

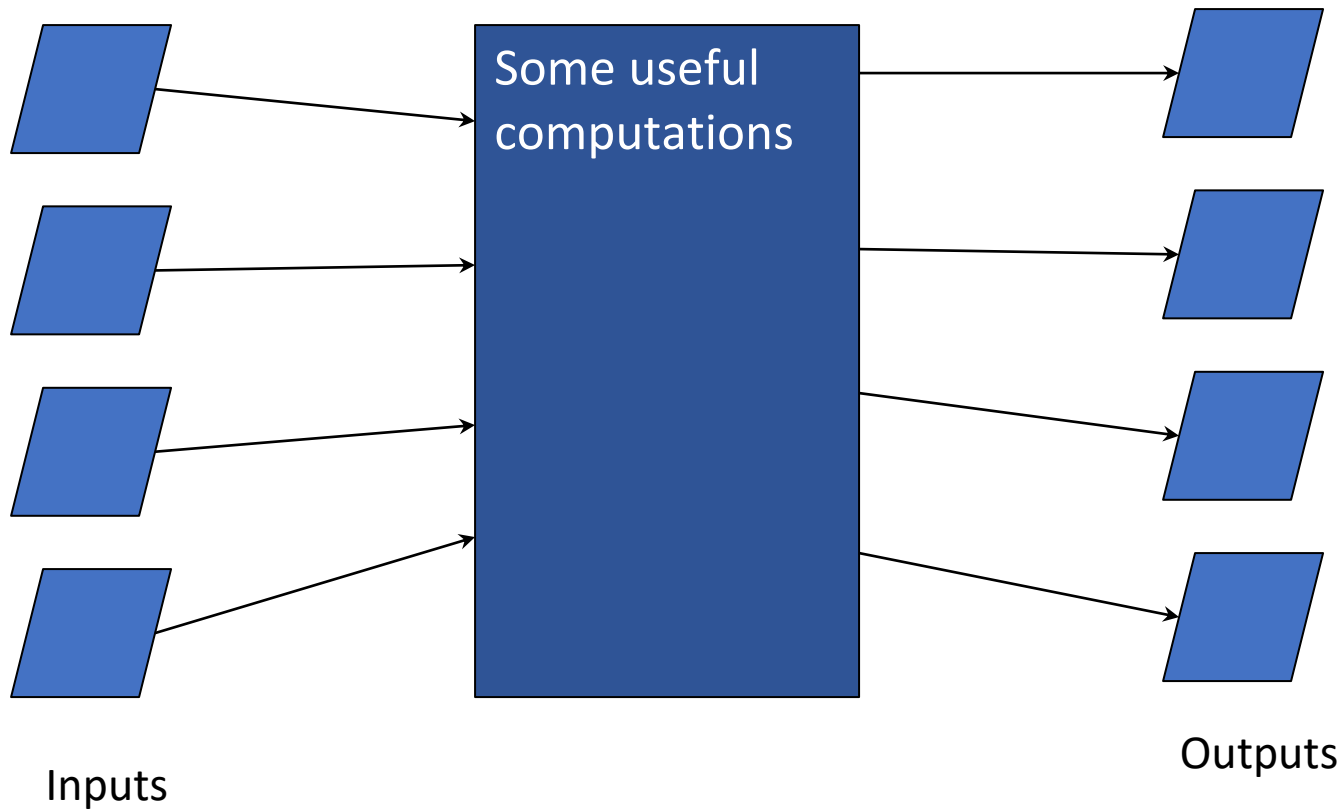
Networks that learn

Lecture 23

by Marina Barsky

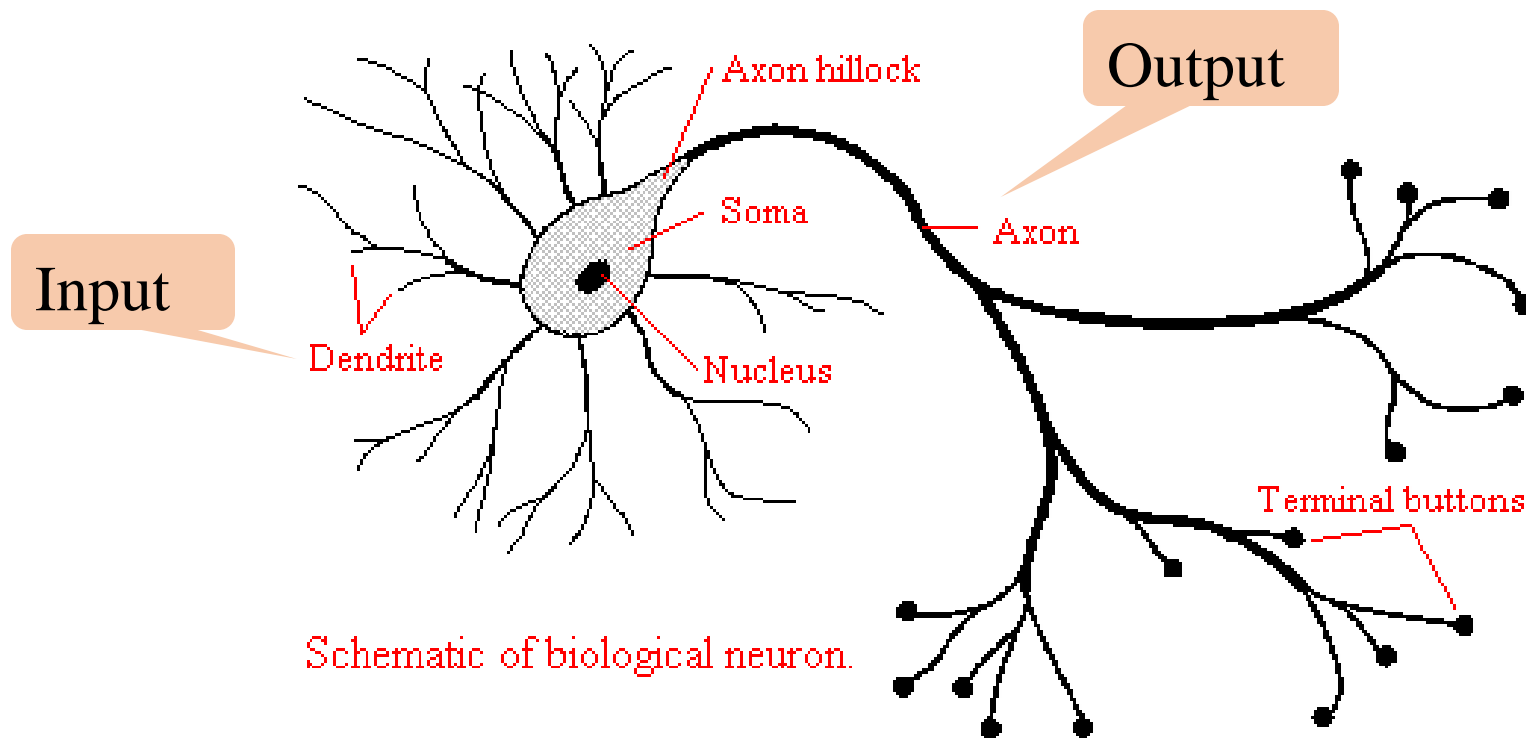
An idea is inspired by
the science of the brain

How computer works



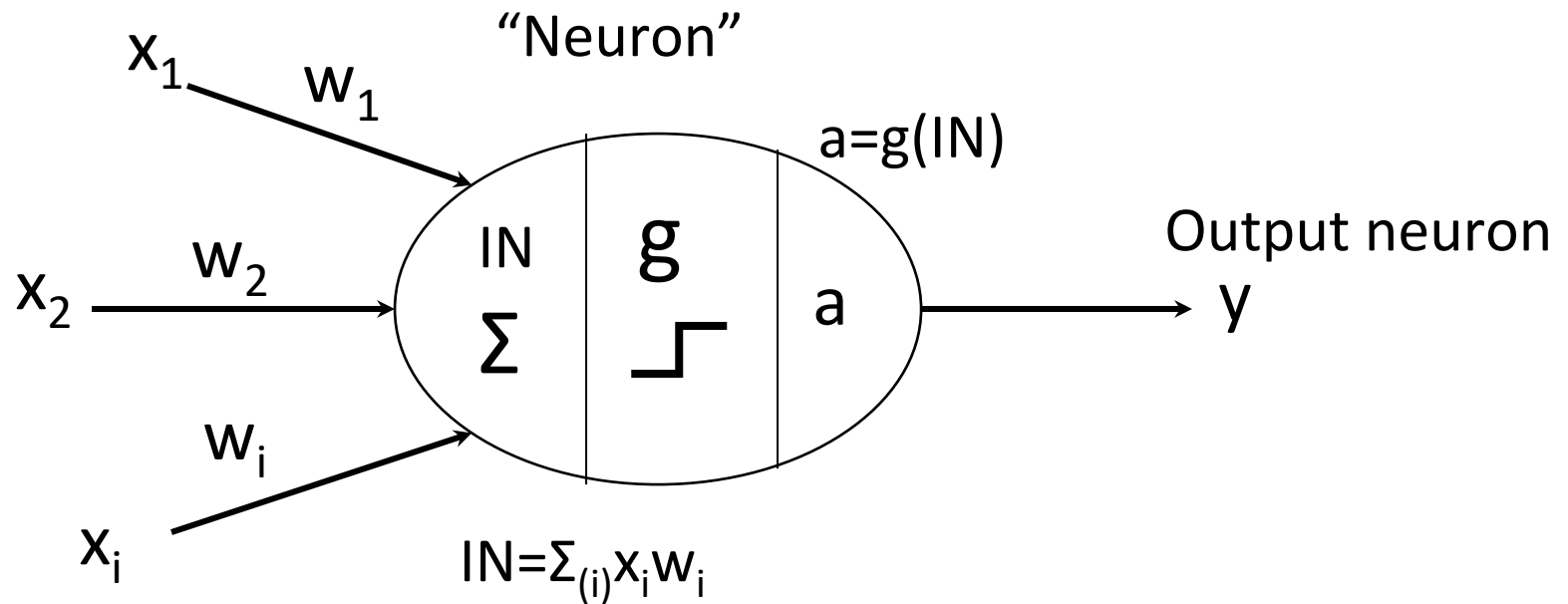
How brain works: neurons

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



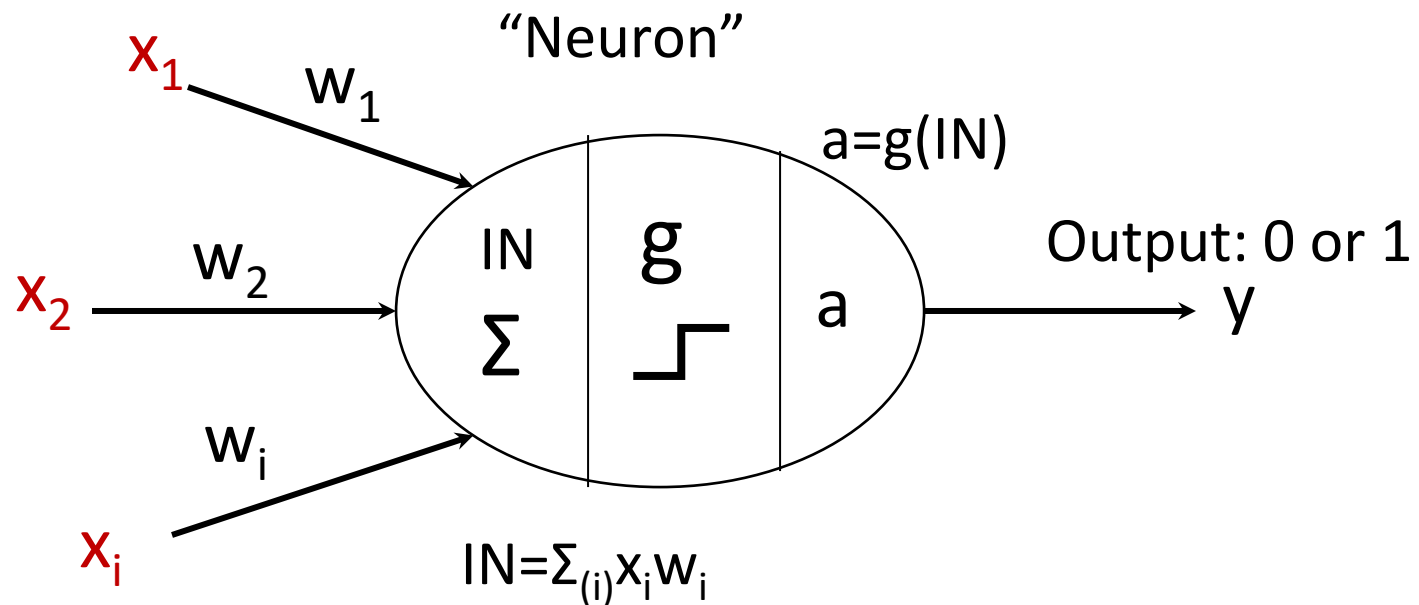
Mathematical model of a neuron (McCulloch and Pitt, 1943)

Input neurons (\mathbf{x})



Input “neurons”

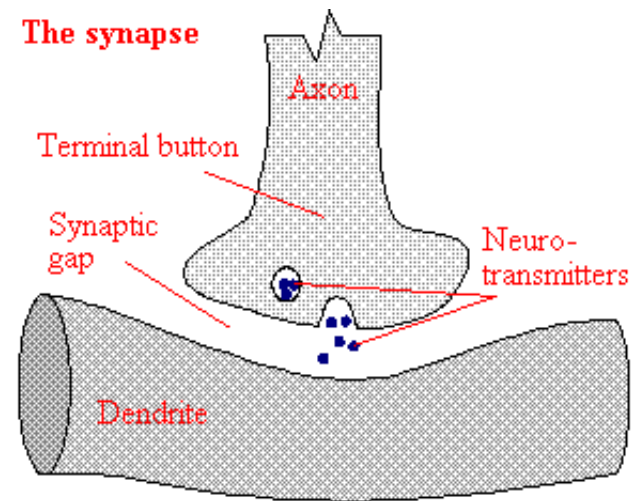
Input vector (\mathbf{x})



- An input vector \mathbf{x} is the data given as one input to the processing “neuron” (corresponds to afferent neurons that transmit information to the brain).

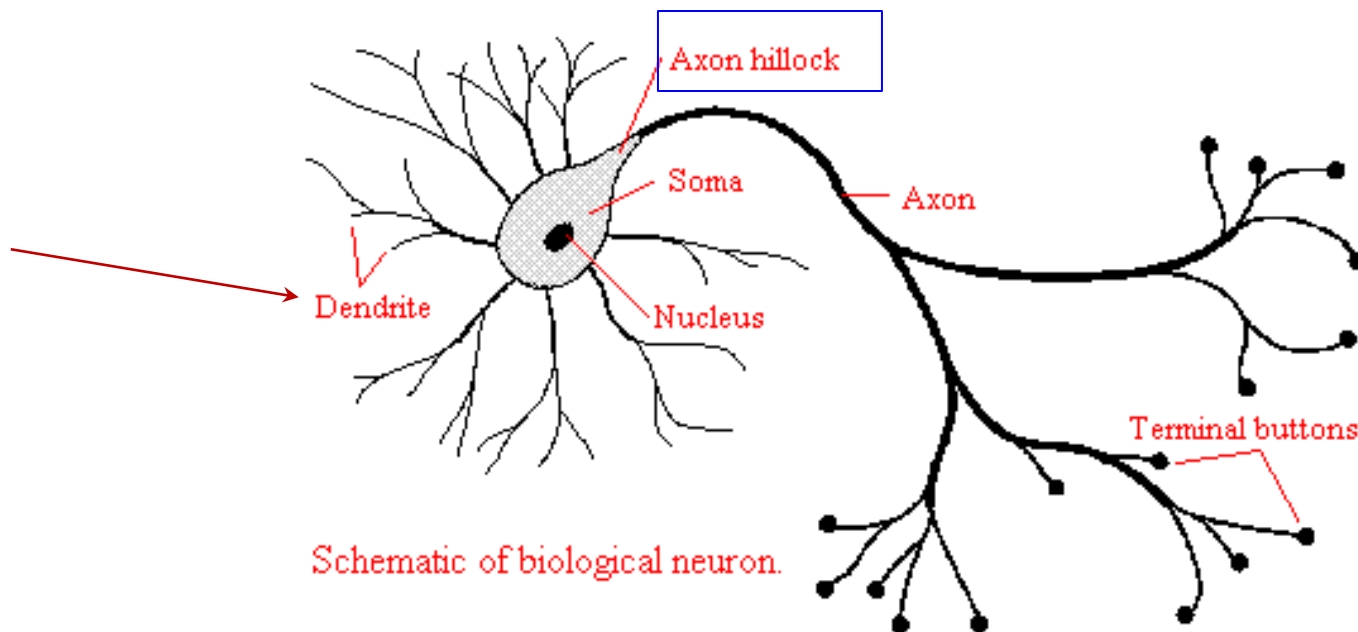
How real neurons communicate

- 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*.



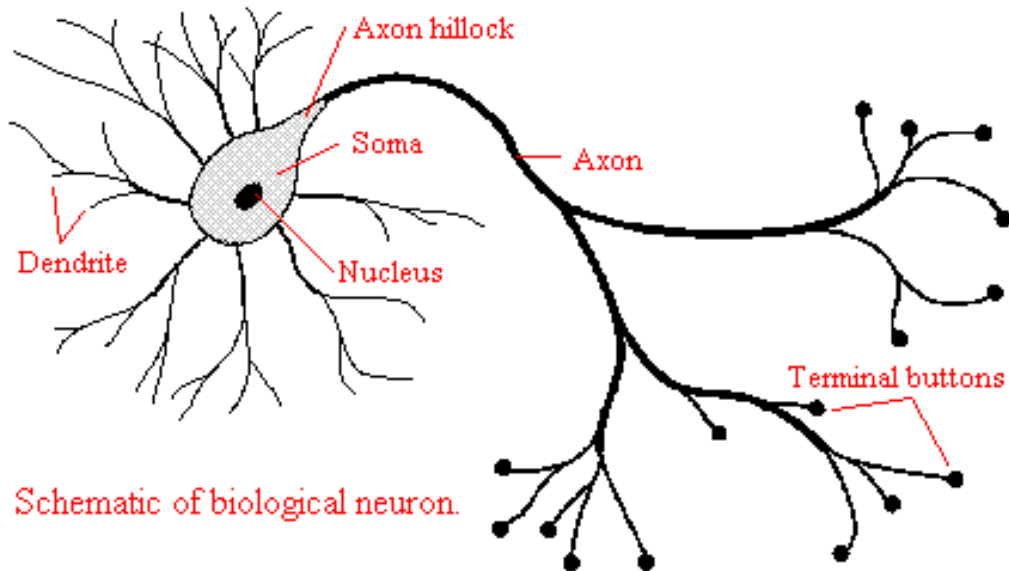
Real neurons: signal summation

- **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**.



Real 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.

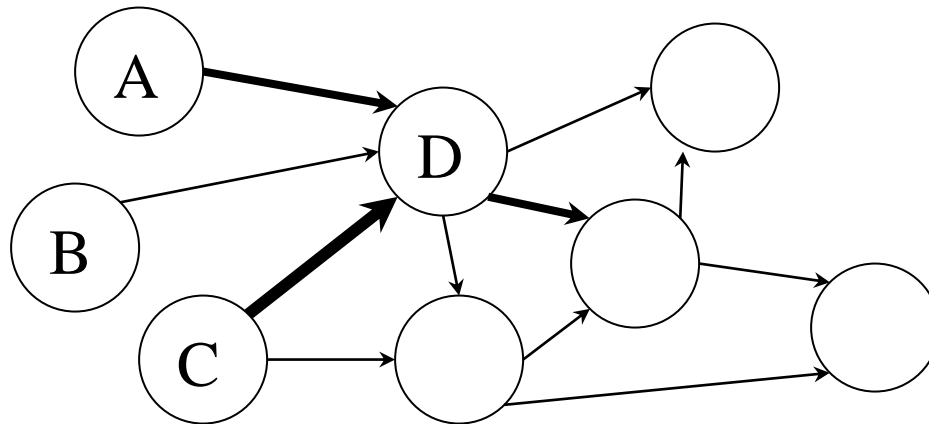


Real neurons: the output signal is constant

- **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.

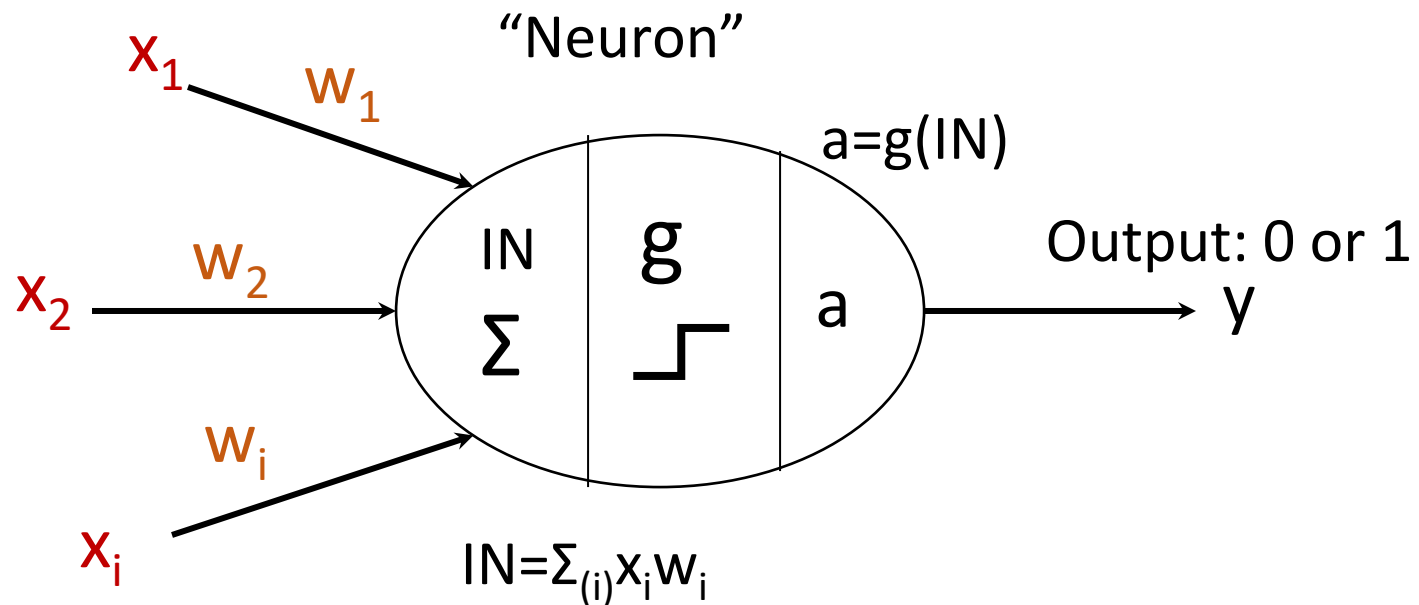
Modeling brain with networks

- 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**.



