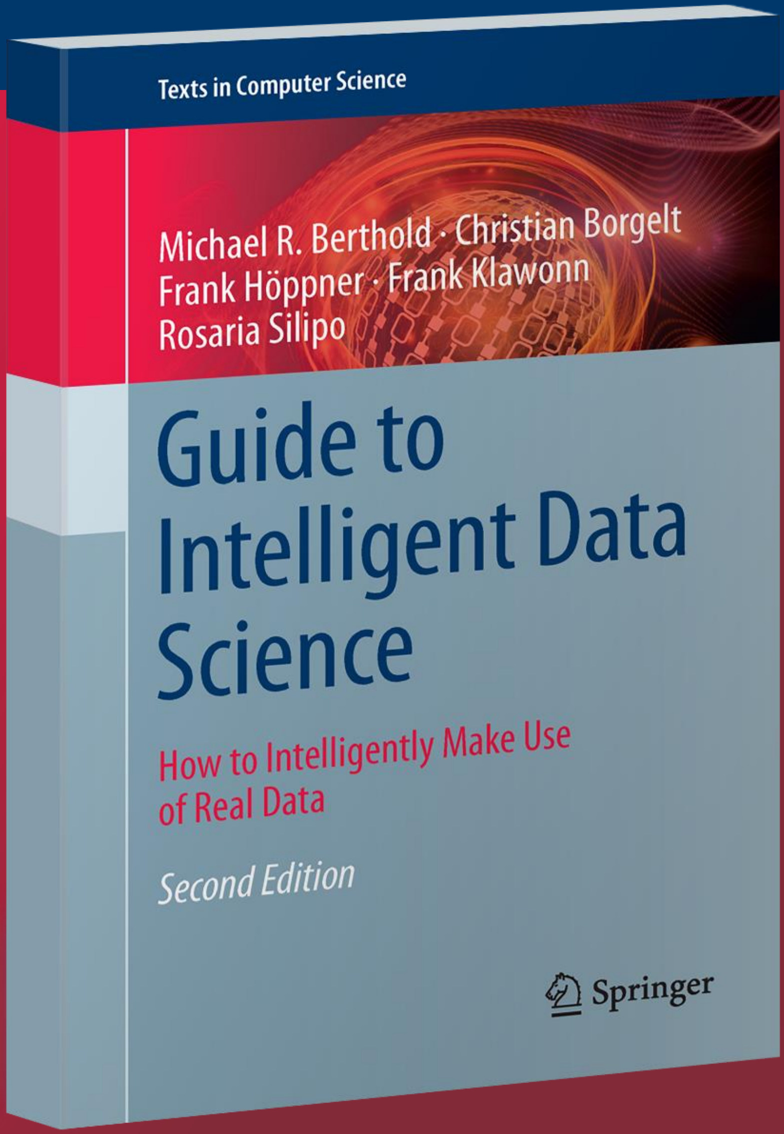
Neural Networks



Caption

Summary of this lesson

"Tell me and I forget. Teach me and I remember. Involve me and I learn."
-Benjamin Franklin

How do machines *learn*?

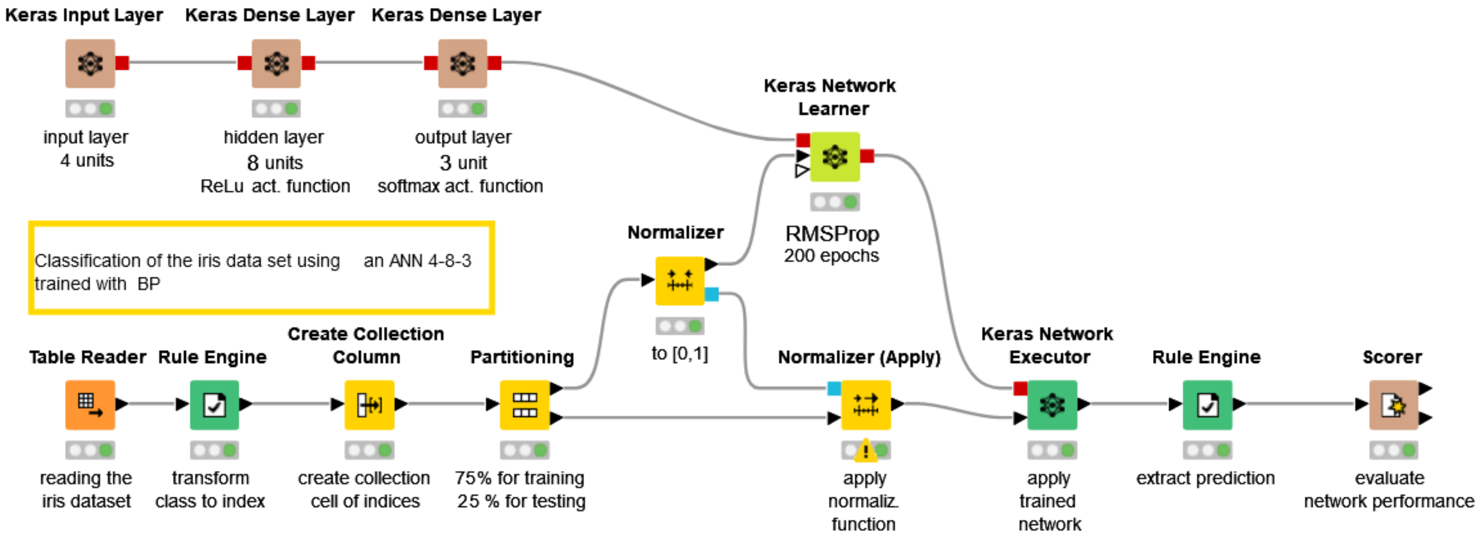
**This lesson refers to chapter 9 of the GIDS book*

