Natural Language Understanding

Lecture 2: Revision of neural networks and backpropagation

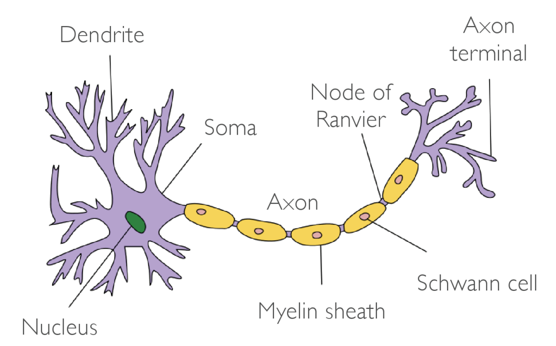
Adam Lopez

Credits: Mirella Lapata and Frank Keller

19 January 2018

School of Informatics
University of Edinburgh
alopez@inf.ed.ac.uk

Biological neural networks



- Neuron receives inputs and combines these in the cell body.
- If the input reaches a threshold, then the neuron may fire (produce an output).
- Some inputs are excitatory, while others are inhibitory.

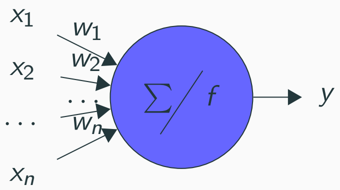
The relationship of artifical neural networks to the brain

The relationship of artifical neural networks to the brain

While the brain metaphor is sexy and intriguing, it is also distracting and cumbersome to manipulate mathematically. (Goldberg 2015)

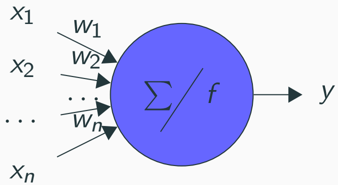
The perceptron: an artificial neuron

Developed by Frank Rosenblatt in 1957.



The perceptron: an artificial neuron

Developed by Frank Rosenblatt in 1957.

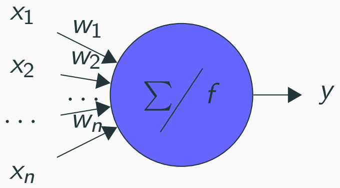


Input function:

$u(\mathbf{x}) = \sum_{i=1}^{n} w_i x_i$

The perceptron: an artificial neuron

Developed by Frank Rosenblatt in 1957.



Input function:

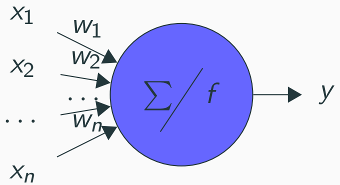
$$u(\mathbf{x}) = \sum_{i=1}^{n} w_i x_i$$

Activation function: threshold

$$y = f(u(\mathbf{x})) = \begin{cases} 1, & \text{if } u(\mathbf{x}) > \theta \\ 0, & \text{otherwise} \end{cases}$$

The perceptron: an artificial neuron

Developed by Frank Rosenblatt in 1957.



Activation function: threshold

Input function:

$$u(\mathbf{x}) = \sum_{i=1}^{n} w_i x_i$$

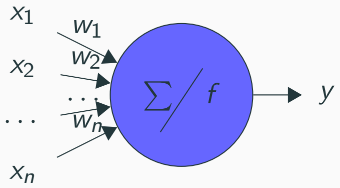
$$y = f(u(\mathbf{x})) = \begin{cases} 1, & \text{if } u(\mathbf{x}) > \theta \\ 0, & \text{otherwise} \end{cases}$$

Activation state:

0 or 1 (-1 or 1)

The perceptron: an artificial neuron

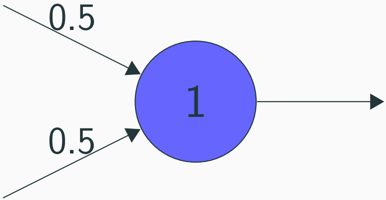
Developed by Frank Rosenblatt in 1957.



- Inputs are in the range [0, 1], where 0 is "off" and 1 is "on".
- Weights can be any real number (positive or negative).