Model: signal strength (weights)

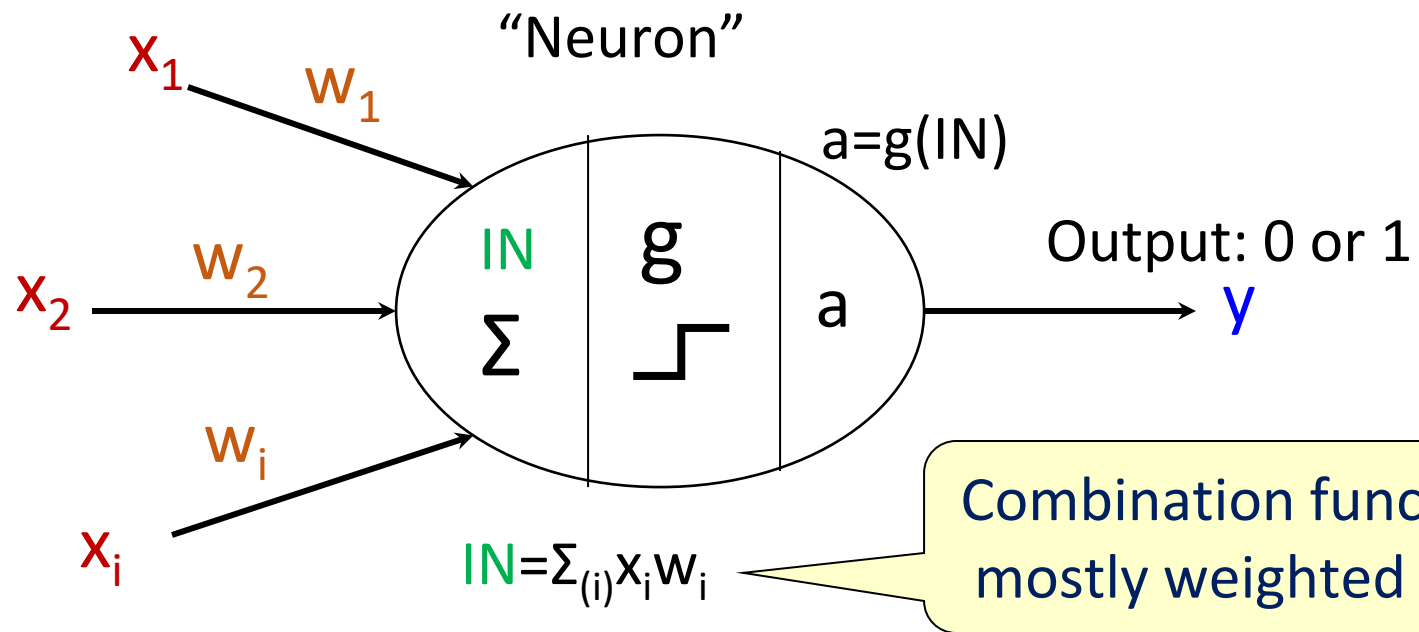
Input vector (\mathbf{x})



- Weights w_i , are the weighted connections between input neurons and the processing neuron (these weights model the strength of synaptic connections in the brain).

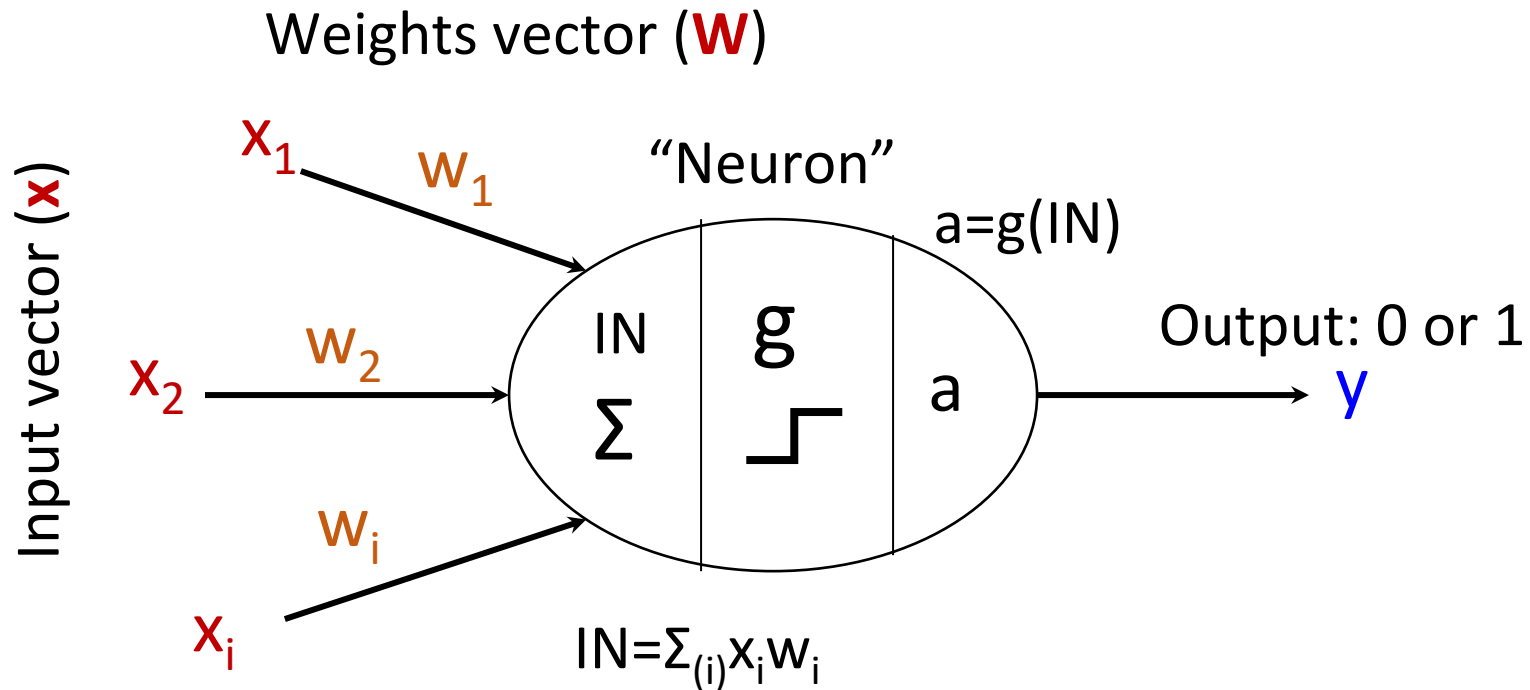
Model: processing “neuron” - signal summation

Input vector (\mathbf{x})



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

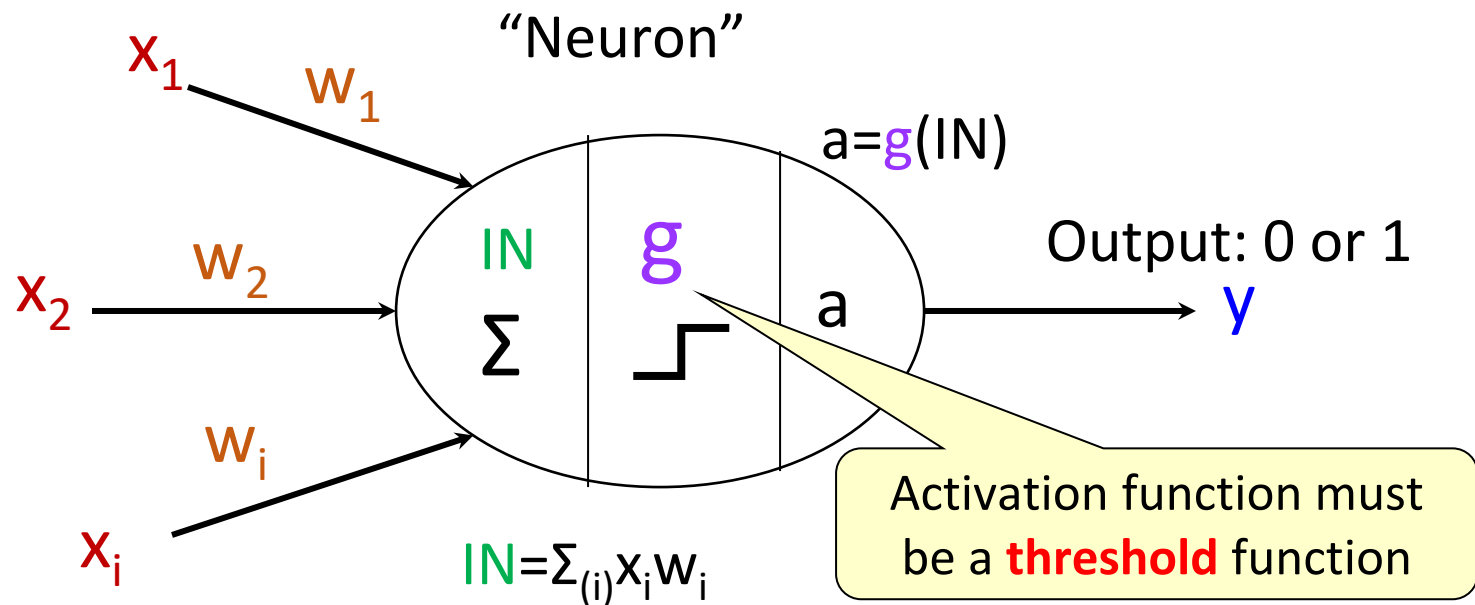
Model: output “neuron”



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

Model: activation threshold

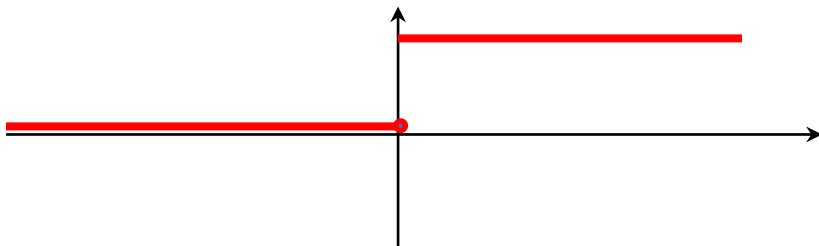
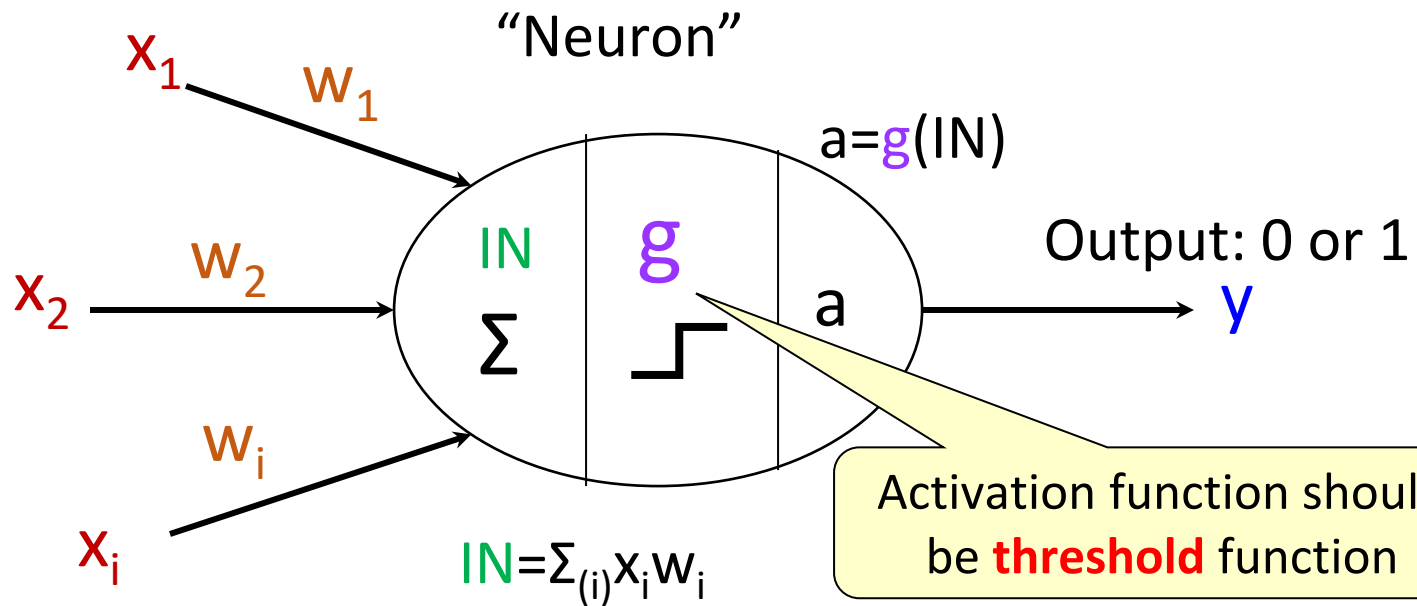
Input vector (\mathbf{x})



- 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.

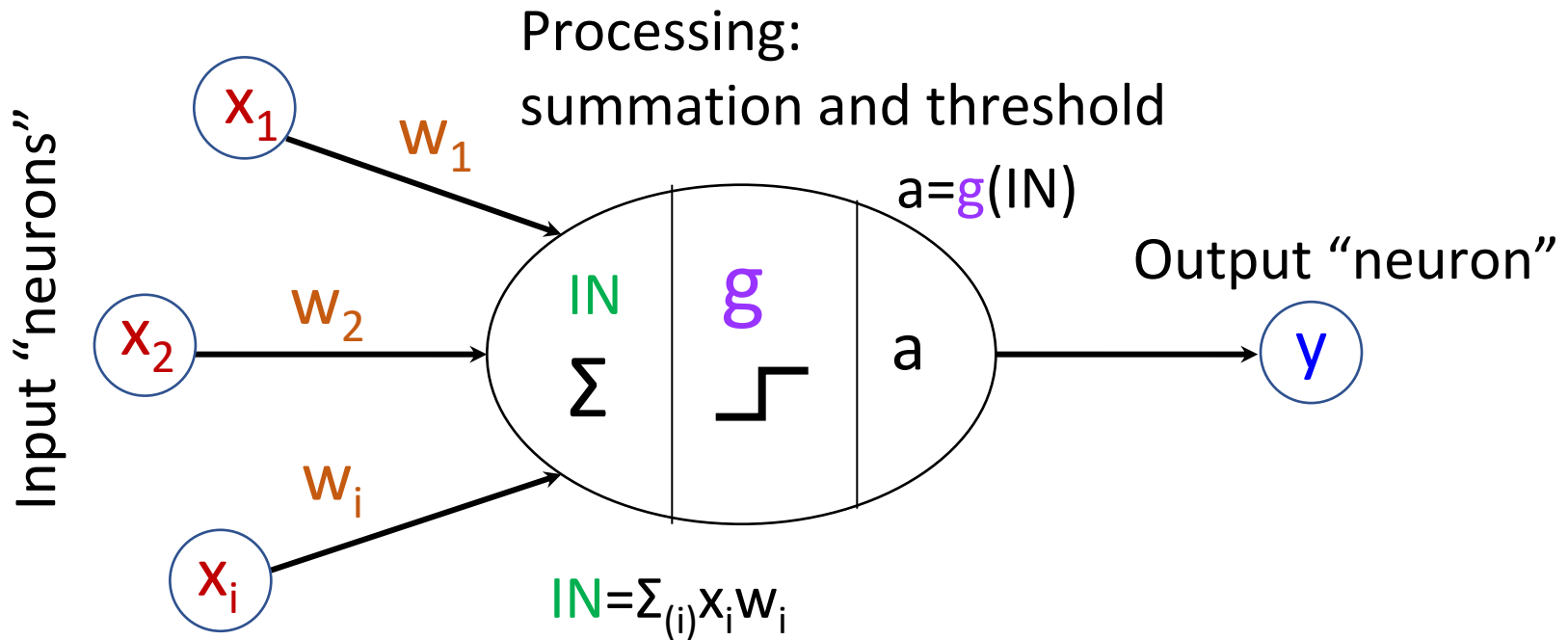
Simple threshold: *sign*

Input vector (\mathbf{x})



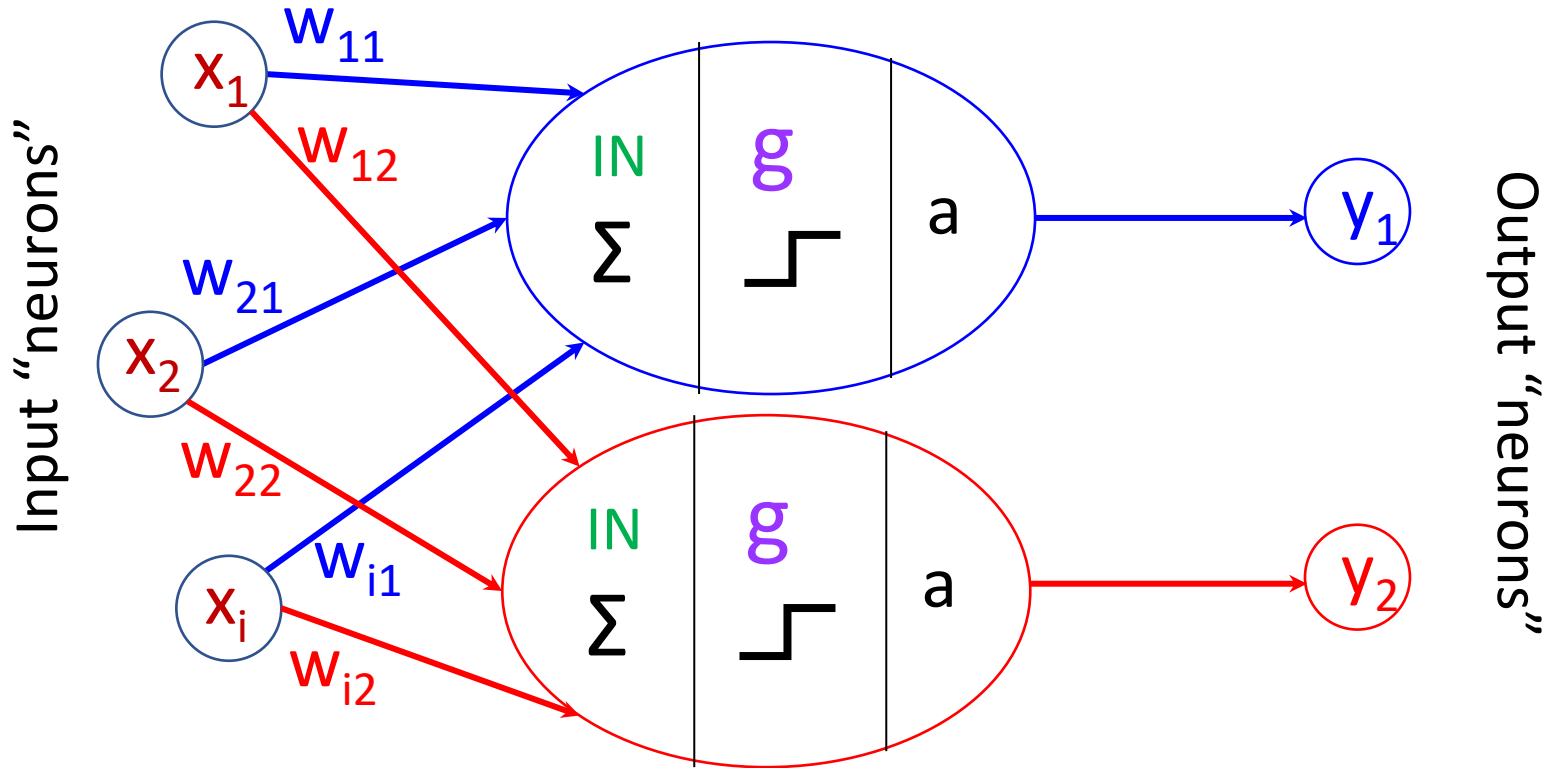
The simplest threshold function: *sign*
 $g(x) = 0$ if $x \leq 0$
 $g(x) = 1$ if $(x > 0)$ (neuron fires)

Model: the goal – predict y



- The model can be used to **predict a target variable y given input vector x** .
- Each input dimension (attribute) can be considered a separate input "neuron"
- Processing happens in the "axon" and based on the result the output neuron "fires" (or not)

Model: multiple predictions

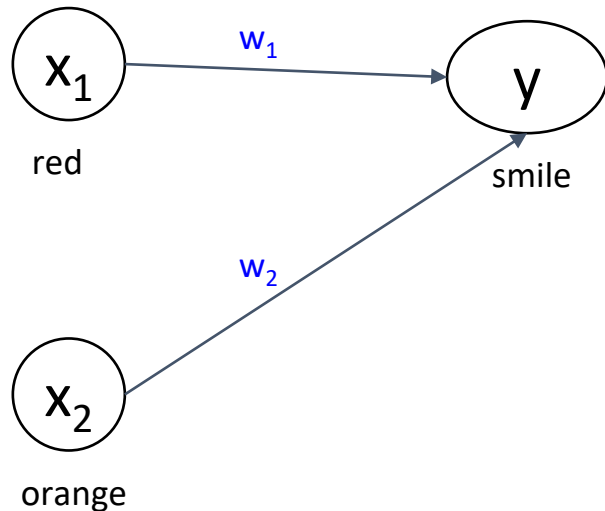


- Conceptually there is no difference between input and output neurons
- So the same input vector can be used to activate multiple output "neurons", using a different set of weights