What you will learn

- Multilayer Perceptrons
  - The Perceptron
  - Why the Perceptron is not enough
  - The MLP
- The Back Propagation algorithm
  - The delta rule to train the output neurons
  - Recursivity to train the hidden neurons
  - Learning rate and local minima
- MLP and BackPropagation
  - Pro‘s and Con‘s
    - Black-box tools
    - Overfitting
    - Techniques to avoid overfitting
    - Special architectures

Datasets

- Datasets used : iris dataset
- Example Workflows:
  - „Classifying the iris data set with ANN“ https://kni.me/w/ei3eX9Sj5-RFEUat
    - Keras layers
    - Multi-layer perceptrion
    - Back propagation



Supervised Learning

- A target attribute is predicted based on other attributes
- Assumption: in addition to the object description **x**, we have also the value for the target attribute **y**

Transparent

- The decision process maps the application domain
- *How did we come to this final medical diagnosis?*

Black-box

- Abstract mathematical procedures not meaningful for the application domain
- *Recover most similar face in a million. How and why is not important.*

Biological Neuron

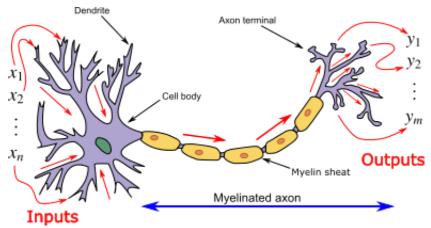
- Artificial Neural Networks (ANN) are among the oldest and most intensely studied Machine Learning approaches
- They took their inspiration from biological neural networks and tried to mimic the learning process of animals and humans
- However, the model of biological processes ended up to be very coarse, and several improvements to the basic approach have even abandoned the biological analogy

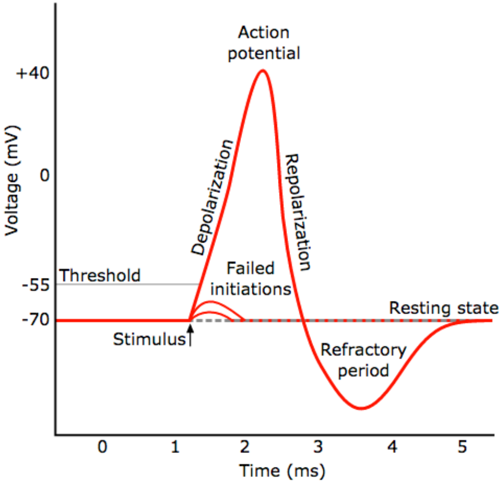
Advantage	**Disadvantage**
Highly flexible => good performance	Black-box models not easy to interpret

