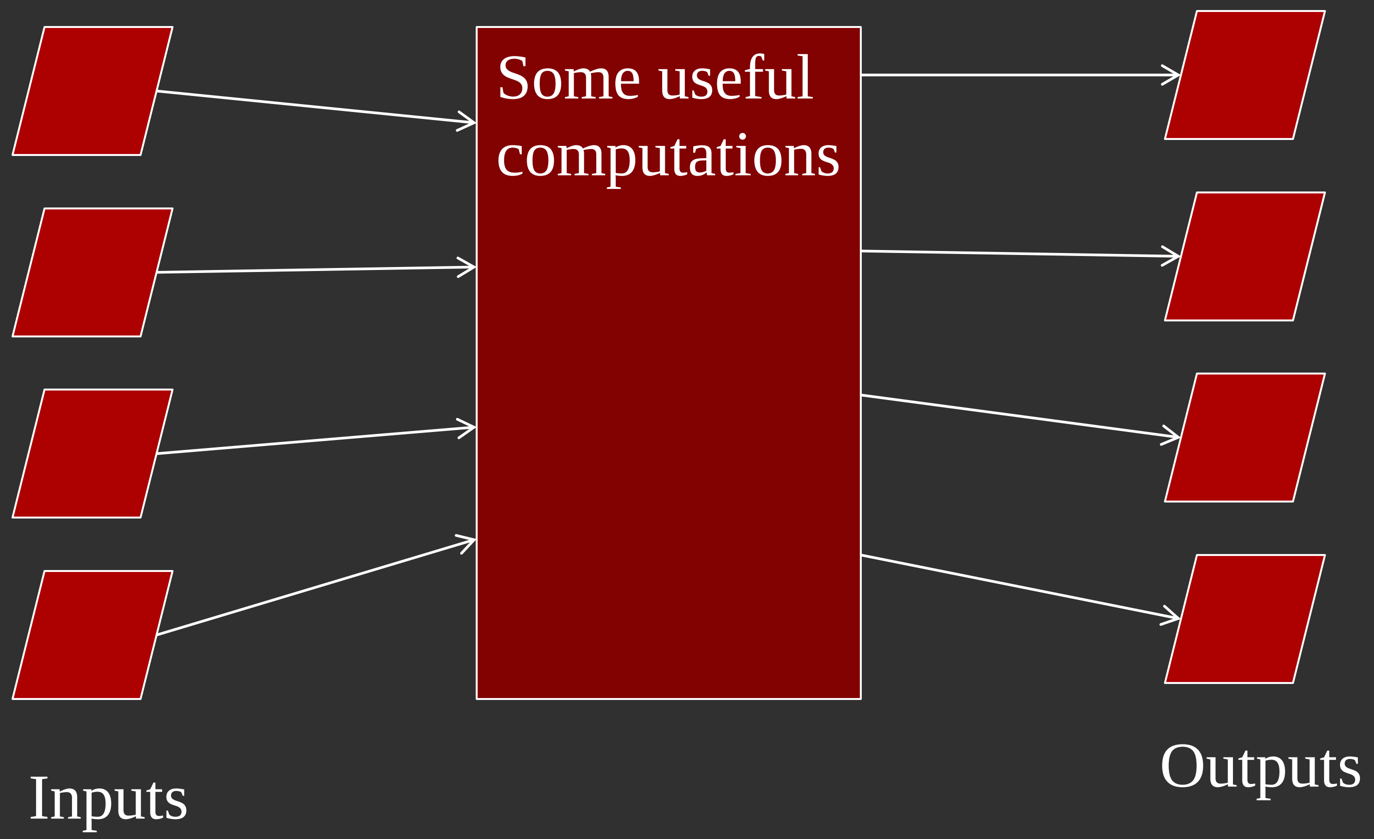
Artificial Neural Networks

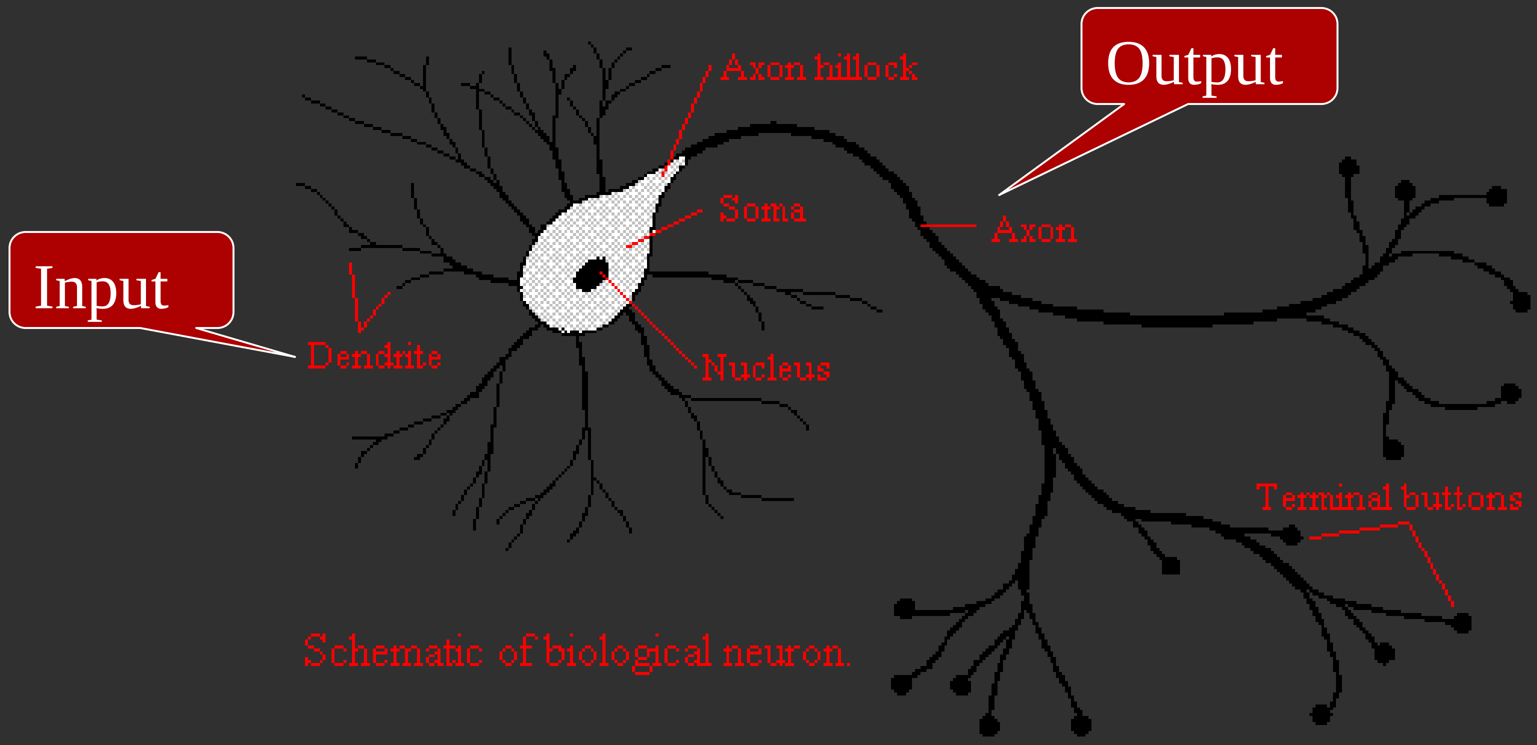
Lecture 23

How computer works



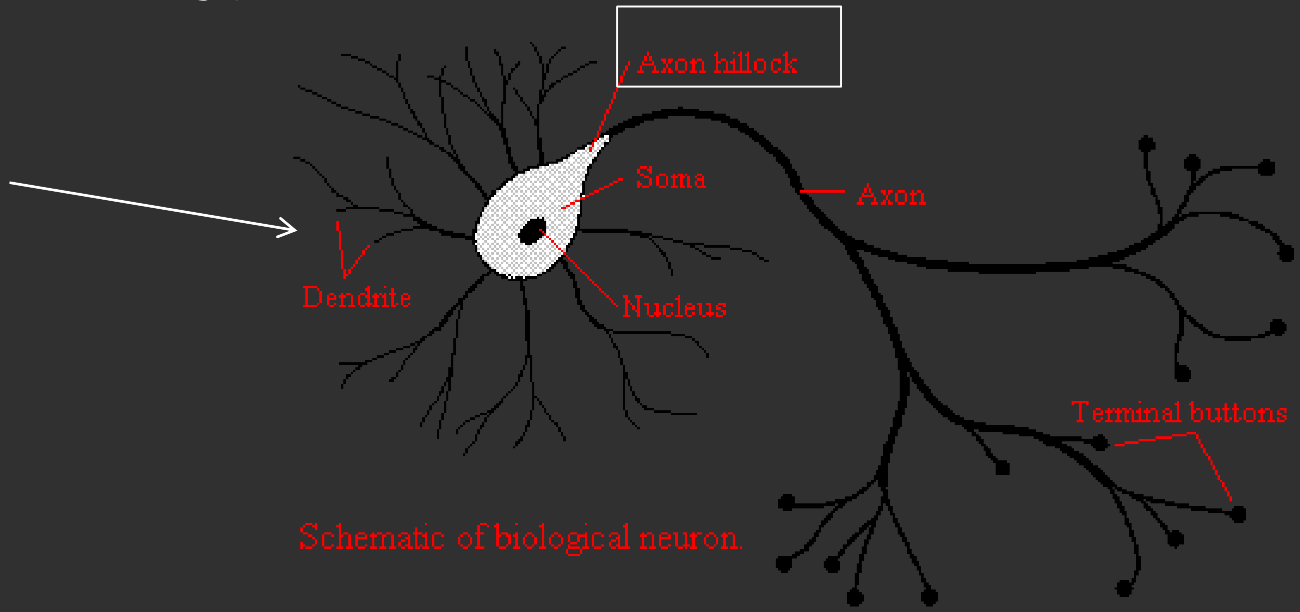
How brain works: neurons

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



Neurons: signal summation

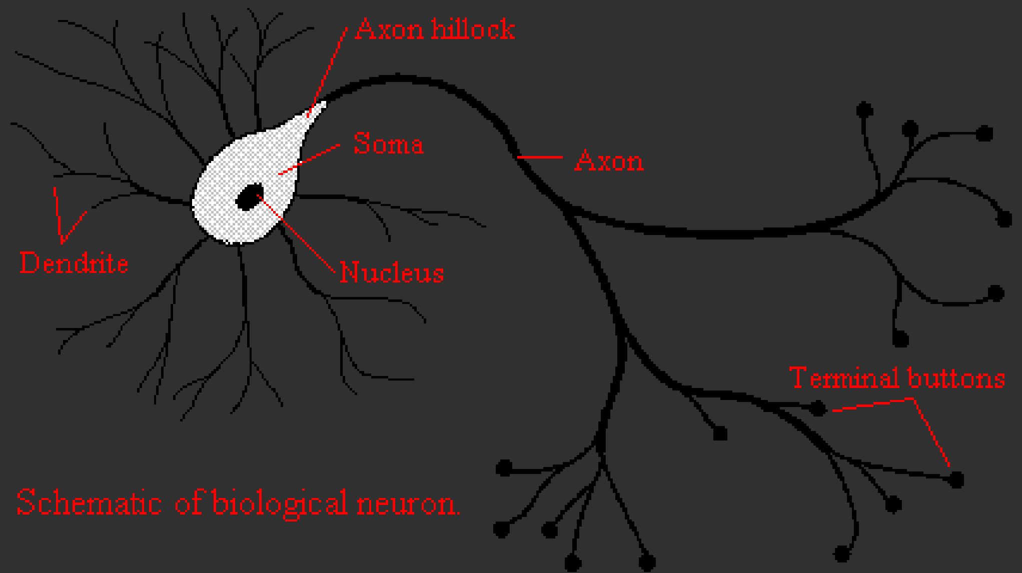
- Dendrite(s) receive an electric charge
- The strengths of all the received charges are added together (spatial and temporal summation). The aggregate value is then passed to the soma (cell body) to axon hillock.



Schematic of biological neuron.

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.

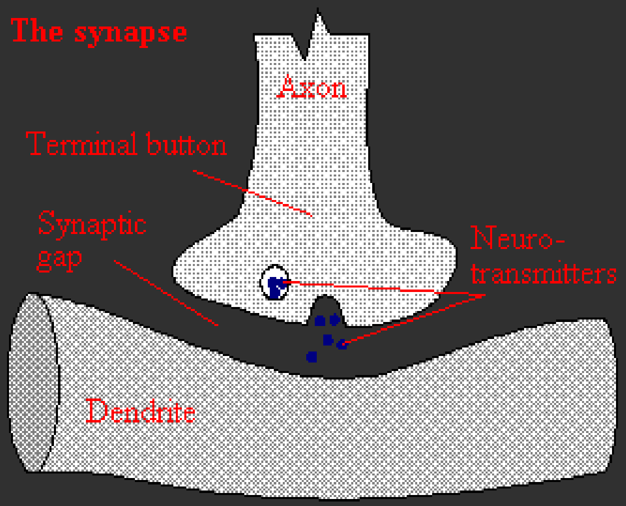


Neurons: the output signal is constant

- 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 than in a digital system.

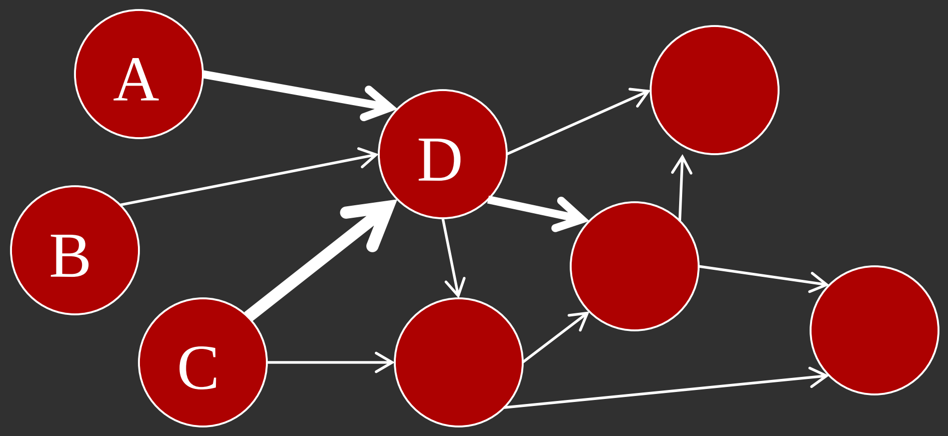
How neurons communicate

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



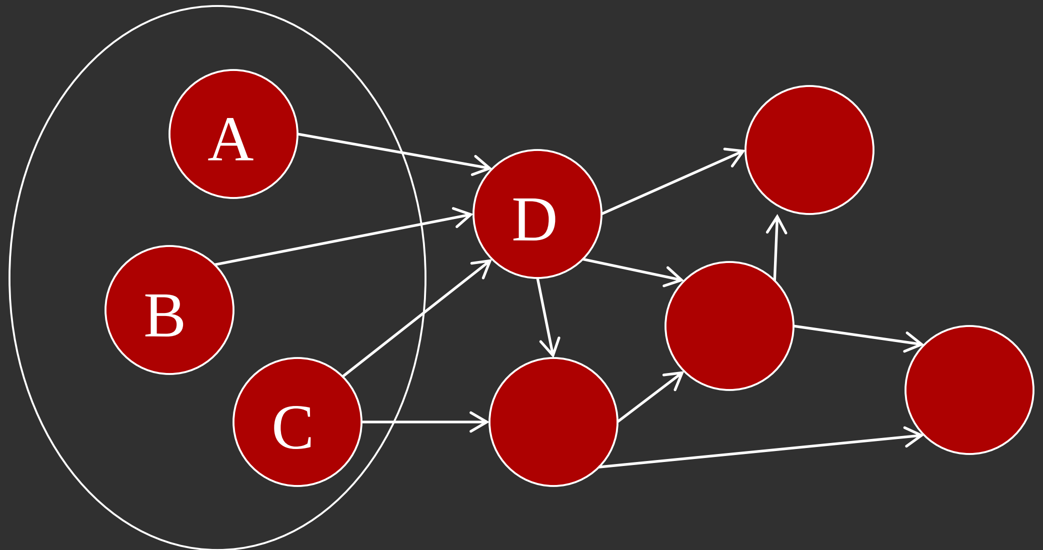
Modeling the brain

- 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 synaptic connection.

