.
- As in real brain, this is a threshold function: neuron either fires, or not.

Terminology

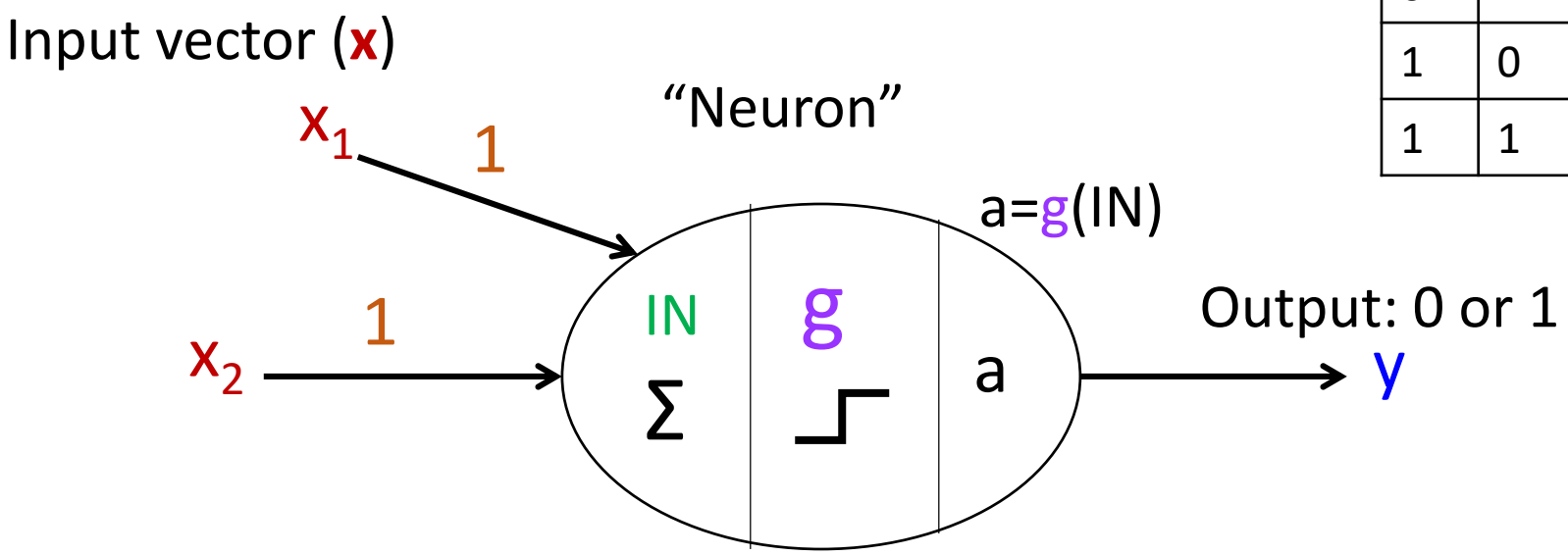




The simplest threshold function: *sign*
g(x)=0 if x<=0
g(x)=1 if (x>0) (neuron fires)

Neuron for OR function

x1	x2	y
0	0	0
0	1	1
1	0	1
1	1	1



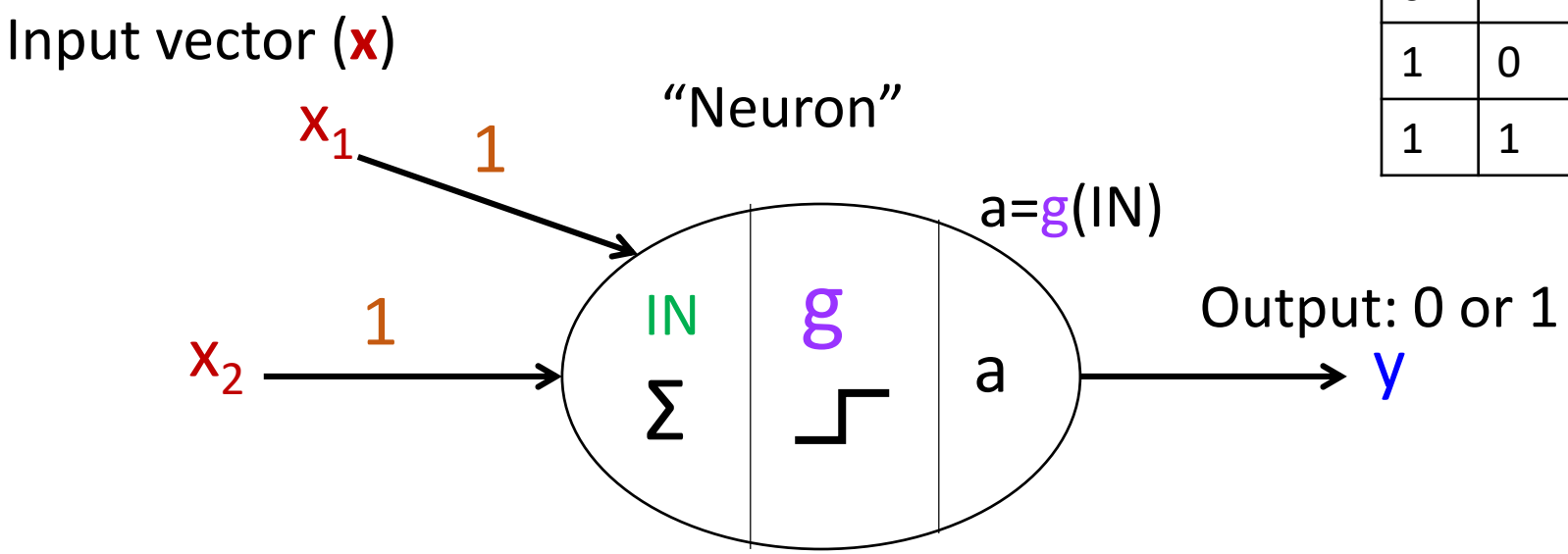
$IN=\Sigma_{(i)}x_i w_i$

g(x)=0 if x<=0
g(x)=1 if (x>0)

x1	x2	IN	g	y
0	0	0	0	0
0	1	1	1	1
1	0	1	1	1
1	1	2	1	1

Neuron for AND function

x1	x2	y
0	0	0
0	1	0
1	0	0
1	1	1



$IN = \Sigma_{(i)} x_i w_i$

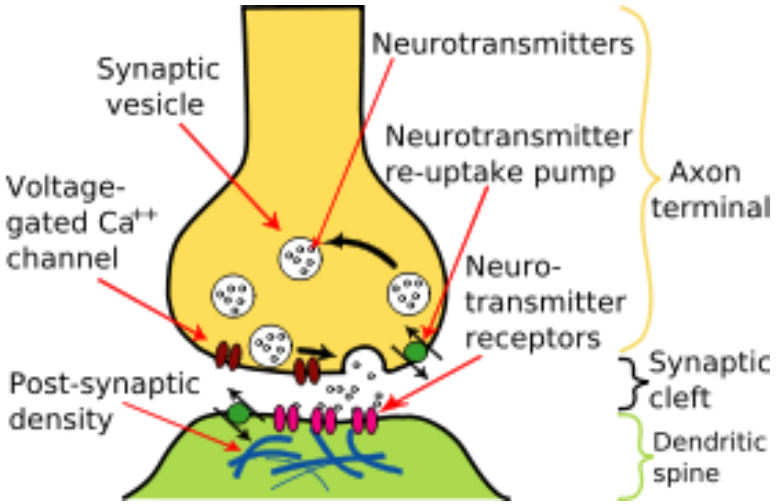
g(x)=0 if x<=**1**
g(x)=1 if (x>**1**)

Just changed the threshold for firing from 0 to 1

x1	x2	IN	g	y
0	0	0	0	0
0	1	1	0	0
1	0	1	0	0
1	1	2	1	1

How do we learn: brain

- Hebbian theory: “Cells that fire together wire together”
- Persistent changes in molecular structures alter synaptic transmission between neurons
- This corresponds to changing weights in Neural Network



Neuron with learning capabilities: Perceptron (Rosenblatt, 1958)

- The network can learn its own weights.
- It is presented with a set of inputs and predefined outputs.
- The actual output is different from the predefined output by some error.
- Adjust the connection weights to produce a smaller error.

Teaching perceptron the concept of AND

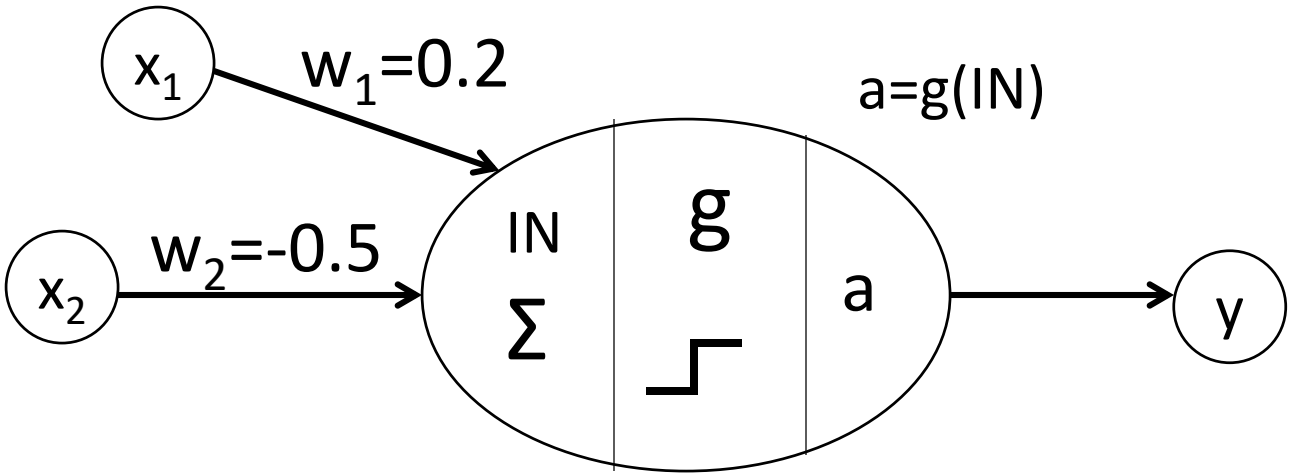
Consider the following simple labeled dataset:

grandma	holiday	present
X1	X2	Y (Class)
0	0	0
0	1	0
1	0	0
1	1	1

Each data record has 2 attributes –X1 and X2, and the record is classified into a binary class

We want to train Perceptron so it will be able to predict the correct label based on the value of X1, X2.