Biological Neural Networks

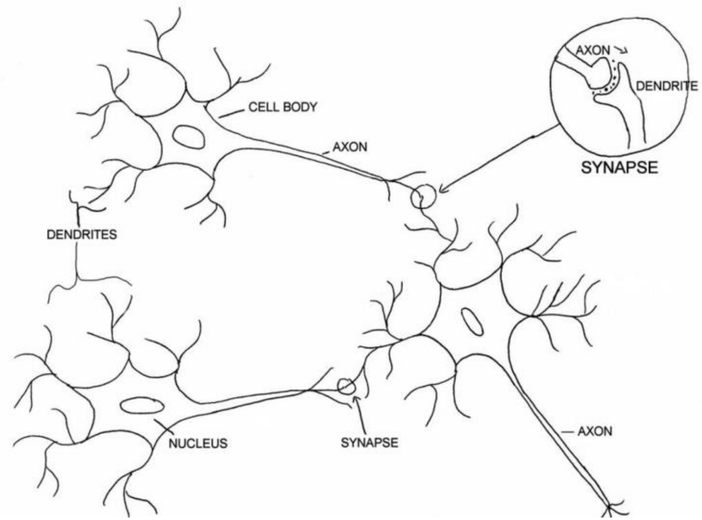
- Human brain: ca. 10^{11} neurons.
- Each neuron connected to 10^4 other neurons on average.
- Switching time of a neuron 10^{-3} sec (computer: 10^{-10} sec ...)
- Neurons compute very basic functions
- Neuron assembly performs complex recognition tasks (faces!) in 10^{-1} sec!
- The human brain: gigantic assembly of highly connected simple processing units...

Biological Neuron





Biological Neural Networks



Perceptron

McCulloch-Pitts Model of a neuron (1943)

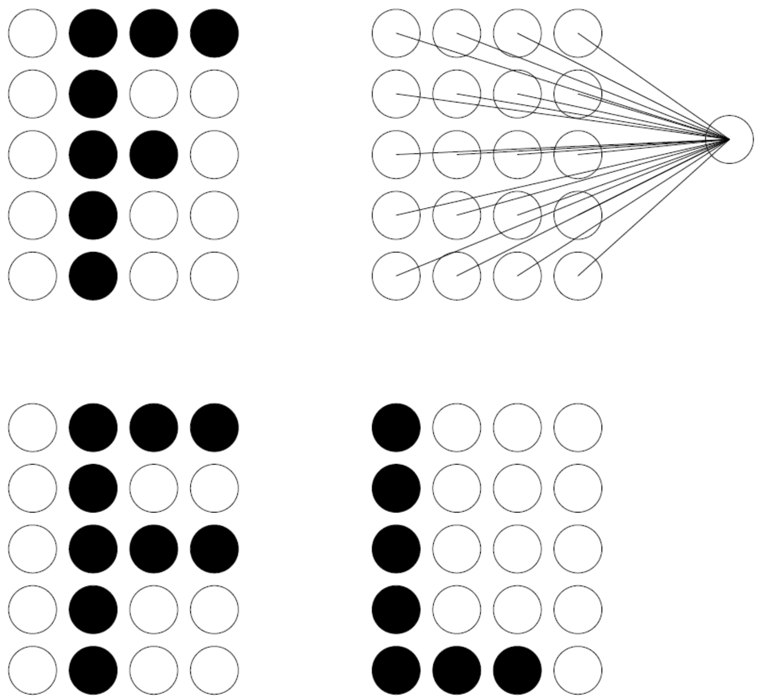
- **Aim**: neurobiological modelling and simulation to understand very elementary functions of neurons and brain.
- A neuron is a binary switch, being either active or inactive.
- Each neuron has a fixed threshold value.
- A neuron receives input signals from excitatory (positive) synapses (connections to other neuron).
- A neuron receives input signals from inhibitory (negative) synapses (connections to other neuron).
- Inputs to a neuron are accumulated (integrated) for a certain time. When the threshold value of the neuron is exceeded, the neuron becomes active and sends signals to its neighbouring neurons via its synapses.

The Perceptron (Rosenblatt, 1958)

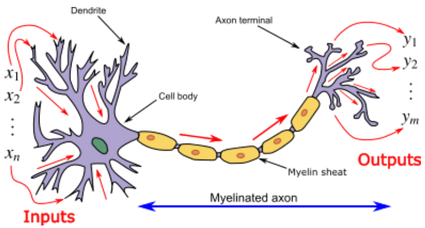
- The perceptron was introduced by Frank Rosenblatt for modelling pattern recognition abilities in 1958.
- **Aim**: Automatic learning of weights and threshold of a model of a retina to correctly classify objects.
- A simplified retina is equipped with receptors (input neurons) that are activated by an optical stimulus.
- The stimulus is passed on to an output neuron via a weighted connection (synapse).
- When the threshold of the output neuron is exceeded, the output is 1, otherwise 0.