Perceptrons can represent logic functions

Perceptron for AND

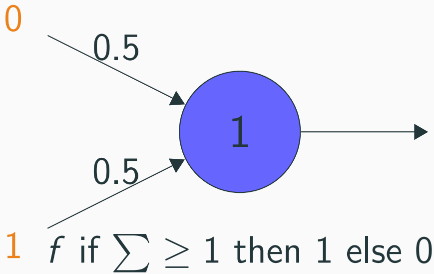


f if $\sum \geq 1$ then 1 else 0

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

Perceptrons can represent logic functions

Perceptron for AND

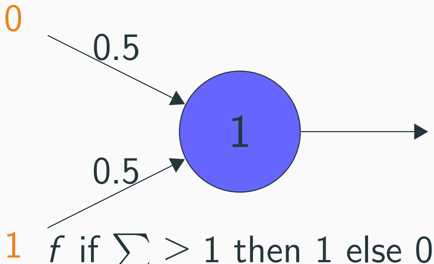


f if $\sum \geq 1$ then 1 else 0

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

Perceptrons can represent logic functions

Perceptron for AND



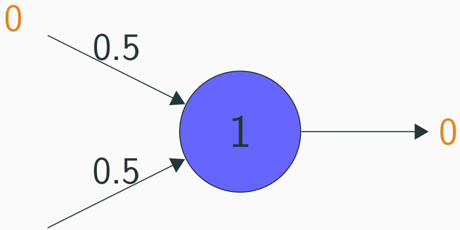
f if $\sum \geq 1$ then 1 else 0

$0 \cdot 0.5 + 1 \cdot 0.5 = 0.5 < 1$

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

Perceptrons can represent logic functions

Perceptron for AND



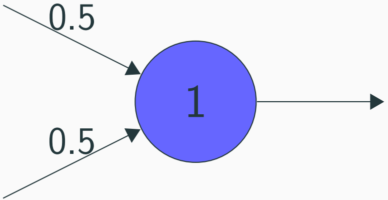
f if $\sum \geq 1$ then 1 else 0

$0 \cdot 0.5 + 1 \cdot 0.5 = 0.5 < 1$

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

Perceptrons can represent logic functions

Perceptron for AND

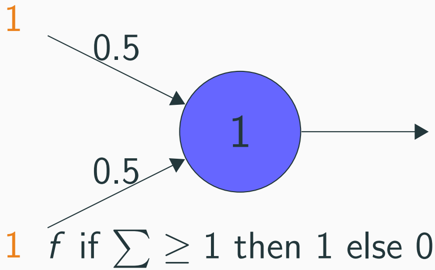


f if $\sum \geq 1$ then 1 else 0

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

Perceptrons can represent logic functions

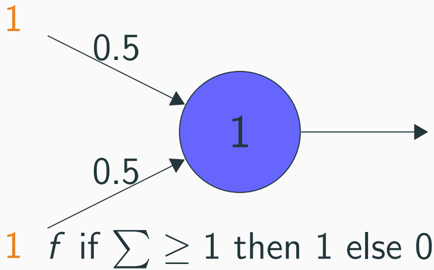
Perceptron for AND



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

Perceptrons can represent logic functions

Perceptron for AND

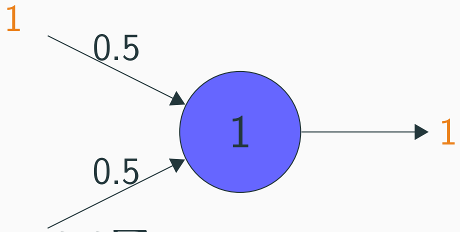


f if $\sum \geq 1$ then 1 else 0

$1 \cdot 0.5 + 1 \cdot 0.5 = 1 = 1$

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

Perceptron for AND



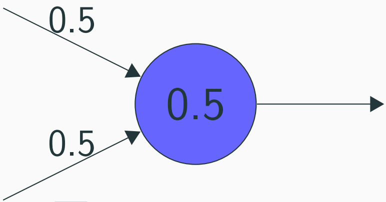
f if $\sum \geq 1$ then 1 else 0

$$1 \cdot 0.5 + 1 \cdot 0.5 = 1 = 1$$

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

Perceptrons can represent logic functions

Perceptron for OR

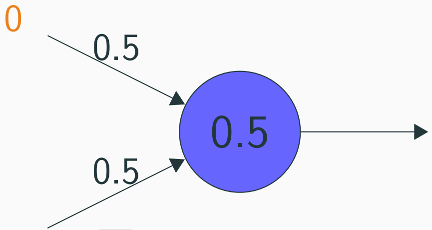


f if $\sum \geq 0.5$ then 1 else 0

x_1	x_2	x_1 OR x_2
0	0	0
0	1	1
1	0	1
1	1	1

Perceptrons can represent logic functions

Perceptron for OR

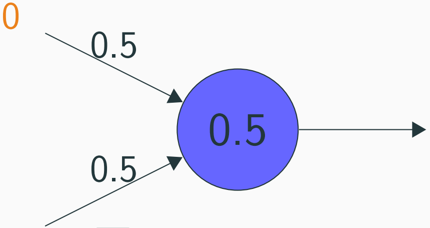


f if $\sum \geq 0.5$ then 1 else 0

x_1	x_2	x_1 OR x_2
0	0	0
0	1	1
1	0	1
1	1	1

Perceptrons can represent logic functions

Perceptron for OR



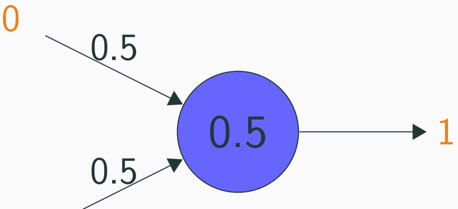
1 f if $\sum \geq 0.5$ then 1 else 0

$$0 \cdot 0.5 + 1 \cdot 0.5 = 0.5 = 0.5$$

x_1	x_2	x_1 OR x_2
0	0	0
0	1	1
1	0	1
1	1	1

Perceptrons can represent logic functions

Perceptron for OR



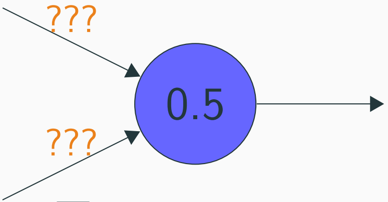
1 f if $\sum \geq 0.5$ then 1 else 0

$$0 \cdot 0.5 + 1 \cdot 0.5 = 0.5 = 0.5$$

x_1	x_2	x_1 OR x_2
0	0	0
0	1	1
1	0	1
1	1	1

How would you represent NOT(OR)?

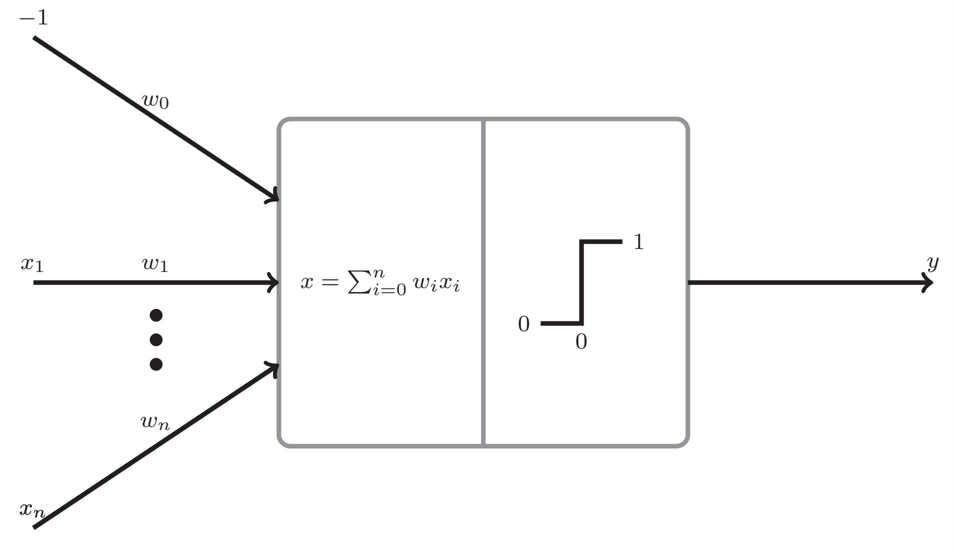
Perceptron for NOT(OR)



f if $\sum \geq$??? then 1 else 0

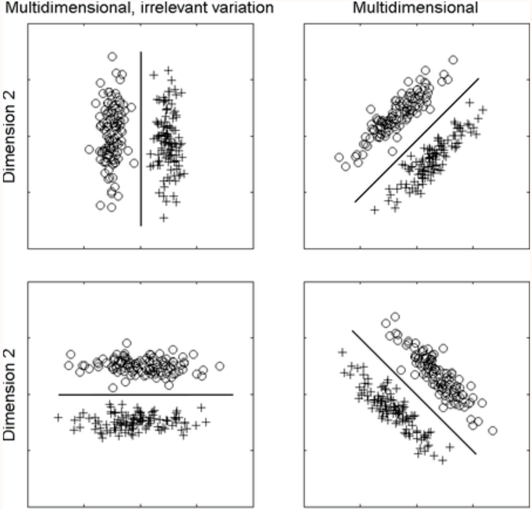
x_1	x_2	x_1 OR x_2
0	0	1
0	1	0
1	0	0
1	1	0

Perceptrons are linear classifiers



Perceptrons are linear classifiers

Perceptrons are linear classifiers, i.e., they can only separate points with a hyperplane (a straight line).



Perceptron can learn logic functions from examples

Give some examples to the Perceptron:

N	**input** x	**target** t
1	(0,1,0,0)	1
2	(1,0,0,0)	0
3	(0,1,1,1)	0
4	(1,0,1,0)	0
5	(1,1,1,1)	1
6	(0,1,0,0)	1
...	...	...

- Input: a vector of 1's and 0's—-a feature vector.
- Output: a 1 or 0, given as the target.

Perceptron can learn logic functions from examples

Give some examples to the Perceptron:

N	**input** x	**target** t	**output** o
1	(0,1,0,0)	1	0
2	(1,0,0,0)	0	0
3	(0,1,1,1)	0	1
4	(1,0,1,0)	0	1
5	(1,1,1,1)	1	0
6	(0,1,0,0)	1	1
...	...	...	...

- Input: a vector of 1's and 0's—-a feature vector.
- Output: a 1 or 0, given as the target.

Perceptron can learn logic functions from examples

Give some examples to the Perceptron:

N	**input** x	**target** t	**output** o
1	(0,1,0,0)	1	0
2	(1,0,0,0)	0	0
3	(0,1,1,1)	0	1
4	(1,0,1,0)	0	1
5	(1,1,1,1)	1	0
6	(0,1,0,0)	1	1
...	...	...	