Learning algorithm: example



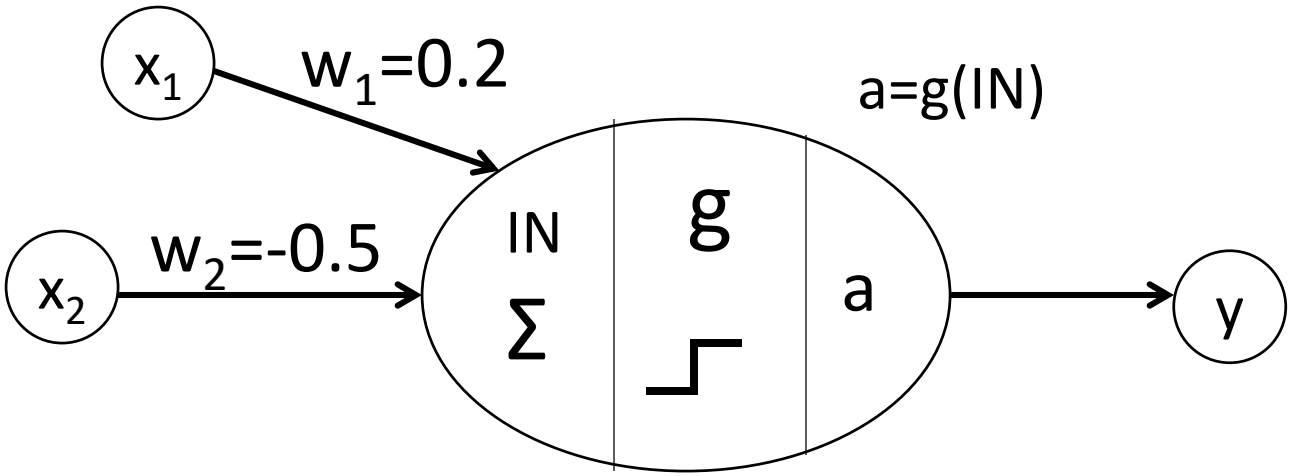
Dataset

X1	X2	Y (Class)
0	0	0
0	1	0
1	0	0
1	1	1

initialize the weights to random values: for example W = [0.2, -0.5]
total error E = ∞
while E != 0:
    E = 0
    for each record
        present network with input vector
        compute output y according to g
        for each x_i, w_i compute classification error $\Delta w_i = (t - y)*x_i$
        increase or decrease w_i to get closer to the target
        E += Δw_i

Learning step: example

X1	X2	Y (Class)
0	0	0
0	1	0
1	0	0
1	1	1



showing vector [1,1]
IN = 0.2 – 0.5
G(IN) < 0 → y=0 (Neuron does not fire)
Our target t = 1
Δ w1 = (t – y)*x1 = 1
We need to increase w1 to get closer to the desired target t

Training Perceptron: learning rate

$\Delta = T - Y$

T – desired output (*target*)

Y – actual output

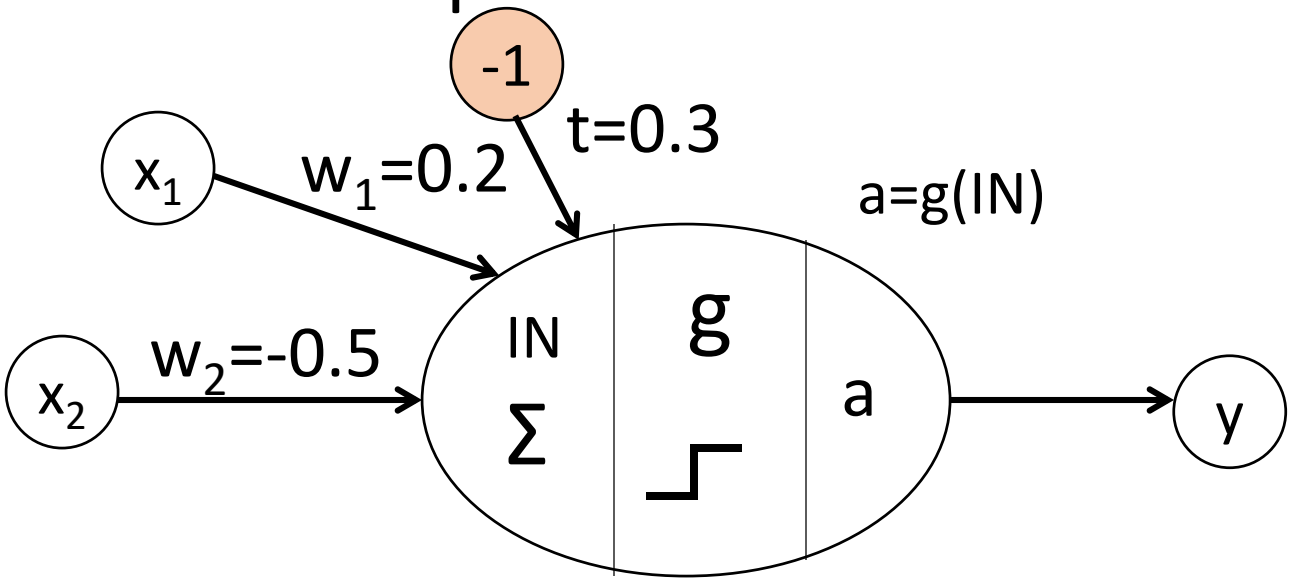
The delta rule:

$w_i \leftarrow w_i + \eta \times x_i \times \Delta$

But do not adjust by the entire value of error, just move slightly into desired direction

η (eta) represents the "learning rate" – the speed with which we move in the direction of the target

Bias input

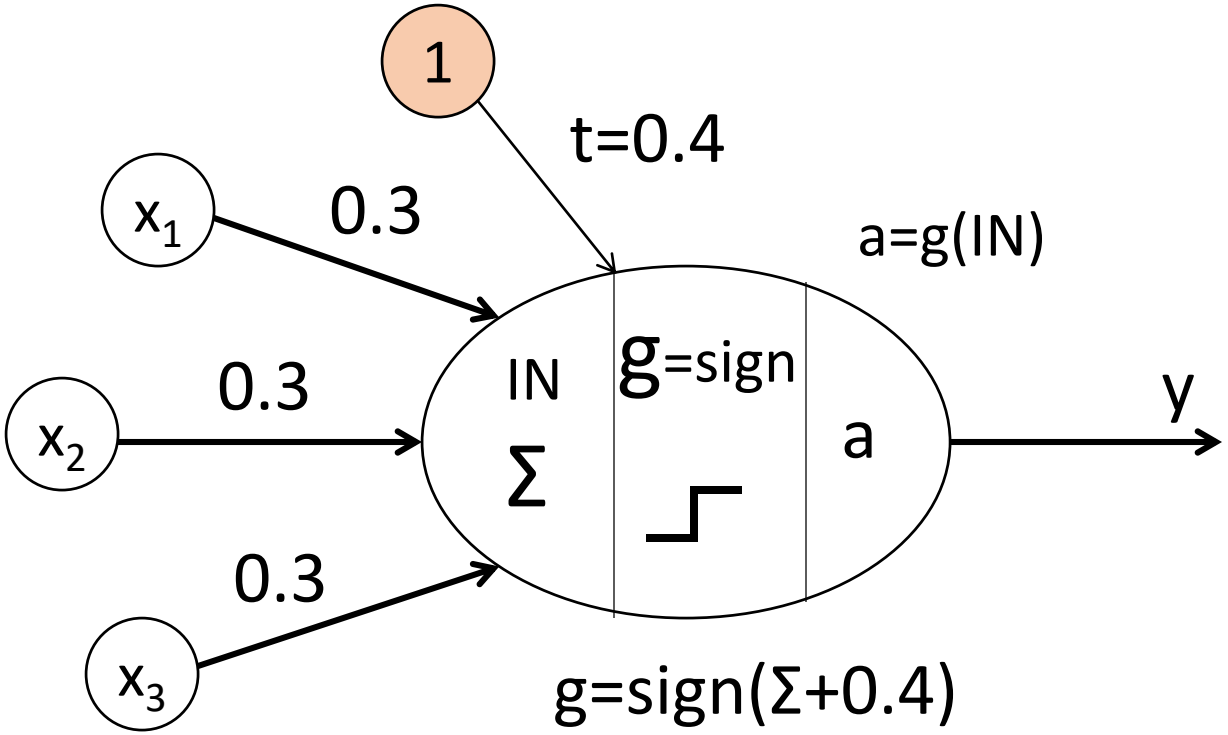


X1	X2	Y (Class)
0	0	0
0	1	0
1	0	0
1	1	1

showing vector [0,0]
IN = 0

- No matter how we adjust the weights the result never changes!
- To avoid this situation we add a "bias" node x_0 with the constant value (for example -1), so we could adjust its weight w_0 to move the value of y closer to t in case that all other values in the input vector are 0

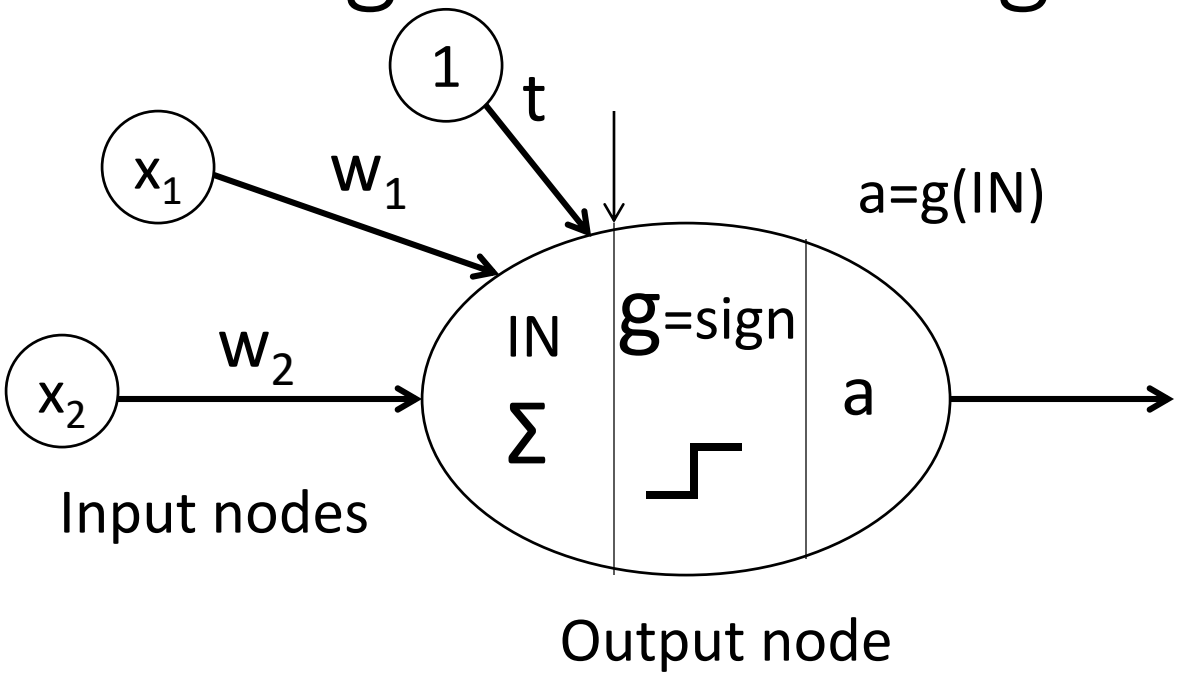
Using bias input: example



x1	x2	x3	y
1	0	0	-1
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	-1
0	1	0	-1
0	1	1	1
0	0	0	-1

y=sign(w1x1+w2x2+w3x3+t)