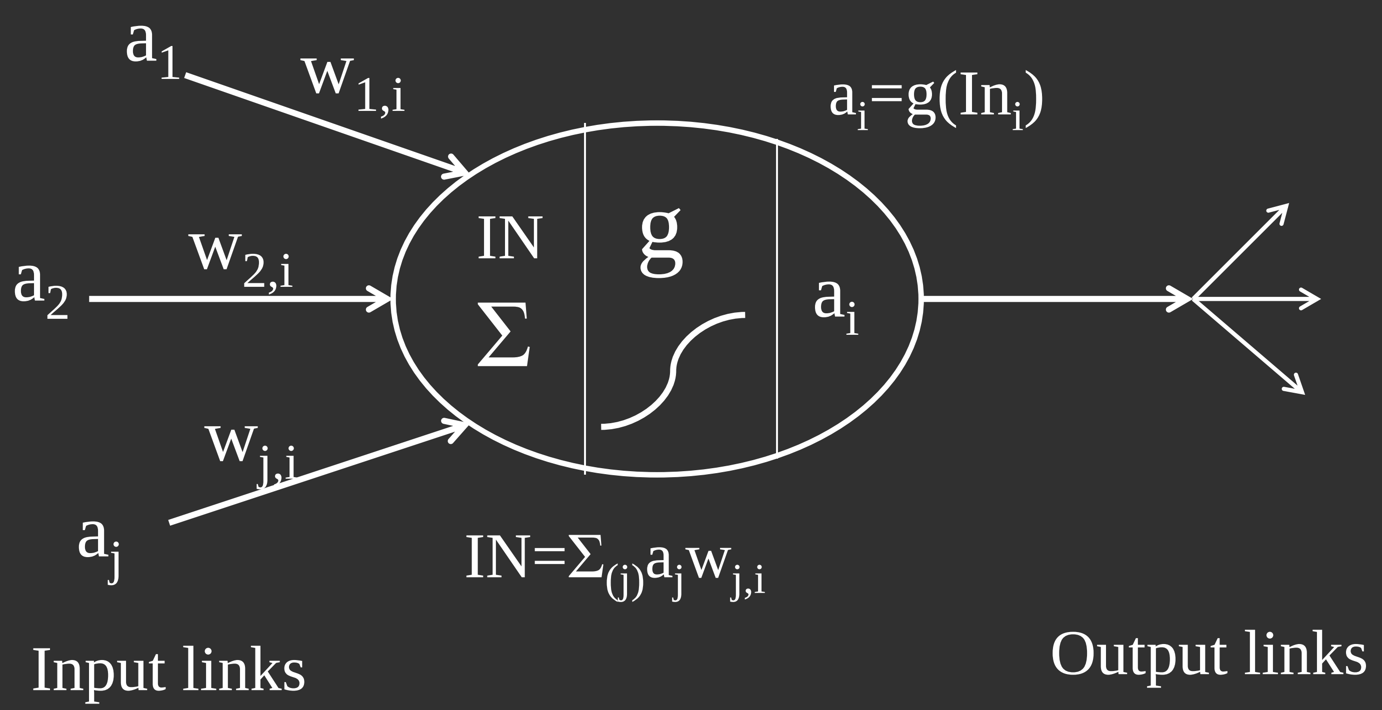


Modeling the brain: input neurons

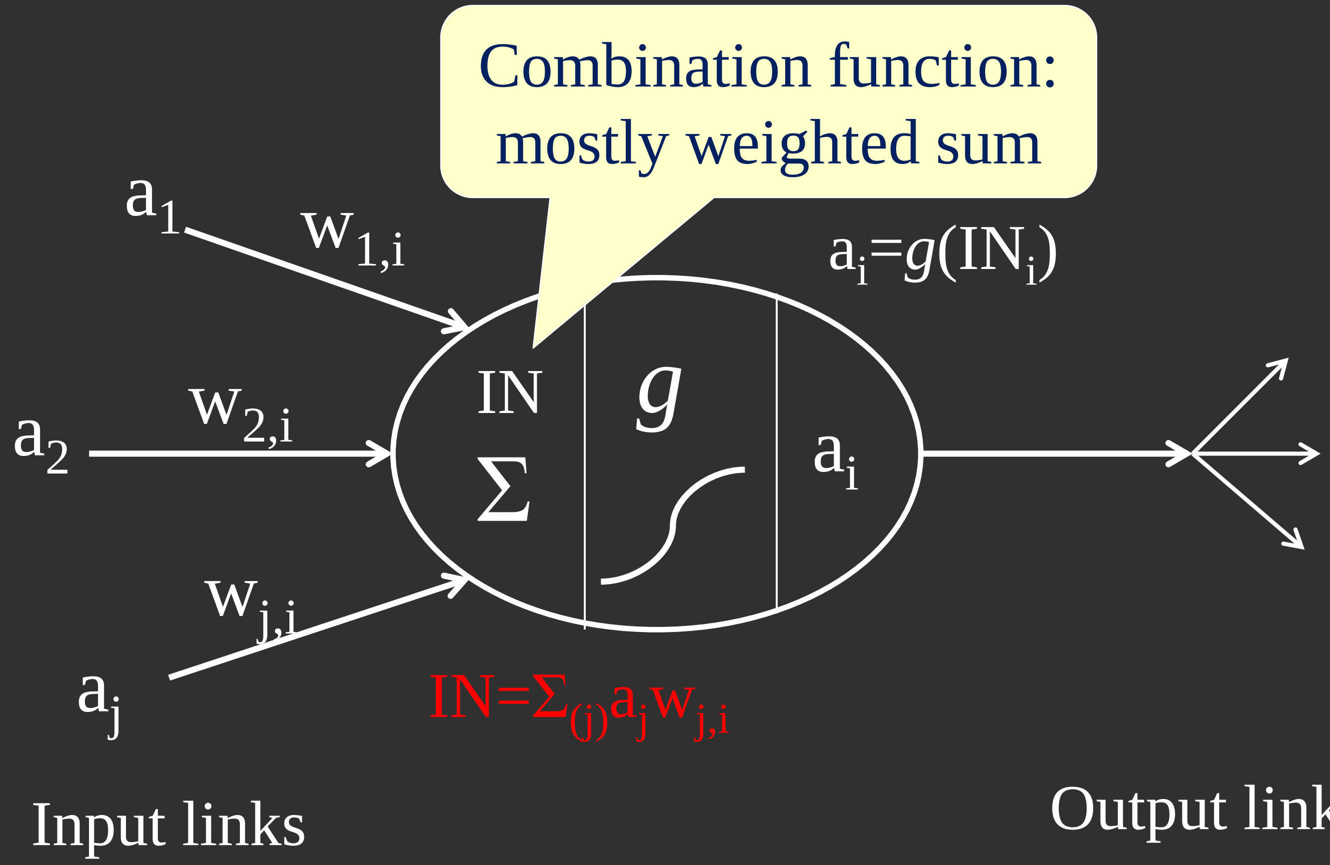
- The input nodes (A, B, C) each have a weight that they contribute to the neuron (D), if the input is active. The neuron can have any number of inputs; neurons in the brain can have as many as a thousand inputs.



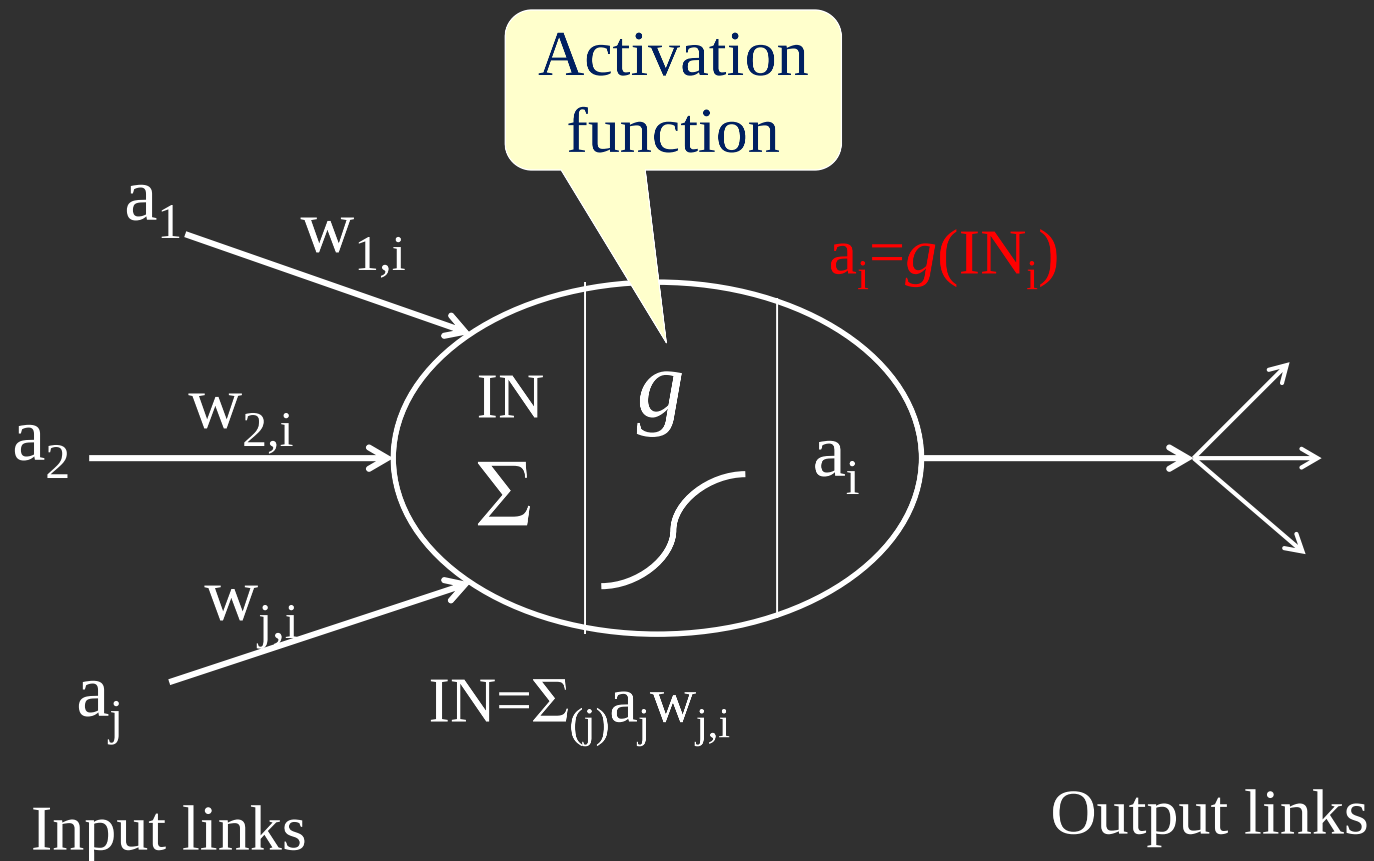
Basic unit of the model: artificial neuron