- Input: a vector of 1's and 0's—-a feature vector.
- Output: a 1 or 0, given as the target.
- How do we efficiently find the weights and threshold?

Learning

Q_1: Choosing weights and threshold θ for the perceptron is not easy! What's an effective to learn the weights and threshold from examples?

A_1: We use a learning algorithm that adjusts the weights and threshold based on examples.

`http://www.youtube.com/watch?v=vGwemZhPlsA&feature=youtu.be`

Simplify by converting θ into a weight

$$\sum_{i=1}^{n} w_i x_i > \theta$$

Simplify by converting θ into a weight

$$\sum_{i=1}^{n} w_i x_i > \theta$$

$$\sum_{i=1}^{n} w_i x_i - \theta > 0$$

Simplify by converting θ into a weight

$$\sum_{i=1}^{n} w_i x_i > \theta$$

$$\sum_{i=1}^{n} w_i x_i - \theta > 0$$

$$w_1 x_1 + w_2 x_2 + \ldots w_n x_n - \theta > 0$$

Simplify by converting θ into a weight

$$\sum_{i=1}^{n} w_i x_i > \theta$$

$$\sum_{i=1}^{n} w_i x_i - \theta > 0$$

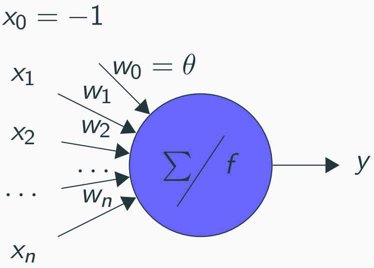
$$w_1 x_1 + w_2 x_2 + \dots w_n x_n - \theta > 0$$

$$w_1 x_1 + w_2 x_2 + \dots w_n x_n + \theta(-1) > 0$$

Simplify by converting θ into a weight

$$\sum_{i=1}^{n} w_i x_i > \theta$$

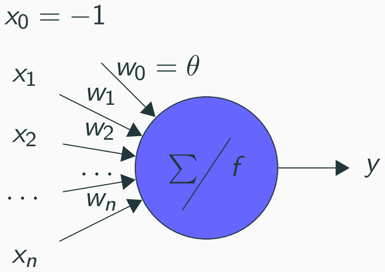
$$\sum_{i=1}^{n} w_i x_i - \theta > 0$$



$$w_1 x_1 + w_2 x_2 + \dots w_n x_n - \theta > 0$$

$$w_1 x_1 + w_2 x_2 + \dots w_n x_n + \theta(-1) > 0$$

Simplify by converting θ into a weight



Let $x_0 = -1$ be the weight of θ. Now our activation function is:

$$y = f(u(\mathbf{x})) = \begin{cases} 1, & \text{if } u(\mathbf{x}) > 0 \\ 0, & \text{otherwise} \end{cases}$$

Learn by adjusting weights whenever output $\neq$ target

Intuition: classification depends on the sign ($+$ or -) of the output. If output has a different sign than the target, adjust weights to move output in the direction of 0.

Learn by adjusting weights whenever output $\neq$ target

Intuition: classification depends on the sign (+ or -) of the output. If output has a different sign than the target, adjust weights to move output in the direction of 0.

$o = 0$ and $t = 0$ Don't adjust weights

Learn by adjusting weights whenever output $\neq$ target

Intuition: classification depends on the sign (+ or -) of the output. If output has a different sign than the target, adjust weights to move output in the direction of 0.

$o = 0$ and $t = 0$	Don't adjust weights
$o = 0$ and $t = 1$	$u(\mathbf{x})$ was too low. Make it bigger!

Learn by adjusting weights whenever output $\neq$ target

Intuition: classification depends on the sign (+ or -) of the output. If output has a different sign than the target, adjust weights to move output in the direction of 0.

$o = 0$ and $t = 0$	Don't adjust weights
$o = 0$ and $t = 1$	$u(\mathbf{x})$ was too low. Make it bigger!
$o = 1$ and $t = 0$	$u(\mathbf{x})$ was too high. Make it smaller!

Learn by adjusting weights whenever output $\neq$ target

Intuition: classification depends on the sign (+ or -) of the output. If output has a different sign than the target, adjust weights to move output in the direction of 0.

$o = 0$ and $t = 0$	Don't adjust weights
$o = 0$ and $t = 1$	$u(\mathbf{x})$ was too low. Make it bigger!
$o = 1$ and $t = 0$	$u(\mathbf{x})$ was too high. Make it smaller!
$o = 1$ and $t = 1$	Don't adjust weights

Learn by adjusting weights whenever output $\neq$ target

Intuition: classification depends on the sign (+ or -) of the output. If output has a different sign than the target, adjust weights to move output in the direction of 0.

$o = 0$ and $t = 0$	Don't adjust weights
$o = 0$ and $t = 1$	$u(\mathbf{x})$ was too low. Make it bigger!
$o = 1$ and $t = 0$	$u(\mathbf{x})$ was too high. Make it smaller!
$o = 1$ and $t = 1$	Don't adjust weights

Notice: the sign of $t - o$ is the direction we want to move in.

Learn by adjusting weights whenever output $\neq$ target

Perceptron Learning Rule

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

$$\Delta w_i = \eta(t - o)x_i$$

- η, $0 < \eta \leq 1$ is a constant called the learning rate.
- t is the target output of the current example.
- o is the output of the Perceptron with the current weights.

Perceptron Learning Rule

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

$$\Delta w_i \ = \eta(t - o)x_i$$

$o = 1$ and $t = 1$
$o = 0$ and $t = 1$

- Learning rate η is positive; controls how big changes Δw_i are.
- If $x_i > 0$, $\Delta w_i > 0$. Then w_i increases in an so that $w_i x_i$ becomes larger, increasing $u(\mathbf{x})$.
- If $x_i < 0$, $\Delta w_i < 0$. Then w_i reduces so that the absolute value of $w_i x_i$ becomes smaller, increasing $u(\mathbf{x})$.

Perceptron Learning Rule

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

$$\Delta w_i = \eta(t - o)x_i$$

$$\begin{aligned} o = 1 \text{ and } t = 1 \quad \Delta w_i &= \eta(t - o)x_i = \eta(1 - 1)x_i = 0 \\ o = 0 \text{ and } t = 1 \quad \Delta w_i &= \eta(t - o)x_i = \eta(1 - 0)x_i = \eta x_i \end{aligned}$$

- Learning rate η is positive; controls how big changes Δw_i are.
- If $x_i > 0$, $\Delta w_i > 0$. Then w_i increases in an so that $w_i x_i$ becomes larger, increasing $u(\mathbf{x})$.
- If $x_i < 0$, $\Delta w_i < 0$. Then w_i reduces so that the absolute value of $w_i x_i$ becomes smaller, increasing $u(\mathbf{x})$.

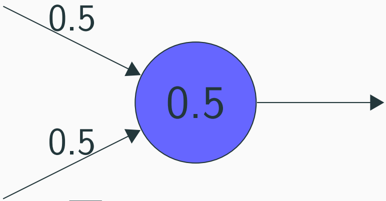
Learning Algorithm

Initialize all weights randomly.
repeat
    for each training example **do**
        Apply the learning rule.
    end for
until the error is acceptable or a certain number of iterations is reached

This algorithm is guaranteed to find a solution with error zero in a limited number of iterations as long as the examples are linearly separable.