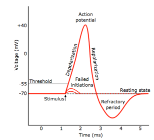
Identifying the letter F

– 2 positive and 1 negative example

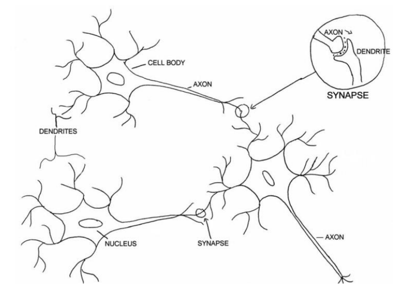


Biological Neuron

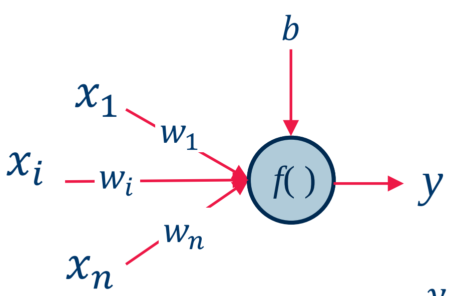




Biological Neural Networks



Artificial Neuron (**Perceptron**)



$$a(\boldsymbol{x}) = \sum_{i=1}^{n} x_i w_i$$

$$y = f(\boldsymbol{x}) = \begin{cases} 1 & if \;\; a(\boldsymbol{x}) \geq b \\ 0 & if \;\; a(\boldsymbol{x}) < b \end{cases}$$

- Numerical input attributes x_i
- Binary output class (0 or 1).
- Classifier for two-class problem
- For multiclass problems use one perceptron per class

Perceptron Learning Algorithm

- Initialise the weight and the threshold values randomly.

- For each data object in the training data set, check whether the perceptron predicts the correct class.

- If the perceptron predicts the wrong class, adjust the weights and threshold value to improve the prediction.

- Repeat this until no changes occur

The delta rule

- Whenever the perceptron makes a wrong classification => change weights and threshold in ”**appropriate direction**”.

- If the desired output is 1 and the perceptron’s output is 0, the threshold is not exceeded, although it should be. Therefore, lower the threshold and adjust the weights depending on the sign and magnitude of the inputs.

- If the desired output is 0 and the perceptron’s output is 1, the threshold is exceeded, although it should not be. Therefore, increase the threshold and adjust the weights depending on the sign and magnitude of the inputs.

- The **delta rule** recommends to adjust the weight and the threshold values as:

$$w_i^{new} = w_i^{old} + \Delta w_i$$

$$b^{new} = b^{old} + \Delta b$$

- w_i: A weight of the perceptron
- b : The threshold value of the perceptron
- $(x_1, x_2, \dots, x_n)$: An input vector
- t: the desired output for input vector $(x_1, x_2, \dots, x_n)$
- y: the real output of the Perceptron for input vector $(x_1, x_2, \dots, x_n)$
- $\eta > 0$: the Learning rate

- The **delta rule** recommends to adjust the weight and the threshold values as:

$$\Delta w_i = \begin{cases} 0 & if\ y = t \\ +\eta x_i & if\ y = 0\ and\ t = 1 \\ -\eta x_i & if\ y = 1\ and\ t = 0 \end{cases}$$

$$\Delta b = \begin{cases} 0 & if\ y = t \\ +\eta & if\ y = 0\ and\ t = 1 \\ -\eta & if\ y = 1\ and\ t = 0 \end{cases}$$

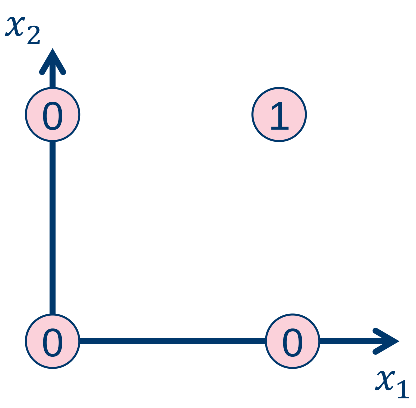
Example: Learning the logical operator AND

- Training Data:

Inputs

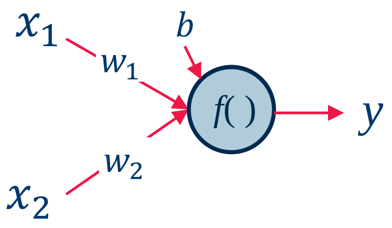
{((0,0),0), ((0,1),0), ((1,0),0), ((1,1), 1)}

target



- Learning rate $\eta = 1$
- Initialization: $w_1 = w_2 = b = 0$

	$\boldsymbol{x}$	t	y	Δw_1	Δw_2	Δb	w_1^{new}	w_2^{new}	b^{new}
Epoch 1	0 0	0	0	0	0	0	0	0	0
	0 1	0	0	0	0	0	0	0	0
	1 0	0	0	0	0	0	0	0	0
	1 1	1	0	1	1	-1	1	1	-1