Neuron: combination function

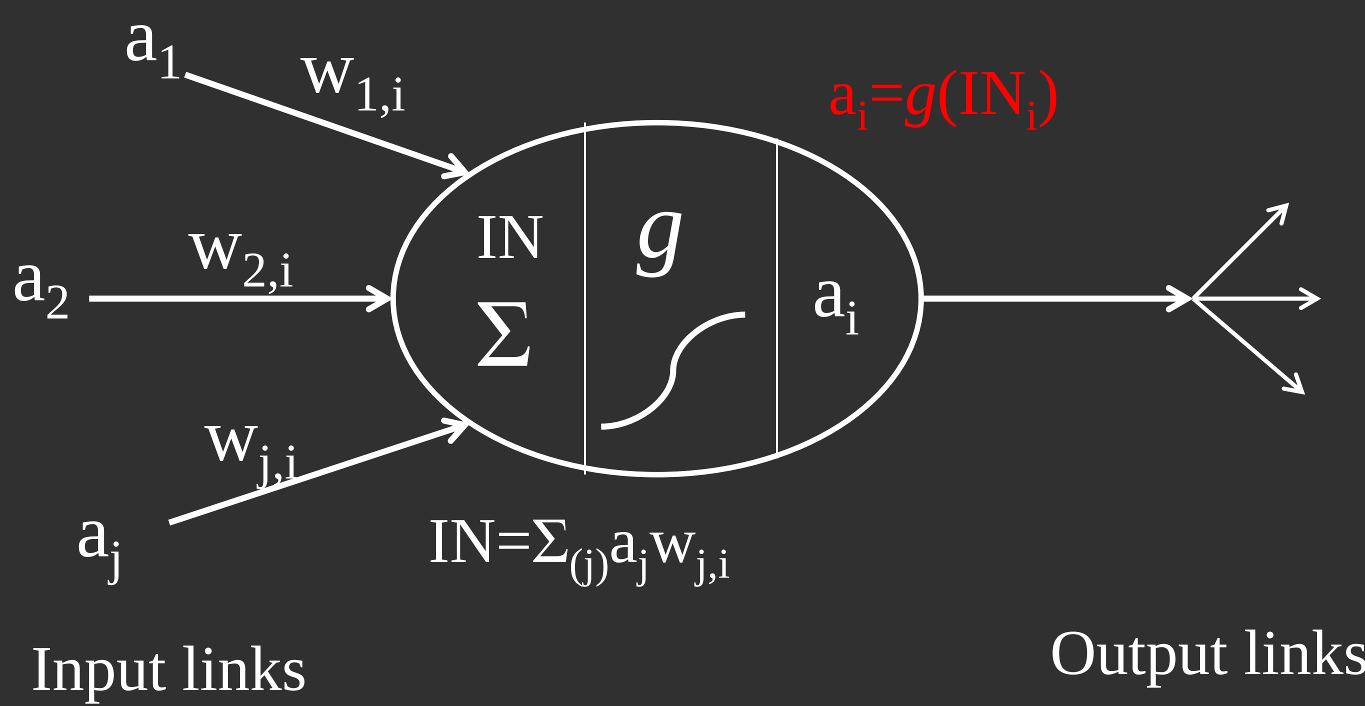


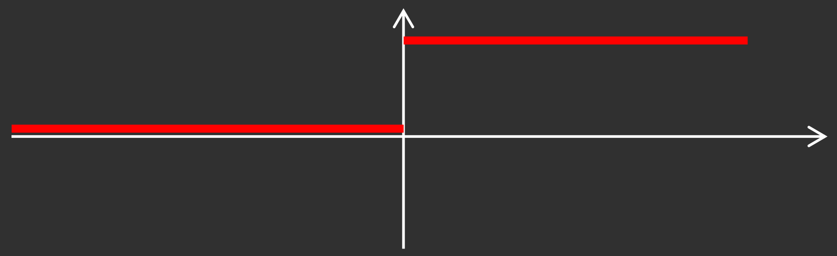
Neuron: activation function



Activation function should be threshold function

The simplest threshold function: *sign*





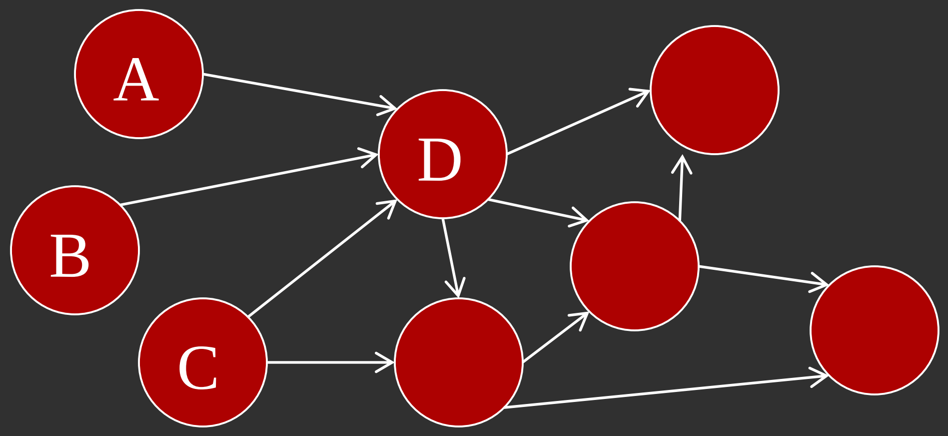
Example of threshold function:

$y(x)=0$ if $x<0$

$y(x)=1$ if $(x\geq 0)$ (neuron fires)

Model of neuron networks

- Nodes and edges. Each edge not only permits to transfer the value, but has an additional parameter: weight
- Node takes input and triggers other nodes through connections
- Node D needs to think if it wants to transfer the value
- The decision is made from the output of transfer function (0 or 1)



Make computers as capable as humans?

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

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

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

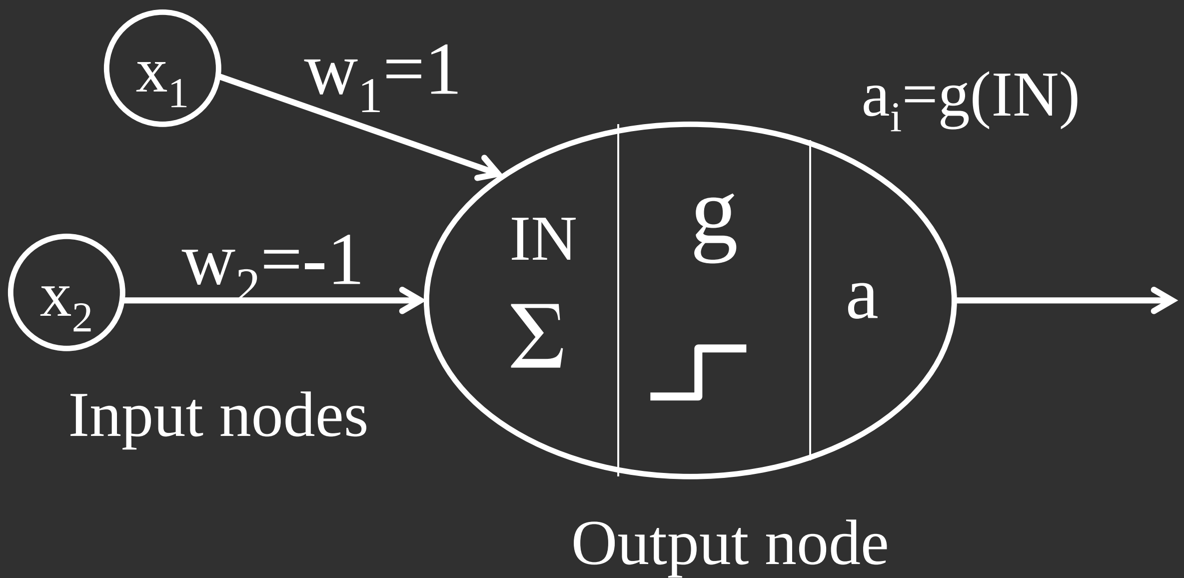
Artificial neural network is abstract – media-independent

- 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 behaviour, 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.

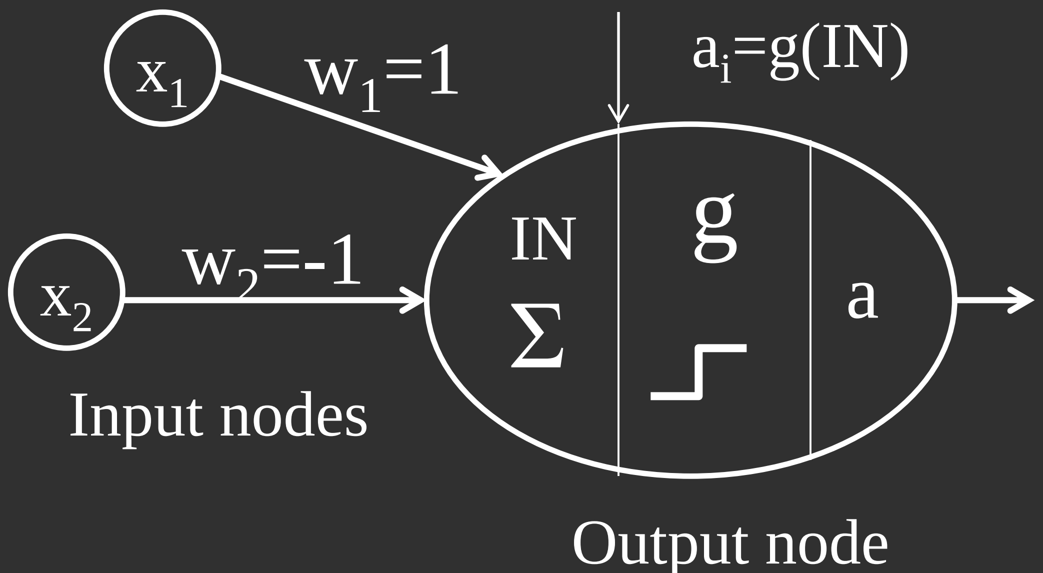
Example of a simple ANN



g(IN)=IN-0.5

a=1 if g≥0.5 – neuron gets activated only if the value of g is ≥ 0.5
a=0 if g<0.5

Example of a simple ANN



If $x1=1$ and $x2=1$ then

$\Sigma=0$

$g=-0.5$ – no activation

If $x1=0$ and $x2=1$ then

$\Sigma=-1$

$g=-1.5$ – no activation

If $x1=1$ and $x2=0$ then

$\Sigma=1$

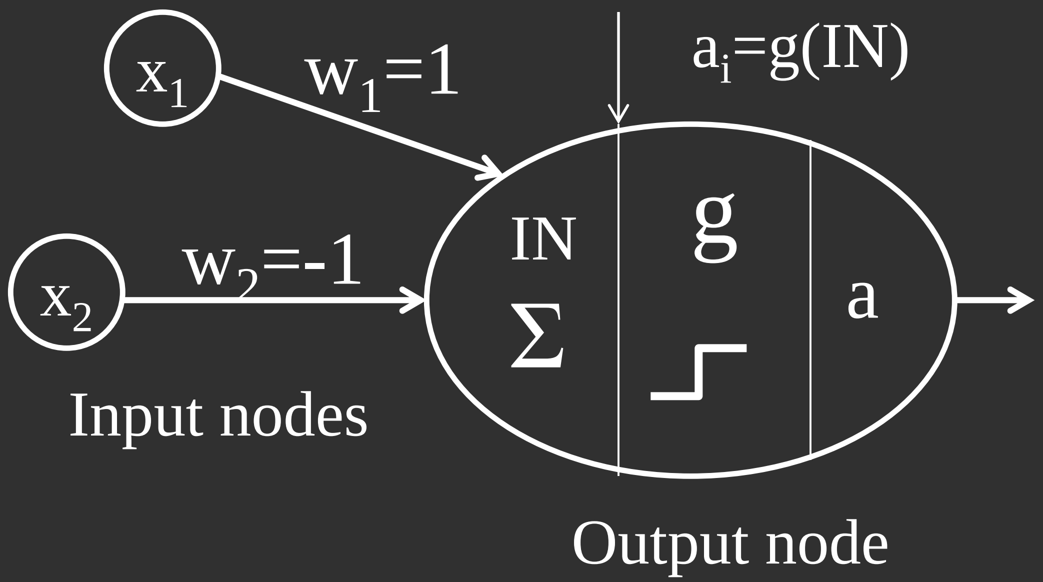
$g=0.5$ – activation

$g(IN)=IN-0.5$

$a=1$ if $g\geq0.5$

$a=0$ if $g<0.5$

Example of a simple ANN



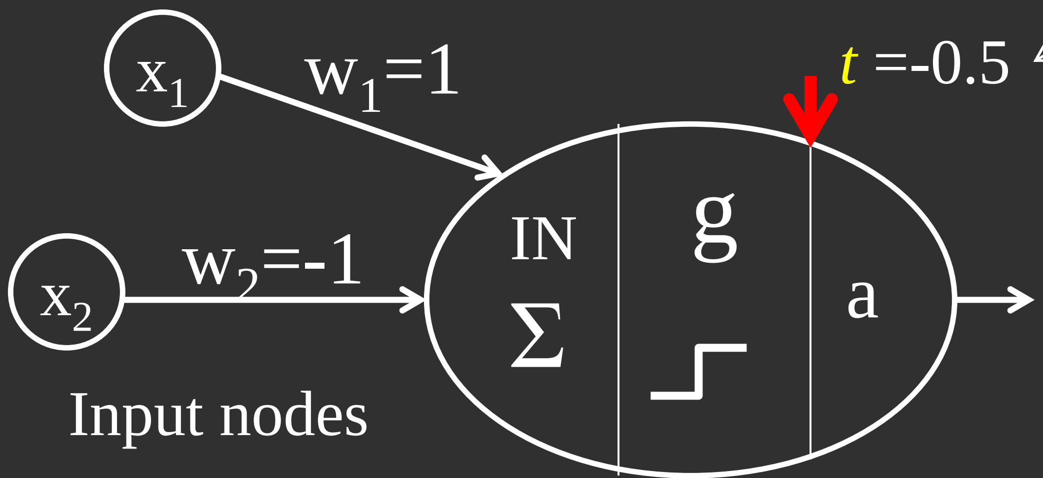
This single neuron and its input weighting performs the logical expression *x1 AND NOT x2.*

$g(IN)=IN-0.5$

$a=1$ if $g \geq 0.5$

$a=0$ if $g<0.5$

Example of a simple ANN: bias factor



$a_i = g(IN) + t$

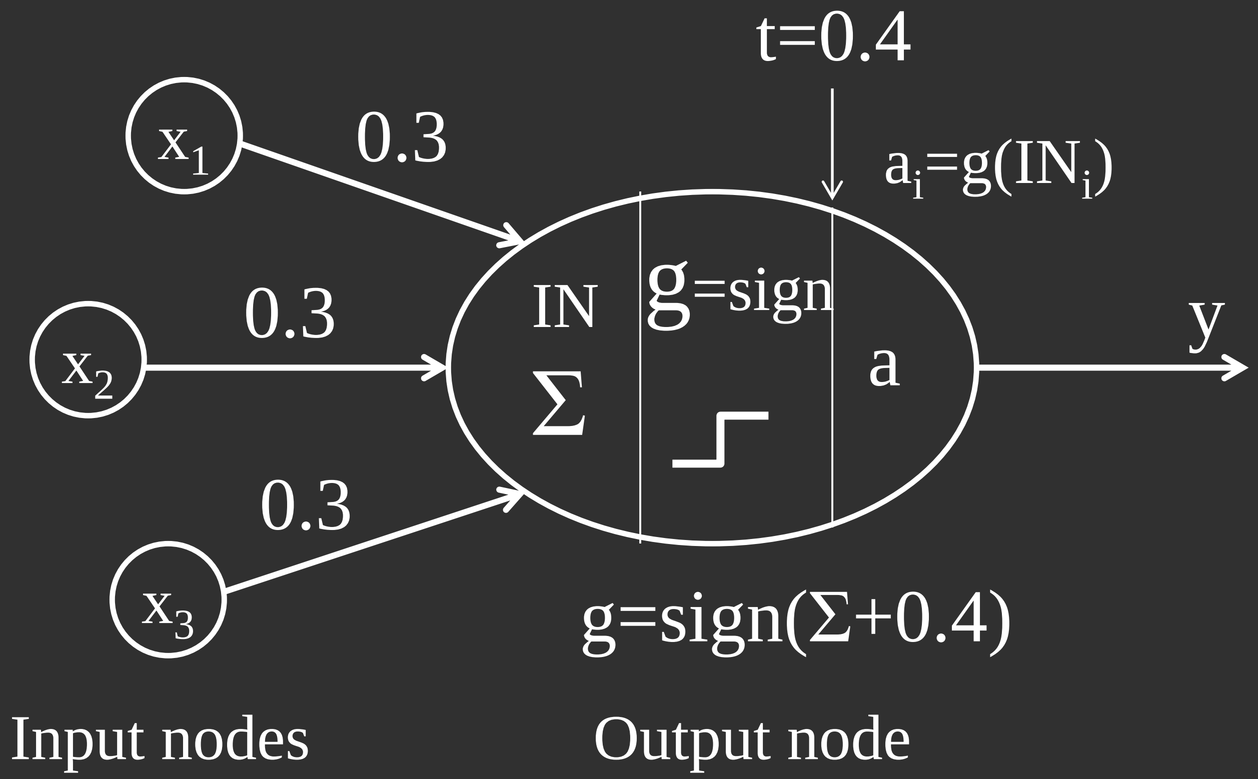
It is more convenient for computation to use *sign* function (> 0 and not > 0.5)