Perceptrons can represent some logic functions... but not all!

Perceptron for XOR



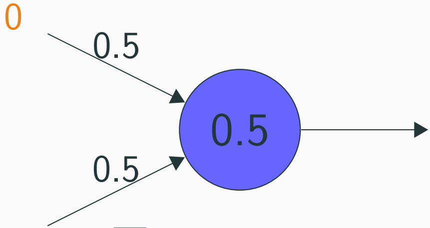
f if $\sum \geq \theta$ then 1 else 0

x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

XOR is an exclusive OR because it only returns a true value of 1 if the two values are exclusive, i.e., they are both different.

Perceptrons can represent some logic functions... but not all!

Perceptron for XOR



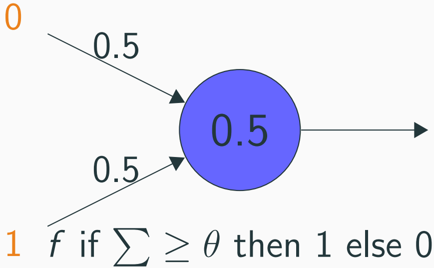
f if $\sum \geq \theta$ then 1 else 0

x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

XOR is an exclusive OR because it only returns a true value of 1 if the two values are exclusive, i.e., they are both different.

Perceptrons can represent some logic functions... but not all!

Perceptron for XOR



f if $\sum \geq \theta$ then 1 else 0

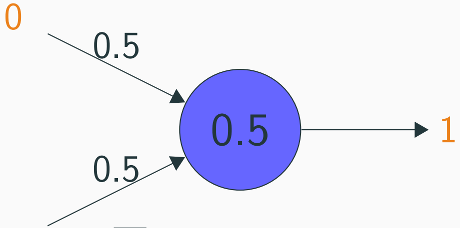
$$0 \cdot 0.5 + 1 \cdot 0.5 = 0.5$$

x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

XOR is an exclusive OR because it only returns a true value of 1 if the two values are exclusive, i.e., they are both different.

Perceptrons can represent some logic functions... but not all!

Perceptron for XOR



f if $\sum \geq \theta$ then 1 else 0

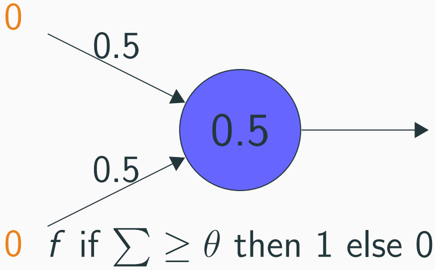
$0 \cdot 0.5 + 1 \cdot 0.5 = 0.5$

x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

XOR is an exclusive OR because it only returns a true value of 1 if the two values are exclusive, i.e., they are both different.

Perceptrons can represent some logic functions... but not all!

Perceptron for XOR

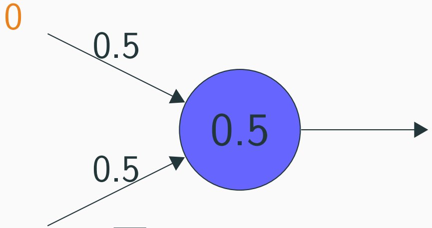


x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

XOR is an exclusive OR because it only returns a true value of 1 if the two values are exclusive, i.e., they are both different.

Perceptrons can represent some logic functions... but not all!

Perceptron for XOR



0 f if $\sum \geq \theta$ then 1 else 0

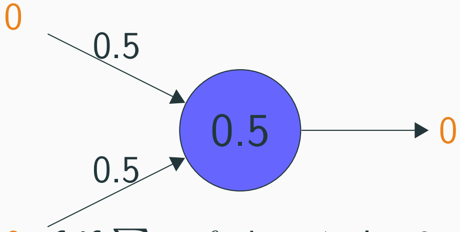
$$0 \cdot 0.5 + 0 \cdot 0.5 = 0$$

x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

XOR is an exclusive OR because it only returns a true value of 1 if the two values are exclusive, i.e., they are both different.

Perceptrons can represent some logic functions... but not all!

Perceptron for XOR



f if $\sum \geq \theta$ then 1 else 0

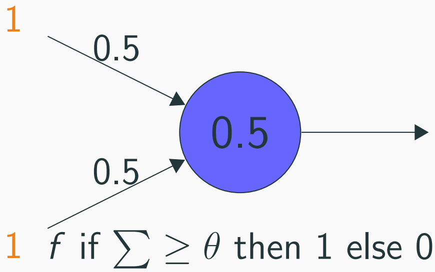
$$0 \cdot 0.5 + 0 \cdot 0.5 = 0$$

x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

XOR is an exclusive OR because it only returns a true value of 1 if the two values are exclusive, i.e., they are both different.

Perceptrons can represent some logic functions... but not all!

Perceptron for XOR

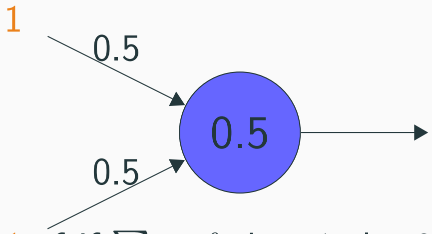


x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

XOR is an exclusive OR because it only returns a true value of 1 if the two values are exclusive, i.e., they are both different.

Perceptrons can represent some logic functions... but not all!

Perceptron for XOR



f if $\sum \geq \theta$ then 1 else 0

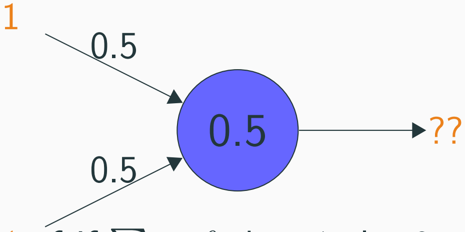
$$1 \cdot 0.5 + 1 \cdot 0.5 = 1$$

x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

XOR is an exclusive OR because it only returns a true value of 1 if the two values are exclusive, i.e., they are both different.

Perceptrons can represent some logic functions... but not all!

Perceptron for XOR



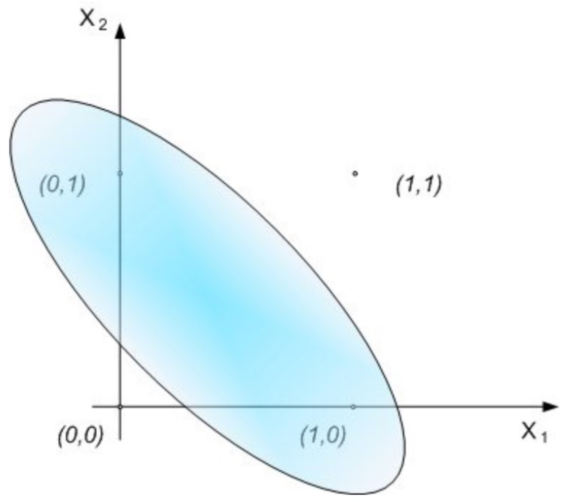
f if $\sum \geq \theta$ then 1 else 0

$$1 \cdot 0.5 + 1 \cdot 0.5 = 1$$

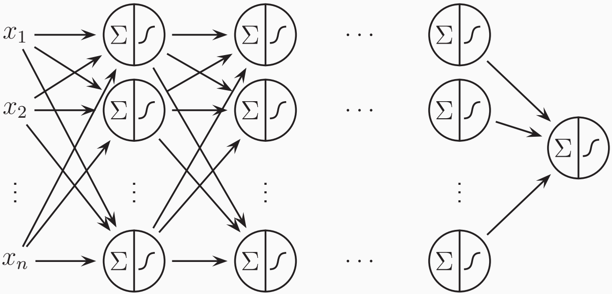
x_1	x_2	x_1 XOR x_2
0	0	0
0	1	1
1	0	1
1	1	0

XOR is an exclusive OR because it only returns a true value of 1 if the two values are exclusive, i.e., they are both different.