The goal of training



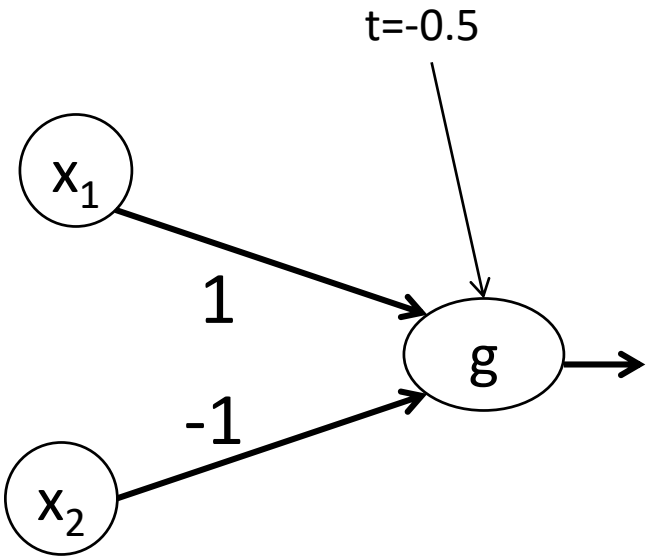
The output node gets activated only if $\Sigma x_i w_i + t > 0$

In 2D this can be expressed as points above and below the line: $w_1 x_1 + w_2 x_2 + t$

In *N* dimensions – it is a hyperplane, which separates all positive examples from negative examples

Objective of Perceptron learning:
To determine the optimal values of weights to separate all labeled instances by a hyperplane

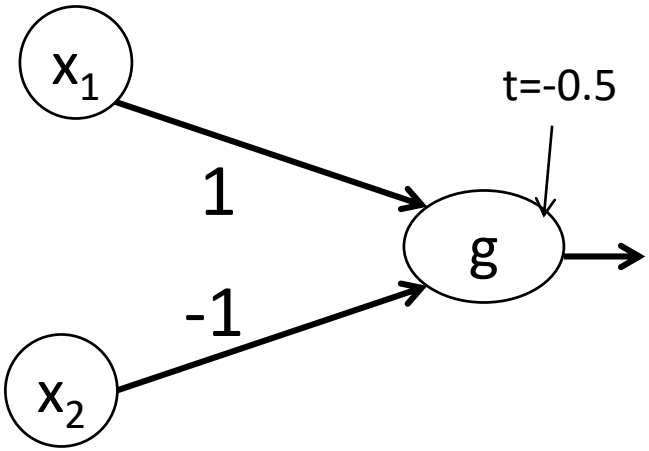
Perceptron learned AND NOT



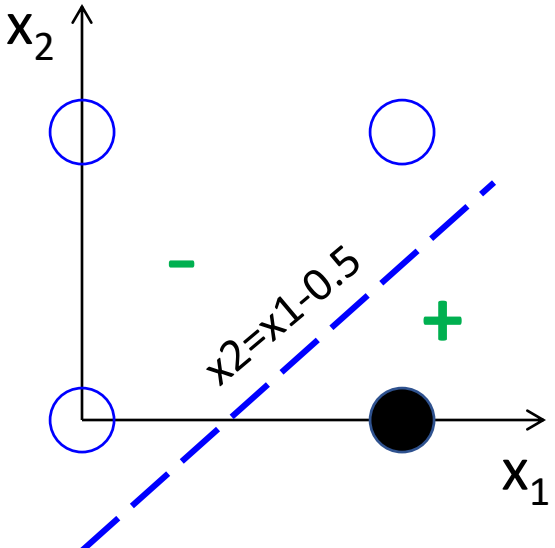
y= x1 AND NOT x2		
x1	x2	y
0	0	<0
0	1	<0
1	0	≥0
1	1	<0

y=x1w1+x2w2+t
Let t=-0.5, w1=1, w2=-1
y(0,0)=-0.5
y(0.1)=-1.5
y(1.0)=0.5
y(1.1)=-0.5

This means:
Perceptron found a separating line

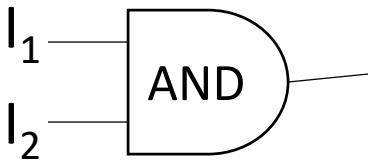


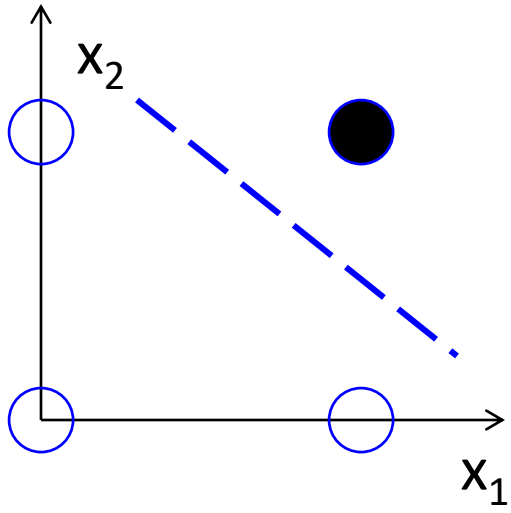
y= x1 AND NOT x2		
x1	x2	y
0	0	<0
0	1	<0
1	0	≥0
1	1	<0

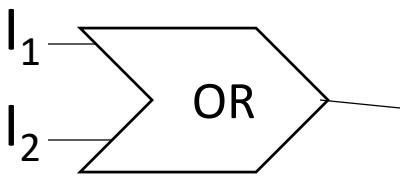


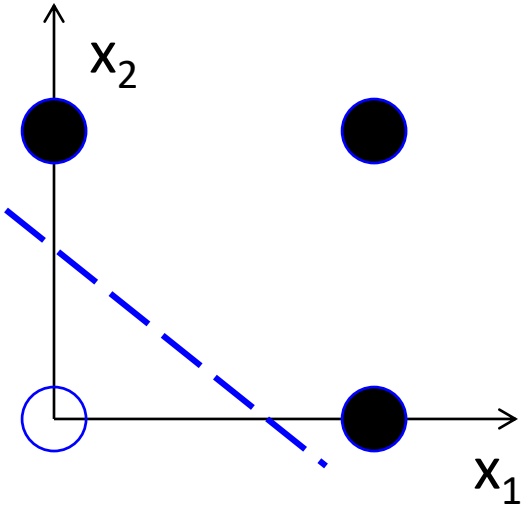
y=x1w1+x2w2+t
t=-0.5, w1=1, w2=-1
x1-x2-0.5=0
x2=x1-0.5

Perceptron can learn only linearly-separable functions



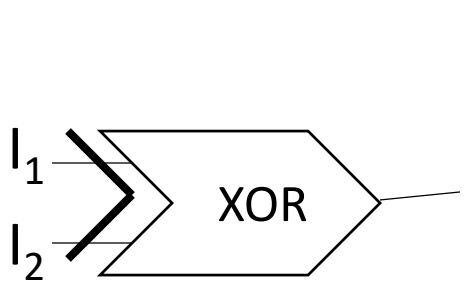


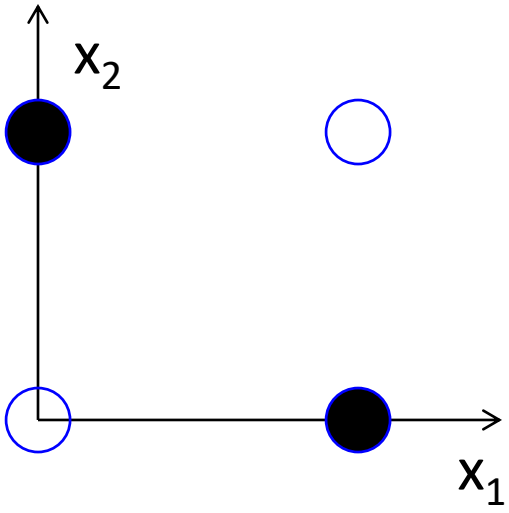




Experiment with *perceptron.py*

Non linearly-separable: *exclusive OR* (*XOR*)



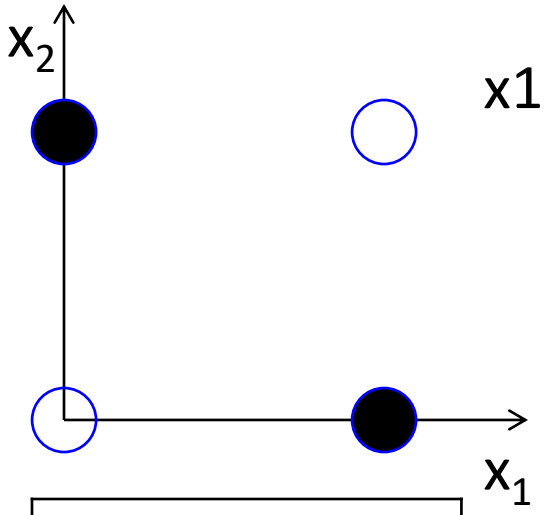


XOR table

x1	x2	z
0	0	0
0	1	1
1	0	1
1	1	0

One possible solution – add more neurons

Adding neuron z



x1 XOR x2 = x1 OR x2 AND NOT (x1 AND x2)

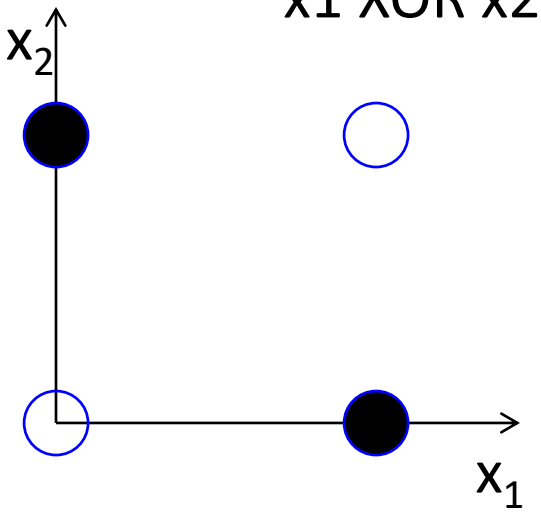
y1=x1 OR x2		
x1	x2	y1
0	0	0
0	1	1
1	0	1
1	1	1

y2=not (x1 AND x2)		
x1	x2	y2
0	0	1
0	1	1
1	0	1
1	1	0

z=y1 AND y2		
y1	y2	z
0	1	0
1	1	1
1	1	1
1	0	0

Combining outputs of two perceptrons

x1 XOR x2 = x1 OR x2 AND NOT (x1 AND x2)