-0.5 is then added as a constant *bias factor*

$a=1$ if $g \geq 0$ – neuron gets activated only if $g \geq 0$

$a=0$ if $g<0$

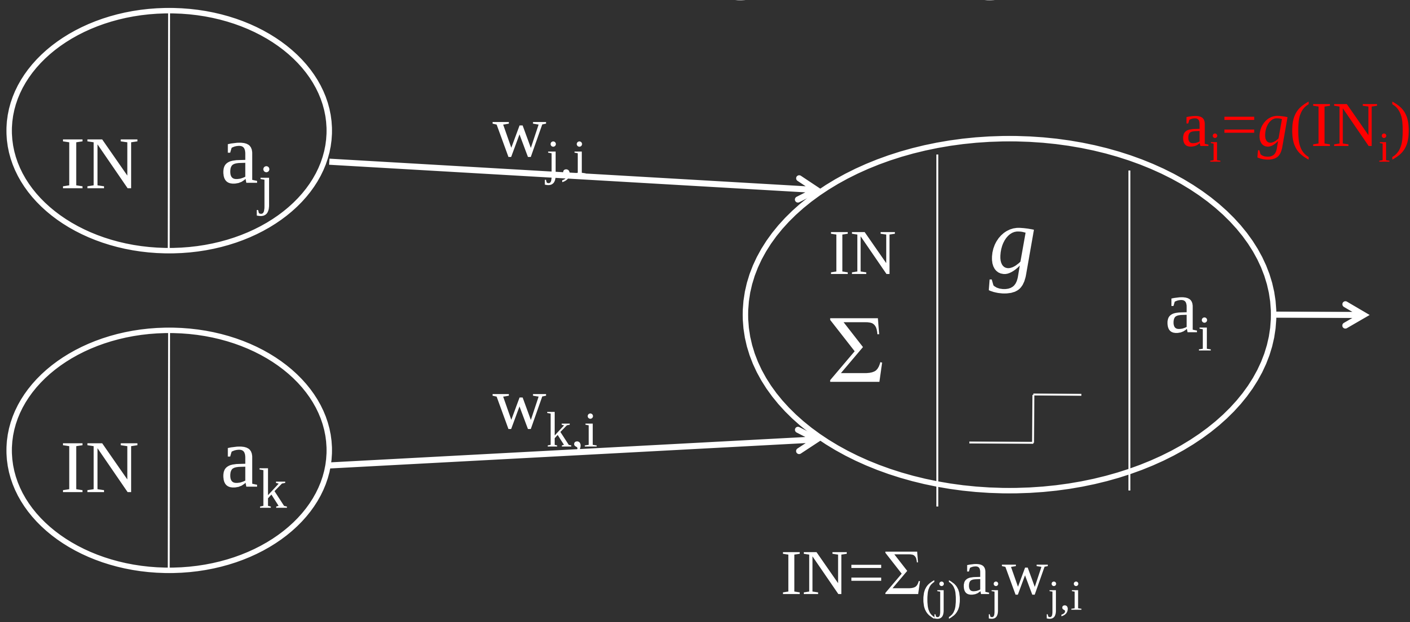
Single-layer NN - *Perceptron*



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

$$y=\text{sign}(w1x1+w2x2+w3x3+t)$$

Training Perceptron: learning weights



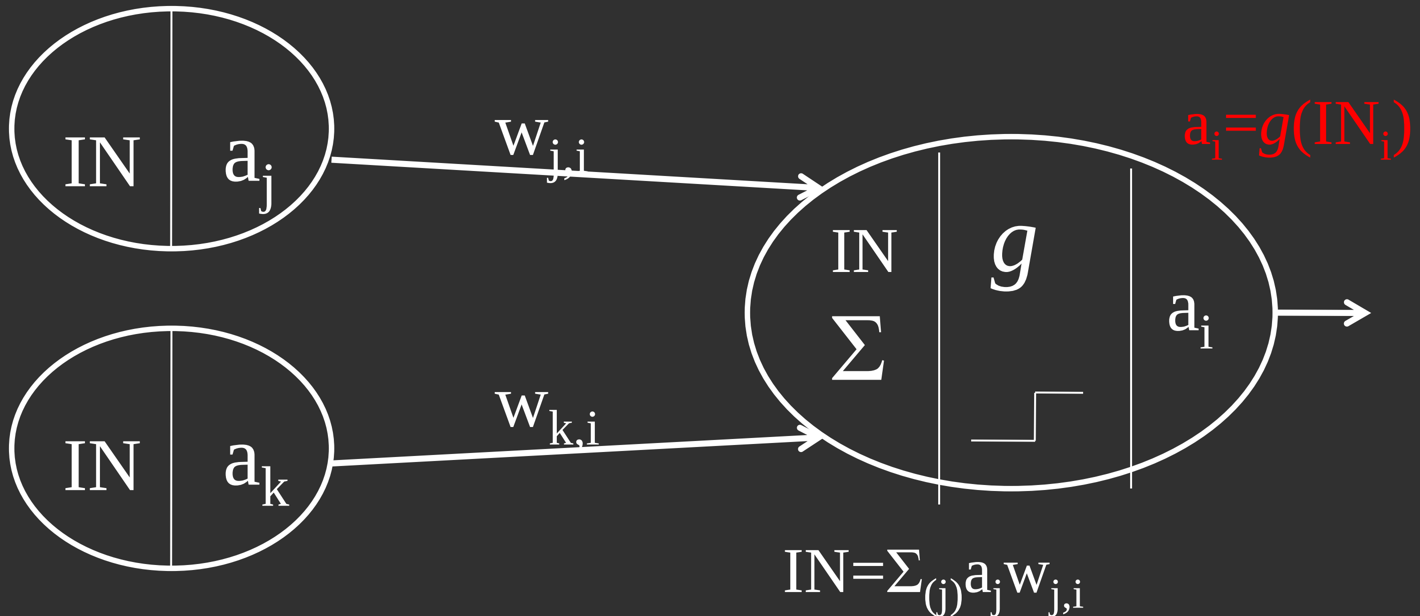
Start with random weights

Training record has attribute values a_j, a_k and class T

Perceptron classifies it as class O

Err=T - O

Training Perceptron: learning weights



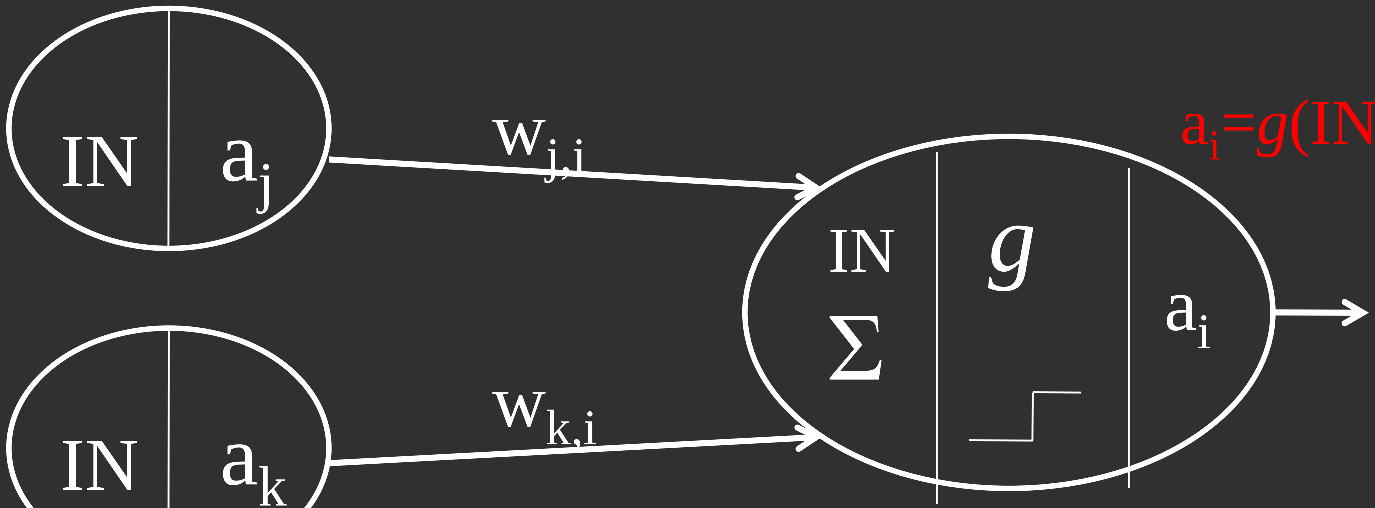
Classification error:

Err=T - O

T – desired output (*target*)

O – actual output

Training Perceptron: learning weights



$IN = \Sigma_{(j)} a_j w_{j,i}$

Classification error:

Err=T - O

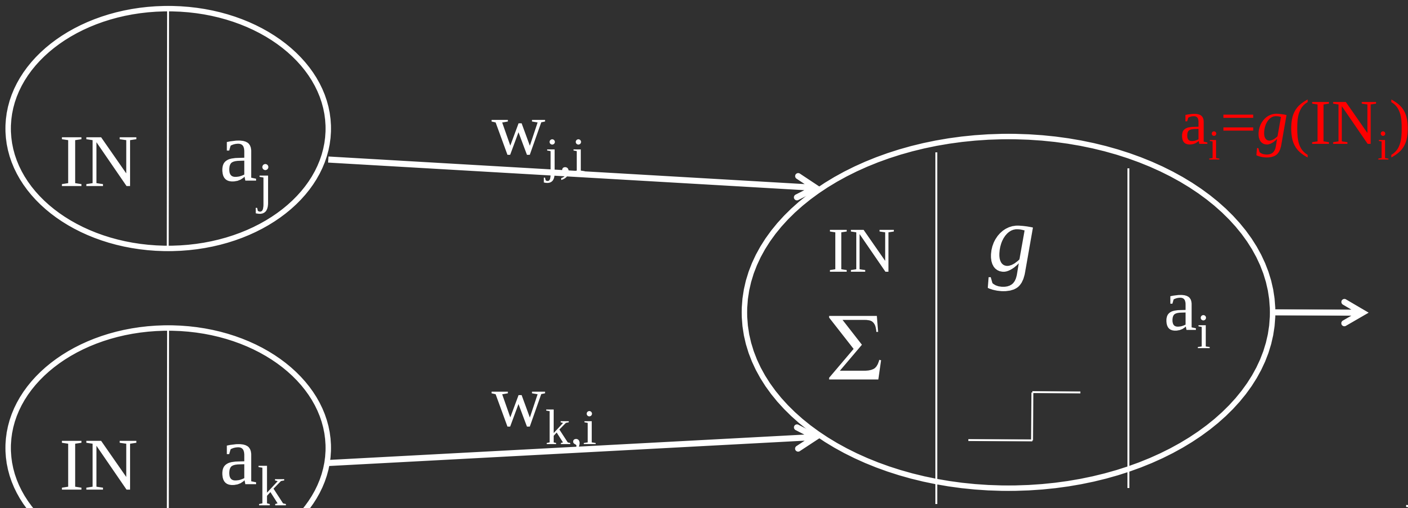
T – desired output (*target*)

O – actual output

Adjust each weight by Δ:

$\Delta (w_{j,i}) = a_j \times Err$

Training Perceptron: learning weights



T – desired
O – actual

Err=T - O

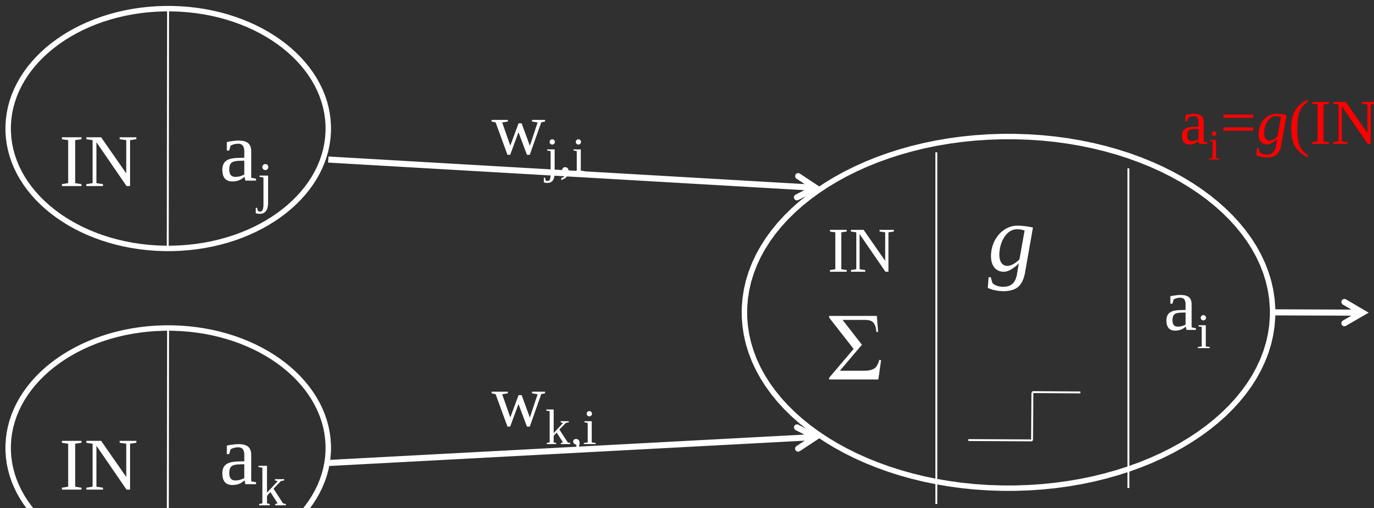
$IN = \Sigma_{(j)} a_j w_{j,i}$

Adjust each weight by Δ:

$\Delta (w_{j,i}) = a_j \times Err$

Each weight is adjusted by multiplying its contribution (value) by the error.