Let's build some neural networks

Networks that know the meaning of lights

Predicting smiles



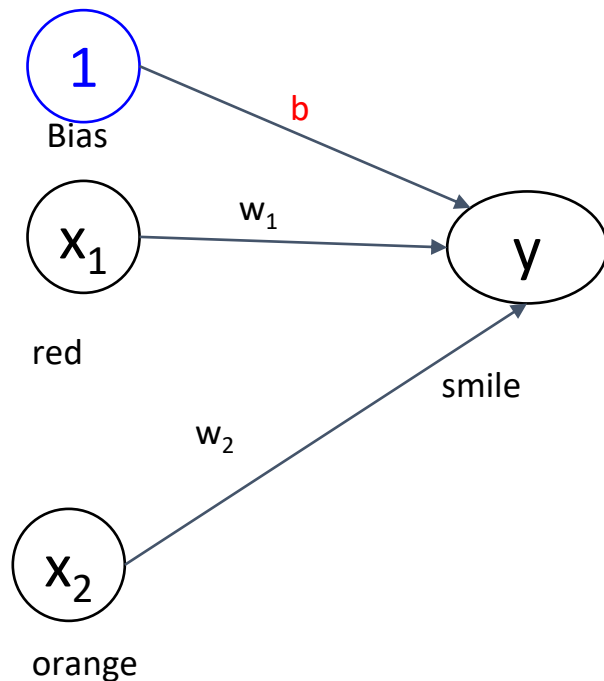
Dataset

red	orange	smile
		neg
		pos
		pos
		pos

- We record people's reaction to lights into a table (dataset)
- Can we set up a single network which when presented with a combination of lights will correctly predict if a person will smile?
- **Setting up the network** means labeling the edges with **correct weights**

Bias node

- When we are presenting the network with combination $[0, 0]$ - then the weights do not matter: the data vector $[0,0]$ is ignored by the network
- To prevent this information loss, we add to the input a special **bias node** which always has a **constant** value, and we assign to it weight b

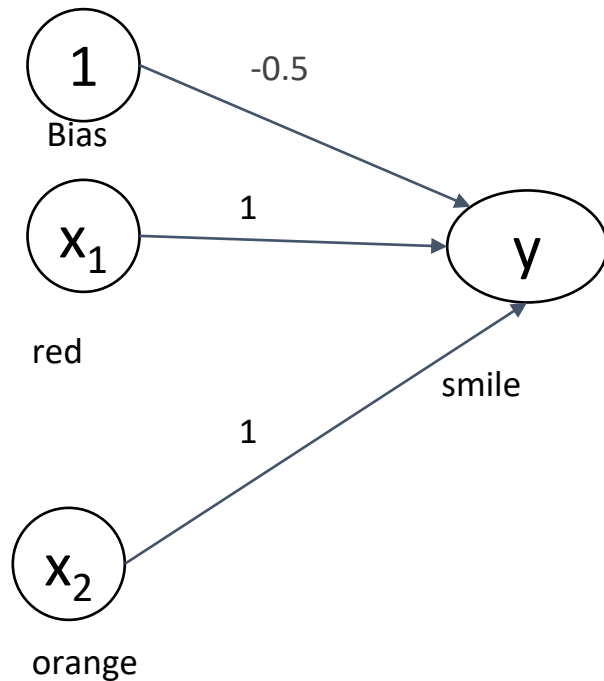


red	orange	smile
		neg
		pos
		pos
		pos

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

Neetwork that predicts smiles

Assigning sample weights

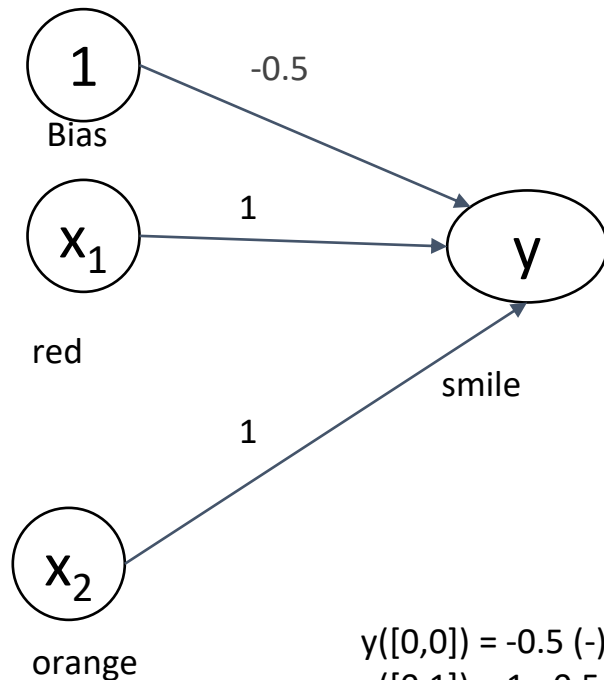


red	orange	smile
		neg
		pos
		pos
		pos

x ₁	x ₂	y
0	0	0
0	1	1
1	0	1
1	1	1

Network that predicts smiles

Checking correctness of predictions



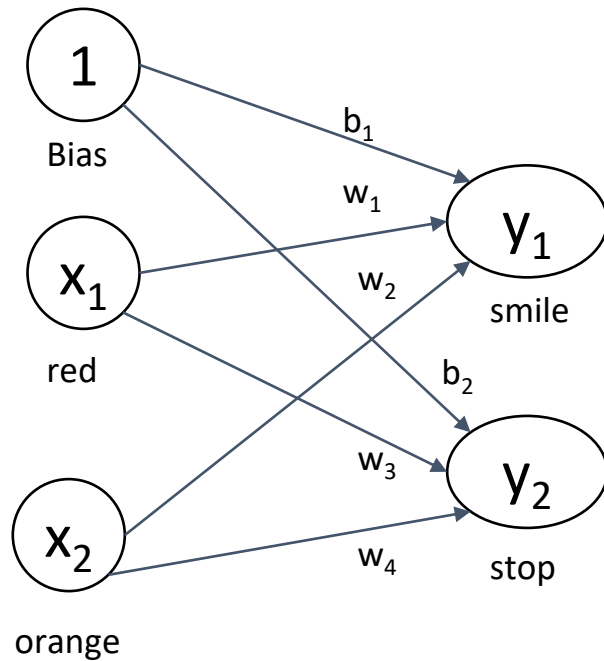
$$\begin{aligned}y([0,0]) &= -0.5 \text{ (-)} \\y([0,1]) &= 1 - 0.5 = 0.5 \text{ (+)} \\y([1,0]) &= 1 - 0.5 = 0.5 \text{ (+)} \\y([1,1]) &= 2 - 0.5 = 1.5 \text{ (+)}\end{aligned}$$



red	orange	smile
		neg
		pos
		pos
		pos

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

Predicting two outputs



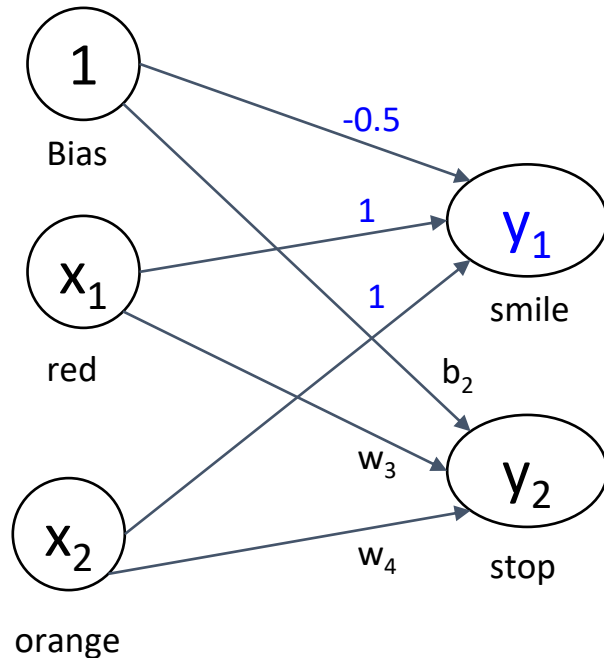
red	orange	smile
<input type="radio"/>	<input type="radio"/>	neg
<input type="radio"/>	<input checked="" type="radio"/>	pos
<input checked="" type="radio"/>	<input type="radio"/>	pos
<input checked="" type="radio"/>	<input checked="" type="radio"/>	pos


red	orange	stop
<input type="radio"/>	<input type="radio"/>	neg
<input type="radio"/>	<input checked="" type="radio"/>	neg
<input checked="" type="radio"/>	<input type="radio"/>	neg
<input checked="" type="radio"/>	<input checked="" type="radio"/>	pos

There is no conceptual difference between input and output nodes

Predicting both smiles and stops

Assigning sample weights for y_1











We already know that this prediction is correct: 









$$y_1([0,0]) = -0.5 (-)$$

$$y_1([0,1]) = 1 - 0.5 = 0.5 (+)$$

$$y_1([1,0]) = 1 - 0.5 = 0.5 (+)$$

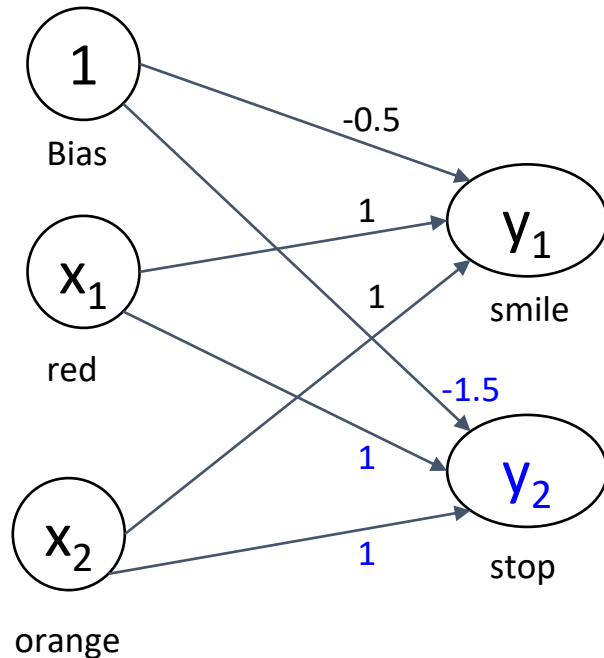
$$y_1([1,1]) = 2 - 0.5 = 1.5 (+)$$


red	orange	smile
		neg
		pos
		pos
		pos

red	orange	stop
		neg
		neg
		neg
		pos

Predicting both smiles and stops

Assigning sample weights for y_2











We already know that this prediction is correct: 









$$y_1([0,0]) = -0.5 \text{ (-)}$$

$$y_1([0,1]) = 1 - 0.5 = 0.5 \text{ (+)}$$

$$y_1([1,0]) = 1 - 0.5 = 0.5 \text{ (+)}$$

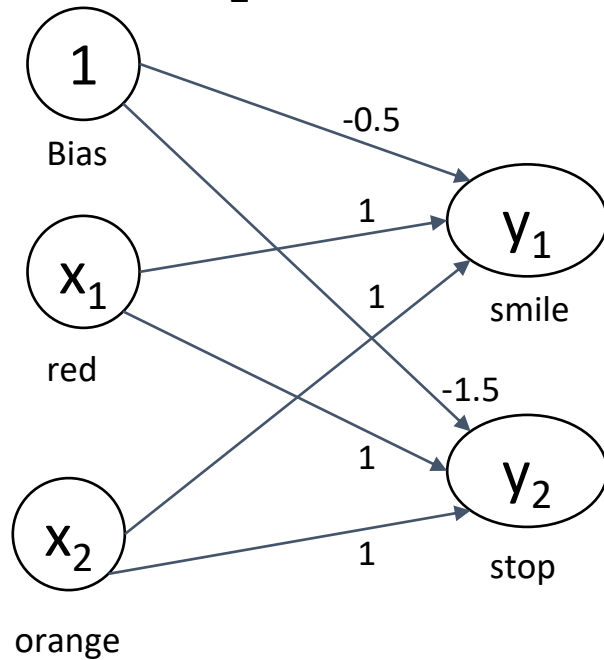
$$y_1([1,1]) = 2 - 0.5 = 1.5 \text{ (+)}$$

red	orange	smile
		neg
		pos
		pos
		pos

red	orange	stop
		neg
		neg
		neg
		pos

Predicting both smiles and stops

Checking y_2



$$y_1([0,0]) = -0.5 \text{ (-)} \quad \checkmark$$

$$y_1([0,1]) = 1 - 0.5 = 0.5 \text{ (+)}$$

$$y_1([1,0]) = 1 - 0.5 = 0.5 \text{ (+)}$$

$$y_1([1,1]) = 2 - 0.5 = 1.5 \text{ (+)}$$

$$y_2([0,0]) = -1.5 \text{ (-)} \quad \checkmark$$

$$y_2([0,1]) = 1 - 1.5 = -0.5 \text{ (-)}$$

$$y_2([1,0]) = 1 - 1.5 = -0.5 \text{ (-)}$$

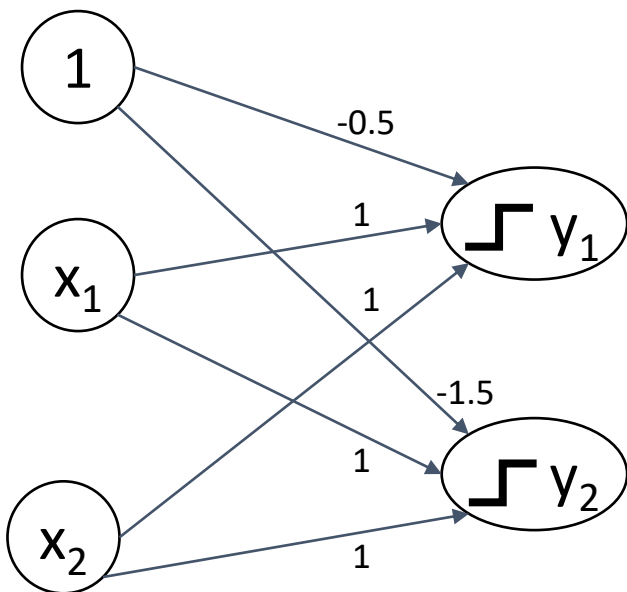
$$y_2([1,1]) = 2 - 1.5 = 0.5 \text{ (+)}$$

red	orange	smile
		neg
		pos
		pos
		pos

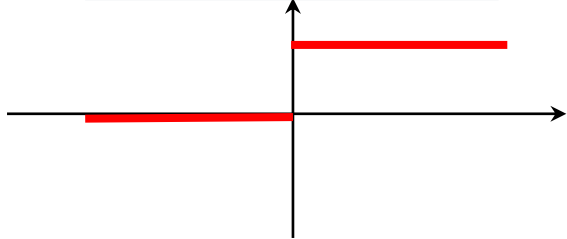
red	orange	stop
		neg
		neg
		neg
		pos

We have built the system that recognizes OR and AND

Apply *sign* function to the output



Function g : *sign*
 $g(x)=0$ if $x \leq 0$
 $g(x)=1$ if $(x > 0)$



Truth table for OR

x_1	x_2	y_1
0	0	0
0	1	1
1	0	1
1	1	1

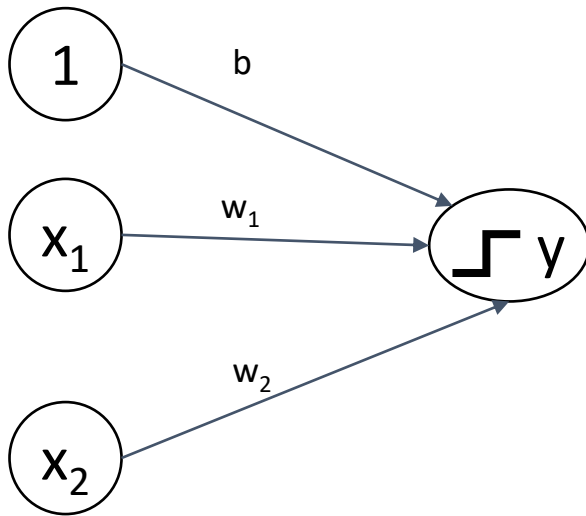
Truth table for AND

x_1	x_2	y_1
0	0	0
0	1	0
1	0	0
1	1	1

$y_1([0,0]) = \text{sign}(-0.5) = 0$ ✓
 $y_1([0,1]) = \text{sign}(1 - 0.5) = 1$
 $y_1([1,0]) = \text{sign}(1 - 0.5) = 1$
 $y_1([1,1]) = \text{sign}(2 - 0.5) = 1$

$y_2([0,0]) = \text{sign}(-1.5) = 0$ ✓
 $y_2([0,1]) = \text{sign}(1 - 1.5) = 0$
 $y_2([1,0]) = \text{sign}(1 - 1.5) = 0$
 $y_2([1,1]) = \text{sign}(2 - 1.5) = 1$

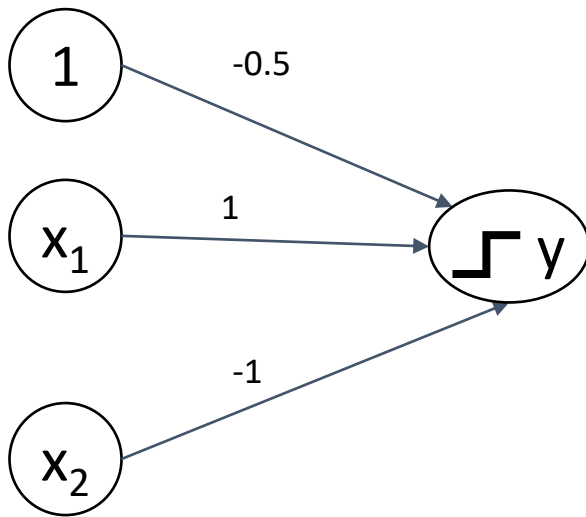
Can we build a system that recognizes: x_1 AND NOT x_2 ?



Truth table for AND NOT

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

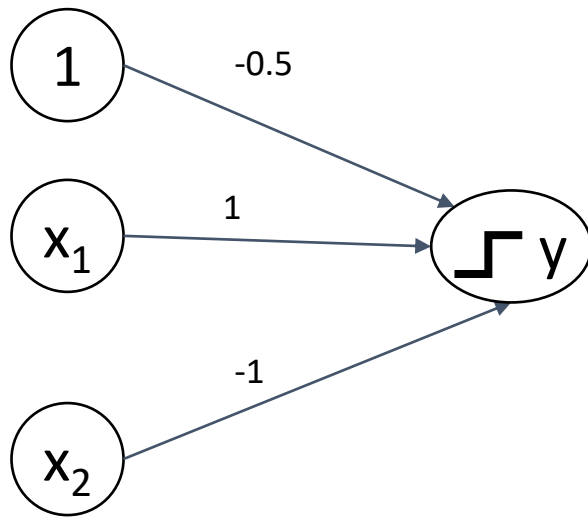
System that recognizes: x_1 AND NOT x_2



Truth table for AND NOT

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

System that recognizes: x_1 AND NOT x_2



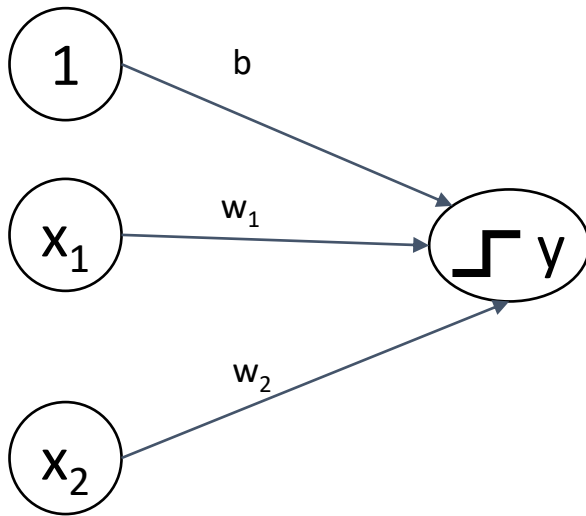
Truth table for AND NOT

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

$$\begin{aligned}y([0,0]) &= \text{sign}(-0.5) = 0 \\y([0,1]) &= \text{sign}(0 - 1 - 0.5) = 0 \\y([1,0]) &= \text{sign}(1 + 0 - 0.5) = 1 \\y([1,1]) &= \text{sign}(1 - 1 - 0.5) = 0\end{aligned}$$



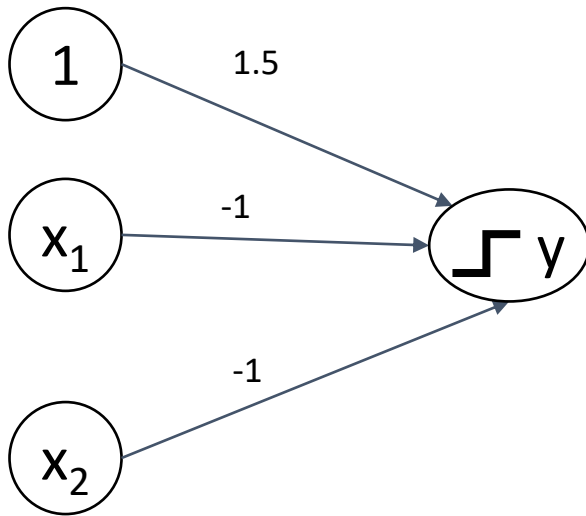
How about: NOT (x_1 AND x_2)



Truth table for NOT AND

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

System that recognizes: NOT (x_1 AND x_2)