Example: Learning the logical operator AND

	$\boldsymbol{x}$	t	y	Δw_1	Δw_2	Δb	w_1^{new}	w_2^{new}	b^{new}
Epoch 2	0 0	0	1	0	0	1	1	1	0
	0 1	0	1	0	-1	1	1	0	1
	1 0	0	0	0	0	0	1	0	1
	1 1	1	0	1	1	-1	2	1	0

	$\boldsymbol{x}$	t	y	Δw_1	Δw_2	Δb	w_1^{new}	w_2^{new}	b^{new}
Epoch 3	0 0	0	0	0	0	0	2	1	0
	0 1	0	1	0	-1	1	2	0	1
	1 0	0	1	-1	0	1	1	0	2
	1 1	1	0	1	1	-1	2	1	1

Example: Learning the logical operator AND

	$\boldsymbol{x}$	t	y	Δw_1	Δw_2	Δb	w_1^{new}	w_2^{new}	b^{new}
Epoch 4	0 0	0	0	0	0	0	2	1	1
	0 1	0	0	0	0	0	2	1	1
	1 0	0	1	-1	0	1	1	1	2
	1 1	1	0	1	1	-1	2	2	1

	$\boldsymbol{x}$	t	y	Δw_1	Δw_2	Δb	w_1^{new}	w_2^{new}	b^{new}
Epoch 5	0 0	0	0	0	0	0	2	2	2
	0 1	0	1	0	-1	1	2	1	2
	1 0	0	0	0	0	0	2	1	2
	1 1	1	1	0	0	0	2	1	2

Example: Learning the logical operator AND

	$\boldsymbol{x}$	t	y	Δw_1	Δw_2	Δb	w_1^{new}	w_2^{new}	b^{new}
Epoch 6	0 0	0	0	0	0	0	2	1	2
	0 1	0	0	0	0	0	2	1	2
	1 0	0	0	0	0	0	2	1	2
	1 1	1	1	0	0	0	2	1	2

Perceptron Convergence

If, for a given data set with two classes, there exists a perceptron that can classify all patterns correctly, then the delta rule will adjust the weights and the threshold after a finite number of steps in such way that all patterns are classified correctly.

What classification problems can a perceptron solve?

- Consider a perceptron with two input neurons.
- Let y be the output of the perceptron for input (x_1, x_2)
- Then:

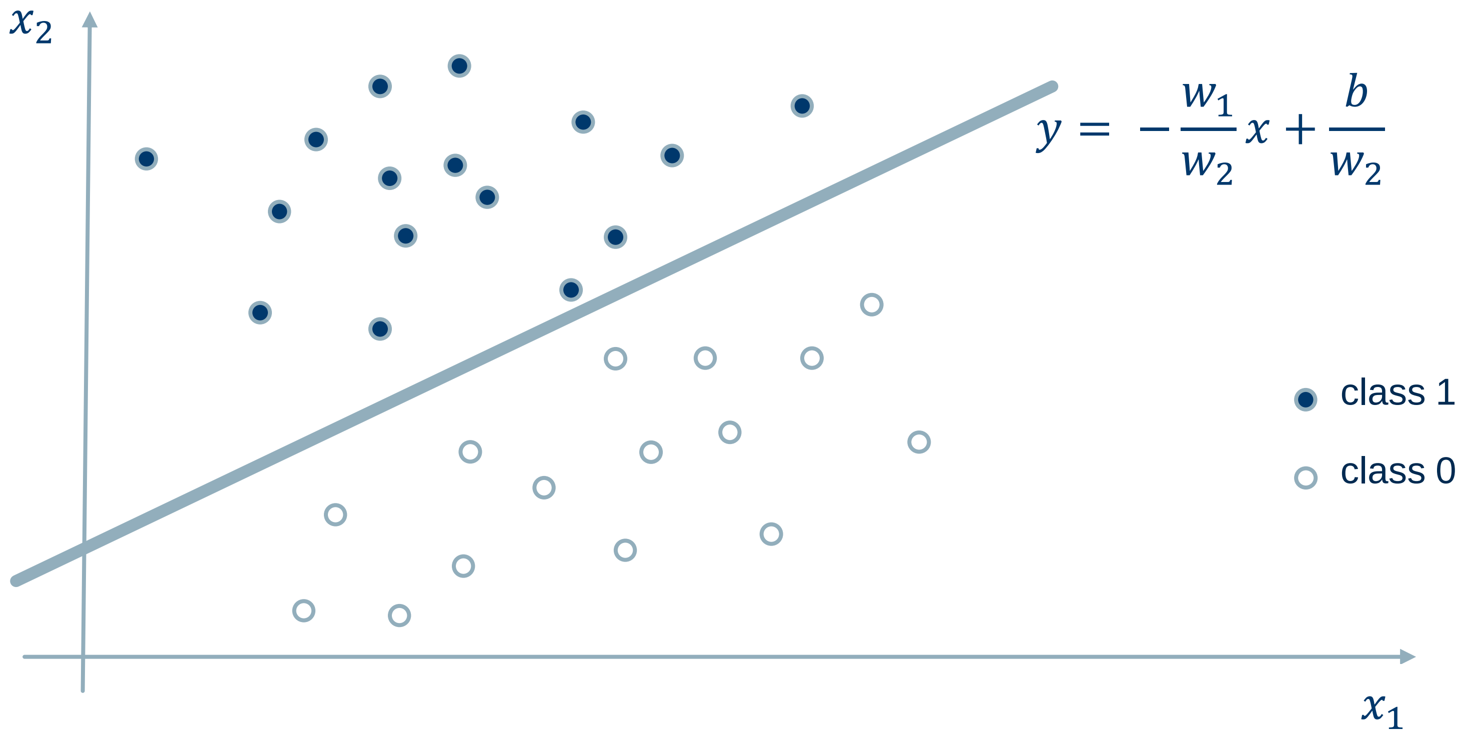
$$y = 1 \iff w_1 \cdot x_1 + w_2 \cdot x_2 > b$$