2 small perceptrons will be connected to the third, which will combine their values

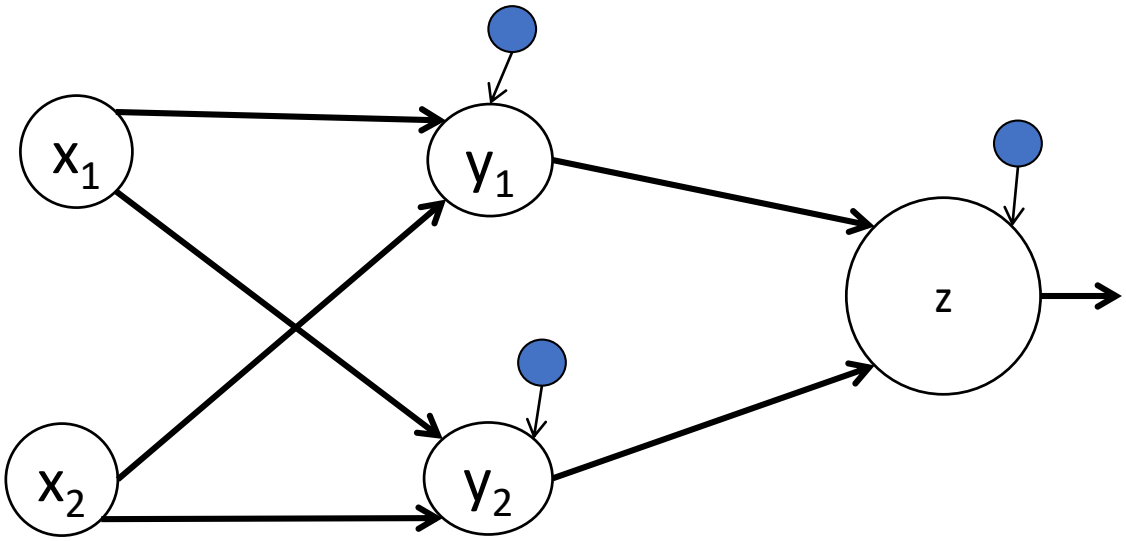
y1=x1 OR x2		
x1	x2	y1
0	0	0
0	1	1
1	0	1
1	1	1

y2=not (x1 AND x2)		
x1	x2	y2
0	0	1
0	1	1
1	0	1
1	1	0

z=y1 AND y2		
y1	y2	z
0	1	0
1	1	1
1	1	1
1	0	0

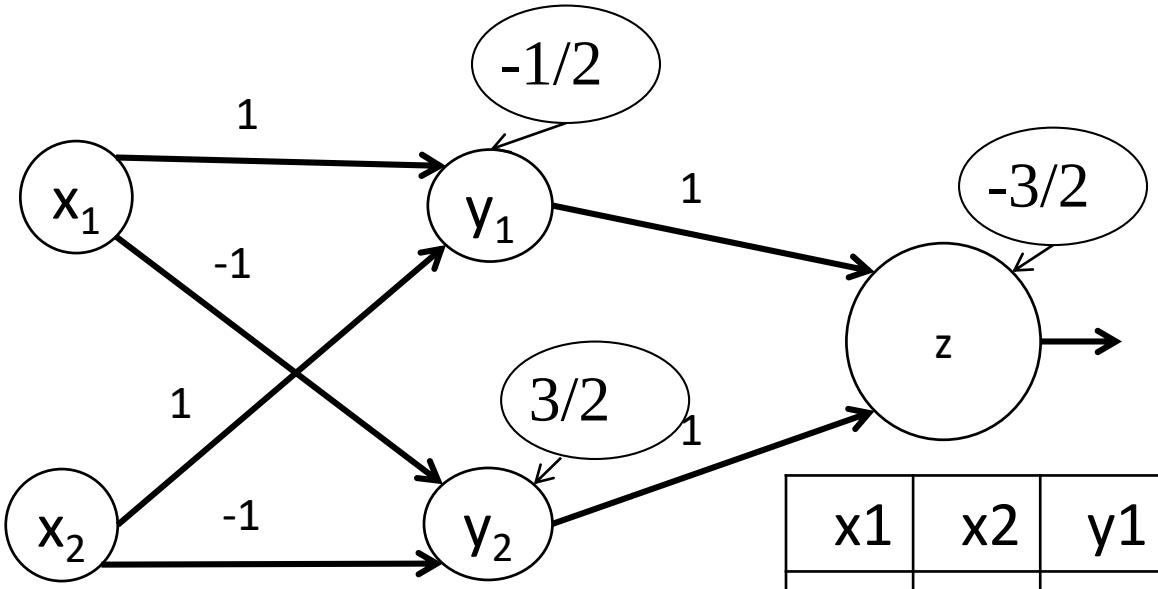
XOR ANN topology

x1 XOR x2 = x1 OR x2 AND NOT (x1 AND x2)



XOR ANN: weights

x1 XOR x2 = x1 OR x2 AND NOT (x1 AND x2)



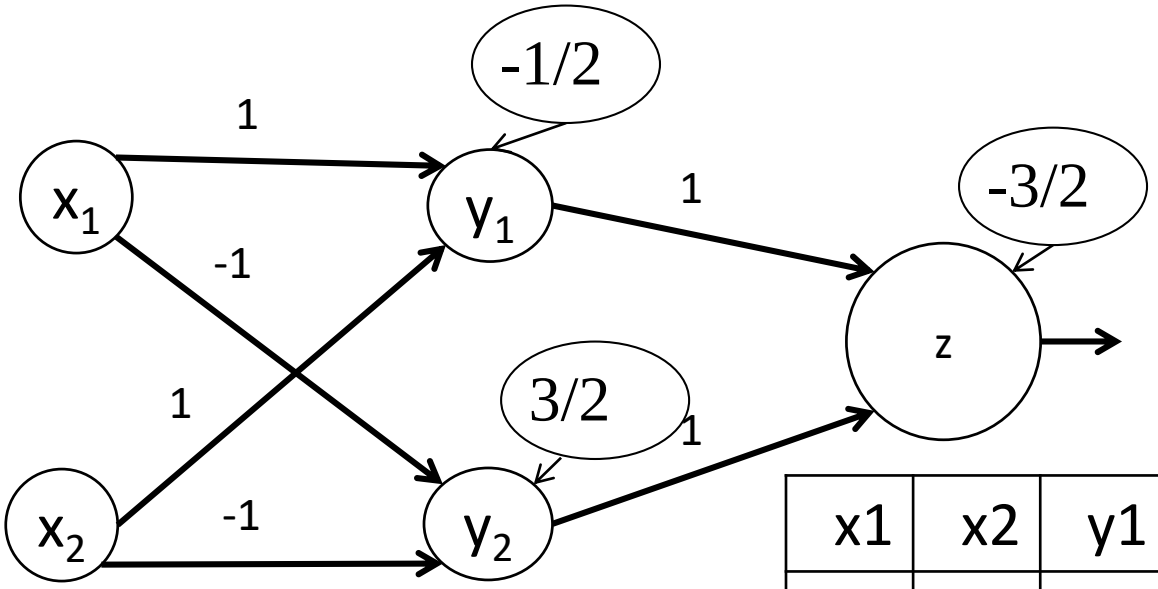
x1	x2	y1	y2	z
0	0	-3/20	1	
0	1	1/21	1	
1	0	1/21	1	
1	1	3/21	0	

Threshold is 0

g(x)=0 if x<=**0**
g(x)=1 if (x>**0**)

XOR ANN: y1

x1 XOR x2 = x1 OR x2 AND NOT (x1 AND x2)



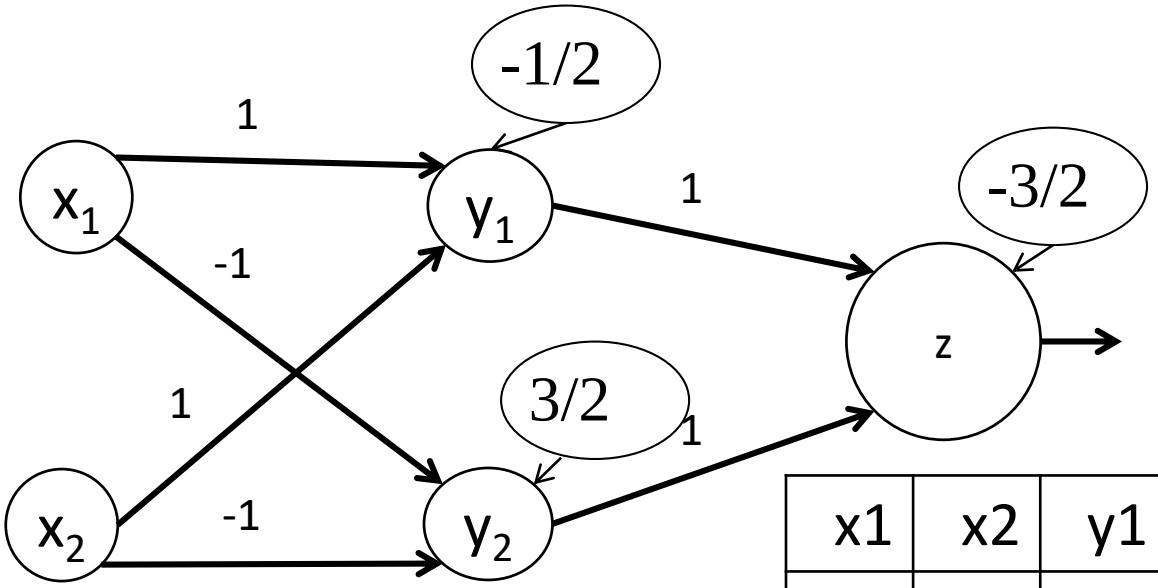
x1	x2	y1	y2	z
0	0	-1/2 0	1	
0	1	1/2 1	1	
1	0	1/2 1	1	
1	1	3/2 1	0	

Threshold is 0

g(x)=0 if x<=**0**
g(x)=1 if (x>**0**)

XOR ANN: y2

x1 XOR x2 = x1 OR x2 AND NOT (x1 AND x2)