Training Perceptron: learning weights



$a_i = g(IN_i)$

T – desired
O – actual

$IN = \Sigma_{(j)} a_j w_{j,i}$

Err=T - O

Adjust each weight by Δ:

$\Delta(w_{j,i}) = a_j \times Err$

if T-O<0 (actual > target) then decrease weight

if T-O>0 (actual<target) then increase weight

Training Perceptron: adjusting weights

Err=T - O

T – desired output (*target*)

O – actual output

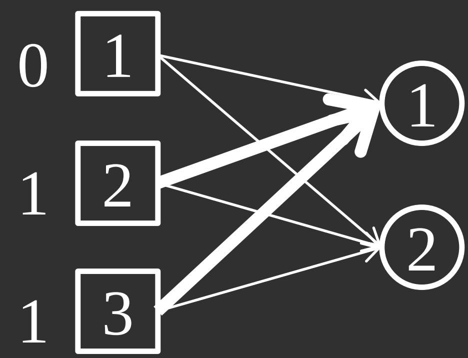
But do not adjust by the entire value of error, just move slightly into desired direction

The delta rule:

$w_{j,i} \leftarrow w_{j,i} + \eta \times a_i \times Err$

Learning rate (eta)

Training Perceptron: adjusting weights



O1=1
T1=-1

Slightly reduce weights on inputs with 1
Slightly Increase weight on input with 0

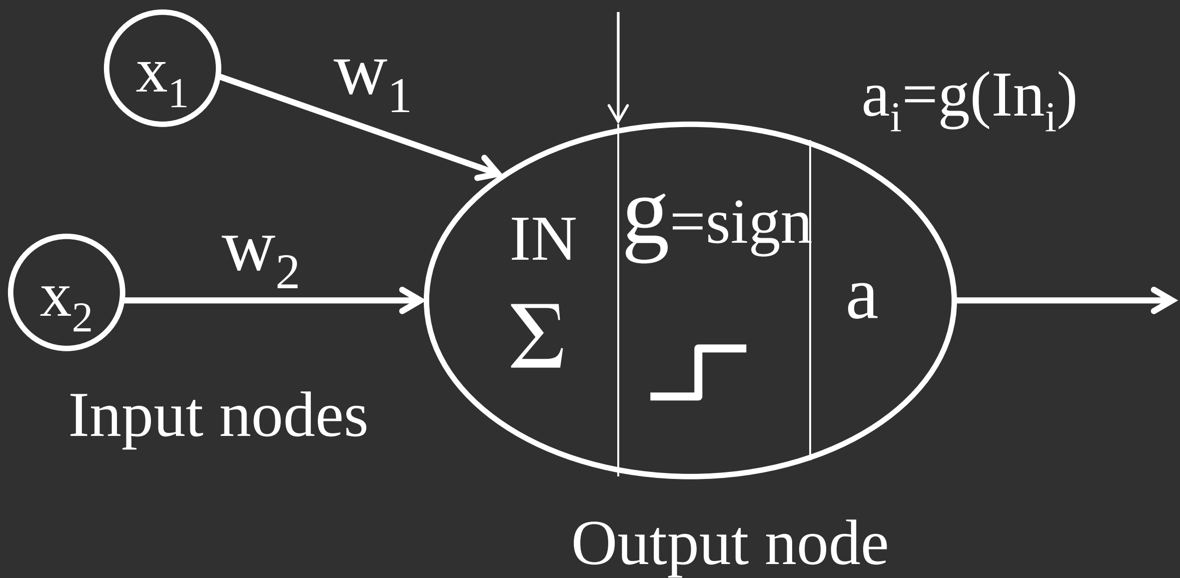
The delta rule:

$$w_j \leftarrow w_j + \eta \times x_i \times Err$$

Learning rate

The learning is performed with a slow rate

The goal of training



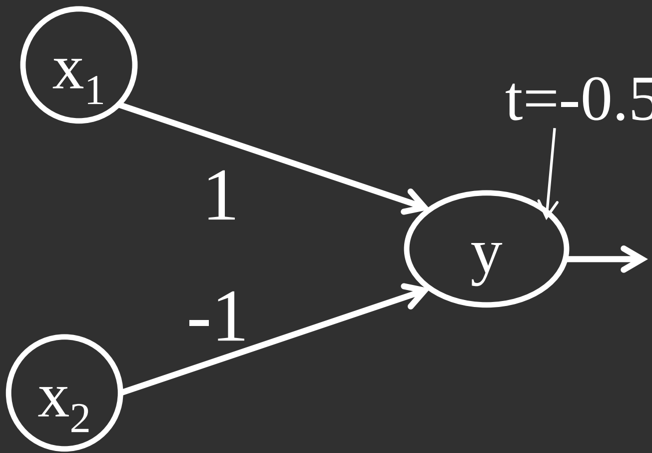
The output node gets activated only if $\Sigma x_i w_i + t > 0$

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: determine the optimal values of weights to separate all labeled instances by a hyperplane

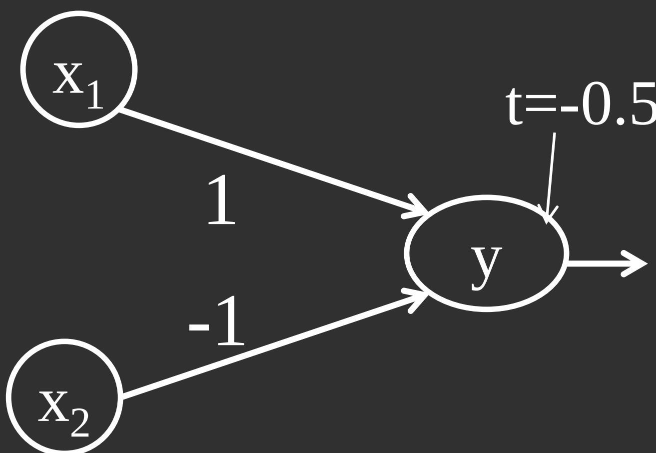
Perceptron learned AND NOT



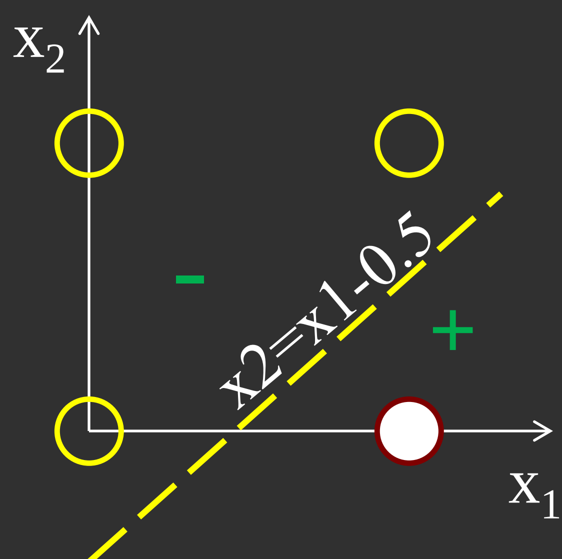
y= x1 AND NOT x2		
x1	x2	y
0	0	<0
0	1	<0
1	0	≥0
1	1	<0

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



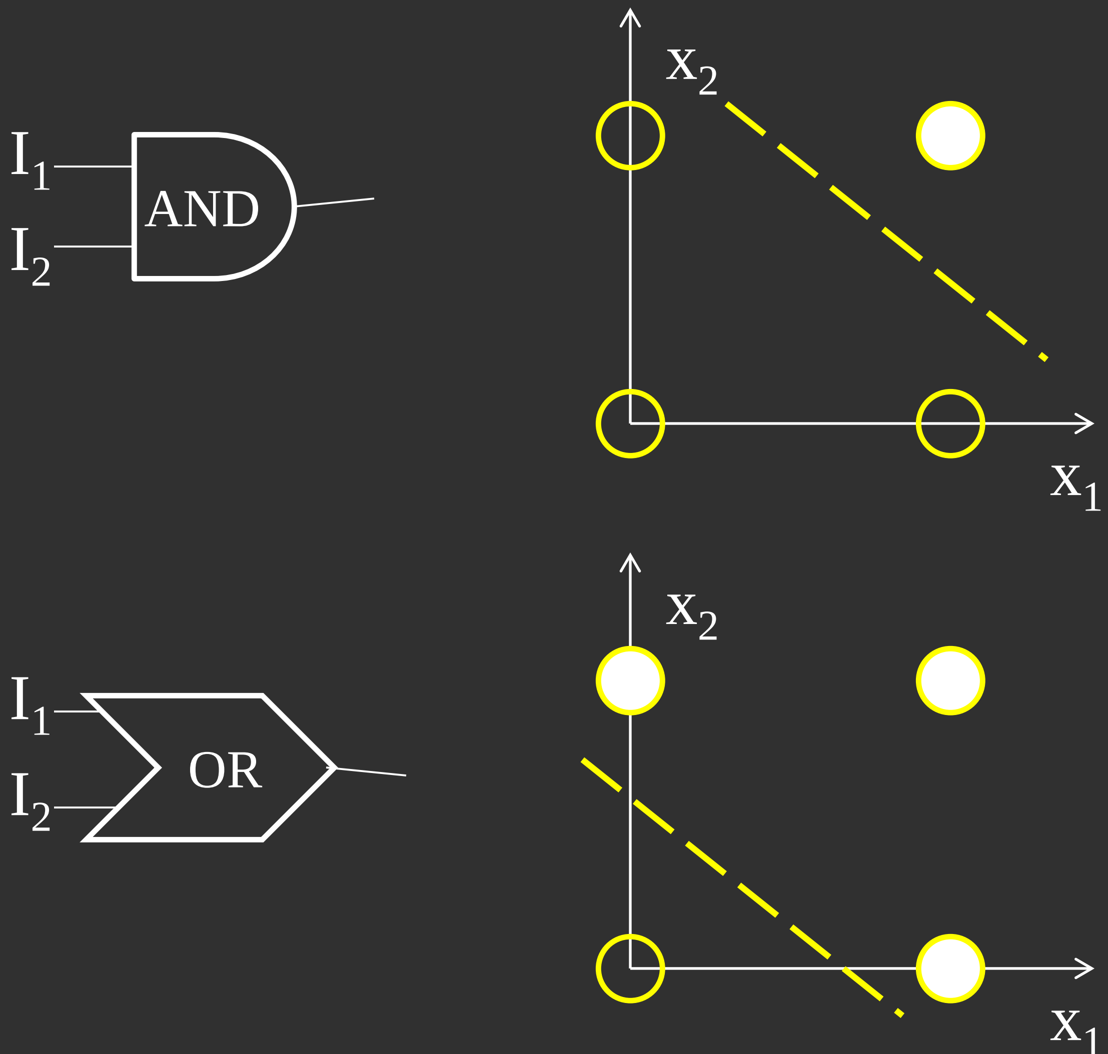
y=x1w1+x2w2+t

t=-0.5, w1=1, w2=-1

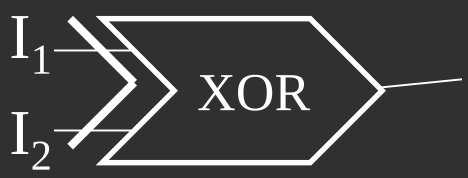
x1-x2-0.5=0

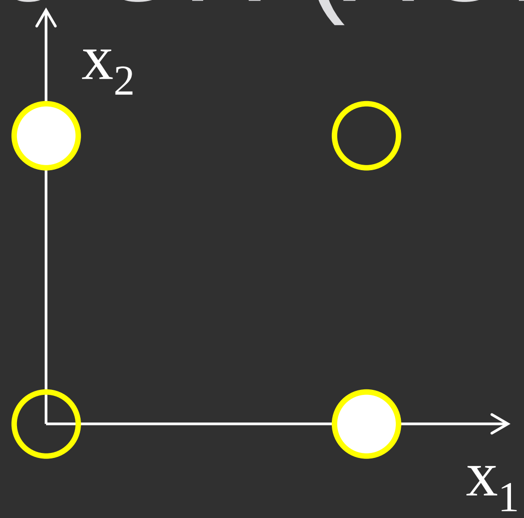
x2=x1-0.5

Perceptron can learn only linearly-separable functions



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



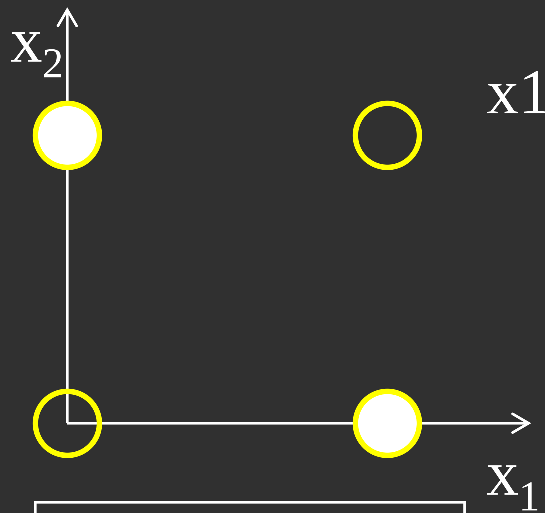


XOR table

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

Solution – add more layers

Building multi-layer perceptron for *XOR*



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

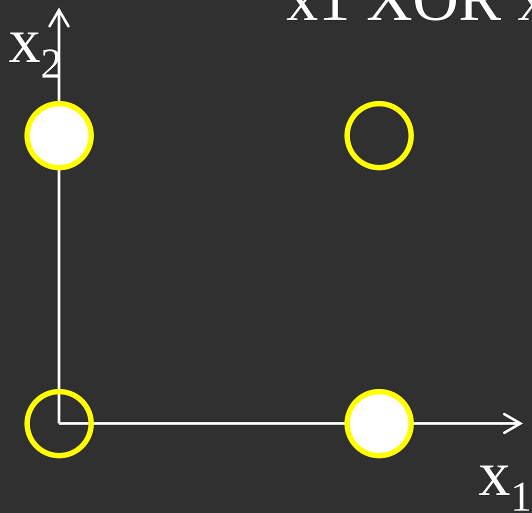
y1=x1 OR x2		
x1	x2	y1
0	0	0
0	1	1
1	0	1
1	1	1

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

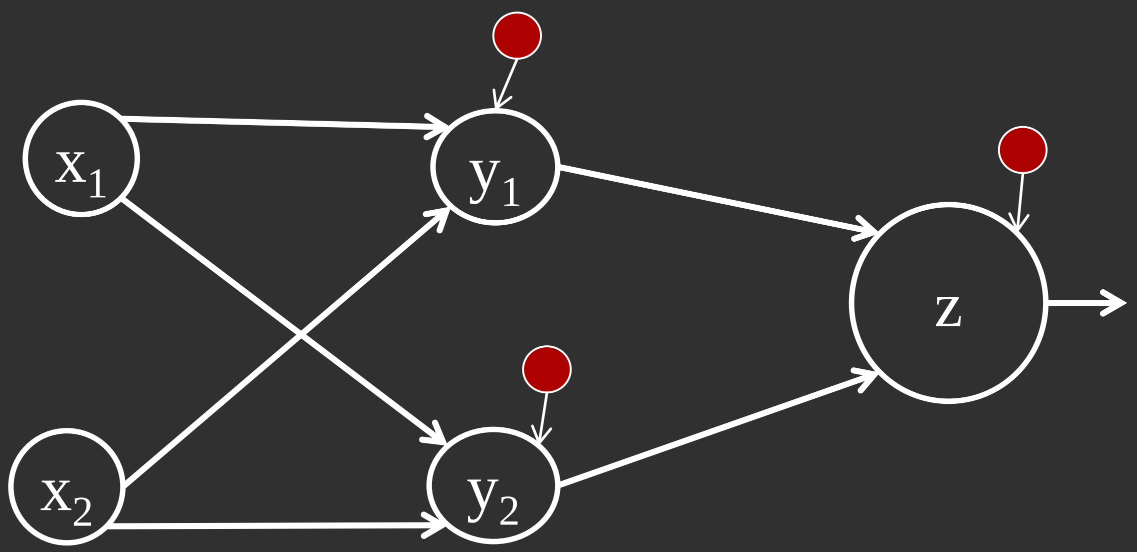
y1=x1 OR x2		
x1	x2	y1
0	0	0
0	1	1
1	0	1
1	1	1