$$\iff x_2 > -\frac{w_1}{w_2} x_1 + \frac{b}{w_2}$$

- The perceptron output is 1 if and only if the input vector (x_1, x_2) is above the line:

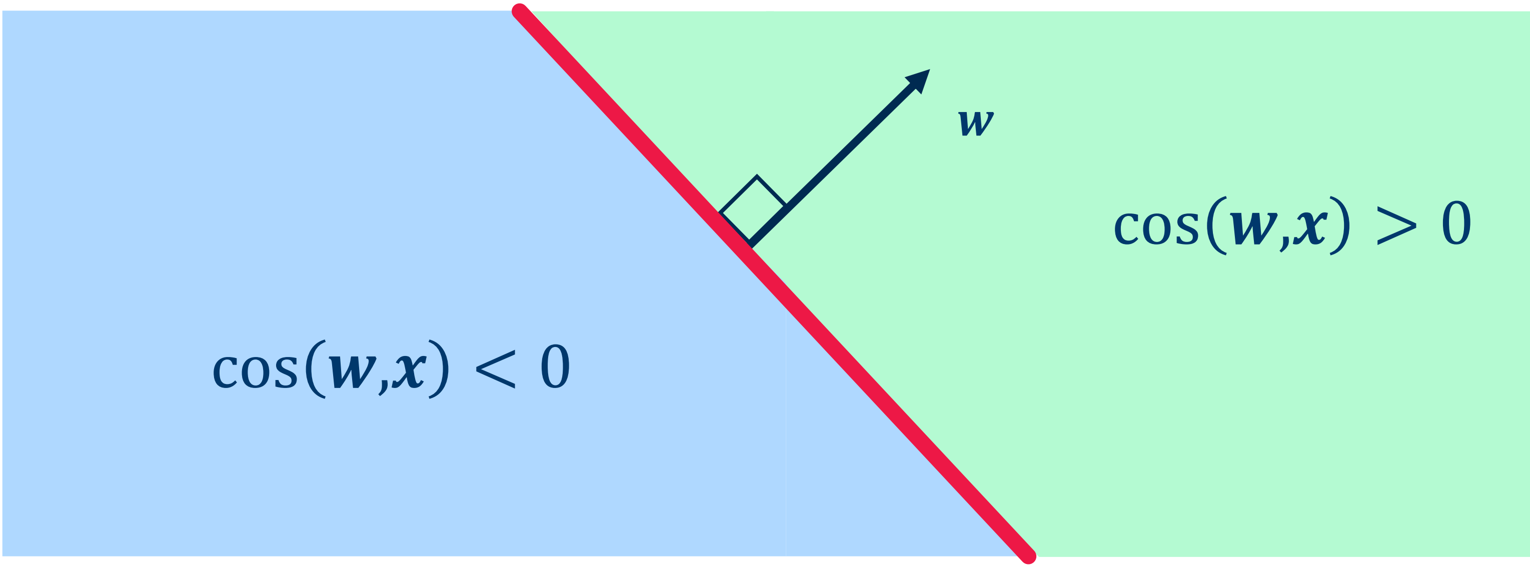
$$y = -\frac{w_1}{w_2} x + \frac{b}{w_2}$$