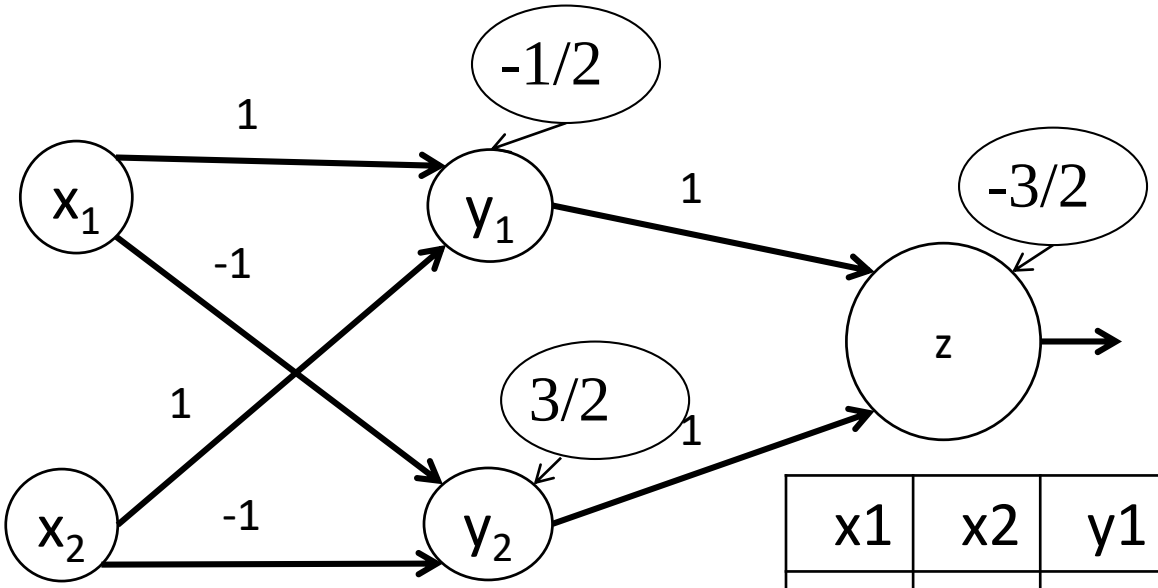
x1	x2	y1	y2	z
0	0	-1/20	3/21	
0	1	1/21	1/21	
1	0	1/21	1/21	
1	1	3/21	-1/20	

Threshold is 0

g(x)=0 if x<=**0**
g(x)=1 if (x>**0**)

XOR ANN: z

x1 XOR x2 = x1 OR x2 AND NOT (x1 AND x2)



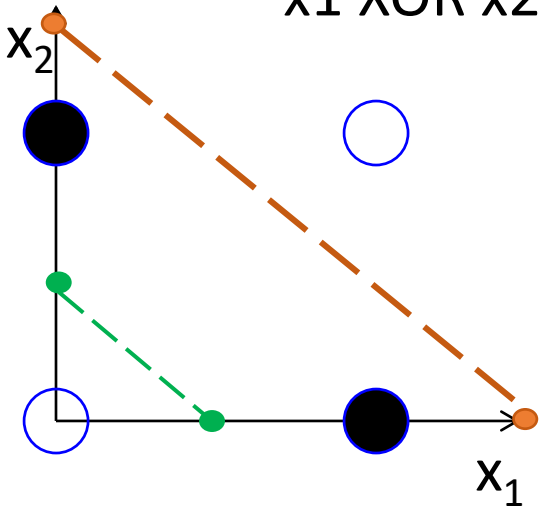
x1	x2	y1	y2	z
0	0	-1/20	3/21	-1/20
0	1	1/21	1/21	1/21
1	0	1/21	1/21	1/21
1	1	3/21	-1/20	-1/20

Threshold is 0

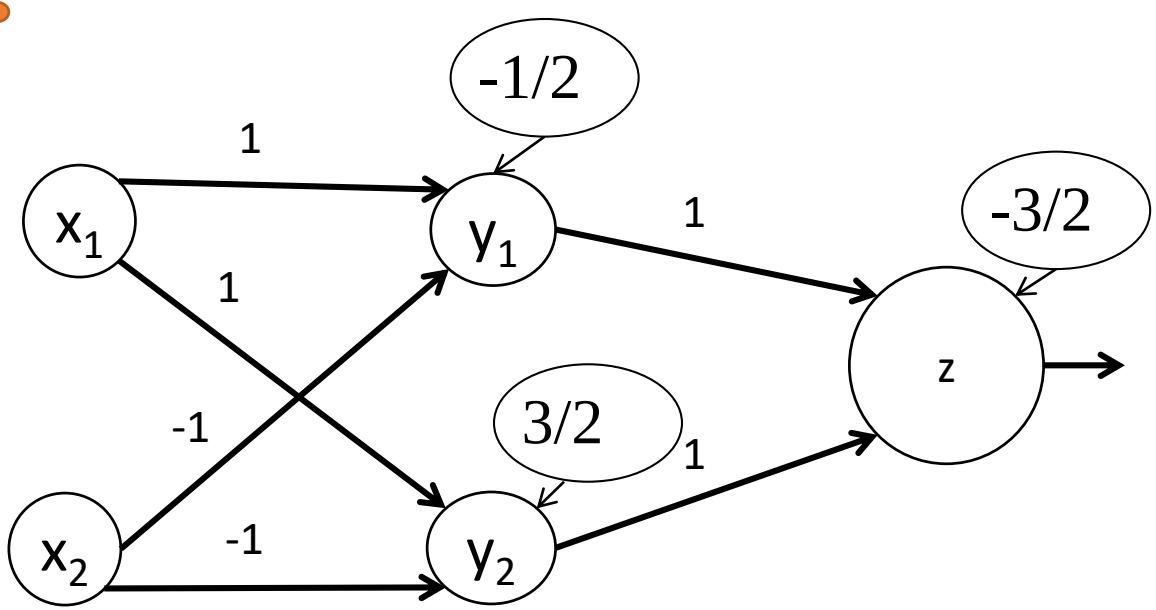
g(x)=0 if x<=**0**
g(x)=1 if (x>**0**)

Separating with 2 linear separators

x1 XOR x2 = x1 OR x2 AND NOT (x1 and x2)