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

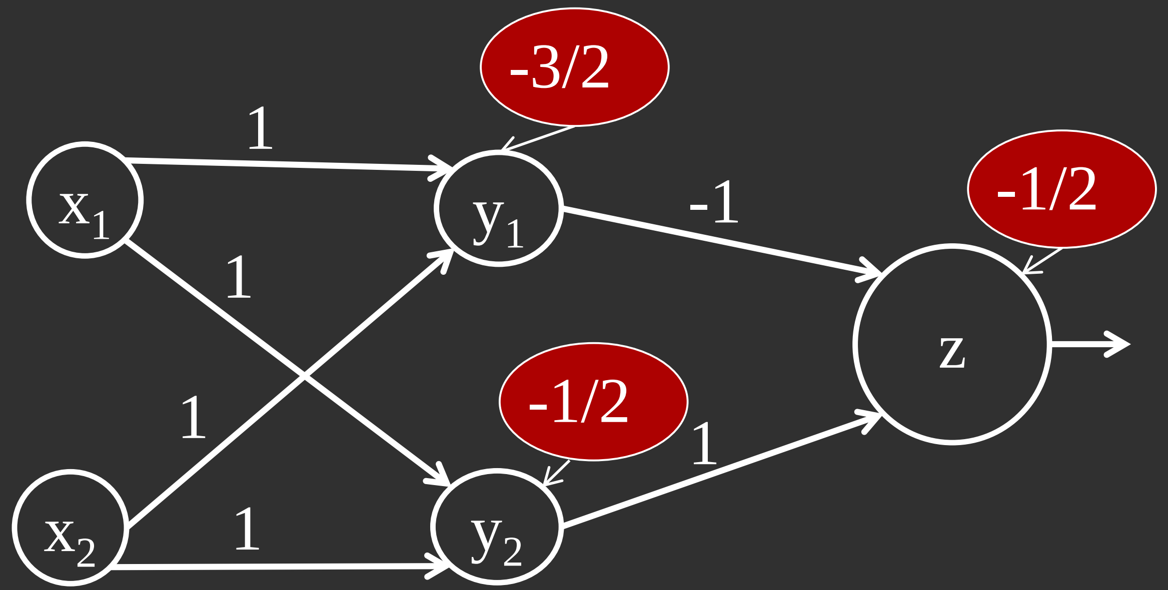
XOR ANN topology

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



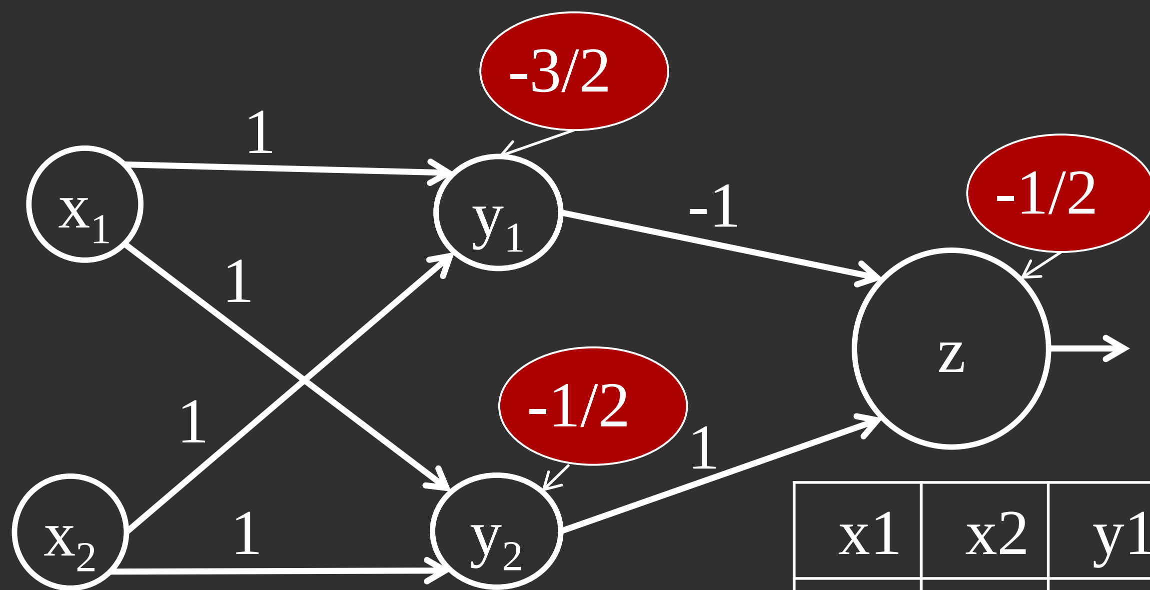
XOR ANN weights

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



XOR ANN: y1

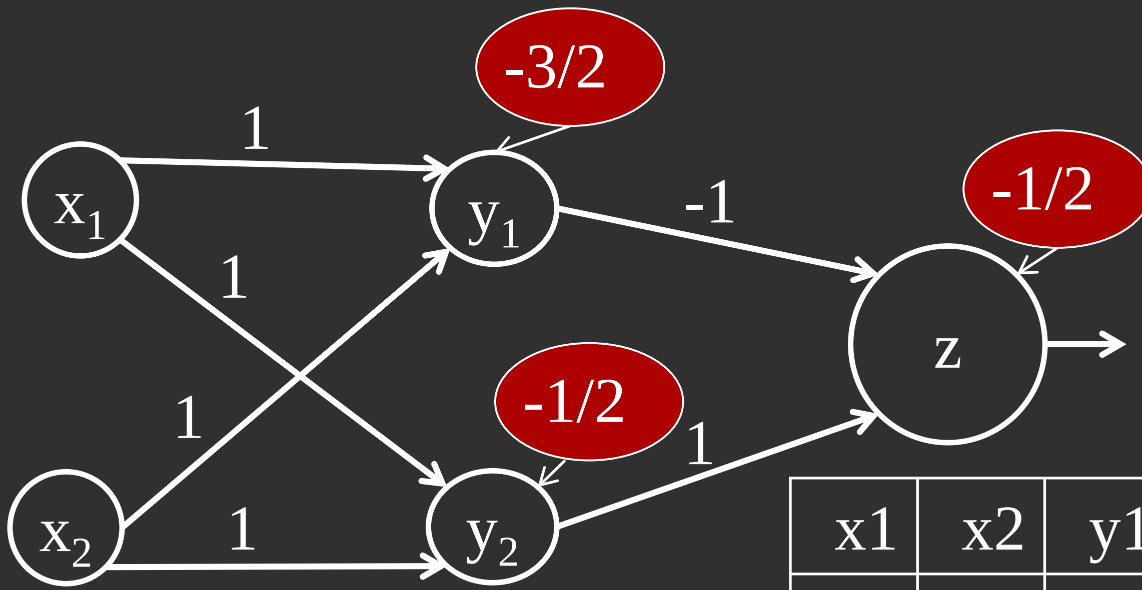
x1 XOR x2 = x1 OR x2 AND NOT (x1 AND x2)



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

XOR ANN: y2

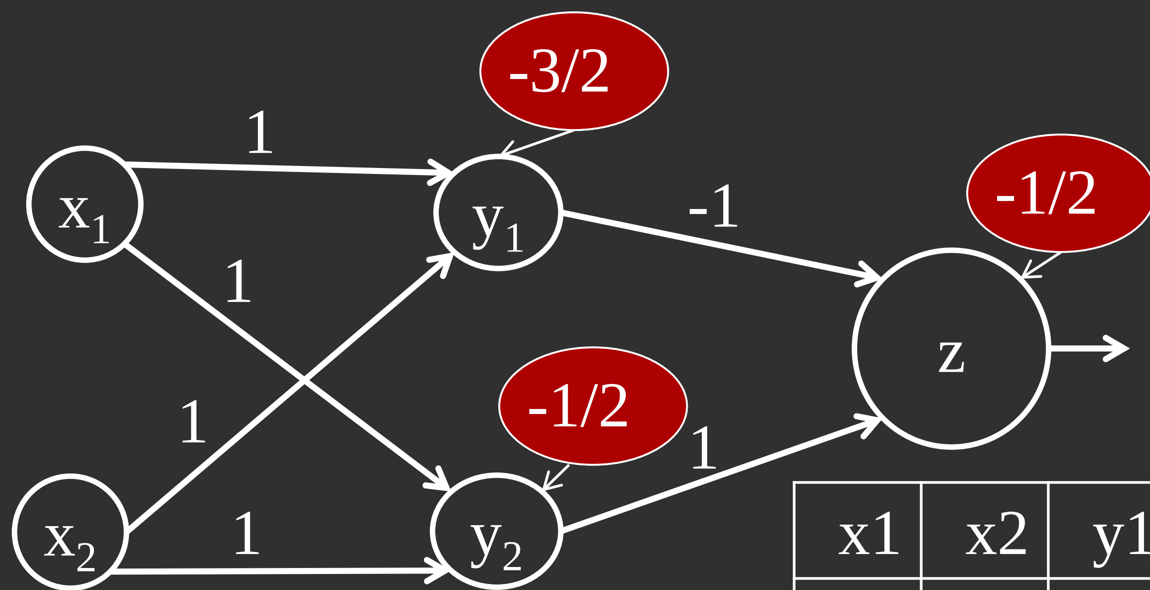
x1 XOR x2 = x1 OR x2 AND NOT (x1 AND x2)



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

XOR ANN: z

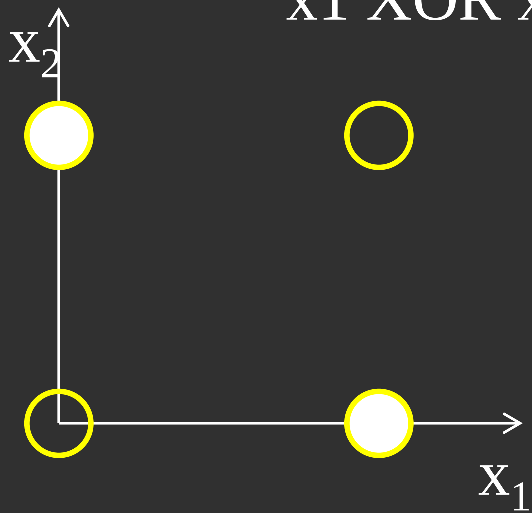
x1 XOR x2 = x1 OR x2 AND NOT (x1 AND x2)



x1	x2	y1	y2	z
0	0	$_{-3/2}$0	$_{-1/2}$0	0
0	1	$_{-3/2}$0	$_{1/2}$1	$_{1/2}$1
1	0	$_{-1/2}$0	$_{1/2}$1	$_{1/2}$1
1	1	$_{1/2}$1	$_{3/2}$1	$_{-1/2}$0

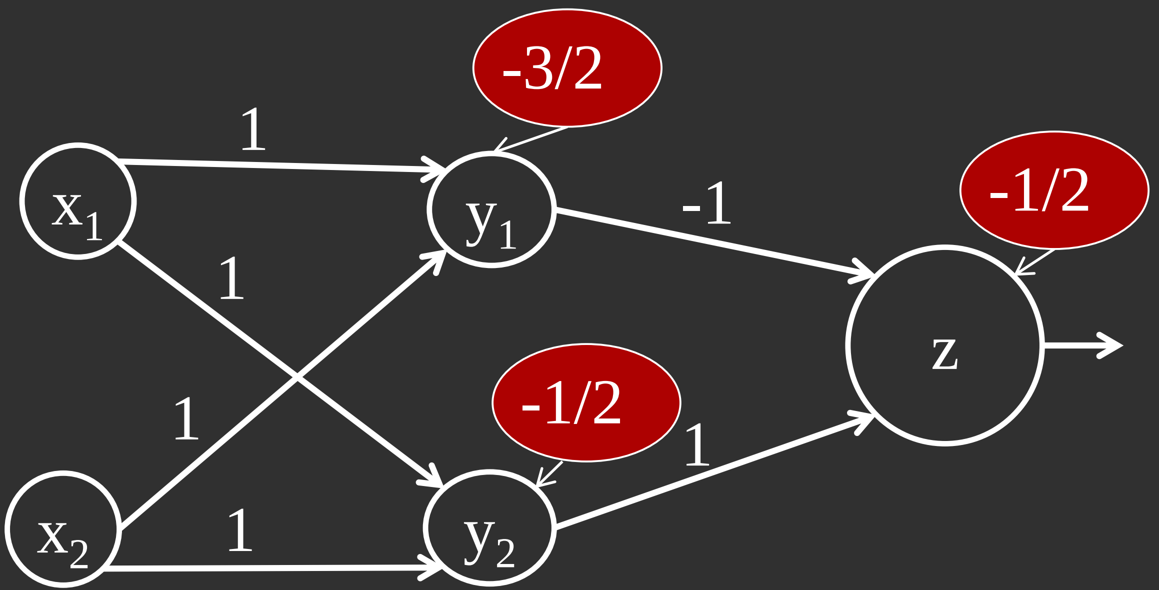
Separating with 2 linear separators

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

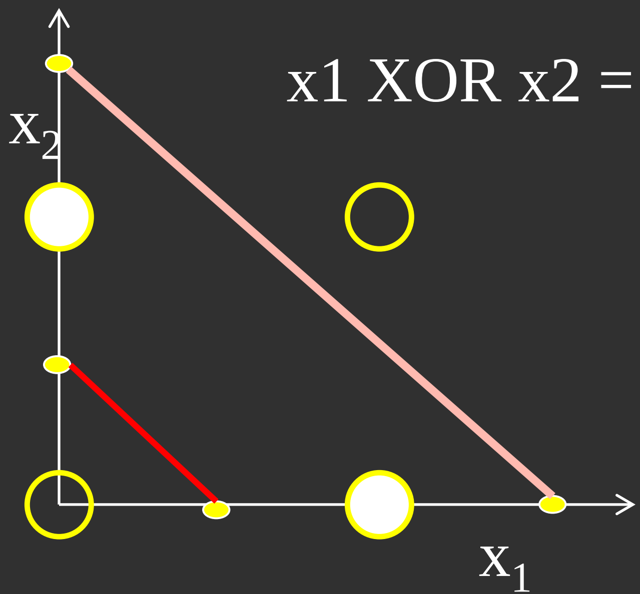


$y1=x1+x2-3/2$

$y2=x1+x2-1/2$



Separating with 2 linear separators



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

y1=x1+x2-3/2

y2=x1+x2-1/2

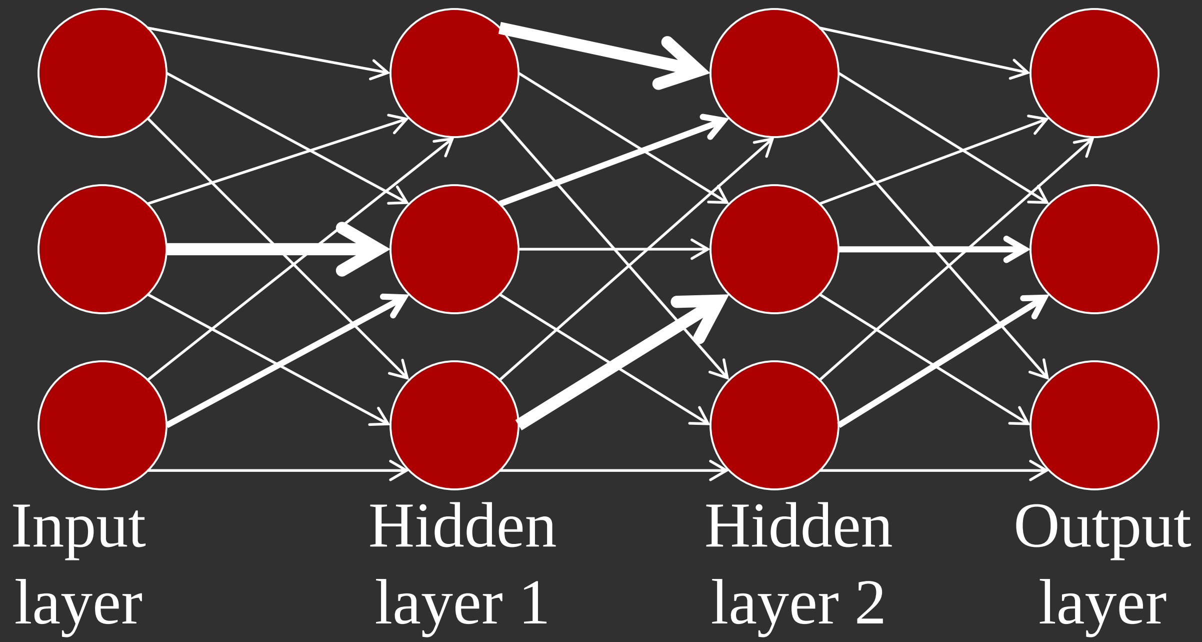
Separating lines (2D hyperplanes):