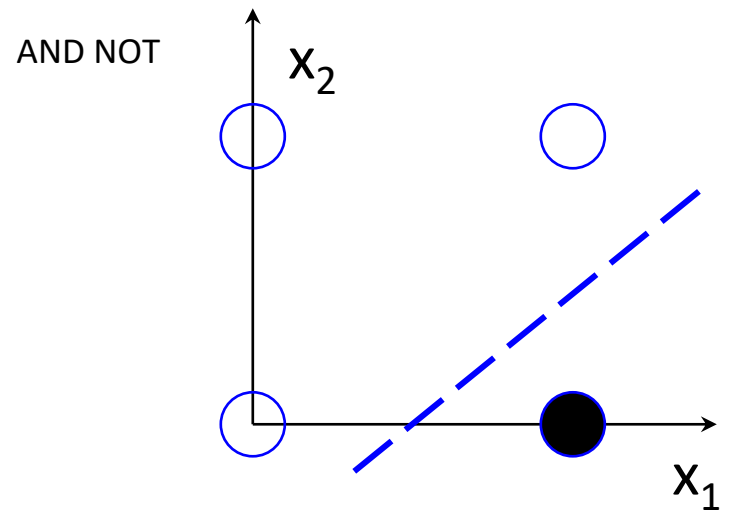
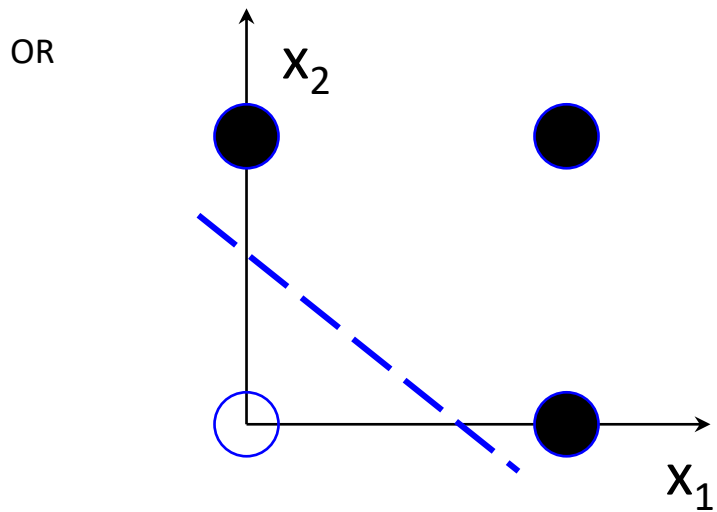
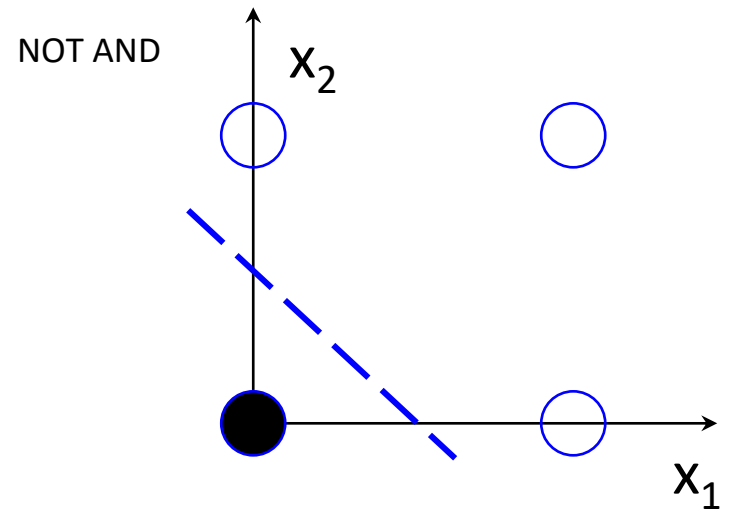
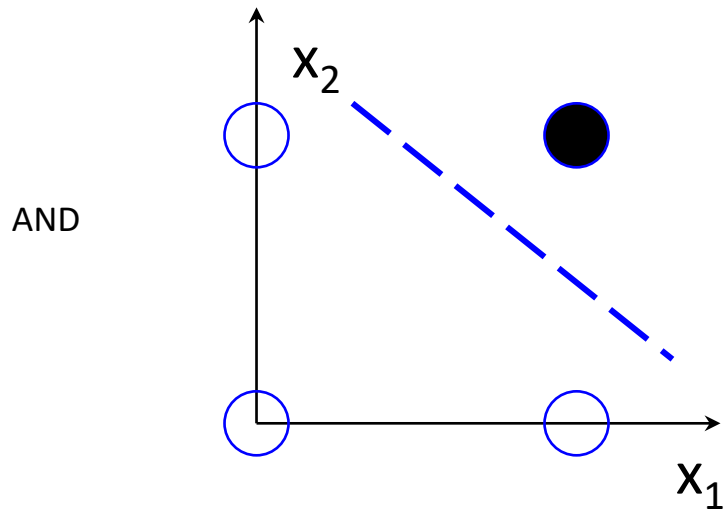


Truth table for NOT AND

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

$y([0,0]) = \text{sign}(1.5) = 1$ ✓
 $y([0,1]) = \text{sign}(-1 + 1.5) = 1$
 $y([1,0]) = \text{sign}(-1 + 1.5) = 1$
 $y([1,1]) = \text{sign}(-2 + 1.5) = 0$

Our network is able to recognize linearly-separable binary classes



Why it works

- The network assumes that there is a **linear correlation** between the input vector \mathbf{x} and the output y
- We just need to discover the equation of the separating line (hyperplane) $y = \mathbf{w}\mathbf{x} + b$, which expresses this linear correlation

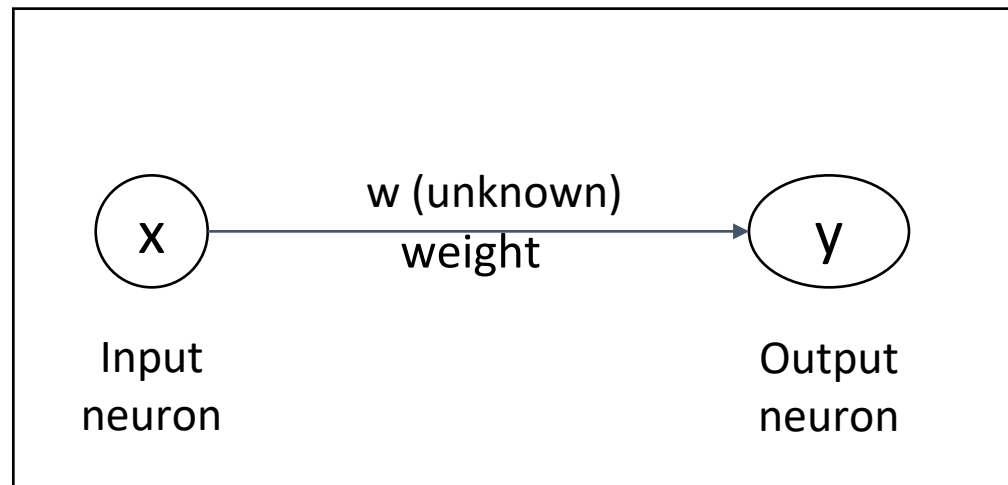
Can machines learn the network for a given problem automatically?

Yes, by looking at the labeled dataset (**supervised learning**)

Predict → **Compare** → **Learn** from errors

Neuron with learning capabilities: *Perceptron* (Rosenblatt, 1958)

- The network can learn its own weights
- It is presented with a set of inputs and known outputs
- Originally the predicted output is different from the actual output by some error
- We adjust the connection weights to produce a smaller error



Most basic Perceptron

Adjusting the weights with gradient descent: error

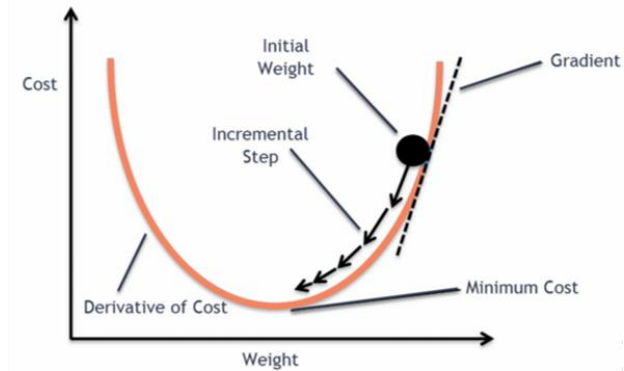


Objective error function - in this case:

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

where $y = w * x$ (predicted value), and t is the actual value of y , known from the labeled dataset

$$\frac{\partial E}{\partial y} = \frac{1}{2} * 2 (y - t) = y - t$$



The error depends on weight

Adjusting the weights with gradient descent: derivative



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

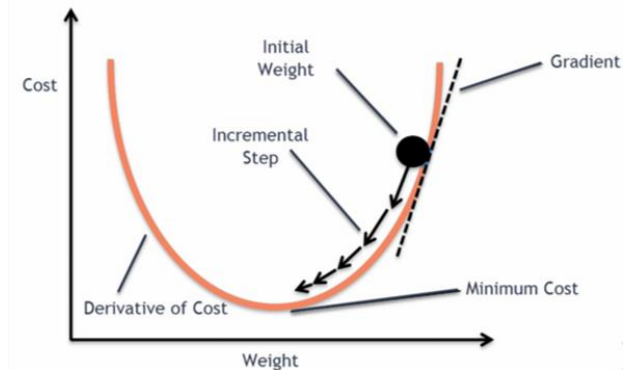
$$y = w * x$$

$$\partial E / \partial y = y - t$$

To determine how to change weight w take derivative of E at point w

$$\Delta = \partial E / \partial w = \partial E / \partial y * \partial y / \partial w = (w * x - t) * x$$

If derivative is positive (function on the rise) we need to decrease the weight, if it is negative - we need to increase the weight



The error depends on weight

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

Chain rule!

Adjusting the weights with gradient descent: delta rule

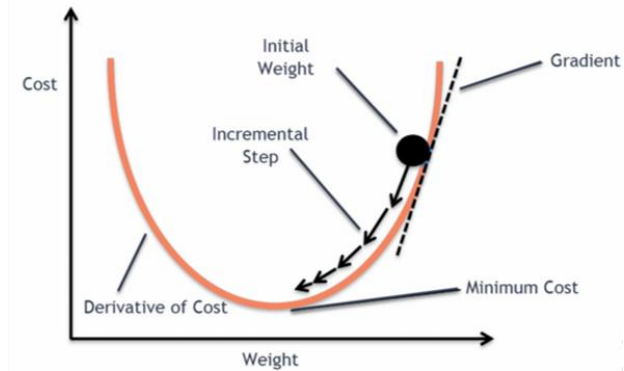


$$E = \frac{1}{2} (y - t)^2$$

$$y = w * x$$

$$\frac{\partial E}{\partial y} = y - t$$

$$\Delta = \frac{\partial E}{\partial w} = (w * x - t) * x$$



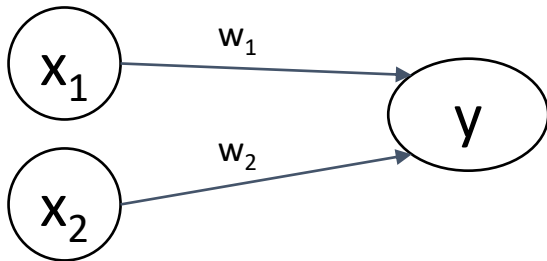
The error depends on weight

Delta rule: adjust weight w by Δ

$$w = w - \frac{\partial E}{\partial w} = w - \Delta = w - (w * x - t) * x$$

More input dimensions - more weights to adjust

The network transforms input feature vector into target using two weights



The principle is the same:

$$E = \frac{1}{2} (w_1 * x_1 + w_2 * x_2 - t)^2$$

Which weight contributed more to the error?

Partial derivatives with respect to each weight:

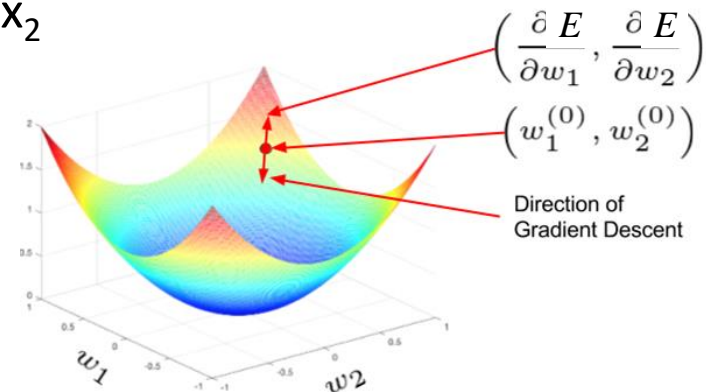
$$\frac{\partial E}{\partial w_1} = (w_1 * x_1 - t) * x_1$$

$$\frac{\partial E}{\partial w_2} = (w_2 * x_2 - t) * x_2$$

Delta rules: update weights

$$w_1 = w_1 - \frac{\partial E}{\partial w_1}$$

$$w_2 = w_2 - \frac{\partial E}{\partial w_2}$$



There is also a **bias** node, of course

Objective function: $E = \frac{1}{2} (w_1 * x_1 + w_2 * x_2 + b - t)^2$

$$\frac{\partial E}{\partial w_1} = (w_1 * x_1 - t) * x_1$$

$$\frac{\partial E}{\partial w_2} = (w_2 * x_2 - t) * x_2$$

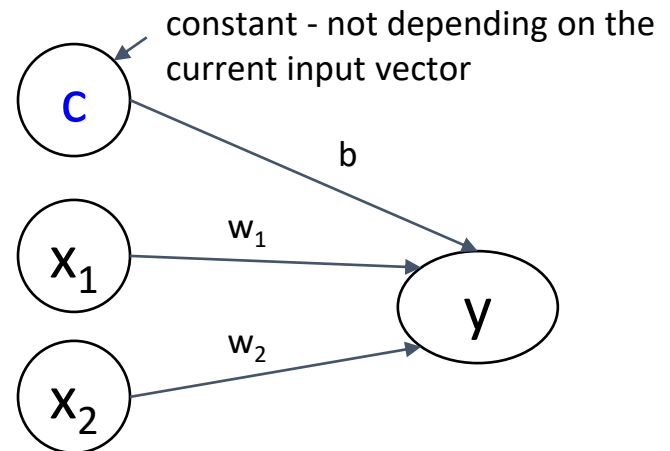
$$\frac{\partial E}{\partial b} = (b * c - t) * c$$

Delta rule:

$$w_1 = w_1 - \frac{\partial E}{\partial w_1}$$

$$w_2 = w_2 - \frac{\partial E}{\partial w_2}$$

$$b = b - \frac{\partial E}{\partial b}$$



Incorporating learning rate η (eta)

$$w_1 = w_1 - \eta * \partial E / \partial w_1$$

$$w_2 = w_2 - \eta * \partial E / \partial w_2$$

$$b = b - \eta * \partial E / \partial b$$

where:

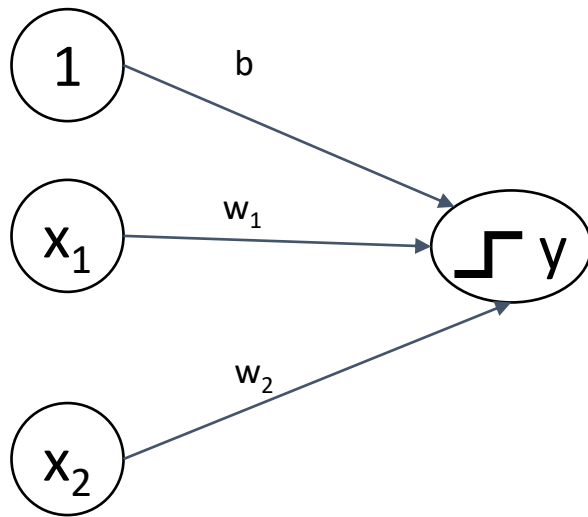
$$\partial E / \partial w_1 = (w_1 * x_1 - t) * x_1$$

$$\partial E / \partial w_2 = (w_2 * x_2 - t) * x_2$$

$$\partial E / \partial b = (cb - t) * c$$

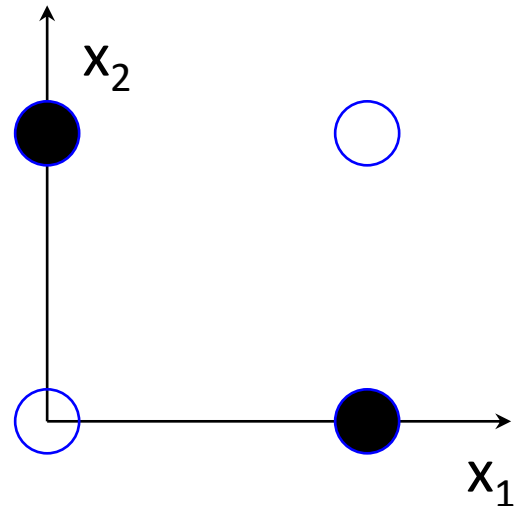
Experiment with *basic perceptron*
[here](#)

Let's try to build a perceptron that recognizes XOR

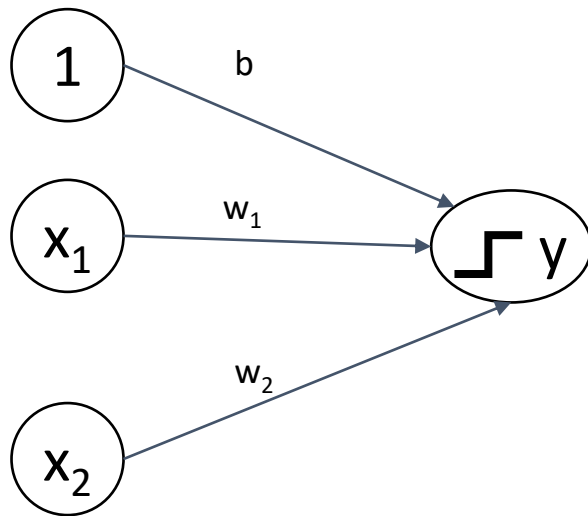


Truth table for XOR

x_1	x_2	o
0	0	0
0	1	1
1	0	1
1	1	0



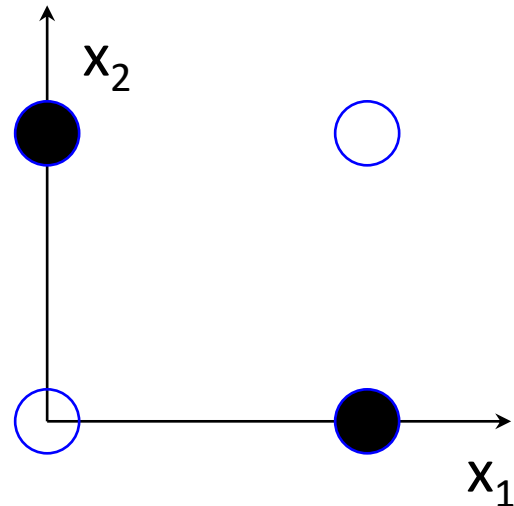
Let's try to build a perceptron that recognizes XOR



Truth table for XOR

x_1	x_2	o
0	0	0
0	1	1
1	0	1
1	1	0

We can't!

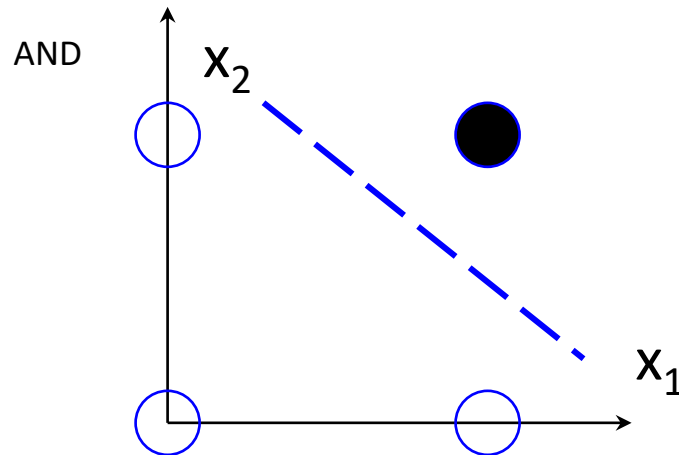
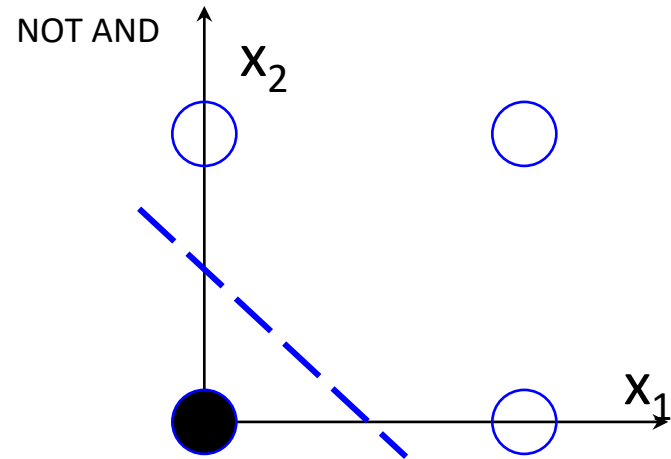
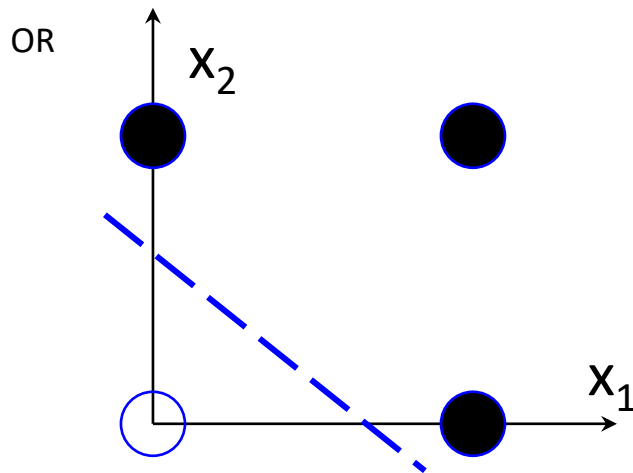


This failure caused a major delay in developing the idea of ANN in the 60s

Idea:

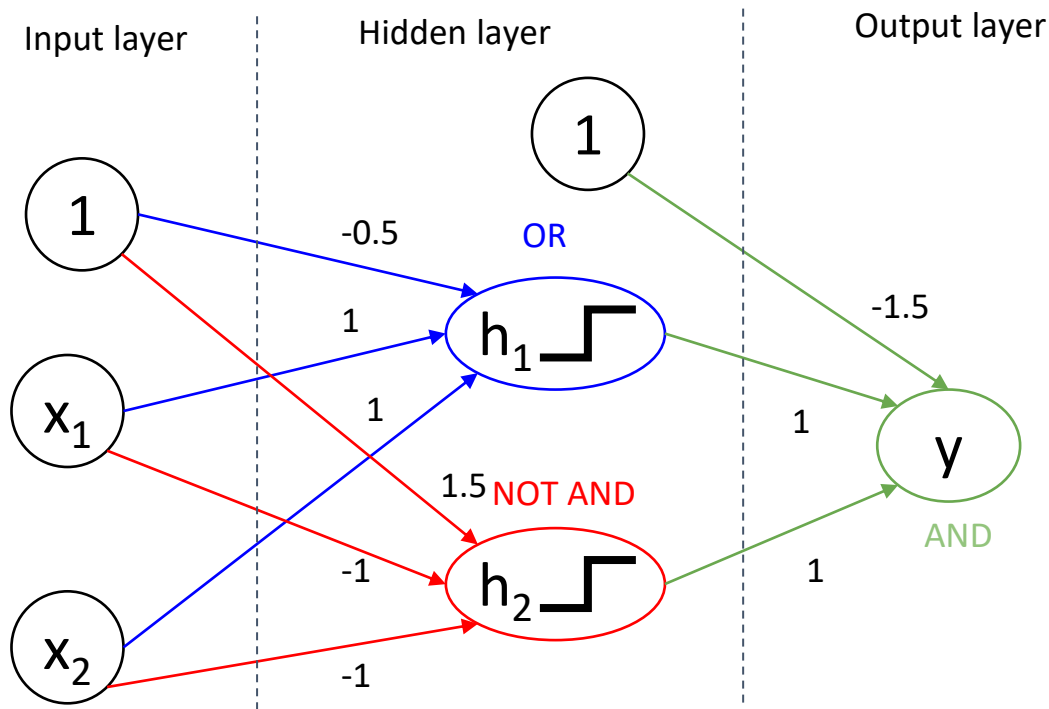
express XOR through known solutions

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



Add more layers

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



Truth table for XOR

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

$$\begin{aligned} h_1([0,0]) &= -0.5 (-) && \rightarrow 0 \\ h_1([0,1]) &= 1 - 0.5 = 0.5 (+) && \rightarrow 1 \\ h_1([1,0]) &= 1 - 0.5 = 0.5 (+) && \rightarrow 1 \\ h_1([1,1]) &= 2 - 0.5 = 1.5 (+) && \rightarrow 1 \end{aligned}$$

$$\begin{aligned} h_2([0,0]) &= 1.5 (+) && \rightarrow 1 \\ h_2([0,1]) &= -1 + 1.5 = 0.5 (+) && \rightarrow 1 \\ h_2([1,0]) &= -1 + 1.5 = 0.5 (+) && \rightarrow 1 \\ h_2([1,1]) &= -2 + 1.5 = -0.5 (-) && \rightarrow 0 \end{aligned}$$