Linear Separability



The parameters w_1, w_2, define the line. All input patterns above this line are assigned to class 1, all input patterns below the line to class 0.

$$y(\boldsymbol{x}) = w_0 + \sum_{i=1}^{n} w_i \ x_i = w_0 + \boldsymbol{w}^T \cdot \boldsymbol{x} = w_0 + |\boldsymbol{x}| \, |\boldsymbol{w}| \cos(\boldsymbol{w}, \boldsymbol{x})$$



Perceptrons implements hyperplanes in the feature space

Linear Separability

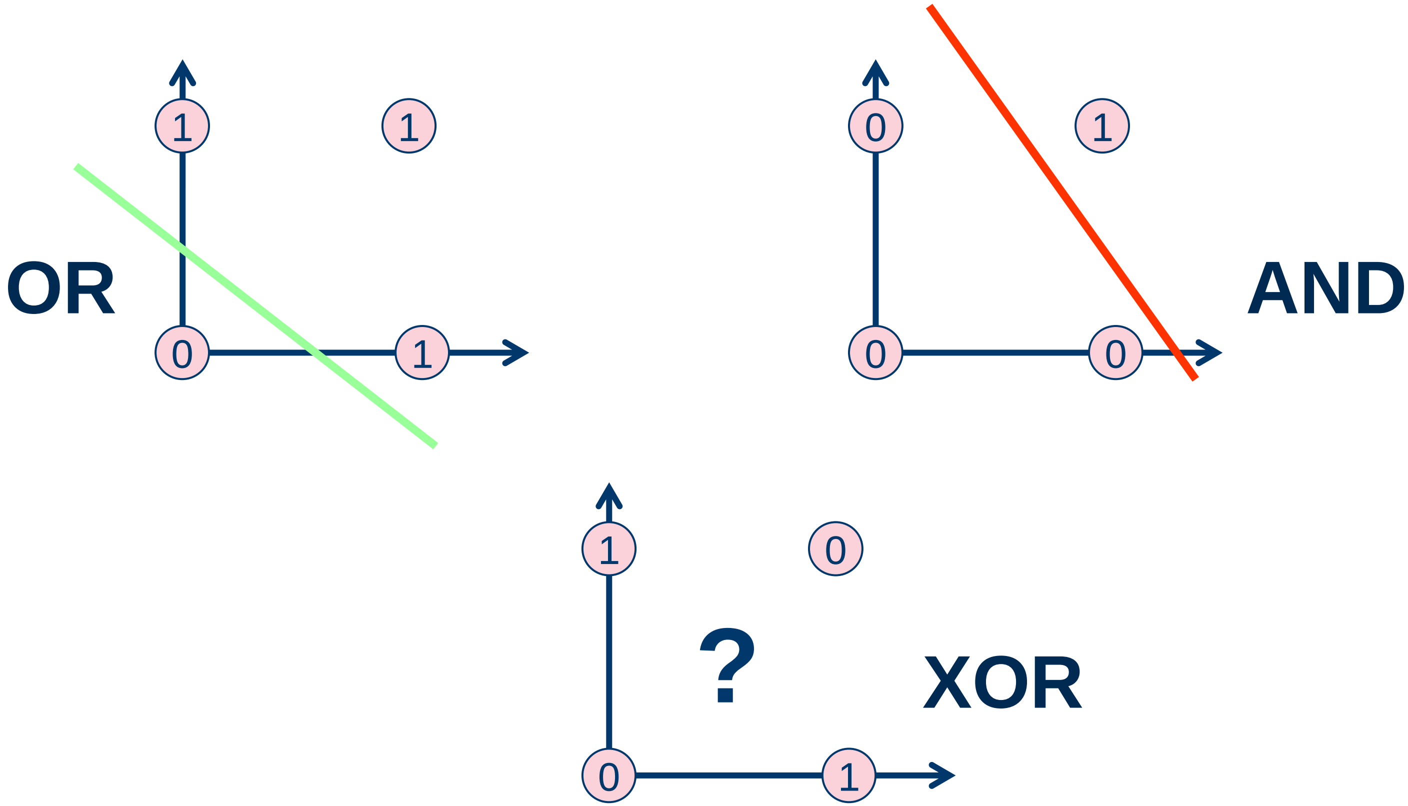
- A Perceptron with n input neurons can classify all examples from a dataset with n input variables and two classes correctly, if there exists a hyperplane separating the two classes