x2=-x1+3/2

x2=-x1+1/2

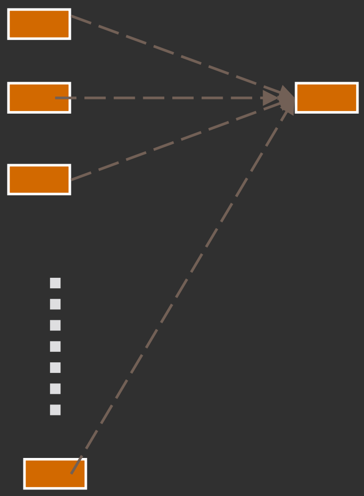
Summary: Multi-layer ANNs

- As in a regular computer: inputs and outputs, only now we call them neurons. Added: *hidden nodes*
- Nodes are organized into layers. Edges are directed and carry weight
- No connections inside the layer



Properties of architecture

- No connections within a layer

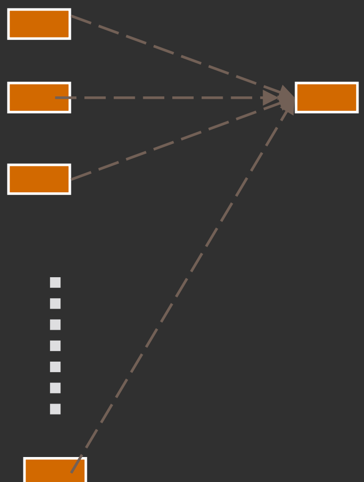


Each unit is a perceptron

$$y_i = f\left(\sum_{j=1}^{m} w_{ij} x_j + b_i\right)$$

Properties of architecture

- No connections within a layer
- No direct connections between input and output layers
-

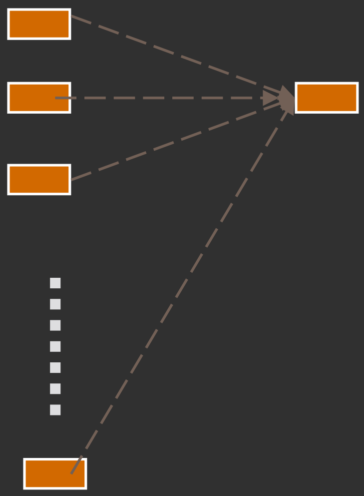


Each unit is a perceptron

$$y_i = f\left(\sum_{j=1}^{m} w_{ij} x_j + b_i\right)$$

Properties of architecture

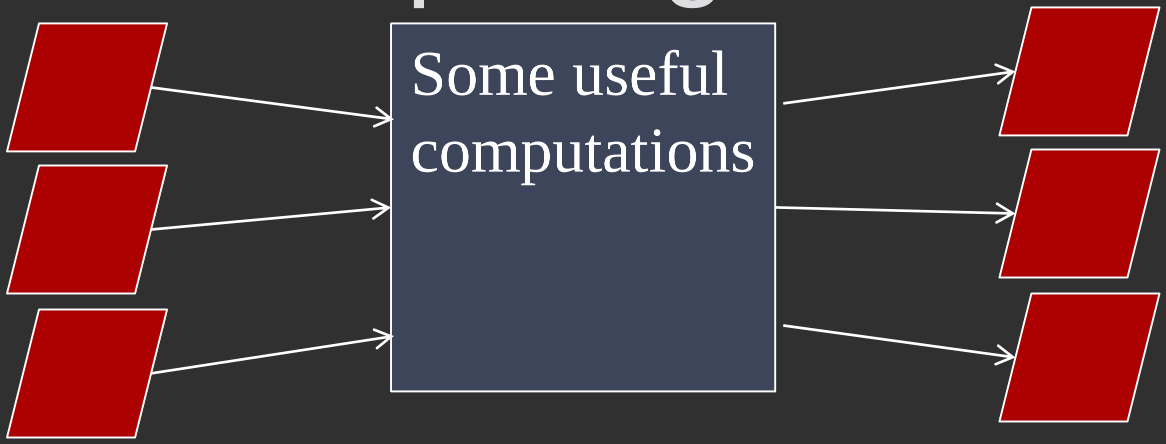
- No connections within a layer
- No direct connections between input and output layers
- Fully connected between layers

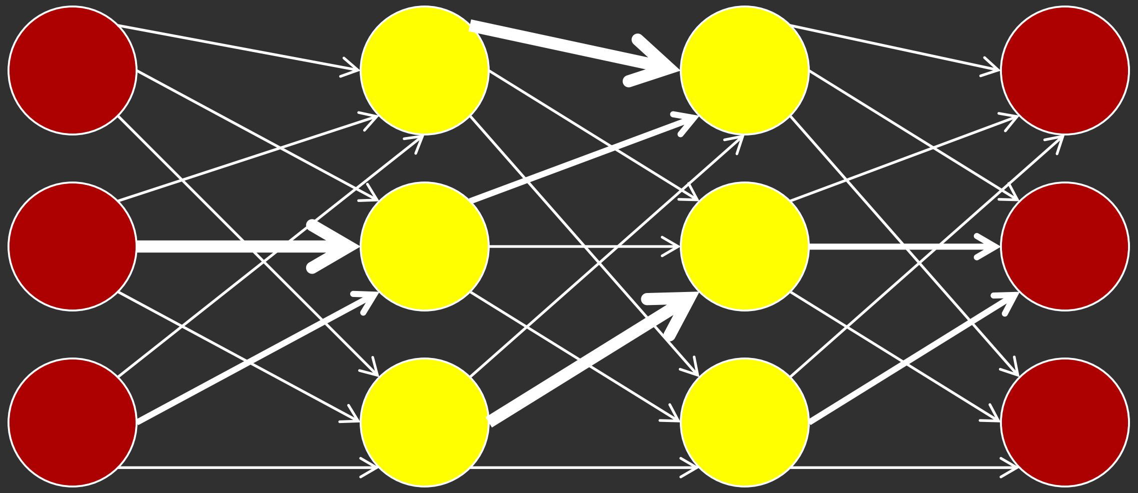


Each unit is a perceptron

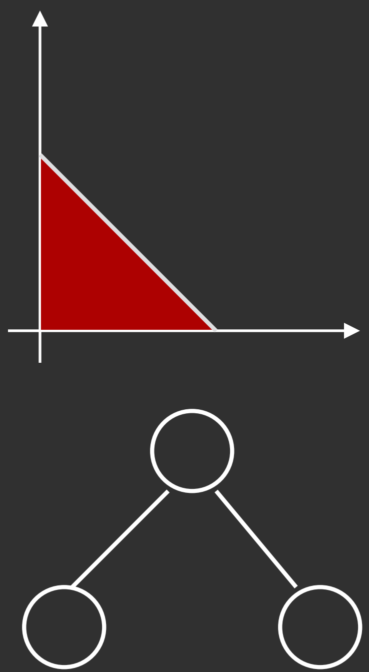
$$y_i = f\left(\sum_{j=1}^{m} w_{ij} x_j + b_i\right)$$

ANN model vs. regular computing model

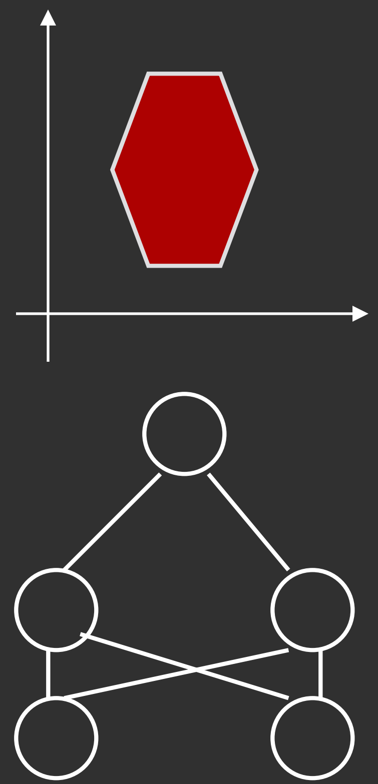




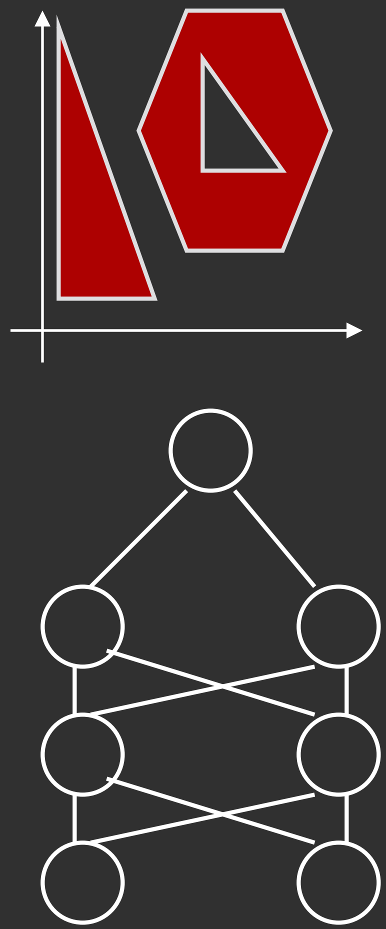
What do we gain from the extra layers



1st layer draws
linear boundaries



2nd layer combines
the boundaries

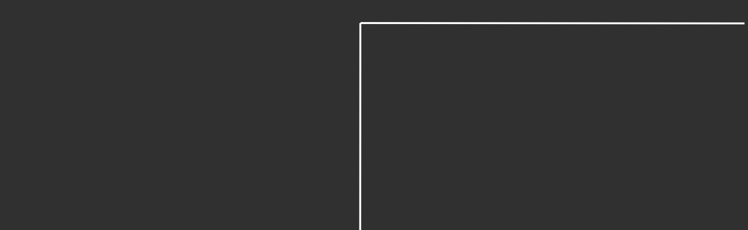


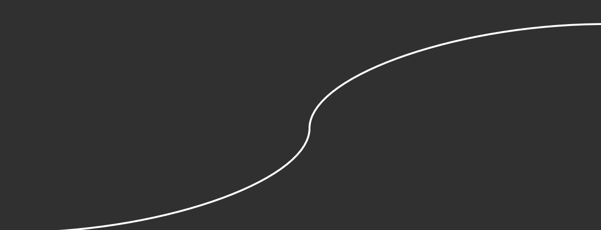
3rd layer can generate
arbitrarily complex boundaries

Can also view 2nd layer as using local knowledge while 3rd layer does global

Activation function does not need to be linear or *sign*

- Recall: brain is highly complex, non-linear, massively-parallel system
- We can use more complex non-linear function: sigmoidal functions



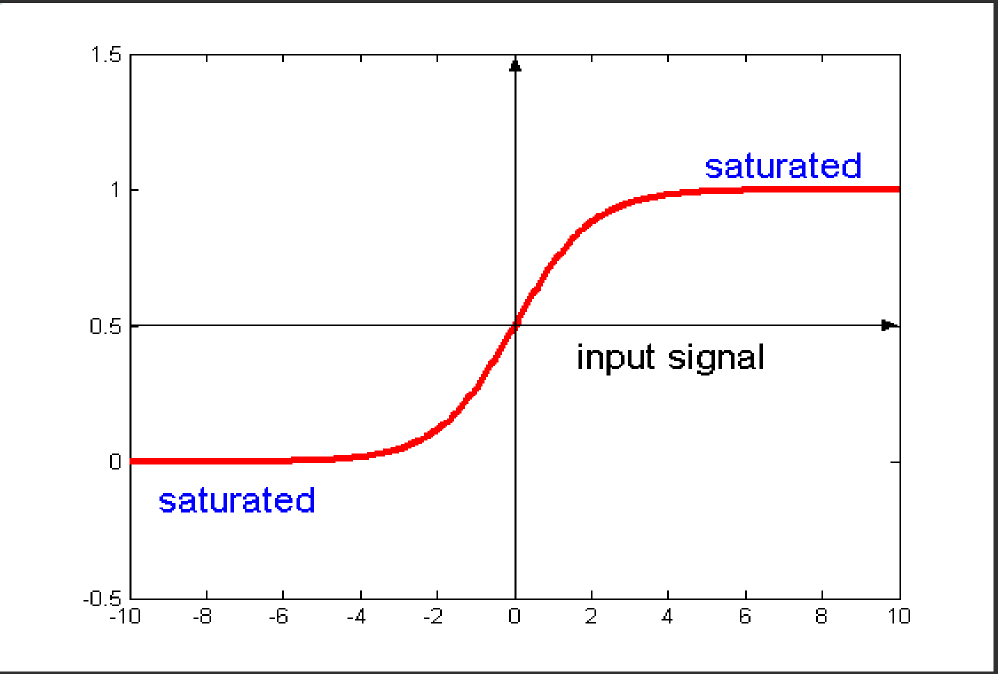


Non-linear activation functions

Sigmoidal (logistic) function-common in ANN

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

where k is a positive constant



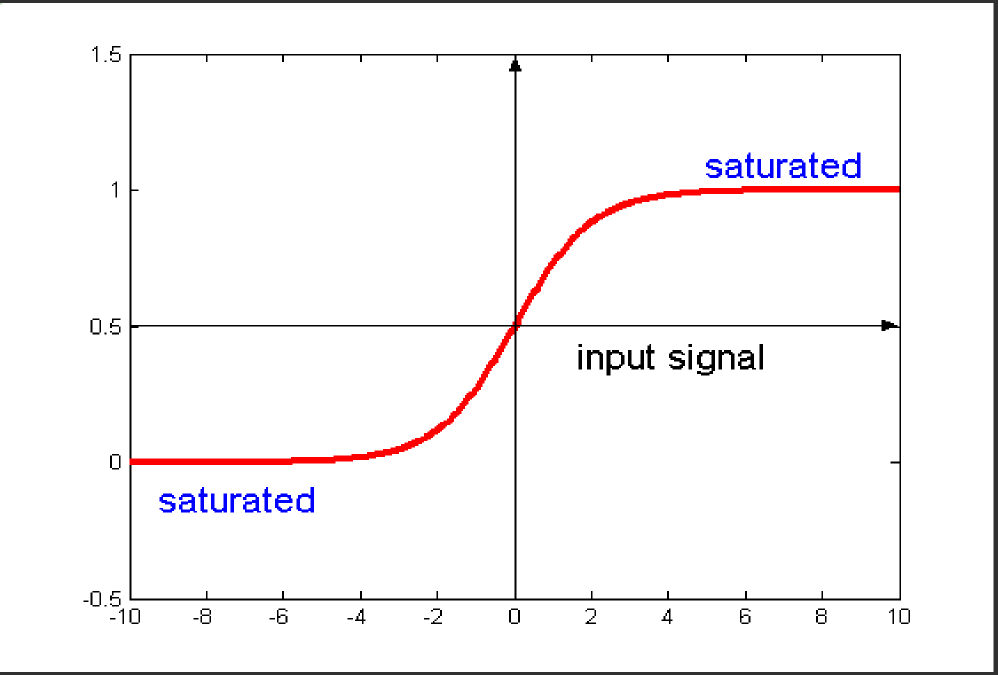
The sigmoidal function gives a value in range of 0 to 1.