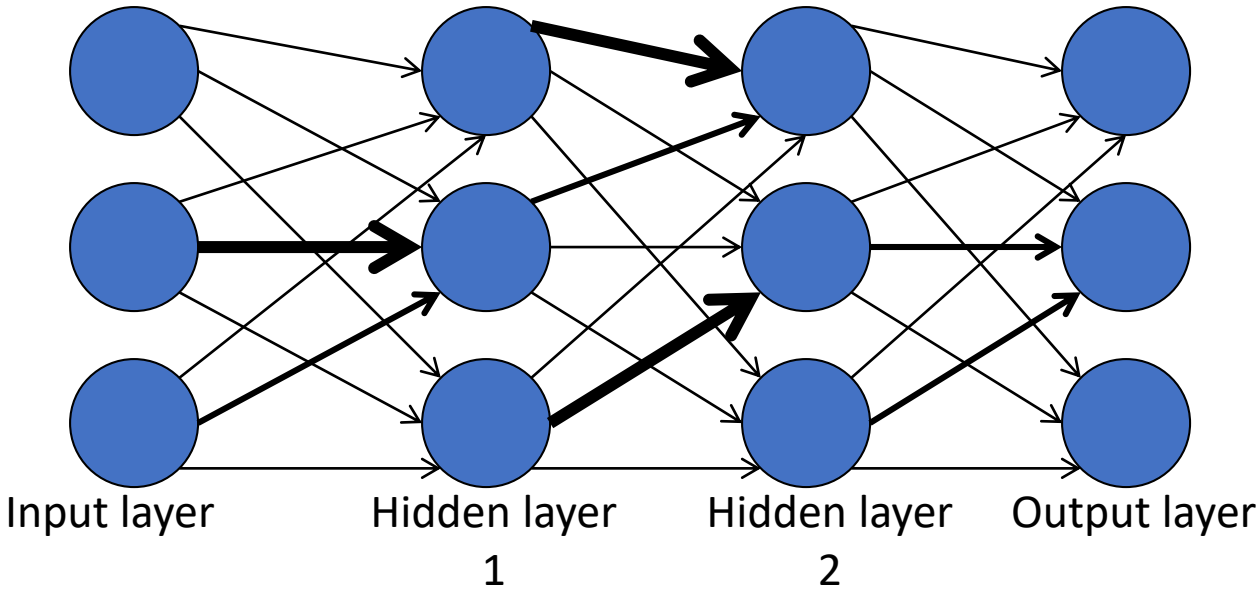


y1=x1+x2-1/2
y2=-x1-x2+3/2

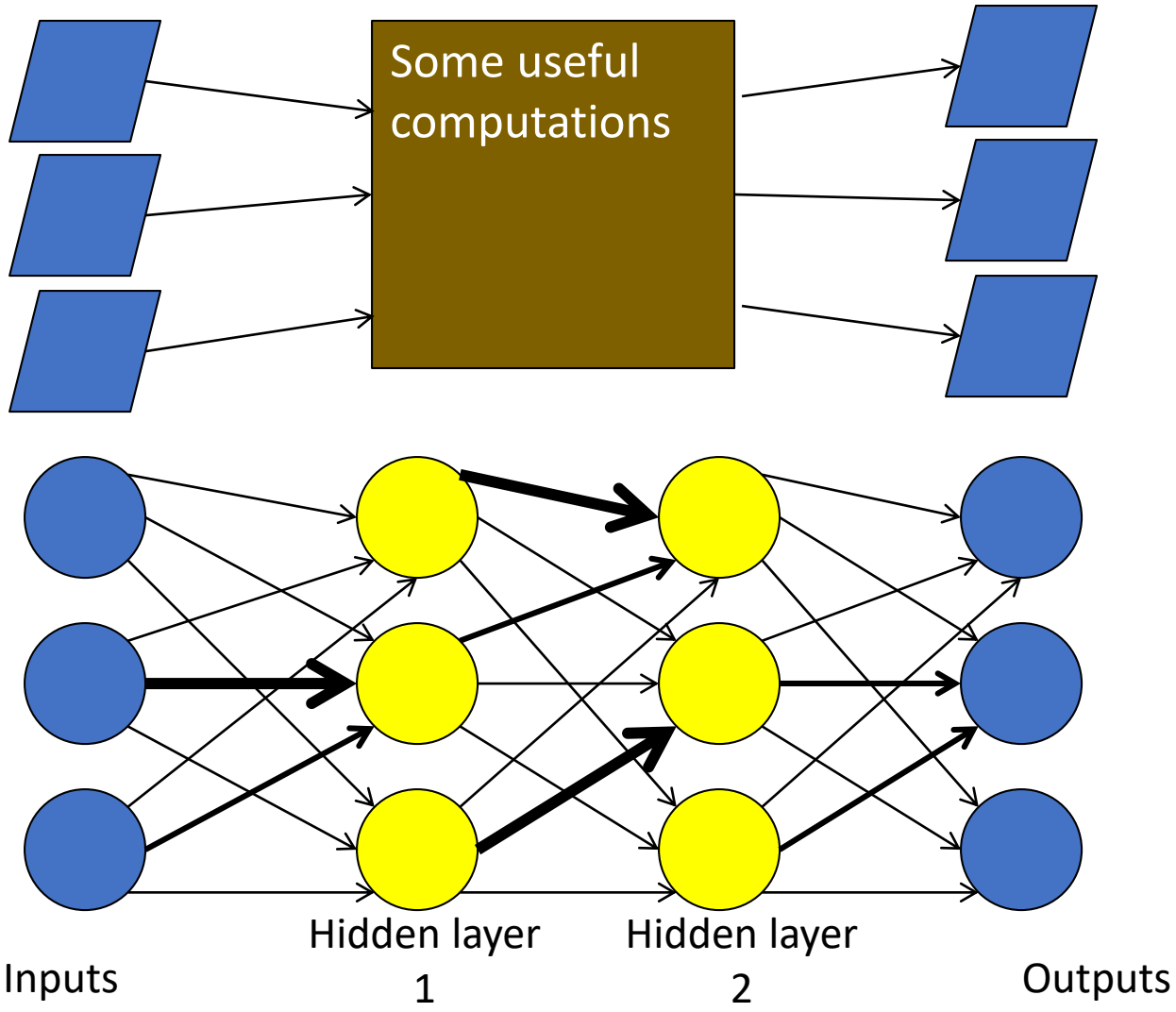


Multi-layer Perceptron

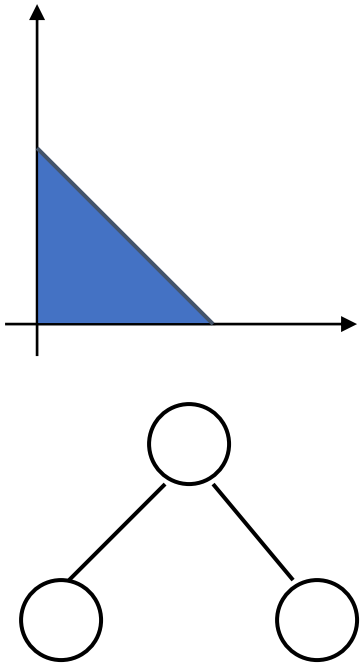
- Added: *hidden nodes*
- Nodes are organized into layers. Edges are directed and carry weight
- No connections inside the layer



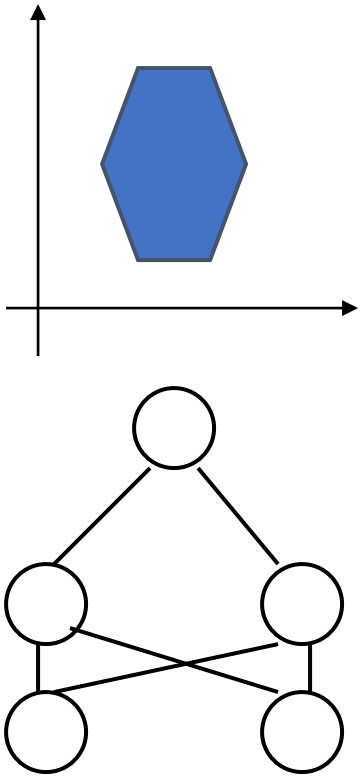
Multi-layer Perceptron vs. regular computing



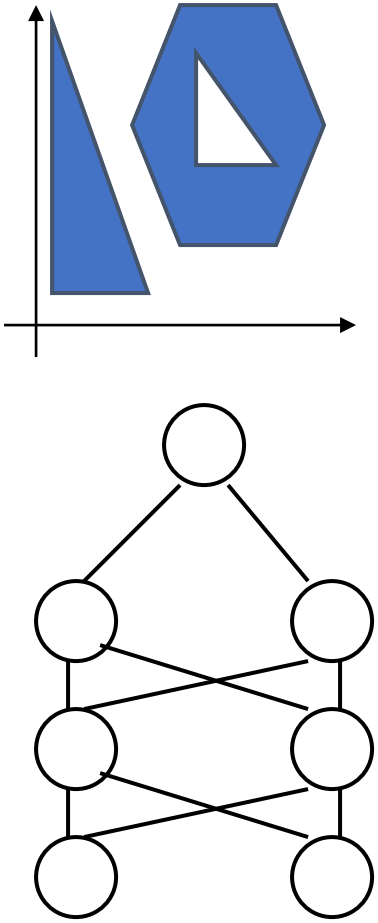
What do we gain from the extra layers



1st layer draws linear boundaries



2nd layer combines the boundaries



3rd layer can generate arbitrarily complex boundaries

Phases of learning

- Training the MLP consists of two parts:
  - Working out what the outputs are for the given inputs and the current weights – Forward phase
  - Updating the weights according to the error, which is a function of the difference between the outputs and the targets – Backward phase

Going forward did not change

- We start at the left by filling in the values for the input vector
- We then use the input values and the first level of weights to calculate the activations of each neuron in the hidden layer
- Then we use those activations and the next set of weights to calculate the activations of the output layer
- Now that we've got the outputs of the network, we can compare them to the targets and compute the error

Learning weights in 3-layer networks: from hidden to output

- From the delta rule, we know how to adjust weights between the output and the hidden layer
- But if we only apply this rule, the weights from input to hidden units *never change*!

- **We do not have the value of error for hidden units**

So how do we adjust weights between input and hidden layer?

Backpropagation learning algorithm ‘BP’

Rumelhart, Hinton, Williams, McClelland (1986)

Forward pass phase: computes ‘functional signal’, feedforward propagation of input pattern signals through network

Backward pass phase: computes ‘error signal’, *propagates* the error *backwards* through network starting at output units (where the error is the difference between actual and desired output values)

We need a new error function

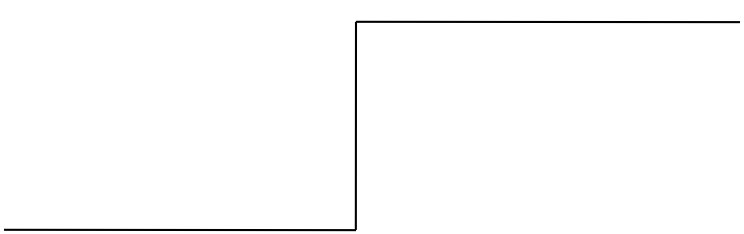
- We now compute the total error of the network using

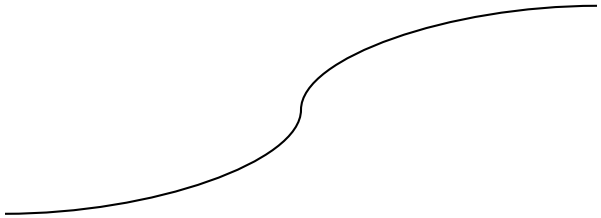
$$E(t, y) = \frac{1}{2} \sum_{k=1}^{N} (y_k - t_k)^2$$

- If we differentiate an error function with respect to each weight, we get the gradient of the error.
- Since the purpose of learning is to minimize the error, following the error function downhill (in other words, in the direction of the negative gradient) will give us what we want.
- This is called “gradient descent”

We need a new activation function

- We also need to change the activation functions to something which is differentiable
- We can use more complex non-linear functions: sigmoidal functions

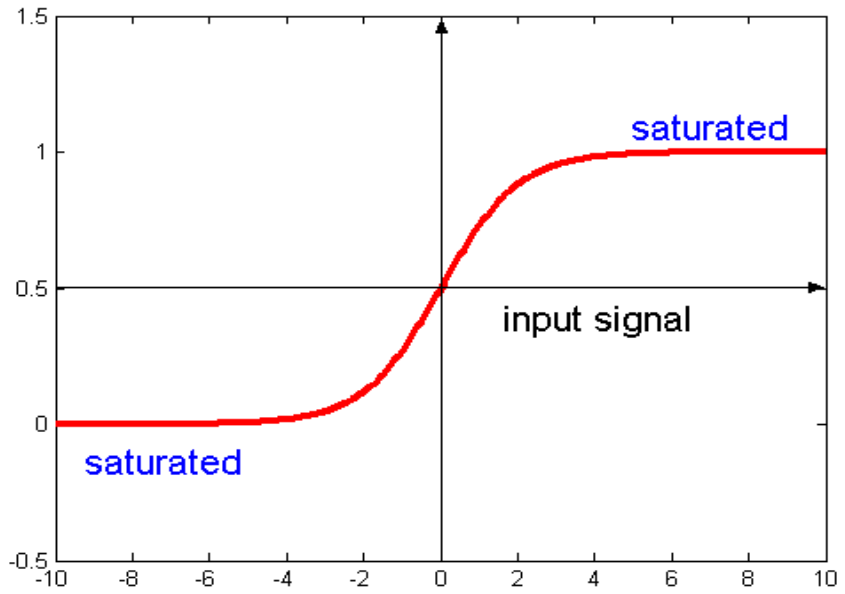




Non-linear activation functions

$$g(a_i(t)) = \frac{1}{1+\exp(-k a_i(t))} = \frac{1}{1+e^{-k a_i(t)}}$$

where k is a positive constant



The sigmoidal function gives a value in range of 0 to 1.

Alternatively can use tanh(ka) which has the same shape but in range -1 to 1.