Problem: XOR is not linearly separable



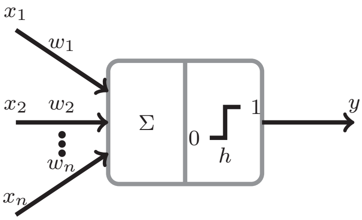
Multilayer Perceptrons (MLPs) are more expressive



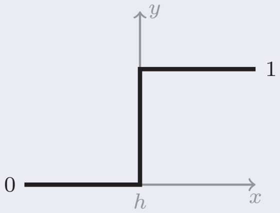
- MLPs are feed-forward neural networks, organized in layers.
- One input layer, one or more hidden layers, one output layer.
- Each node in a layer connected to all other nodes in next layer.
- Each connection has a weight (can be zero).
- Universal function approximators: can represent XOR.

Q: How would you represent XOR?

We can use activation functions other than thresholds

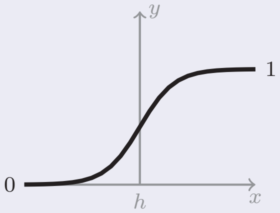


Step function



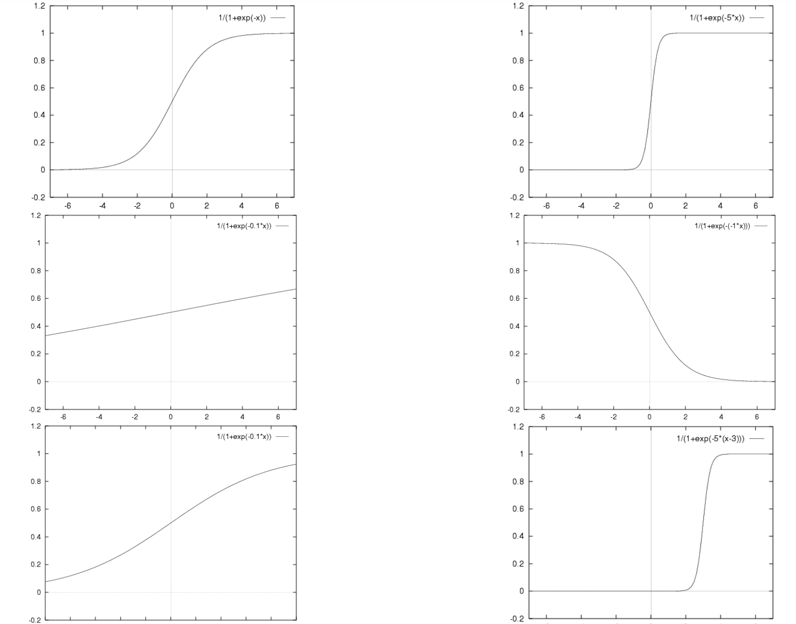
Outputs 0 or 1.

Sigmoid function

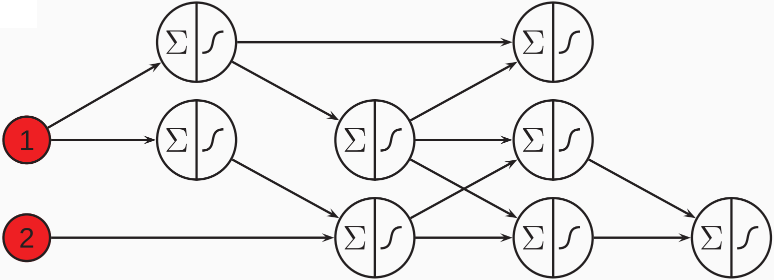


Outputs a real value between 0 and 1.

Sigmoid can be made sharper or smoother

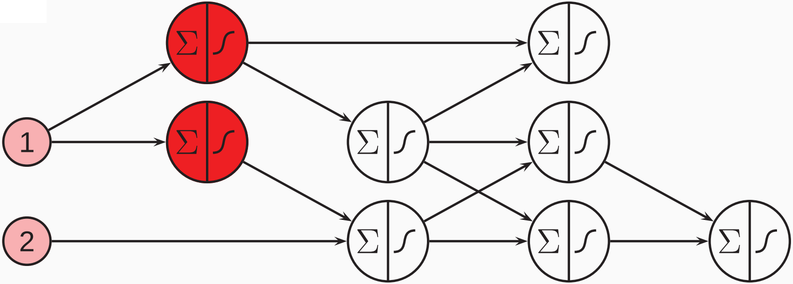


Forward Pass



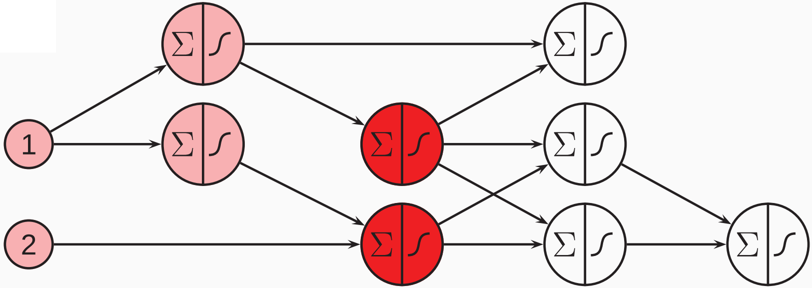
1. Present the pattern at the input layer.

Forward Pass



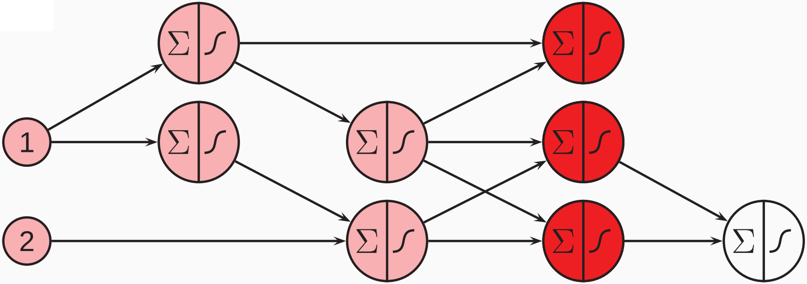
1. Present the pattern at the input layer.
2. Calculate activation of input neurons

Forward Pass



1. Present the pattern at the input layer.
2. Calculate activation of input neurons
3. Propagate forward activations step by step.

Forward Pass

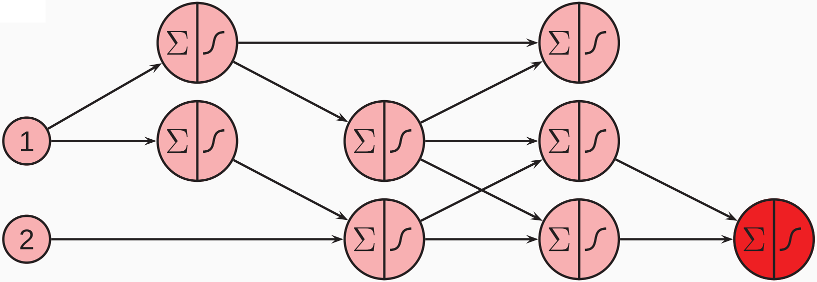


1. Present the pattern at the input layer.
2. Calculate activation of input neurons
3. Propagate forward activations step by step.