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

Note: when net = 0, f = 0.5

Weight adjustment for non-linear activation functions

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



Derivative of activation function

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

Universal Function Approximation

How good is a Multi-Layer model?

Universal Approximation Theorem

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 $H(x_1, \ldots, x_m) = \Sigma^k_{i=1} a_i f(\Sigma^m_{j=1} w_{ij} x_j + b_i)$

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

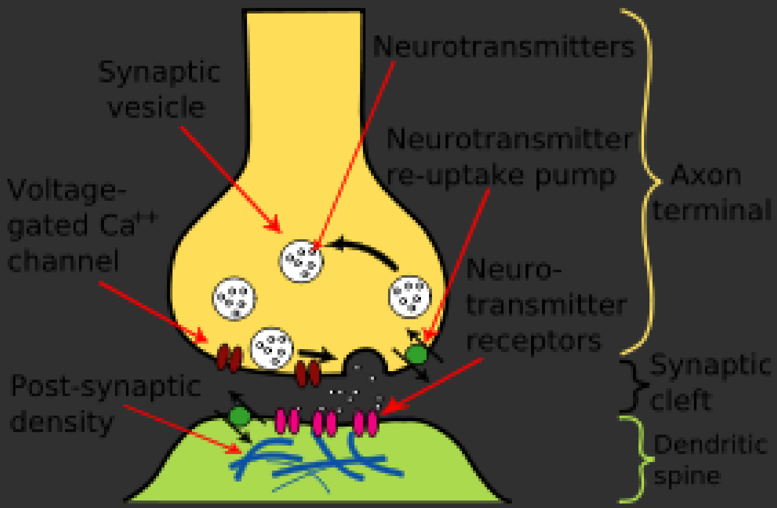
Proof by thinking of superposition of sigmoids

Not practically useful as need arbitrarily large number of units but more of an existence proof

For a 2 layer net: same is true for a 2 layer net providing function is continuous and from one finite dimensional space to another

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 ANN



How does ANN learn

- 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

Learning weights in 3-layer networks

- When we input attribute values of a training record, the activation values are propagated through hidden layer neurons to output neurons.
- The actual network outputs are compared with the desired output, we end up with the error in each of the output units. We want to bring this error to zero.

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

- The simplest method is a greedy method: 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

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

- The solution 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.

Backpropagation learning algorithm ‘BP’

Solution to credit assignment problem in ML NN

Rumelhart, Hinton and Williams (1986)

BP has two phases:

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

Backpropagation learning algorithm ‘BP’

Solution to credit assignment problem in ML NN

Rumelhart, Hinton and Williams (1986)

BP has two phases:

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)

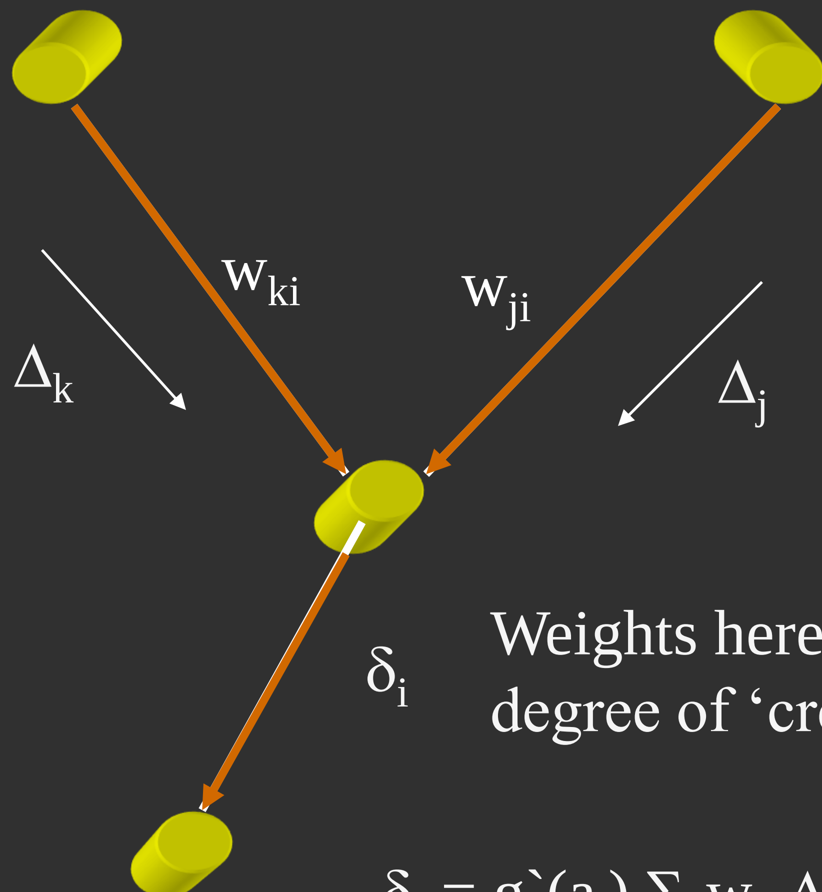
Backpropagation: intuition

- 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

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

Backward Pass



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

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

BP Algorithm (sequential)

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 weights using:

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

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

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

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

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

weight changes will be greatest when units
receives 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 be static: does not change anymore - learned.

Summary of (sequential) BP learning algorithm

Set learning rate η

Set initial weight values (incl. biases): w, v

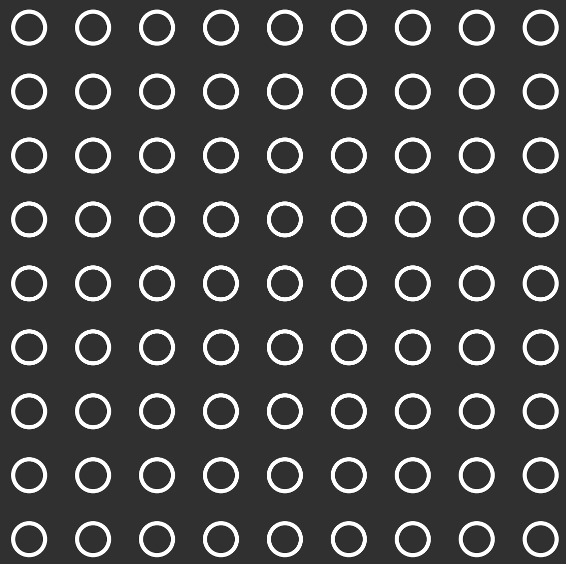
Loop until stopping criteria satisfied:

present input pattern to input units
compute functional signal for hidden units
compute functional signal for output units

present Target response to output units
compute error signal for output units
compute error signal for hidden units
update all weights at the same time
increment n to n+1 and select next input and target

end loop

Application:
Handwriting recognition

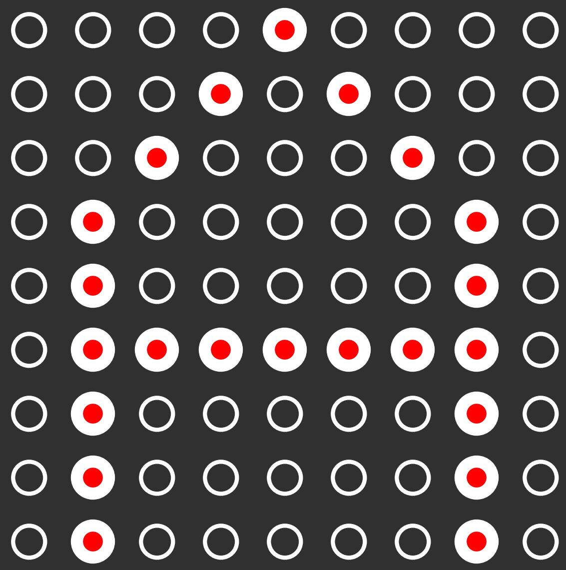


Dataset: collection of handwritings

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

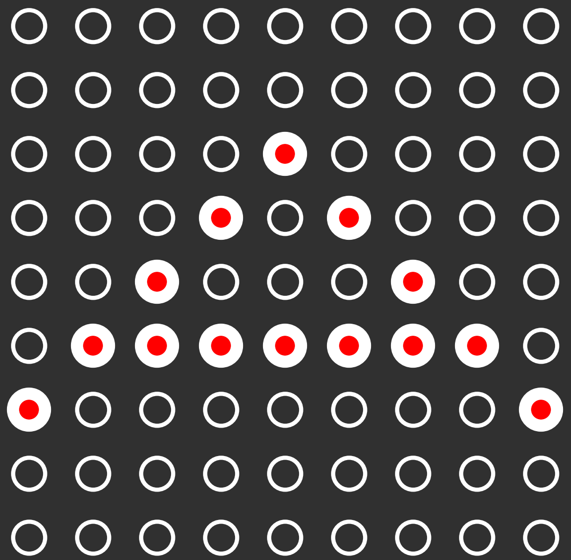
Class: actual letter meant by the writer

Application: Handwriting recognition



Sample training record for class capital A

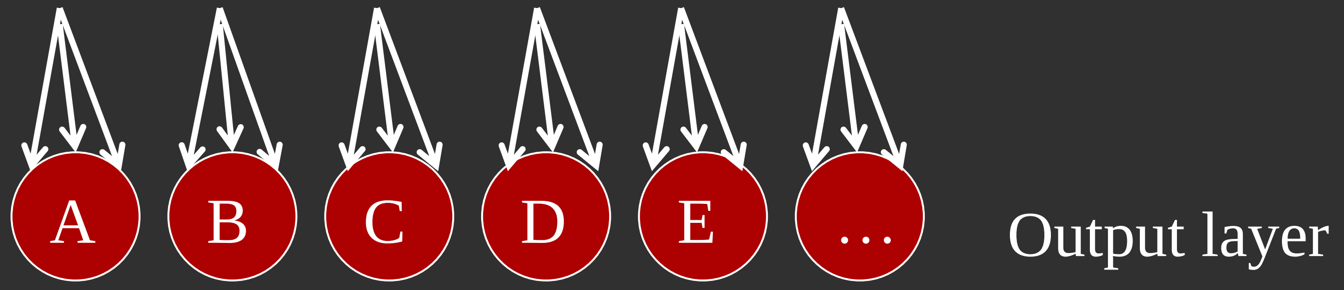
Application: Handwriting recognition



Another training record for class capital letter A

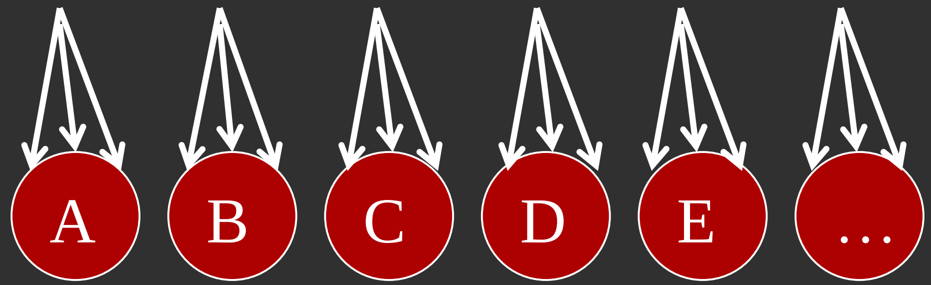
NN for handwriting recognition

- 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

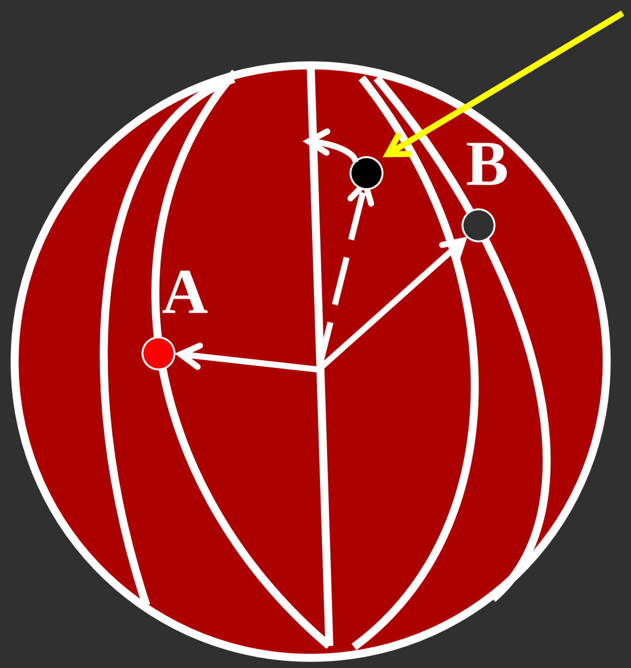
- Multi-class problems are solved by competitive learning
- Initially all weights are random, and each output neuron gets some value
- The class is assigned by the letter with maximum value
- The weights are adjusted in such a way that to increase the correct classification, and to decrease the incorrect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



Output layer

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

Applications of ANNs

- Credit card frauds
- Kinect – gesture recognition
- Facial recognition:
  http://celebrity.myheritage.com/FP/Company/try-face-recognition.php
- Self-driving cars
- …

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