Note: when IN = 0, f = 0.5

Backpropagation: intuition

- The output nodes tell to hidden nodes that there was an error
- The hidden nodes need to decide how to adjust their weights to decrease an error
- Each hidden node needs to calculate its own error to back-propagate it to the input layer

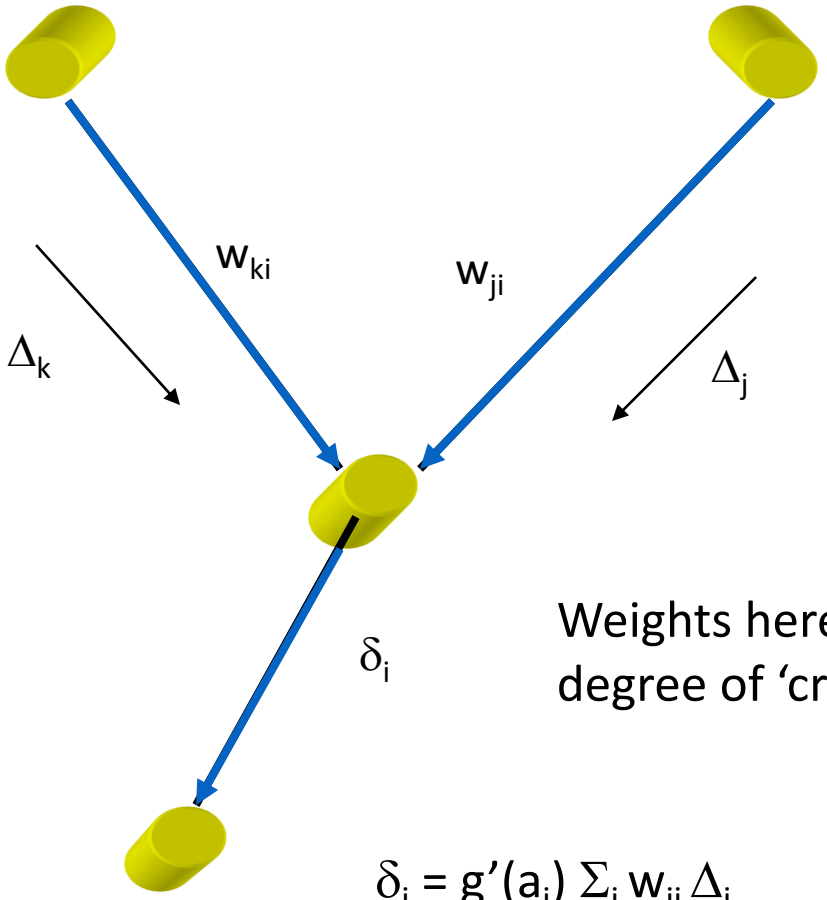
Backpropagation: intuition

- The node calculates its own error (by taking partial derivative of error function by its weight) and pushes it back to the input layer nodes, which need to adjust their weights
- The idea is to find out which of the connections is the most to blame for the error and to adjust its outgoing weight more

Learning weights in 3-layer networks: distributing credit (blame)

- The goal is to distribute error from an output node to all the hidden units connected to it, weighted by this connection.
- i.e. a hidden unit receives a delta from each output unit weighted with (=multiplied by) the weight of the connection between these units.

Backward Pass

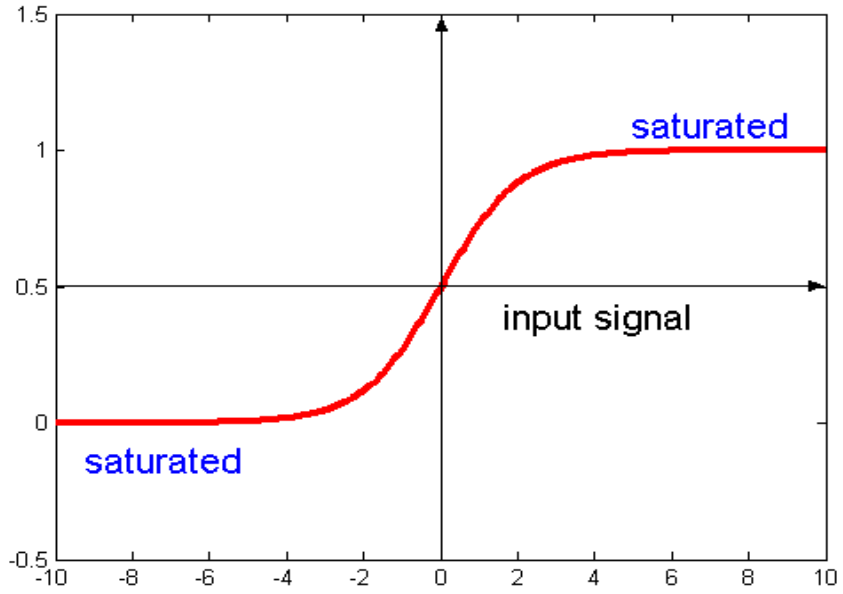


Weights here can be viewed as providing degree of 'credit' or 'blame' to hidden units

$$\delta_i = g'(a_i) \Sigma_j w_{ji} \Delta_j$$

Weight adjustment for non-linear activation functions

$$g(a_i(t)) = \frac{1}{1+\exp(-k a_i(t))} = \frac{1}{1+e^{-k a_i(t)}}$$



Derivative of activation function

$$\Delta_i(t) = (d_i(t) - y_i(t))g'(a_i(t))$$

BP algorithm

1. Apply an input vector (training record) and calculate all activation functions, the output and the error
2. Evaluate Δ_k for all output units via:

$$\Delta_i(t) = (d_i(t) - y_i(t))g'(a_i(t))$$

(Note similarity to perceptron learning algorithm)

3. Backpropagate Δ_ks to get error terms δ for hidden layers using:

$$\delta_i(t) = g'(u_i(t))\sum_k \Delta_k(t) w_{ki}$$

4. Change the weights from inputs to hidden layer and from hidden layer to outputs using:

$$v_{ij}(t+1) = v_{ij}(t) + \eta\delta_i(t)x_j(t)$$

$$w_{ij}(t+1) = w_{ij}(t) + \eta\Delta_i(t)z_j(t)$$

BP algorithm

1. Apply an input vector (training record) and calculate all activation functions, the output and the error
2. Evaluate Δ_k for all output units via:

t for time – training epoch

$$\Delta_i(t) = (d_i(t) - y_i(t))g'(a_i(t))$$

(Note similarity to perceptron learning algorithm)

3. Backpropagate Δ_ks to get error terms δ for hidden layers using:

$$\delta_i(t) = g'(u_i(t))\sum_k \Delta_k(t)w_{ki}$$

4. Change the weights from inputs to hidden layer and from hidden layer to outputs using:

$$v_{ij}(t+1) = v_{ij}(t) + \eta\delta_i(t)x_j(t)$$

$$w_{ij}(t+1) = w_{ij}(t) + \eta\Delta_i(t)z_j(t)$$

BP algorithm

1. Apply an input vector (training record) and calculate all activation functions, the output and the error
2. Evaluate Δ_k for all output units via:

d for desired output – target value

$$\Delta_i(t) = (d_i(t) - y_i(t))g'(a_i(t))$$

(Note similarity to perceptron learning algorithm)

3. Backpropagate Δ_ks to get error terms δ for hidden layers using:

$$\delta_i(t) = g'(u_i(t))\sum_k \Delta_k(t) w_{ki}$$

4. Change the weights from inputs to hidden layer and from hidden layer to outputs using:

$$v_{ij}(t+1) = v_{ij}(t) + \eta\delta_i(t)x_j(t)$$

$$w_{ij}(t+1) = w_{ij}(t) + \eta\Delta_i(t)z_j(t)$$

BP algorithm

1. Apply an input vector (training record) and calculate all activation functions, the output and the error
2. Evaluate Δ_k for all output units via:

Derivative of the activation function for output node *i*

$$\Delta_i(t) = (d_i(t) - y_i(t))g'(a_i(t))$$

(Note similarity to perceptron learning algorithm)

3. Backpropagate Δ_ks to get error terms δ for hidden layers using:

$$\delta_i(t) = g'(u_i(t))\sum_k \Delta_k(t) w_{ki}$$

4. Change the weights from inputs to hidden layer and from hidden layer to outputs using:

$$v_{ij}(t+1) = v_{ij}(t) + \eta\delta_i(t)x_j(t)$$

$$w_{ij}(t+1) = w_{ij}(t) + \eta\Delta_i(t)z_j(t)$$

BP algorithm

1. Apply an input vector (training record) and calculate all activation functions, the output and the error
2. Evaluate Δ_k for all output units via:

$$\Delta_i(t) = (d_i(t) - y_i(t))g'(a_i(t))$$

(Note similarity to perceptron learning algorithm)

3. Backpropagate Δ_ks to get error terms δ for hidden layers using:

$$\delta_i(t) = g'(u_i(t))\sum_k \Delta_k(t)w_{ki}$$

Derivative of the activation function a hidden node *i*

4. Change the weights from inputs to hidden layer and from hidden layer to outputs using:

$$v_{ij}(t+1) = v_{ij}(t) + \eta\delta_i(t)x_j(t)$$

$$w_{ij}(t+1) = w_{ij}(t) + \eta\Delta_i(t)z_j(t)$$

BP algorithm

1. Apply an input vector (training record) and calculate all activation functions, the output and the error
2. Evaluate Δ_k for all output units via:

$$\Delta_i(t) = (d_i(t) - y_i(t))g'(a_i(t))$$

(Note similarity to perceptron learning algorithm)

3. Backpropagate Δ_ks to get error terms δ for hidden layers using:

$$\delta_i(t) = g'(u_i(t))\sum_k \Delta_k(t)w_{ki}$$

That will give the proportional error for each neuron in hidden layer: the degree of blame

4. C eights from inputs to hidden layer and from hidden lay using:

$$v_{ij}(t+1) = v_{ij}(t) + \eta\delta_i(t)x_j(t)$$

$$w_{ij}(t+1) = w_{ij}(t) + \eta\Delta_i(t)z_j(t)$$

BP algorithm

1. Apply an input vector (training record) and calculate all activation functions, the output and the error
2. Evaluate Δ_k for all output units via:

$$\Delta_i(t) = (d_i(t) - y_i(t))g'(a_i(t))$$

(Note similarity to perceptron learning algorithm)

3. Backpropagate Δ_ks to get error terms δ for hidden layers using:

$$\delta_i(t) = g'(u_i(t))\sum_k \Delta_k(t)w_{ki}$$

4. Change the weights from inputs to hidden layer and from hidden layer to outputs using:

$$v_{ij}(t+1) = v_{ij}(t) + \eta\delta_i(t)x_j(t)$$

$$w_{ij}(t+1) = w_{ij}(t) + \eta\Delta_i(t)z_j(t)$$

Now when we know the error both for the output nodes and for hidden nodes, we can adjust weights between all 3 layers

Since degree of weight change is proportional to derivative of activation function,

$$\Delta_i(t) = (d_i(t) - y_i(t))g'(a_i(t))$$

weight changes will be greatest when units receive mid-range functional signal and 0 (or very small) on extremes.
This means that by **saturating** a neuron (making the activation large) the weight can be forced to converge: do not change anymore - learned.

$$\delta_i(t) = g'(u_i(t))\sum_k \Delta_k(t) w_{ki}$$

To derive the formula for backpropagation:

- See attached book chapter for mathematical details

Universal Approximation Theorem

How good is a Multi-Layer model?