Forward Pass

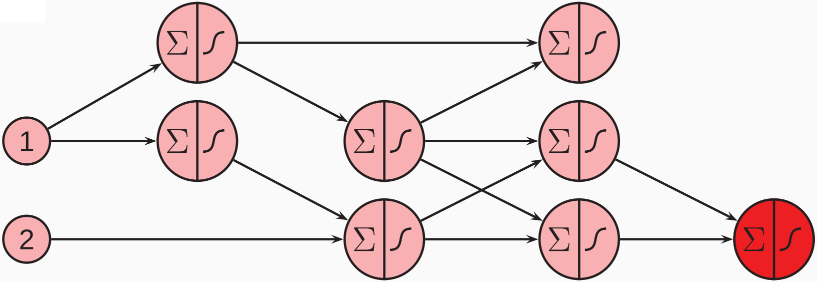
1. Present the pattern at the input layer.
2. Calculate activation of input neurons
3. Propagate forward activations step by step.

Forward Pass



1. Present the pattern at the input layer.
2. Calculate activation of input neurons
3. Propagate forward activations step by step.

Forward Pass



1. Present the pattern at the input layer.
2. Calculate activation of input neurons
3. Propagate forward activations step by step.
4. Read the network output from both output neurons.

Learning in multilayer perceptrons

General Idea: same as in a simple perceptron

1. Send the MLP an input pattern, x, from the **training set**.
2. Get the output from the MLP, y.
3. Compare y with the "right answer", or target t, to get the **error quantity**.
4. Use the error quantity to modify the weights, so next time y will be closer to t.
5. Repeat with another x from the training set.

When updating weights after seeing x, the network doesn't just change the way it deals with x, but other inputs too . . .

Inputs it has not seen yet!

Generalization is the ability to deal accurately with unseen inputs.

Learning as Error Minimization

The perceptron learning rule minimizes the difference between the actual and desired outputs:

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

Generalization of this: Mean Squared Error (MSE)

An **error function** represents such a difference over a set of inputs:

$$E(\vec{w}) = \frac{1}{2N} \sum_{p=1}^{N} (t^p - o^p)^2$$

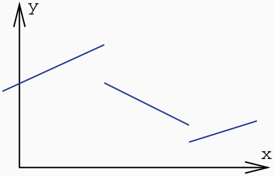
- N is the number of patterns
- t^p is the target output for pattern p
- o^p is the output obtained for pattern p
- the 2 makes little difference, but makes life easier later on!

Minimize error by gradient descent

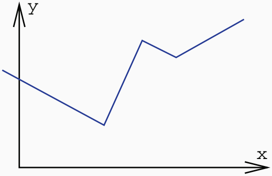
Interpret E just as a mathematical function depending on $\vec{w}$ and forget about its semantics, then we are faced with a problem of mathematical optimization.

$$\underset{\vec{u}}{\text{minimize}}\ f(\vec{u})$$

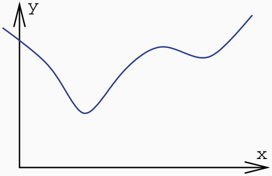
We consider only continuous and differentiable functions.



non continuous function (disrupted)



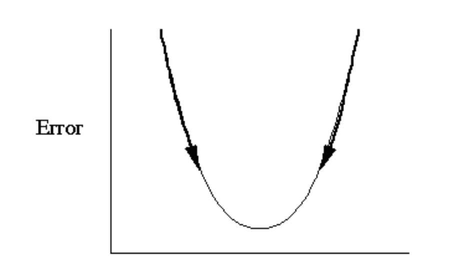
continuous, non differentiable function (folded)

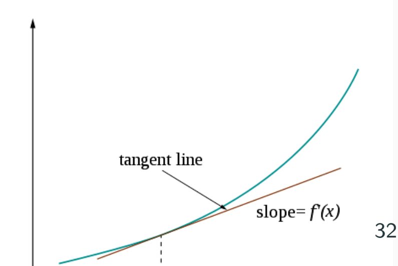


differentiable function (smooth)

Gradient and Derivatives: The Idea

- Gradient descent can be used for minimizing functions.
- The derivative is a measure of the rate of change of a function, as its input changes;
- For function $y = f(x)$, the derivative $\frac{dy}{dx}$ indicates how much y changes in response to changes in x.
- If x and y are real numbers, and if the graph of y is plotted against x, the derivative measures the slope or gradient of the line at each point, i.e., it describes the steepness or incline.





Gradient and Derivatives: The Idea

- So, we know how to use derivatives to adjust one input value.
- But we have several weights to adjust!
- We need to use partial derivatives.
- A partial derivative of a function of several variables is its derivative with respect to one of those variables, with the others held constant.

Example

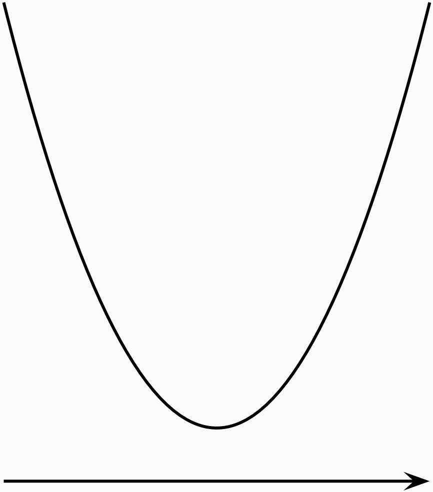
If $y = f(x_1, x_2)$, then we can have $\frac{\partial y}{\partial x_1}$ and $\frac{\partial y}{\partial x_2}$.

Given partial derivatives, update the weights:

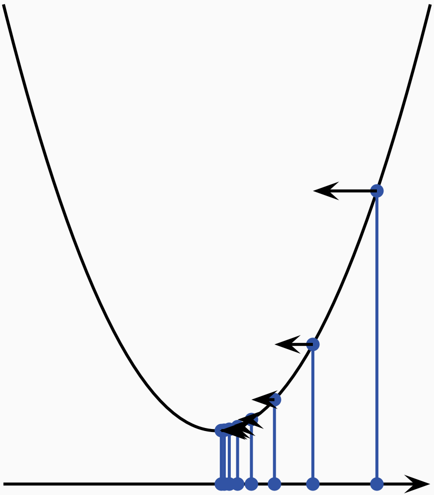
$$w'_{ij} = w_{ij} + \Delta w_{ij}$$

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

Learning Rate

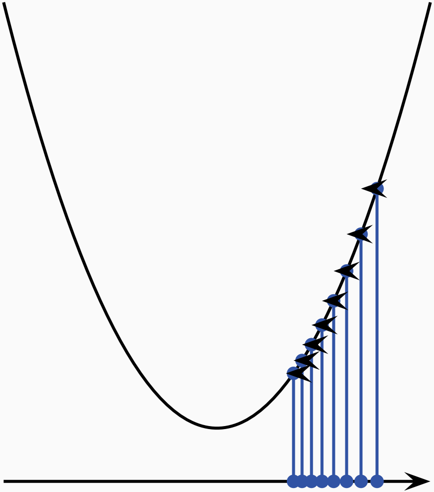


Learning Rate



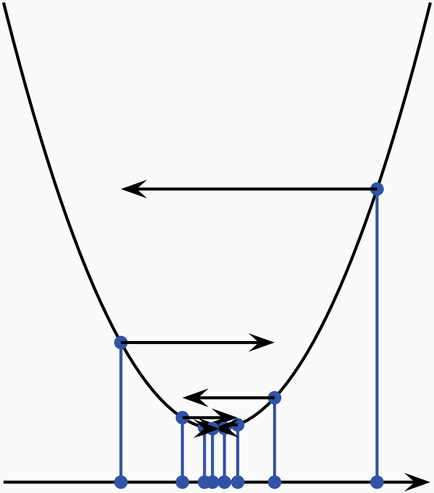
Small η leads to convergence.