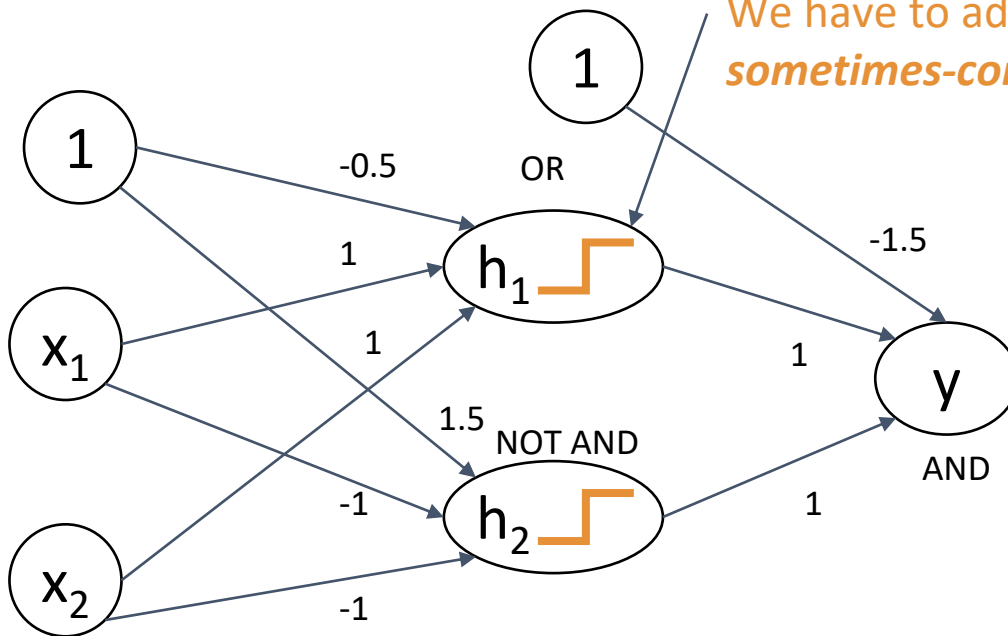
$$\begin{aligned} y([0,0]) &= (-) && \rightarrow 0 \\ y([0,1]) &= (+) && \rightarrow 1 \\ y([1,0]) &= (+) && \rightarrow 1 \\ y([1,1]) &= (-) && \rightarrow 0 \end{aligned}$$



Importance of nonlinearity!

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

Just combining linear separators would not work!
 We have to add some sort of nonlinearity or *sometimes-correlation* between the layers



Truth table for XOR

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

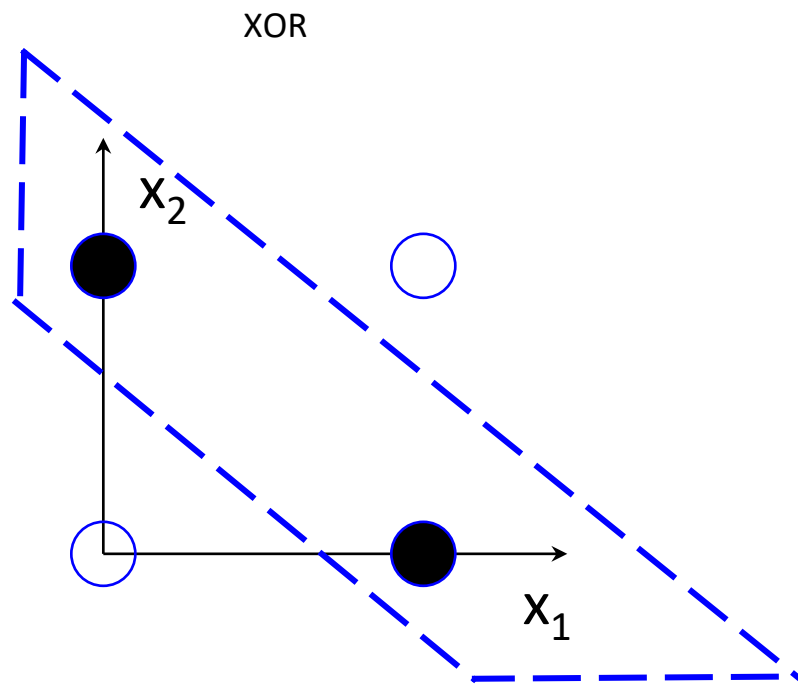
$$\begin{aligned} h_1([0,0]) &= -0.5 (-) \rightarrow 0 \\ h_1([0,1]) &= 1 - 0.5 = 0.5 (+) \rightarrow 1 \\ h_1([1,0]) &= 1 - 0.5 = 0.5 (+) \rightarrow 1 \\ h_1([1,1]) &= 2 - 0.5 = 1.5 (+) \rightarrow 1 \end{aligned}$$

$$\begin{aligned} h_2([0,0]) &= 1.5 (+) \rightarrow 1 \\ h_2([0,1]) &= -1 + 1.5 = 0.5 (+) \rightarrow 1 \\ h_2([1,0]) &= -1 + 1.5 = 0.5 (+) \rightarrow 1 \\ h_2([1,1]) &= -2 + 1.5 = -0.5 (-) \rightarrow 0 \end{aligned}$$

$$\begin{aligned} y([0,0]) &= (-) \rightarrow 0 \\ y([0,1]) &= (+) \rightarrow 1 \\ y([1,0]) &= (+) \rightarrow 1 \\ y([1,1]) &= (-) \rightarrow 0 \end{aligned}$$

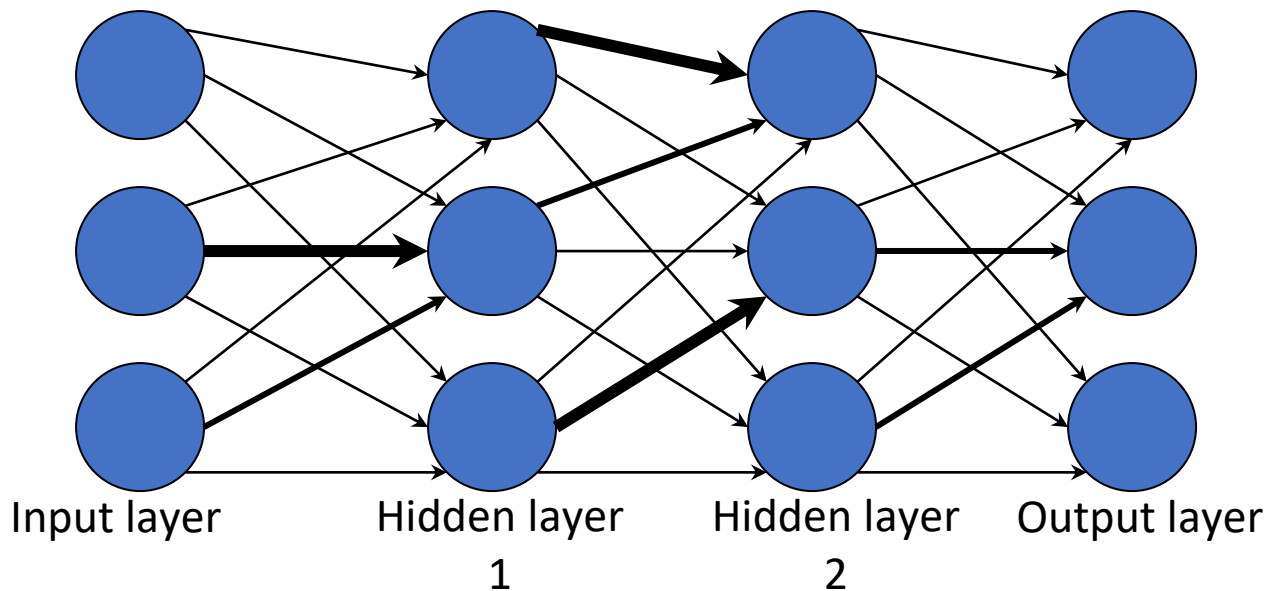


Conclusion: neurons can be combined into multiple layers to create complex shapes from linear boundaries



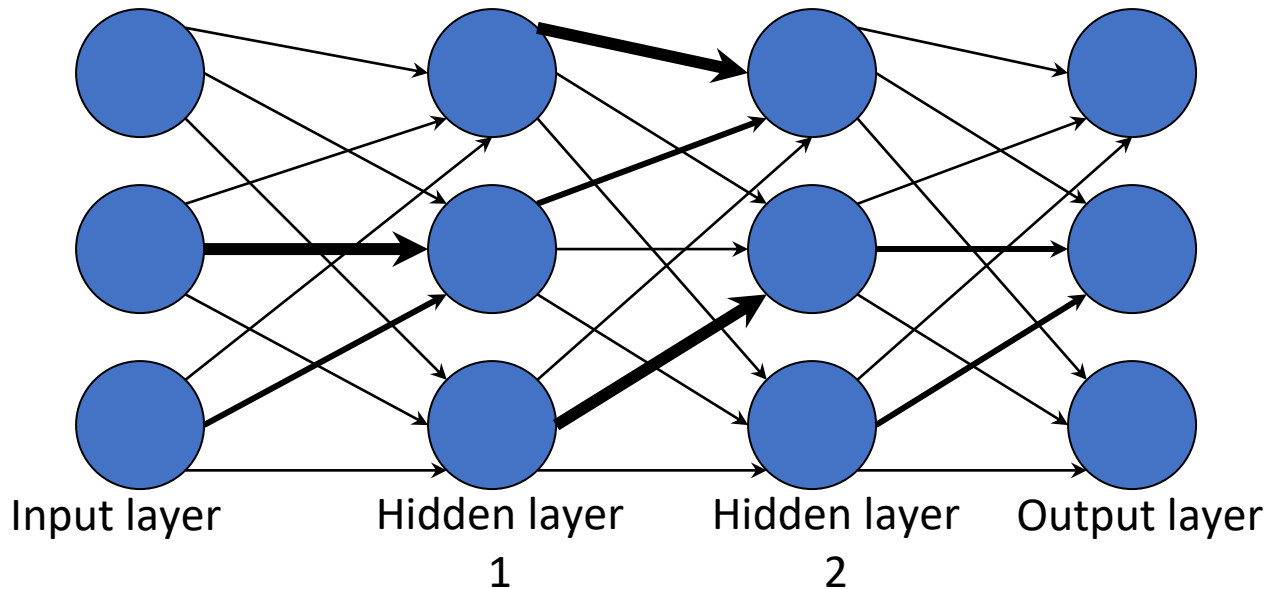
Multi-layer Perceptron (MLP)

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

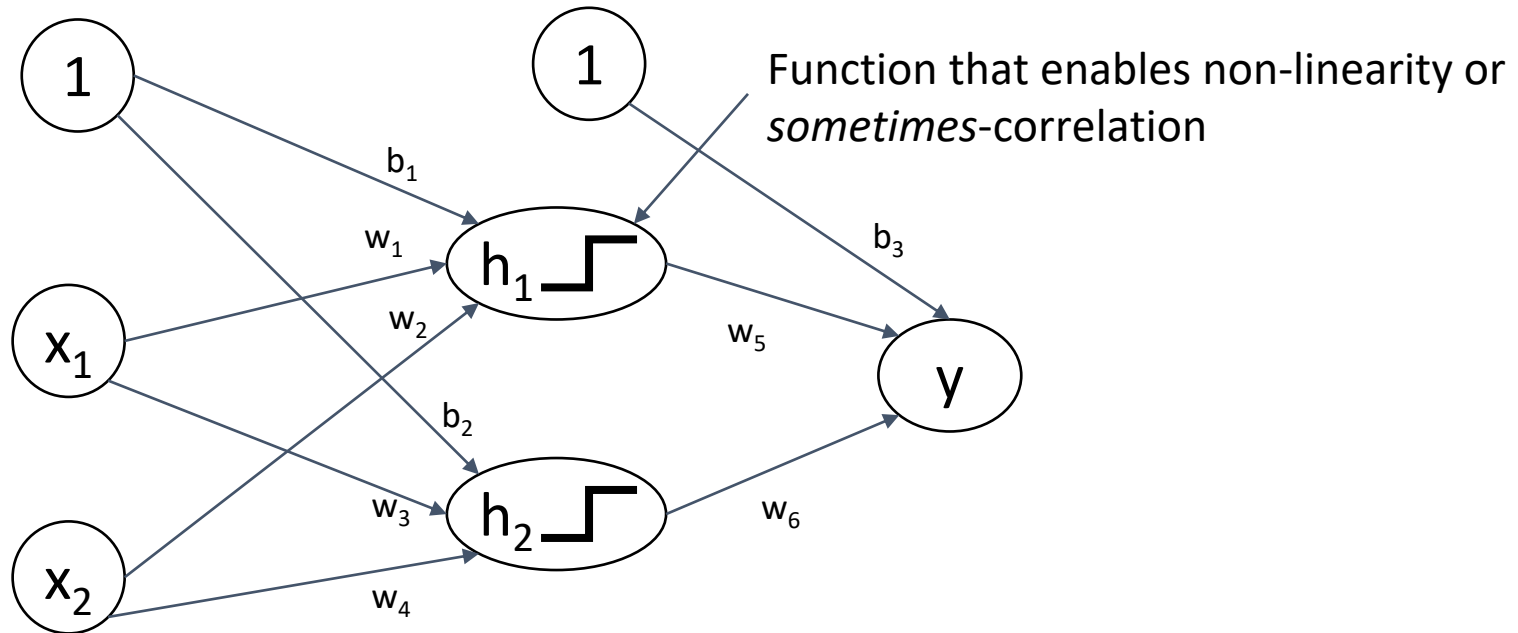


Multi-layer Perceptron: learning

Objective of learning - did not change:
determine the optimal values of weights to separate all
labeled instances by a hyperplane

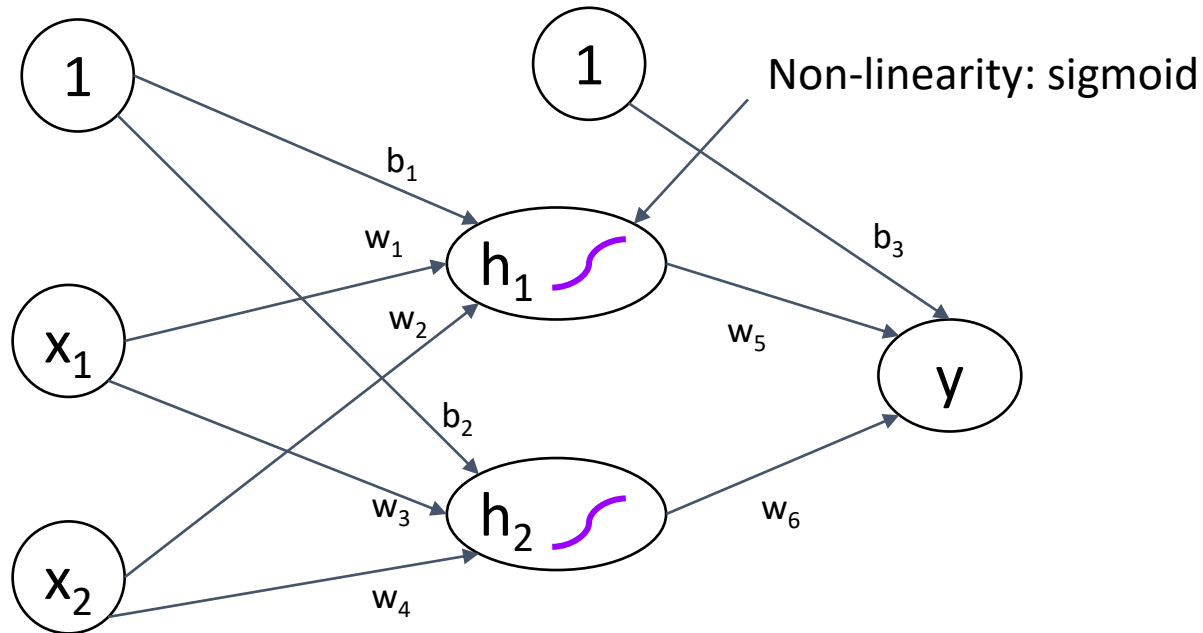


MLP: learning optimal weights



Because we need derivatives: instead of *sign* - use more complex nonlinear functions: **sigmoidal** functions

MLP: learning optimal weights



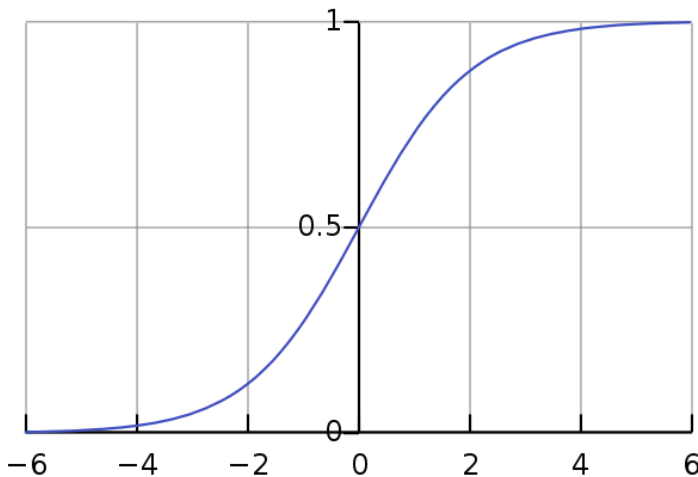
Because we need derivatives: instead of *sign* - use more complex nonlinear functions: **sigmoidal** functions

Non-linear activation functions

Logistic function (sigmoid)

$$g(h) = \frac{1}{1 + e^{-2\beta h}},$$

where β is a positive constant (we generally use $2\beta = 1$ obtaining a standard logistic function)



Sigmoid gives a value in range from 0 to 1.

Note: when $IN = 0$, $f = 0.5$

We consider all values >0 as positive predictions

Alternatively can use **tanh**:

$$\tanh x = \frac{\sinh x}{\cosh x} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = \frac{e^{2x} - 1}{e^{2x} + 1}$$

which has the same shape as sigmoid but in range -1 to 1.

More recently - **rectified linear units**

(ReLU): $f(x) = x^+ = \max(0, x)$

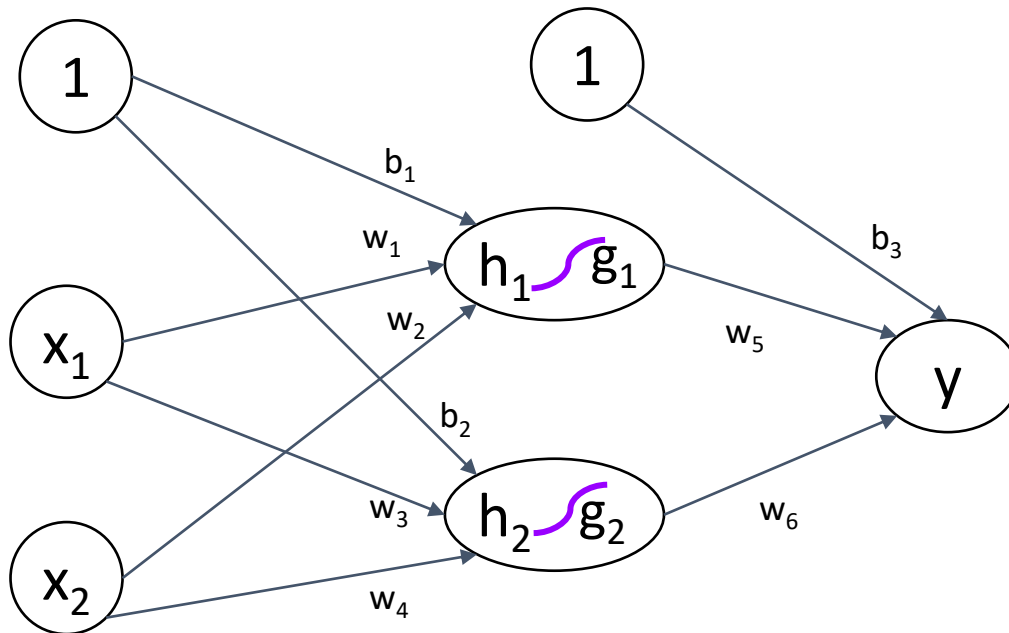
This function is 0 for negative argument values, and some units will yield activations 0, making networks sparse. Moreover, the gradient is particularly simple—either 0 or 1.