For any given constant ε and continuous function $h(x_1,\ldots,x_m)$, there exists a three layer ANN with the property that

$$| h(x_1,\ldots,x_m) - H(x_1,\ldots,x_m) | < \varepsilon$$

where $H(x_1,\ldots,x_m) = \sum_{i=1}^{k} a_i f\left(\sum_{j=1}^{m} w_{ij} x_j + b_i\right)$

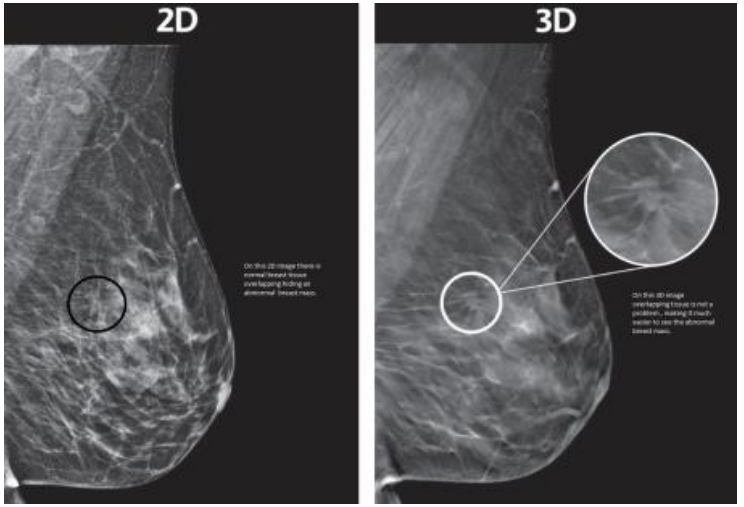
Very powerful model

- With sigmoidal activation functions we can show that a 3-layer net can approximate any function to arbitrary accuracy: property of Universal Approximation
- Proof by thinking of superposition of sigmoids
- Not practically useful as need arbitrarily large number of units but more of an existence proof
- Same is true for a 2-layer net providing function is continuous and from one finite dimensional space to another

- Experiment with *mlp.py*
- See that it can learn the XOR concept easily

Demo: breast cancer diagnosis

- Dataset: https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+(Diagnostic)
- Features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass.
- Diagnosing breast cancer from mammograms is a very hard non-trivial task

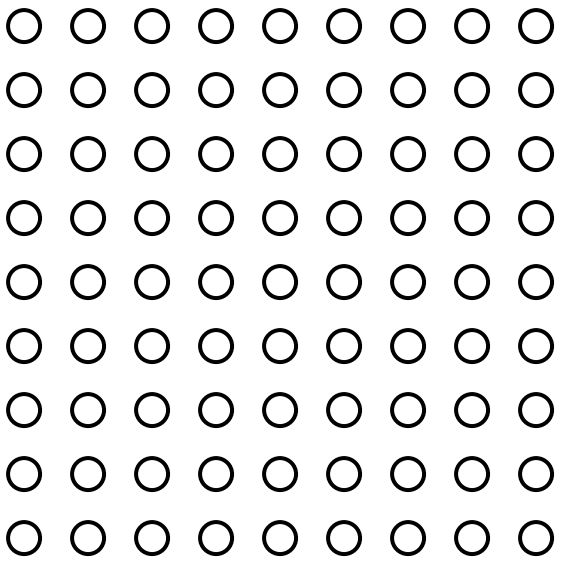


Run *breast_cancer_diagnosis.*py and see how MLP learns to diagnose breast cancer

Applications of ANNs

- Credit card frauds
- Kinect – gesture recognition
- Facial recognition
- Self-driving cars
- …

Example: Handwriting recognition

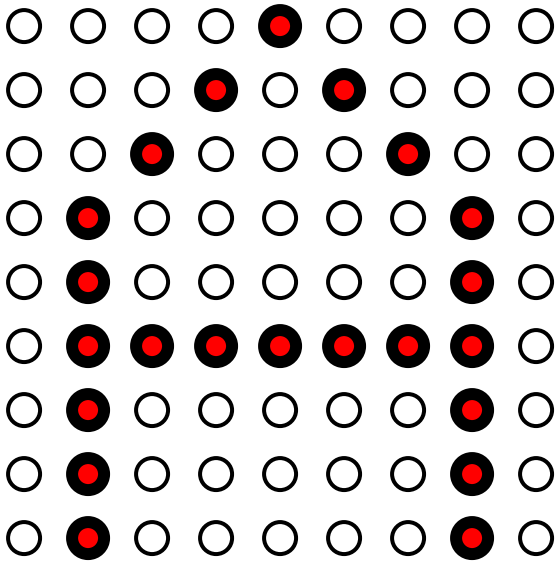


Dataset: collection of handwritings

Attributes: binary values (on-off) of each dot in 2D point matrix

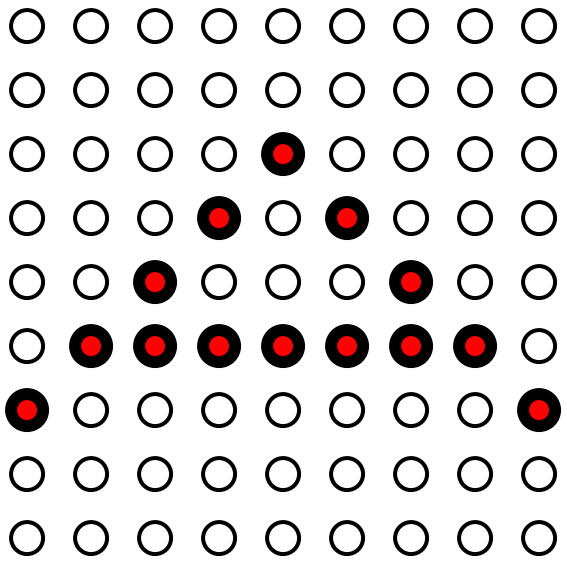
Class: actual letter meant by the writer

Example: Handwriting recognition



Sample training record for class capital letter A

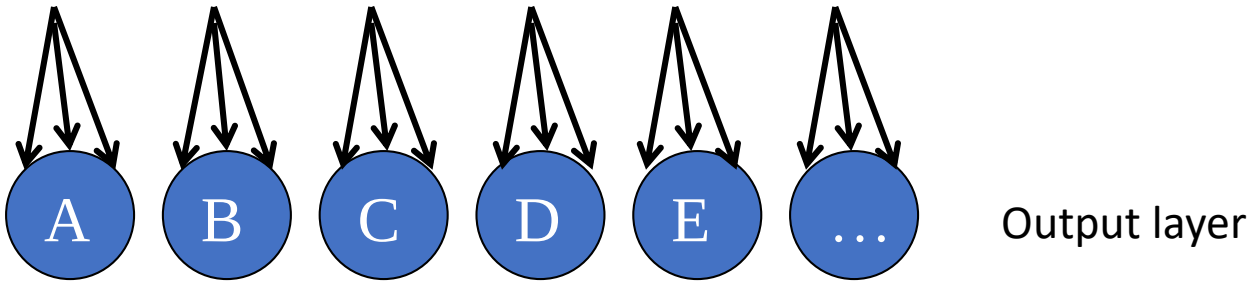
Example: Handwriting recognition



Another training record for class capital letter A

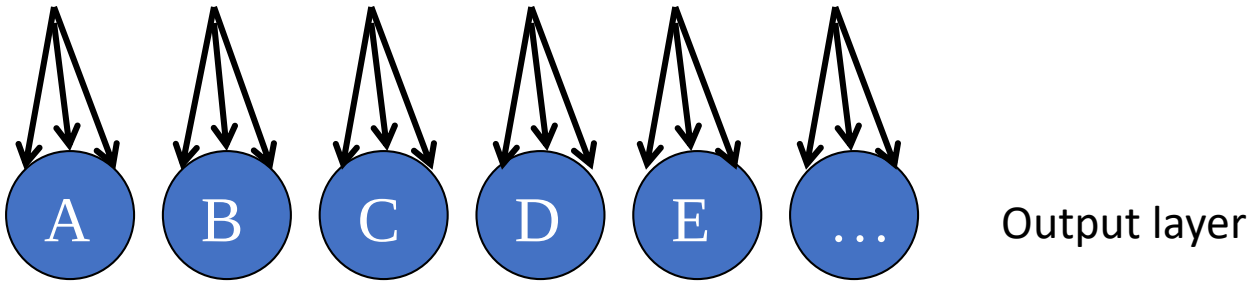
NN for handwriting recognition

- Each dot feeds its value (0 or 1) to a corresponding input neuron
- Each input neuron is connected to the hidden layer
- Each hidden layer neuron is connected to 23 (suppose only for capital English letters) output neurons



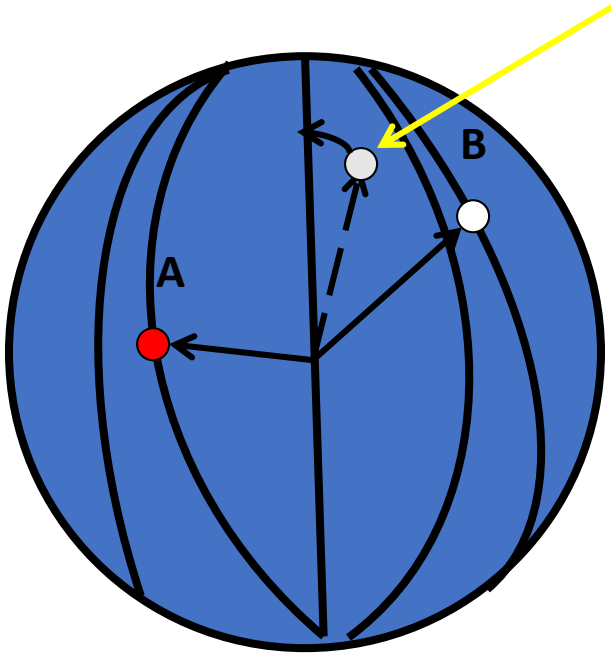
NN for handwriting recognition

- Multi-class problems are solved by competitive learning
- Initially all weights are random, and each output neuron gets some value
- The class is assigned by the letter with maximum value
- The weights are adjusted in such a way that to increase the correct classification, and to decrease the incorrect ones



NN for handwriting recognition

- Each dot is a dimension, and each training record is a vector in 23-D hyperplane



Expected to be A, but falls closer to B
Slightly move vector towards A away from B

Deficiencies of ANNs

- Provide no more insight why the decision was made than dissecting human brain helps to understand how it makes decisions
- Updating with new info – stale – no rules, degrades gracefully.
- As in humans – inference from previous knowledge slows the process of learning new patterns

Make computers as capable as humans?

- Brain is highly complex, non-linear, massively-parallel system
- Response of integrated response circuit:

  1 nanosecond = 10^{-9} sec
- Response of neuron

  1 millisecond = 10^{-3} sec
- The only advantage of the brain: massively parallel – 10 billion neurons with 60 trillions of connections

Artificial neural network is abstract – media-independent

- To simulate the brain we could construct thousands of op-amp circuits in parallel
- We can also simulate them using a program that is executed on a conventional serial processor.
- The solutions are theoretically equivalent since a neuron's medium does not affect its operation.
- By simulating the neural behavior, we created a virtual machine that is functionally identical to a machine that would have been prohibitively complex and expensive to build.

ANN implementation in serial processors is not as powerful as human brain

- We can simulate parallel circuits using a program executing on a conventional serial processor.
- A computer's flexibility makes the creation of one hundred neurons as easy as the creation of one neuron.
- The drawback is that the simulated machine is slower by many orders of magnitude than a *real* neural network since the simulation is being done in a serial manner by the CPU.