Learning Rate



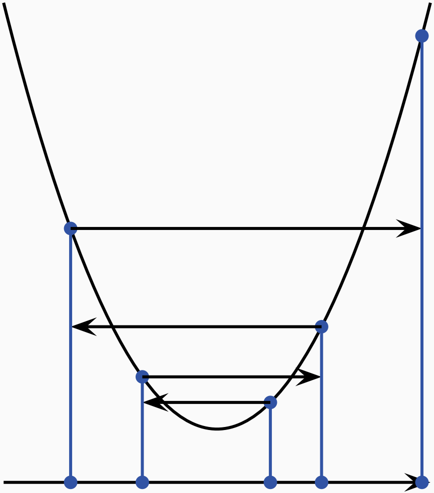
Very small η, convergence may take very long.

Learning Rate



Case of medium size η, also converges.

Learning Rate



Very lage η: divergence.

Gradient Descent (cont.)

- Pure gradient descent is a nice theoretical framework but of limited power in practice.
- Finding the right η is annoying. Approaching the minimum is time consuming.
- Heuristics to overcome problems of gradient descent:
  - gradient descent with momentum
  - individual learning rates for each dimension
  - adaptive learning rates
  - decoupling step length from partial derivates

Summary So Far

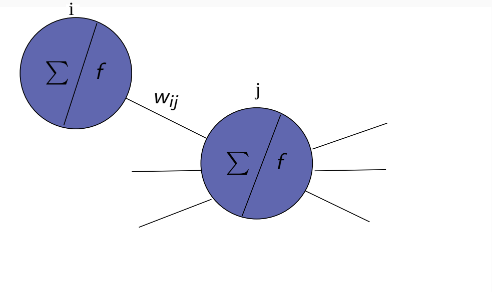
- We learnt what a multilayer perceptron is.
- We know a learning rule for updating weights in order to minimise the error:

$$w'_{ij} = w_{ij} + \Delta w_{ij}$$

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

- Δw_{ij} tells us in which direction and how much we should change each weight to roll down the slope (descend the gradient) of the error function E.
- So, how do we calculate $\frac{\partial E}{\partial w_{ij}}$?

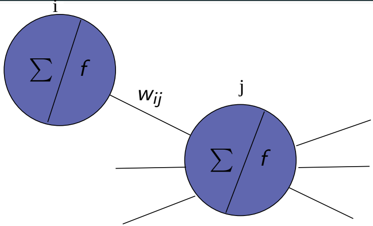
Using Gradient Descent to Minimize the Error



The mean squared error function E, which we want to minimize:

$$E(\vec{w}) = \frac{1}{2N} \sum_{p=1}^{N} (t^p - o^p)^2$$

Using Gradient Descent to Minimize the Error



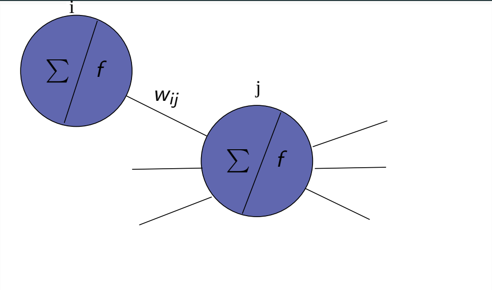
If we use a sigmoid activation function f, then the output of neuron i for pattern p is:

$$o_i^p = f(u_i) = \frac{1}{1 + e^{au_i}}$$

where a is a pre-defined constant and u_i is the result of the input function in neuron i:

$$u_i = \sum_j w_{ij} x_{ij}$$

Using Gradient Descent to Minimize the Error

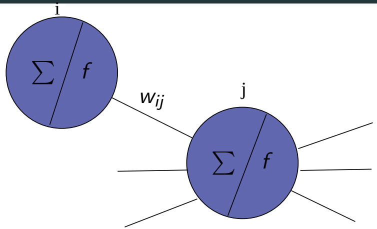


For the pth pattern and the ith neuron, we use gradient descent on the error function:

$$\Delta w_{ij} = -\eta \frac{\partial E_p}{\partial w_{ij}} = \eta(t_i^p - o_i^p) f'(u_i) x_{ij}$$

where $f'(u_i) = \frac{df}{du_i}$ is the derivative of f with respect to u_i.
If f is the sigmoid function, $f'(u_i) = af(u_i)(1 - f(u_i))$.

Using Gradient Descent to Minimize the Error



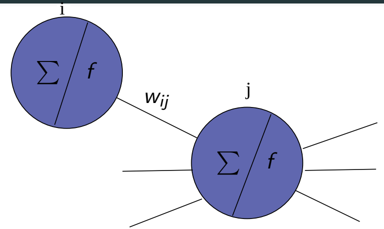
We can update weights after processing each pattern, using rule:

$$\Delta w_{ij} = \eta \, (t_i^p - o_i^p) \, f'(u_i) \, x_{ij}$$

$$\Delta w_{ij} = \eta \, \delta_i^p \, x_{ij}$$

- This is known as the generalized delta rule.
- We need to use the derivative of the activation function f.
  So, f must be differentiable!
- Sigmoid has a derivative which is easy to calculate.

Using Gradient Descent to Minimize the Error



We can update weights after processing each pattern, using rule:

$$\Delta w_{ij} = \eta \; (t_i^p - o_i^p) \; f'(u_i) \; x_{ij}$$

$$\Delta w_{ij} = \eta \; \delta_i^p \; x_{ij}$$

- This is known as the generalized delta rule.
- We need to use the derivative of the activation function f. So, f must be differentiable!
- Sigmoid has a derivative which is easy to calculate.

Updating Output vs Hidden Neurons

We can update output neurons using the generalize delta rule:

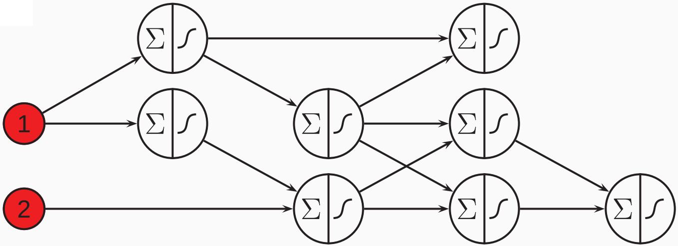
$$\Delta w_{ij} = \eta \; \delta_i^p \; x_{ij}$$
$$\delta_i^p = (t_i^p - o_i^p) f'(u_i)$$

This δ_i^p is only good for the output neurons, it relies on target outputs. But we don't have target output for the hidden nodes!

$$\Delta w_{ki} = \eta \; \delta_k^p \; x_{ik} \qquad \delta_k^p = \sum_{j \in I_k} \delta_j^p w_{kj}$$

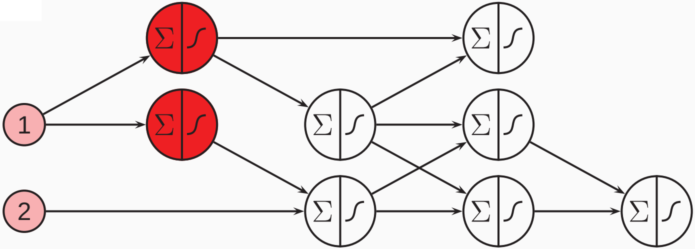
This rule propagates error back from output nodes to hidden nodes. If effect, it blames hidden nodes according to how much influence they had. So, now we have rules for updating both output and hidden neurons!