MLP learning algorithm

Training the MLP consists of **two parts**:

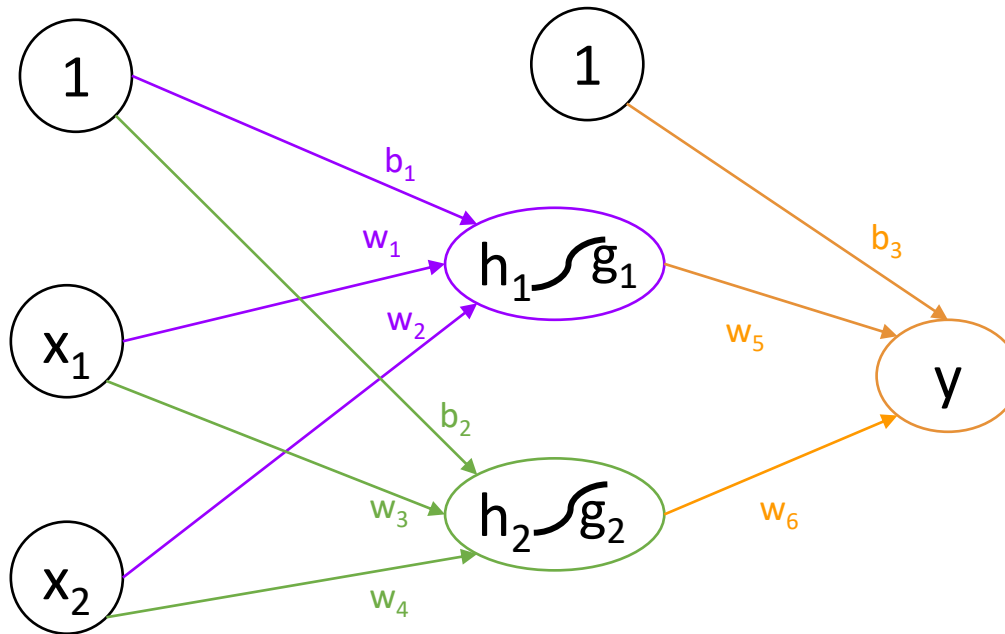
- Working out what the outputs are for the given inputs and the current weights – **Forward** phase
- Updating the weights according to the error, which is a function of the difference between the outputs and the targets – **Backward** phase

Forward: prediction



Forward phase:

1. input-to-hidden layer: summation

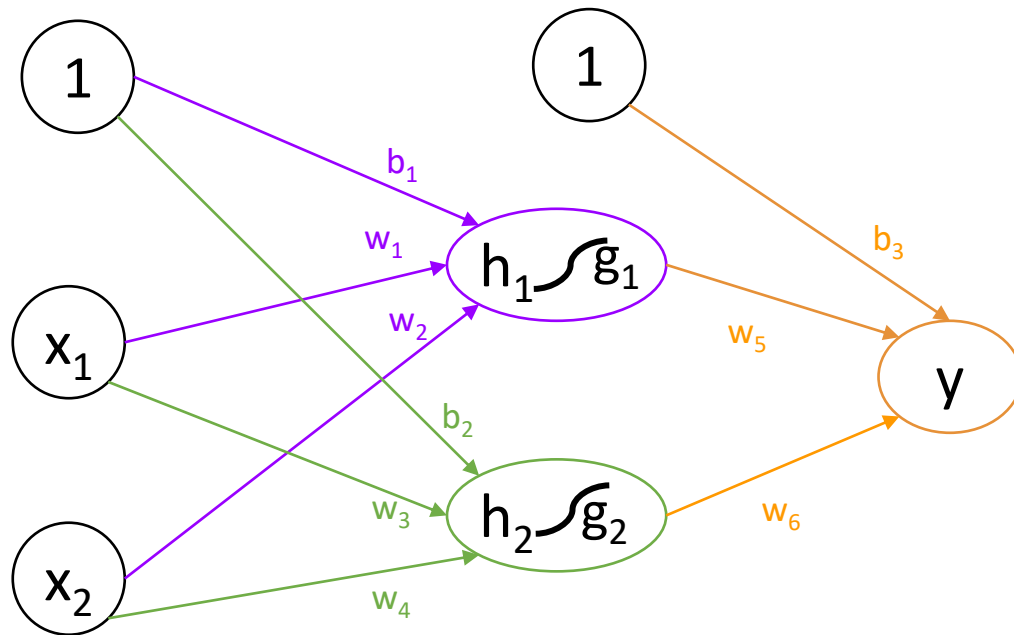


$$h_1 = w_1 * x_1 + w_2 * x_2 + b_1$$

$$h_2 = w_3 * x_1 + w_4 * x_2 + b_2$$

Forward phase:

2. input-to-hidden layer: activation



$$h_1 = w_1 * x_1 + w_2 * x_2 + b_1$$

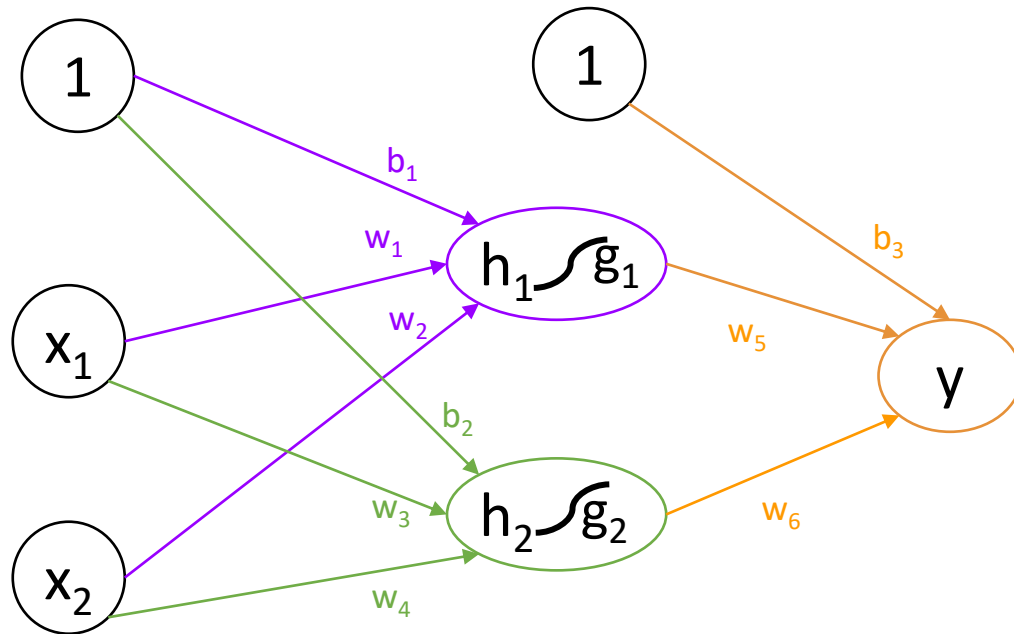
$$h_2 = w_3 * x_1 + w_4 * x_2 + b_2$$

$$g_1 = \sigma(h_1)$$

$$g_2 = \sigma(h_2)$$

Forward phase:

3. hidden-to-output layer: prediction



$$h_1 = w_1 * x_1 + w_2 * x_2 + b_1$$

$$h_2 = w_3 * x_1 + w_4 * x_2 + b_2$$

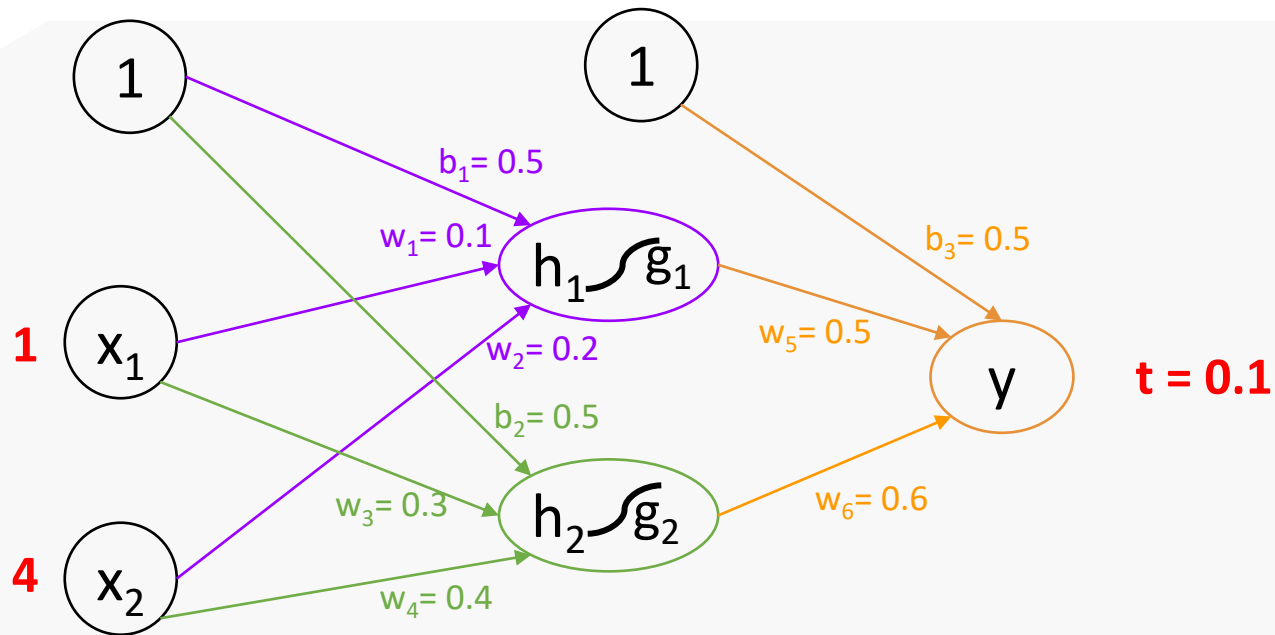
$$g_1 = \sigma(h_1)$$

$$g_2 = \sigma(h_2)$$

$$y = g_1 * w_5 + g_2 * w_6 + b_3$$

Step-by-step example

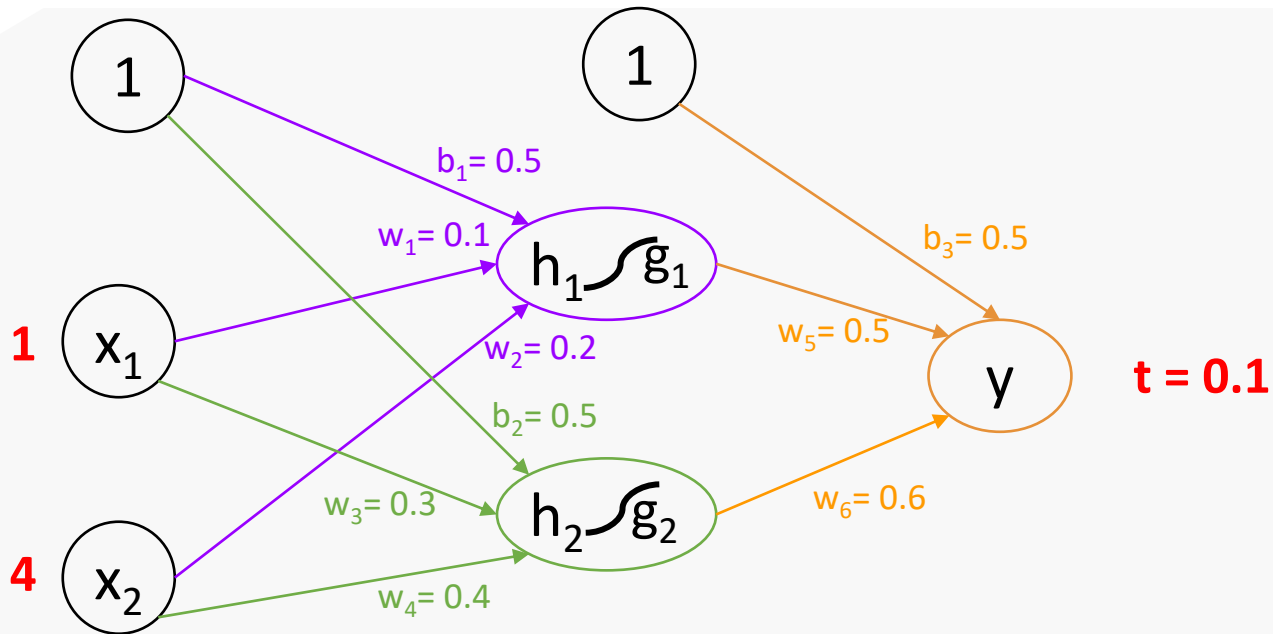
initialize weights at random



The input vector $\mathbf{x} = [1, 4]$, and the actual output $\mathbf{t} = 0.1$

Step-by-step example

1. input to hidden layer: summation

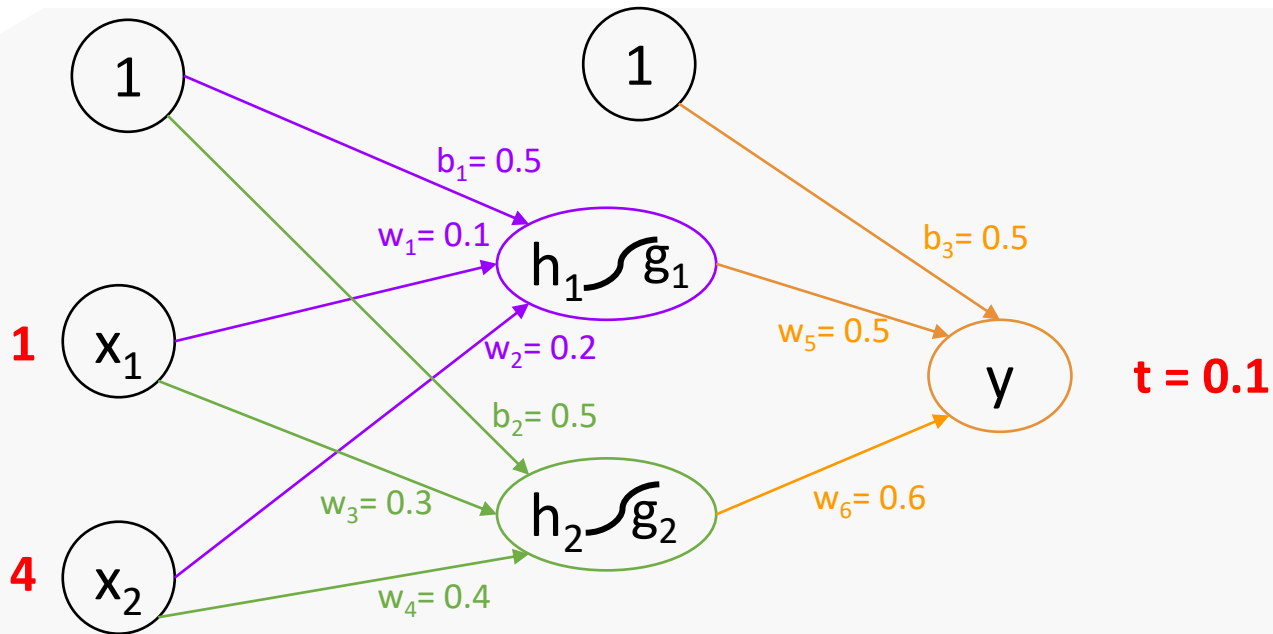


$$h_1 = w_1 * x_1 + w_2 * x_2 + b_1 = 0.5 + 0.1 * 1 + 0.2 * 4 = 1.4$$

$$h_2 = w_3 * x_1 + w_4 * x_2 + b_2 = 0.5 + 0.3 * 1 + 0.4 * 4 = 2.4$$

Step-by-step example

2. input to hidden layer: activation



$$h_1 = 1.4$$

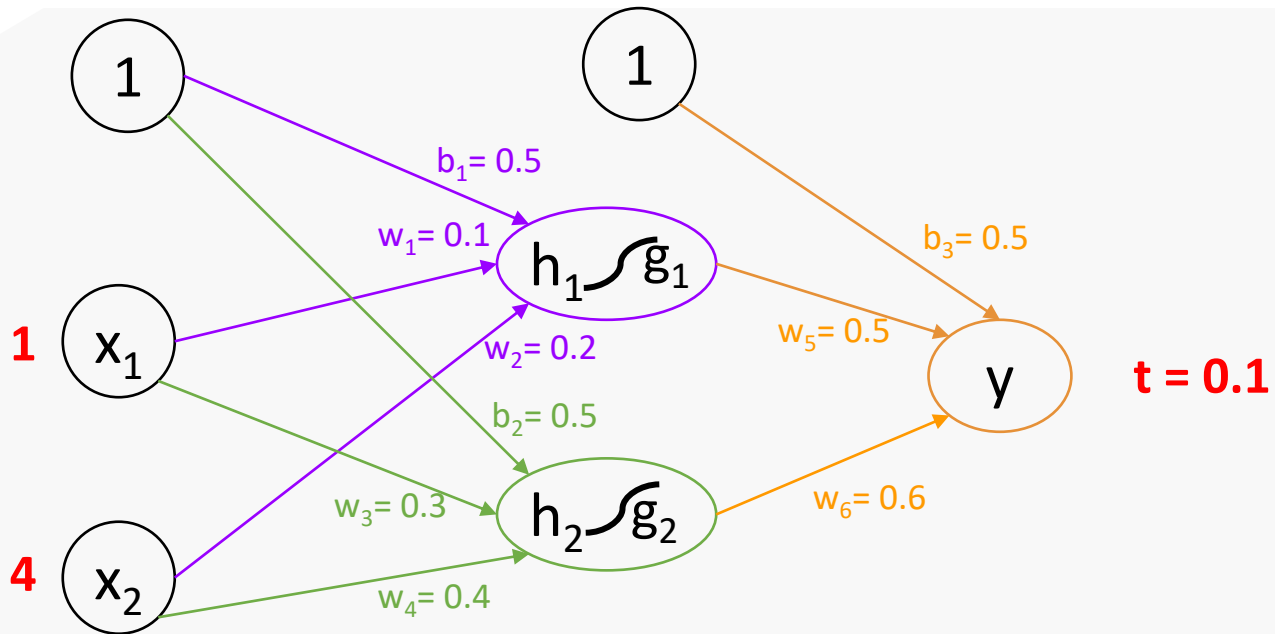
$$h_2 = 2.4$$

$$g_1 = \sigma(h_1) = 0.8021838885585817481543 \approx 0.80$$

$$g_2 = \sigma(h_2) = 0.9168273035060776293371 \approx 0.92$$

Step-by-step example

3. hidden-to-output layer: prediction



$$h_1 = 1.4$$

$$h_2 = 2.4$$

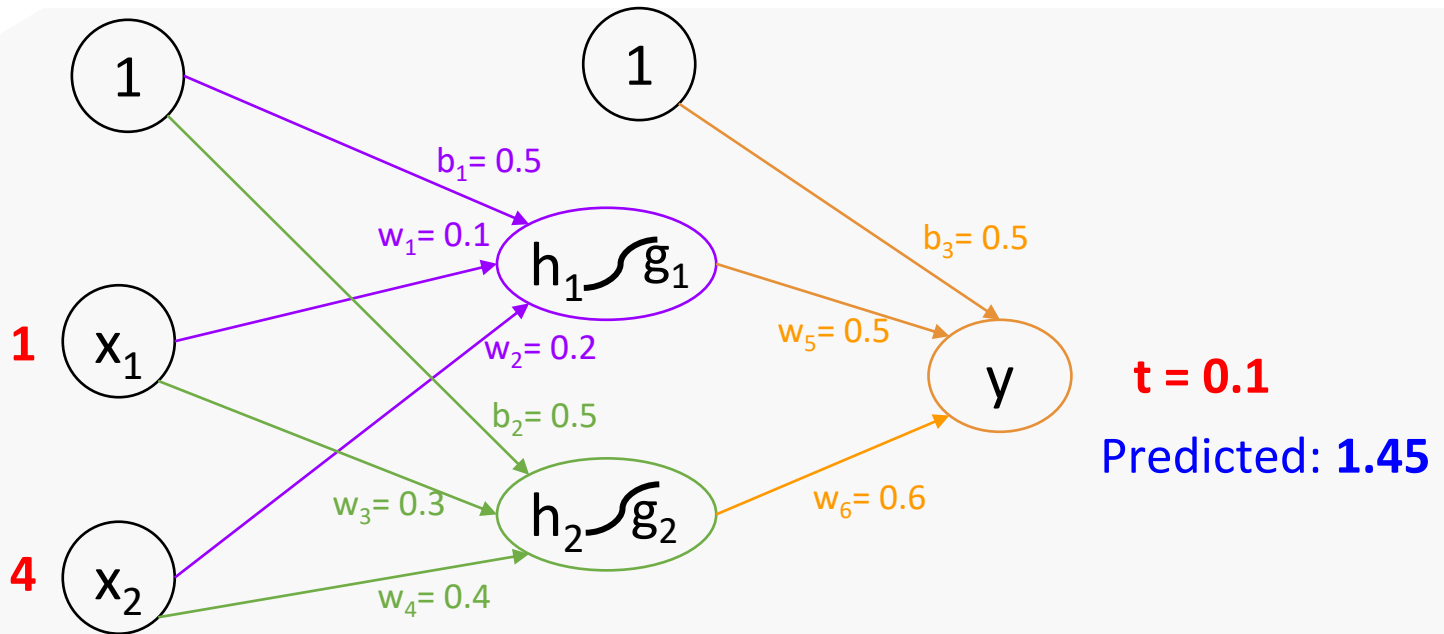
$$g_1 = 0.80$$

$$g_2 = 0.91$$

$$y = g_1 * w_5 + g_2 * w_6 + b_3 = 0.80 * 0.5 + 0.92 * 0.6 + 0.5 \approx 1.45$$

Step-by-step example

compute error



$$h_1 = 1.4$$

$$h_2 = 2.4$$

$$g_1 = 0.80$$

$$g_2 = 0.91$$

$$y = 1.45$$

$$E = \frac{1}{2} (1.45 - 0.1)^2 = 0.845$$

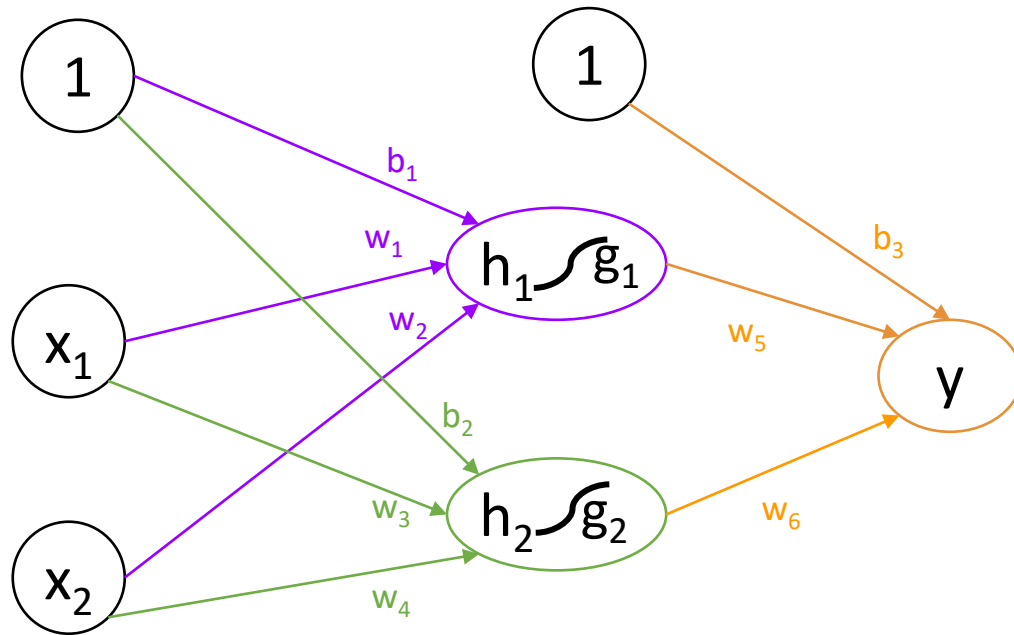
Error **directly** depends on the weights w_5 , w_6 , and b_3

$$E = \frac{1}{2} (0.80 * w_5 + 0.92w_6 + b_3 - 0.1)^2$$

We try to make it smaller by simultaneously adjusting w_5 , w_6 , and b_3

Backward phase:

4. output-to-hidden weight updates



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

$$y = g_1 * w_5 + g_2 * w_6 + b_3$$

To find how to update w_5 , w_6 , and b_3

Partial derivatives:

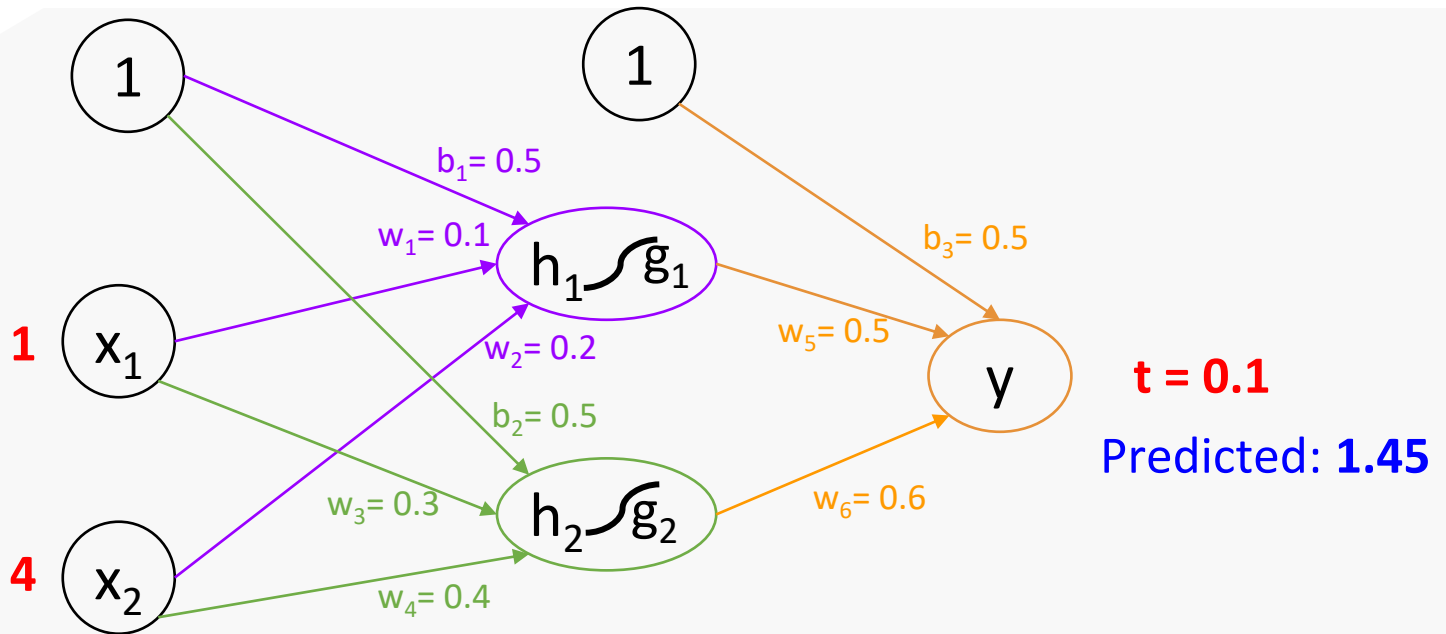
$$\frac{\partial E}{\partial w_5} = \frac{\partial E}{\partial y} * \frac{\partial y}{\partial w_5} = (y - t) * g_1$$

$$\frac{\partial E}{\partial w_6} = (y - t) * g_2$$

$$\frac{\partial E}{\partial b_3} = (y - t) * 1$$

Step-by-step example

4. output-to-hidden weight updates



$h_1 = 1.4$
 $h_2 = 2.4$
 $g_1 = 0.80$
 $g_2 = 0.91$
 $y = 1.45$

$$\frac{\partial E}{\partial w_5} = (y - t) * g_1 = (1.45 - 0.1) * 0.80 = 1.08$$

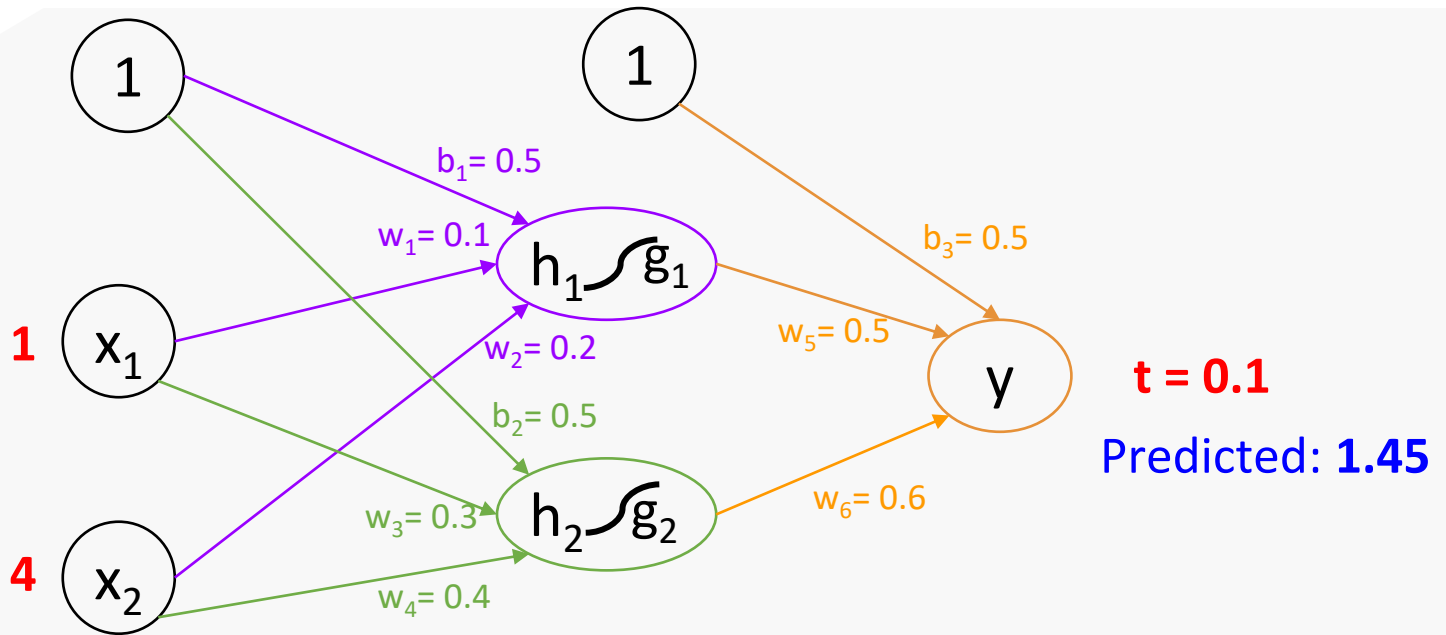
$$\frac{\partial E}{\partial w_6} = (y - t) * g_2 = (1.45 - 0.1) * 0.92 = 1.24$$

$$\frac{\partial E}{\partial b_3} = (y - t) * 1 = 1.45 - 0.1 = 1.35$$

This tells us how much to update w_5 , w_6 , and b_3

Step-by-step example

4. output-to-hidden weight updates



$$h_1 = 1.4$$

$$h_2 = 2.4$$

$$g_1 = 0.80$$

$$g_2 = 0.91$$

$$y = 1.45$$

$$\partial E / \partial w_5 = 1.08$$

$$\partial E / \partial w_6 = 1.24$$

$$\partial E / \partial b_3 = 1.35$$



Update weights $\eta=0.1$:

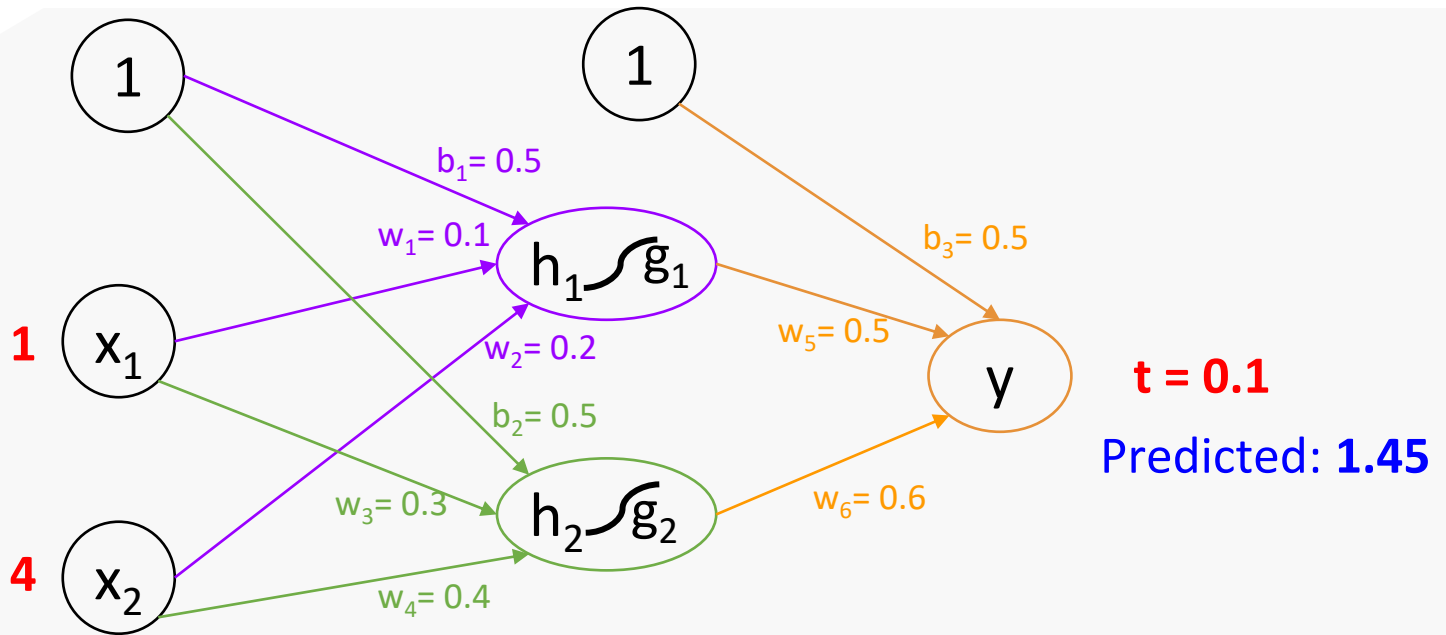
$$w_5 = 0.5 - 1.08 * 0.1 = 0.39$$

$$w_6 = 0.6 - 1.24 * 0.1 = 0.48$$

$$b_3 = 0.5 - 1.35 * 0.1 = 0.37$$

Step-by-step example

4. output-to-hidden weight updates



$$h_1 = 1.4$$

$$h_2 = 2.4$$

$$g_1 = 0.80$$

$$g_2 = 0.91$$

$$y = 1.45$$

$$\partial E / \partial w_5 = 1.08$$

$$\partial E / \partial w_6 = 1.24 \quad \Rightarrow$$

$$\partial E / \partial b_3 = 1.35$$

Update weights $\eta=0.1$:

$$w_5 = 0.5 - 1.08 * 0.1 = 0.39$$

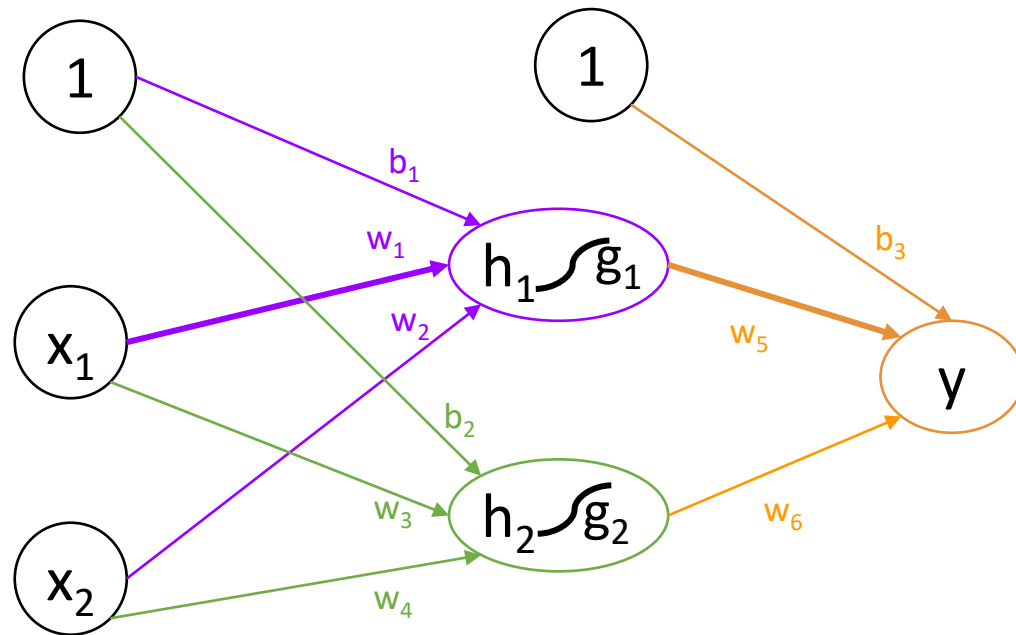
$$w_6 = 0.6 - 1.24 * 0.1 = 0.48$$

$$b_3 = 0.5 - 1.35 * 0.1 = 0.37$$

Note that this step is exactly the same as in a single-layer perceptron!

Backward phase:

5. hidden-to-output weight updates



Error function E **indirectly** depends on $w_1, w_2, w_3, w_4, b_1, b_2$

To find the contribution of each variable: partial derivatives

For example:

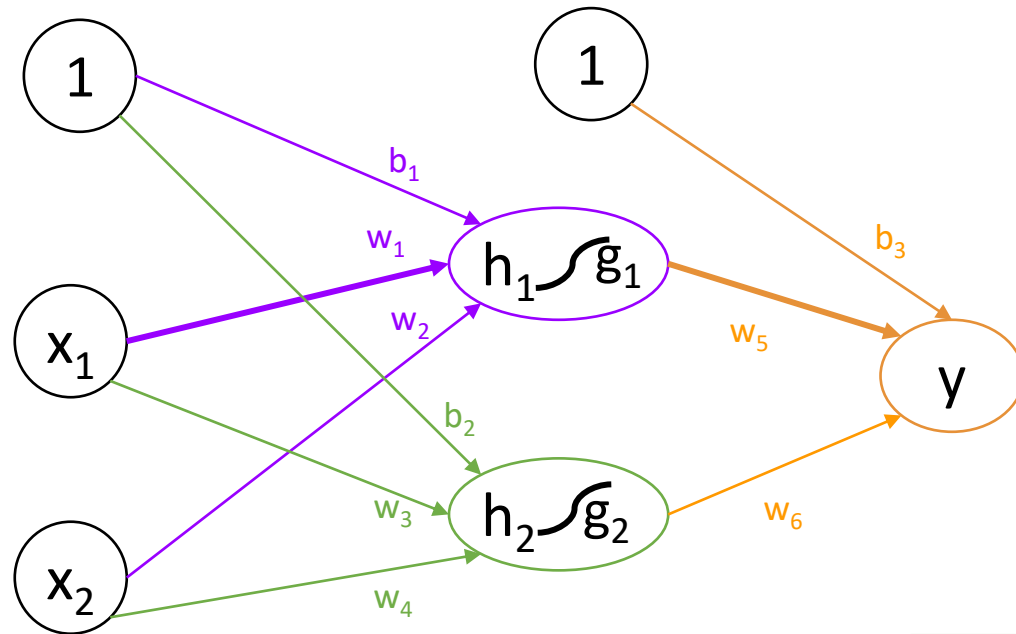
$$\partial E / \partial w_1 = \partial E / \partial y * \partial y / \partial g_1 * \partial g_1 / \partial w_1$$

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

Chain rule!

Backward phase:

5. hidden-to-output weight updates



Computing delta for w_1

$$\partial E / \partial w_1 = \partial E / \partial y * \partial y / \partial g_1 * \partial g_1 / \partial w_1$$

$$E(y) = \frac{1}{2}(y - t)^2 \quad \rightarrow$$

$$y(g_1) = g_1 * w_5 + g_2 * w_6 + b_3 \rightarrow$$

$$g_1(w_1) = \sigma(h_1) = \sigma(w_1 * x_1 + w_2 * x_2 + b_1) \rightarrow$$

$$\partial E / \partial y = y - t$$

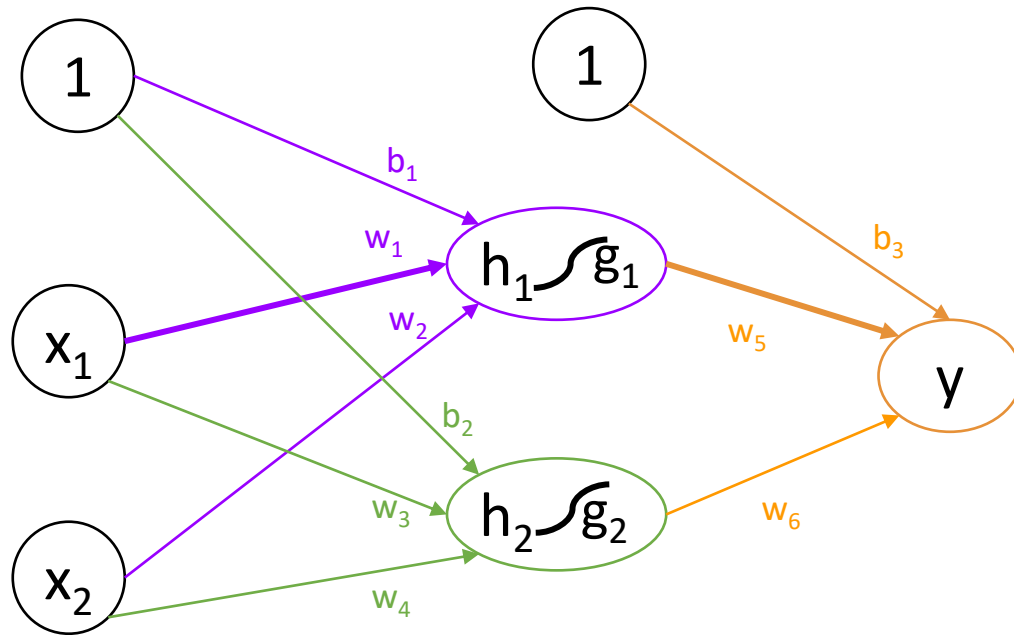
$$\partial y / \partial g_1 = w_5$$

$$\partial g_1 / \partial w_1 = g_1 * (1 - g_1) * x_1$$

$\sigma'(x) = \sigma(x)(1 - \sigma(x))$
sigmoid derivative

Backward phase:

5. hidden-to-output weight updates



Computing delta for w_1

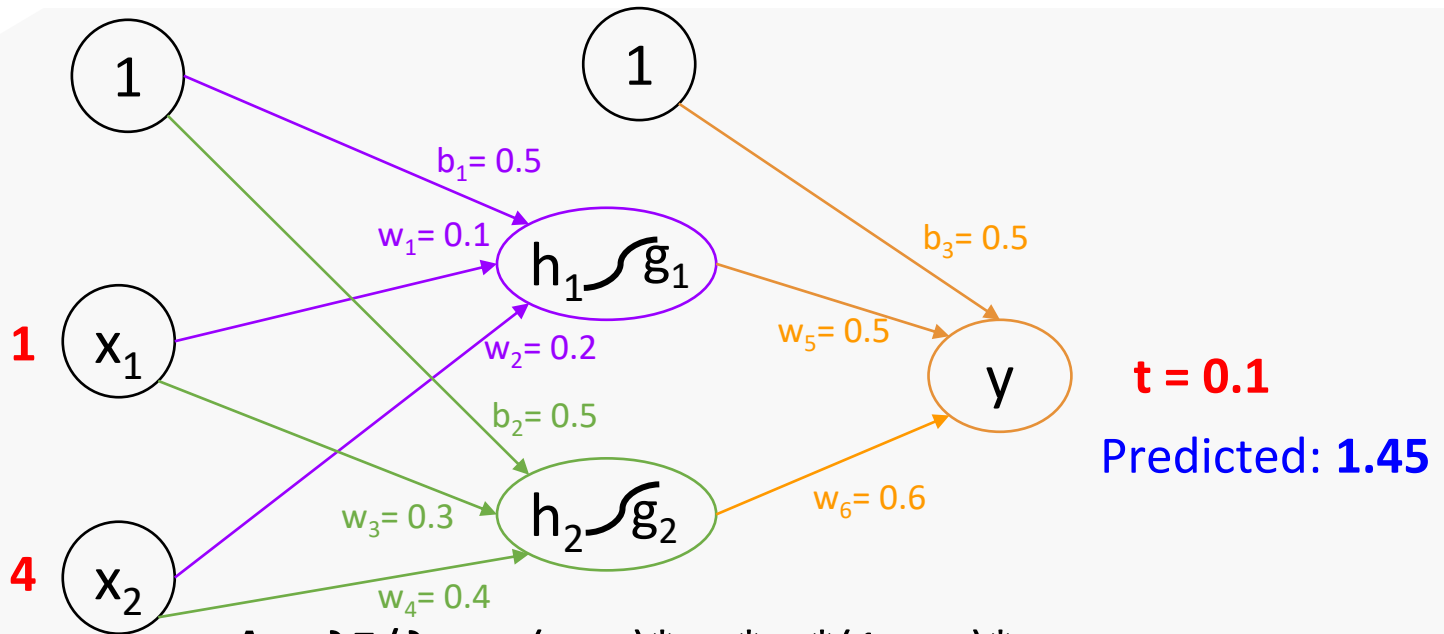
$$\partial E / \partial w_1 = \partial E / \partial y * \partial y / \partial g_1 * \partial g_1 / \partial w_1$$

$$\Delta = \partial E / \partial w_1 = (y - t) * w_5 * g_1 * (1 - g_1) * x_1$$

$$w_1 = w_1 - \eta \Delta$$

Step-by-step example

5. hidden-to-output weight update for w_1



$$h_1 = 1.4$$

$$h_2 = 2.4$$

$$g_1 = 0.80$$

$$g_2 = 0.91$$

$$y = 1.45$$

$$\Delta = \partial E / \partial w_1 = (y - t) * w_5 * g_1 * (1 - g_1) * x_1$$

$$w_1 = w_1 - \eta \Delta$$

$$\Delta = (1.45 - 0.1) * 0.5 * 0.80 * 0.20 * 1 = 0.108$$

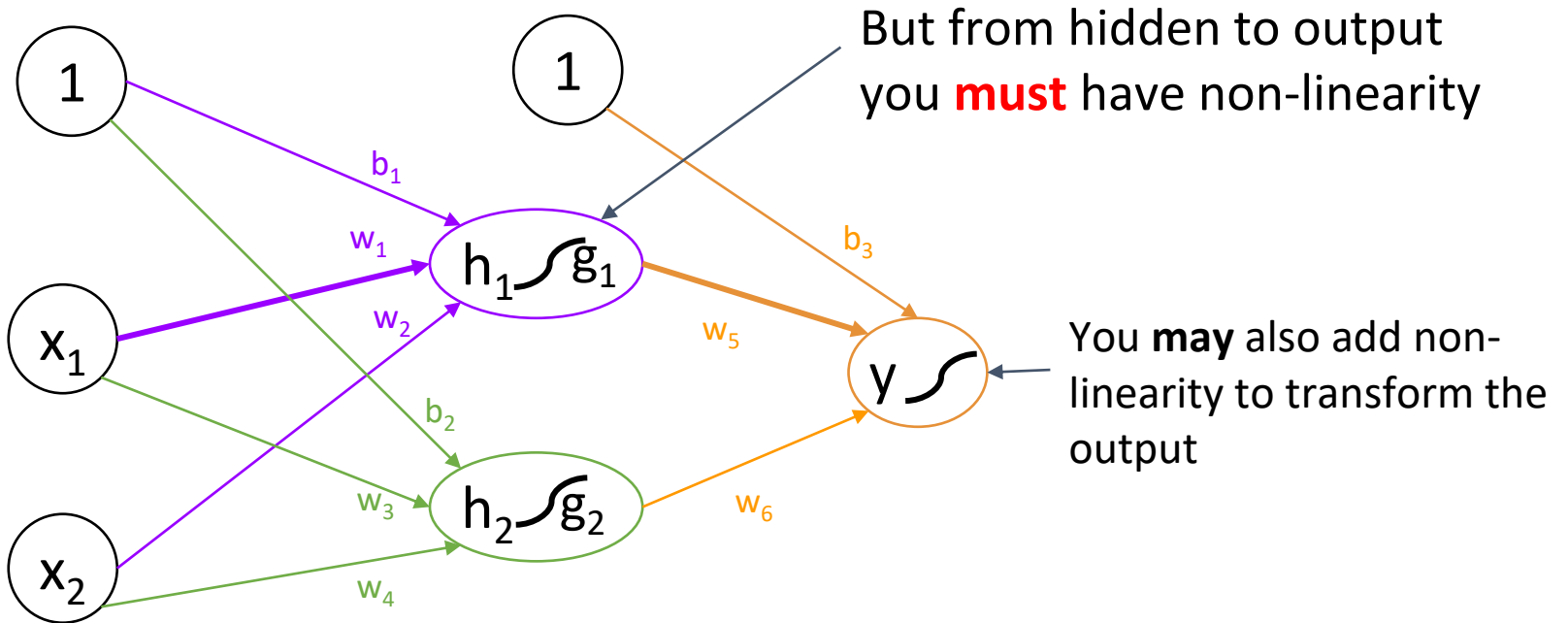
Update w_1 using $\eta = 0.1$:

$$w_1 = 0.1 - 0.1 * 0.108 = 0.0892$$

Quiz 10. Backpropagation

- Use the same network as in the step-by-step example and the same initial parameters to compute new value for w_2 during one backpropagation step.
- Demonstrate your understanding by providing as many details as possible.

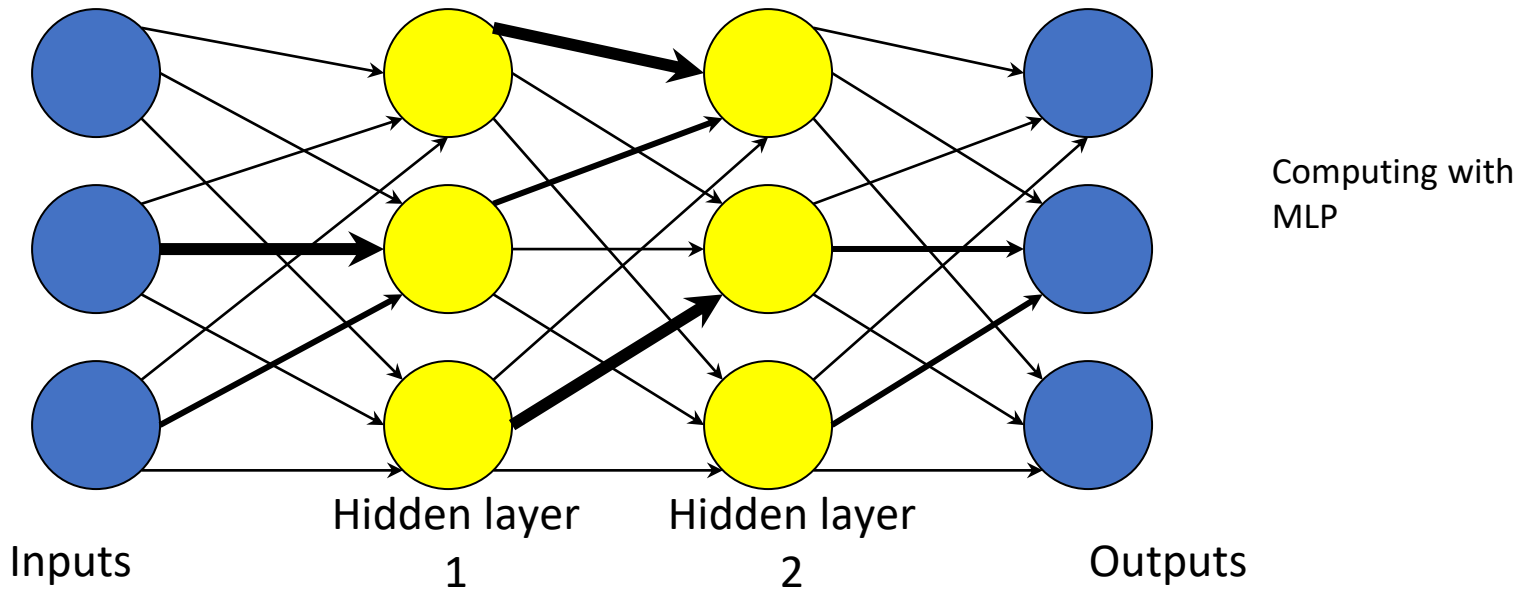
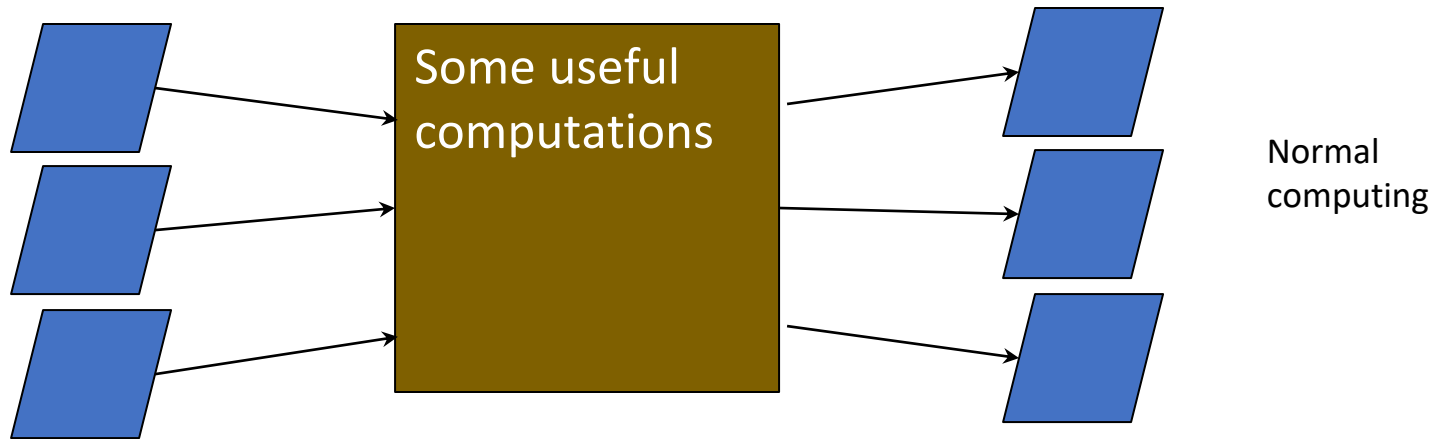
Role of nonlinearity



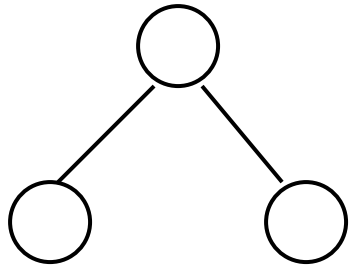
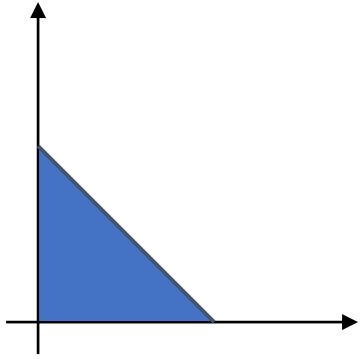
- Somewhere inside the hidden layer we must have a mechanism which will ignore some correlations
- Otherwise the network will serve as a basic linear separator and be no better than a single-layer perceptron

Experiment with *multi-layer-perceptron*
[here](#)

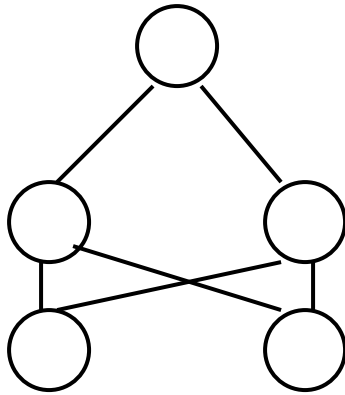
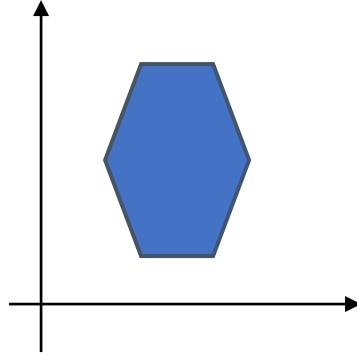
Multi-layer perceptron: vanilla (basic) neural networks



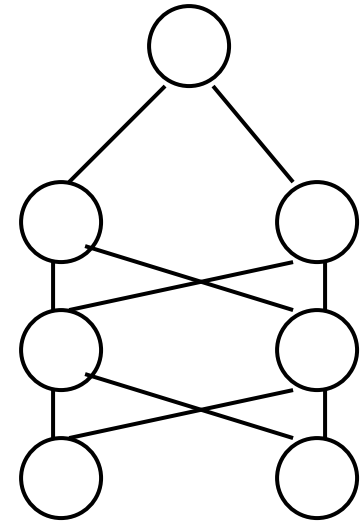
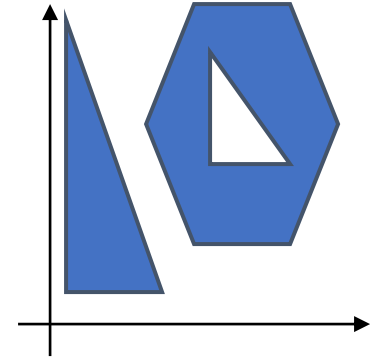
What do we gain from the extra layers



1st layer draws linear boundaries



2nd layer combines the boundaries



3rd layer can generate arbitrarily complex boundaries

Very powerful model

- With sigmoidal activation function 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 we might need arbitrarily large number of neurons - 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

Universal Approximation Theorem

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

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

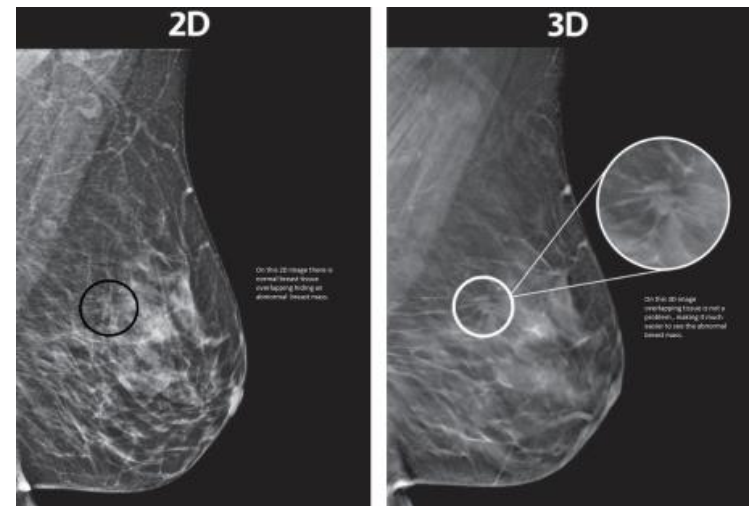
where $H(x_1, \dots, x_m) = \sum_{i=1}^k a_i f(\sum_{j=1}^m w_{ij} x_j + b_i)$

Applications of ANNs

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

Example: breast cancer diagnosis

- Dataset:
[https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+\(Diagnostic\)](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 and see how MLP learns to diagnose breast cancer

Make computers as capable as humans?

Brain is a 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 working together

Artificial neural network is an abstract idea – media-independent

- To simulate the brain we could theoretically construct thousands of circuits working in parallel
- We can simulate them using a program that is executed on a conventional serial processor
- The solutions are *theoretically* equivalent
- We can simulate the neural behavior by a virtual machine which is functionally identical to a real machine that currently is prohibitively complex and expensive to build

Example:

Multi-class classification

- In *multiclass classification*, the output label can be one of the C classes: $y = \{1, \dots, C\}$
- To do this, we can have C output neurons in the output layer: one output neuron for each class
- The model then returns not a label but a score (0...1) that can be interpreted as the probability that the instance belongs to each of C classes
- Then we choose the class with the highest probability as the solution to the classification problem

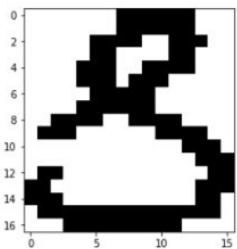
We will try a multi-class classification with *keras* library on **MNIST** dataset

2	2	0	1	7	6	9	0
4	2	6	2	0	3	1	0
3	5	2	8	5	7	0	6
2	0	4	4	0	1	7	6
0	8	4	9	9	7	5	5
1	7	0	2	6	0	7	2
8	6	2	1	5	3	5	2
1	1	5	4	6	1	8	0

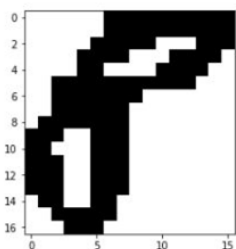
MNIST Dataset: collection of handwritten digits from 0 to 9

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

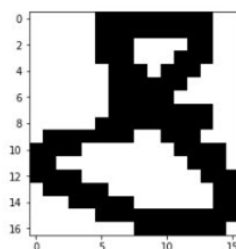
The goal is to predict the class - the actual digit meant by the writer



```
.....*
y      8
y_pred 3
Name: 1519, dtype: int64
```



```
.....*
y      8
y_pred 9
Name: 1153, dtype: int64
```



```
.....*
y      8
y_pred 3
Name: 568, dtype: int64
```

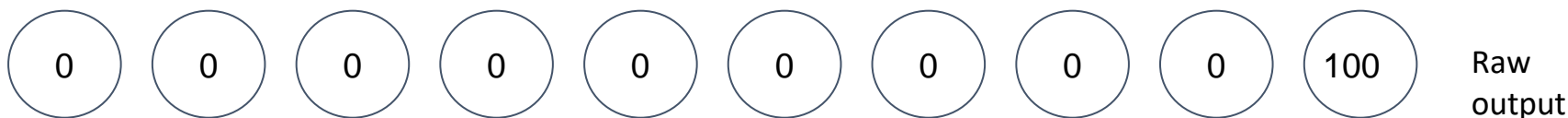
Each black-and-white image is represented by a 28x28 matrix of pixel values

We flatten it into a feature vector of size $28 \times 28 = 784$

We need a new activation function for multi-class classification

- We could train the network with a sigmoid activation function applied to the output layer and simply declare that the highest output probability is the most likely
- However, there is a problem with this approach: sigmoid does not reflect the idea that "The more likely is one label, the less likely is any of the other labels"

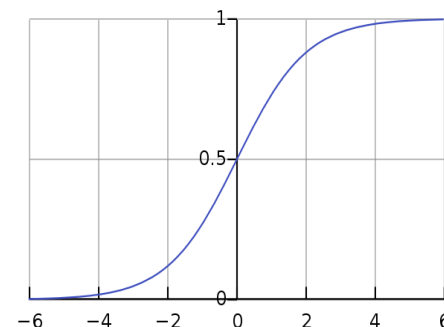
Why not *sigmoid*: 1/3



Consider the raw prediction obtained for digit 9

This is a “perfect” classification

If we apply sigmoid we get:



Suddenly we are **not 100% sure anymore**

Why not *sigmoid*: 2/3

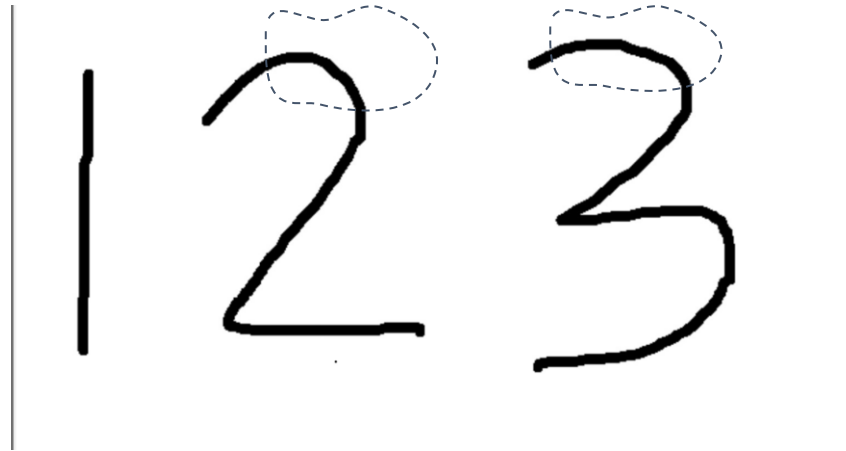


Sigmoid function would lead to a situation when perfectly classified instance will create a **large Mean Square Error**:



- The weights will be updated - even though the prediction was perfect!
- This happens because sigmoid does not take into account relationship between possible outputs
- For sigmoid to reach 0 error, it doesn't just have to predict the highest positive number for the true output, but it has to predict 0 everywhere else (which is very unlikely)

Why not *sigmoid*: 3/3

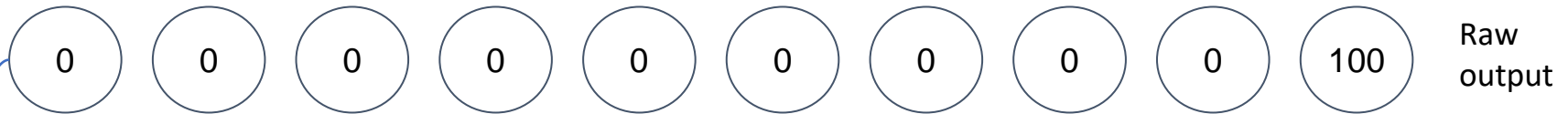


- So for the previous example, the sigmoid would update weights until all the errors are 0 except for 9
- To do this it will penalize all the pixels combinations that simultaneously occur in several numbers
- For example, if it detects the curved region for 2, and only 2, then when it sees 3, it will reject it based on this curve

We want a smooth probability distribution for a given image: each image is classified as all classes with different probabilities

All these probabilities must sum up to 1.0

Softmax activation function



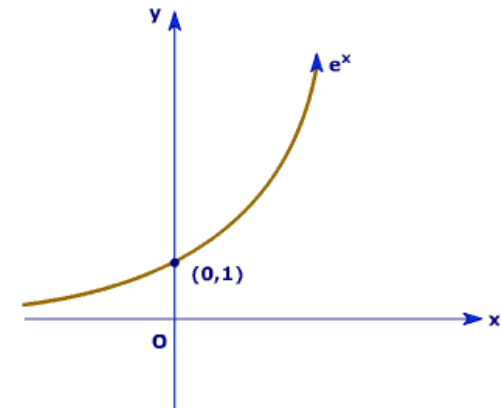
The *softmax* is computed **on the whole output layer**:

1. Raise each value exponentially: each x is transformed into e^x

All the zeros turned into ones, and 100 turned into $e^{100} \approx 2.668 * 10^{43}$

2. Sum up the values for all the nodes in the layer, and divide each value in the layer by that sum

This effectively will make every number 0 except for the value for label 9



This turns every prediction into a positive number: negative numbers turn into very small positive numbers and big positive numbers turn into very big numbers.

Advantages of *softmax* for multi-class classification

- *Softmax* takes into account all the classes at the same time: the higher the network predicts one value, the lower it predicts all the others
- It also increases the differences (sharpness of attenuation): it encourages the network to predict one output with very high probability

If you want to adjust how aggressively it does this - use numbers slightly higher or lower than "e"
- *Softmax* probabilities always sum to 1.0: We can interpret any individual prediction as a global probability that the prediction has a particular class label

Experiment with *keras* library and handwritten digit recognition in *handwriting_classification*

https://github.com/mgbarsky/labs_ml_img_classification/blob/main/handwriting_classification.ipynb