Backpropagation



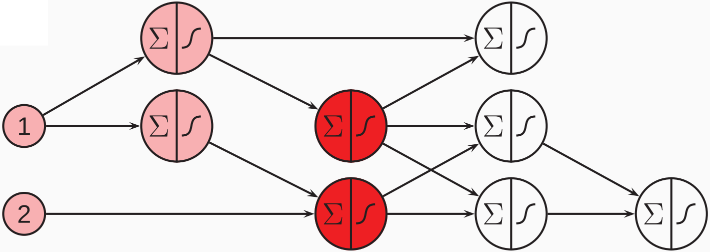
1. Present the pattern at the input layer.

Backpropagation



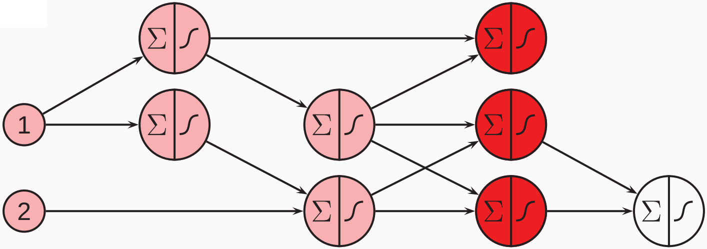
1. Present the pattern at the input layer.
2. Propagate forward activations

Backpropagation



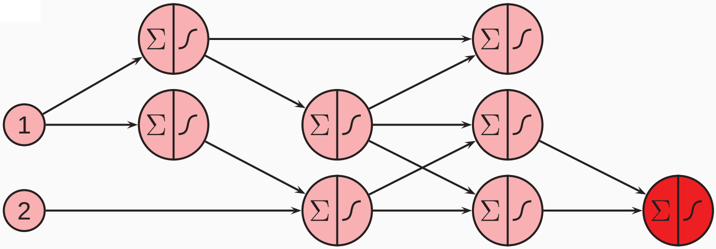
1. Present the pattern at the input layer.
2. Propagate forward activations step

Backpropagation



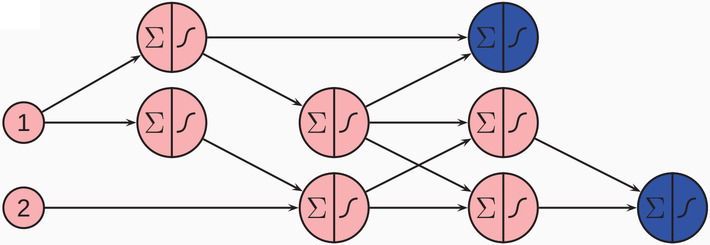
1. Present the pattern at the input layer.
2. Propagate forward activations step by

Backpropagation



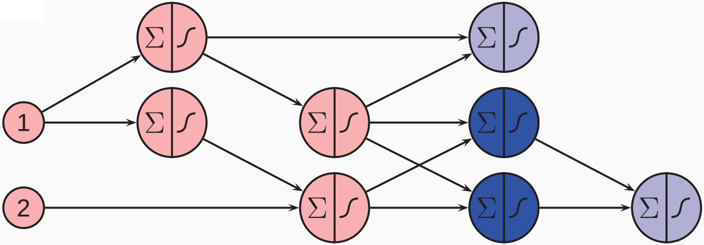
1. Present the pattern at the input layer.
2. Propagate forward activations step by step.

Backpropagation



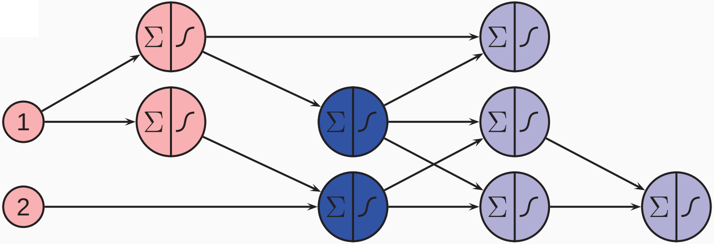
1. Present the pattern at the input layer.
2. Propagate forward activations step by step.
3. Calculate error from both output neurons.

Backpropagation



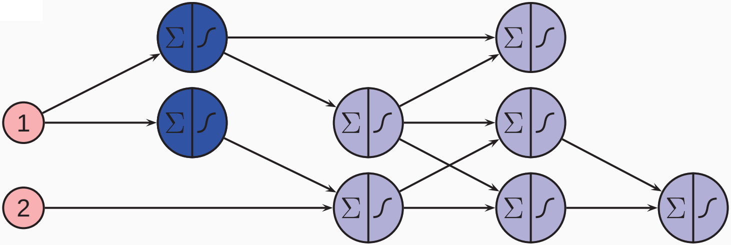
1. Present the pattern at the input layer.
2. Propagate forward activations step by step.
3. Calculate error from both output neurons.
4. Propagate backward error.

Backpropagation



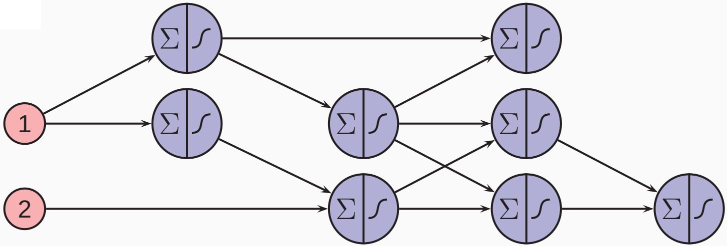
1. Present the pattern at the input layer.
2. Propagate forward activations step by step.
3. Calculate error from both output neurons.
4. Propagate backward error.

Backpropagation



1. Present the pattern at the input layer.
2. Propagate forward activations step by step.
3. Calculate error from both output neurons.
4. Propagate backward error.

Backpropagation



1. Present the pattern at the input layer.
2. Propagate forward activations step by step.
3. Calculate error from both output neurons.
4. Propagate backward error.
5. Calculate $\frac{\partial E}{\partial w_{ij}}$; repeat for all patterns and sum up.

Online Backpropagation

Initialize all weights to small random values.
repeat
    for each training example **do**
        Forward propagate the input features of the example to determine the MLP's outputs.
        Back propagate error to generate Δw_{ij} for all weights w_{ij}.
        Update the weights using Δw_{ij}.
    end for
until stopping criteria reached.

Summary

- We learnt what a multilayer perceptron is.
- We have some intuition about using gradient descent on an error function.
- We know a learning rule for updating weights in order to minimize the error: $\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}}$
- If we use the squared error, we get the generalized delta rule: $\Delta w_{ij} = \eta \delta_i^p x_{ij}$.
- We know how to calculate δ_i^p for output and hidden layers.
- We can use this rule to learn an MLP's weights using the backpropagation algorithm.