- Such classification problems are called **linearly separable**

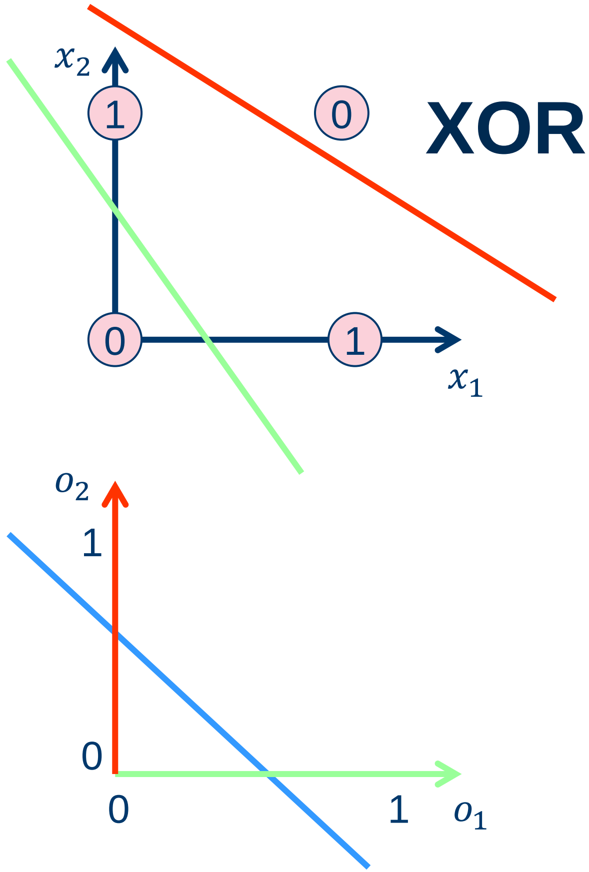
- A Perceptron can only solve linearly separable problems

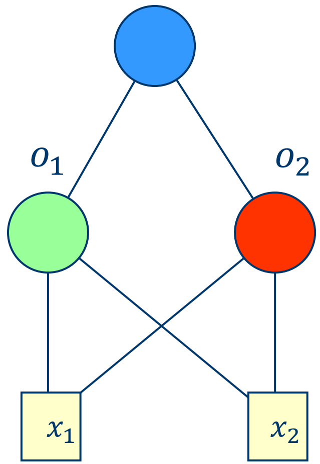
What can a single Perceptron do?

Example: The exclusive OR (XOR) defines a classification task which is not linearly separable.



The exclusive OR (XOR) can be solved adding one more layer





FeedForward Neural Networks

Multi-Layer Perceptron (MLP)

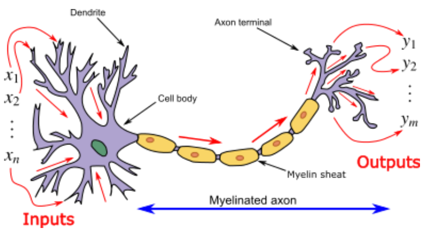
- A Perceptron with more than one layer is a **Multi-Layer Perceptron (MLP)**

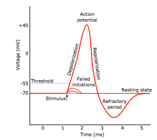
- A MLP is a neural network with:
  - an input layer,
  - one or more hidden layers, and
  - an output layer

- Connections exist only between neurons from one layer to the next layer

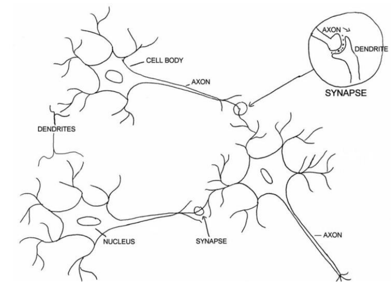
Bias as weight for a constant input

Biological Neuron

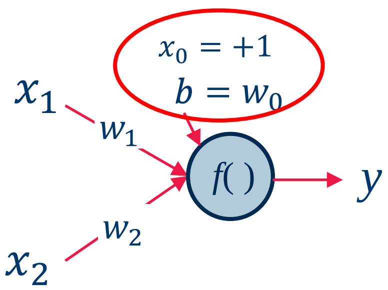




Biological Neural Networks



Artificial Neuron (**Perceptron**)



$$y = f(x_1 w_1 + x_2 w_2 + b)$$

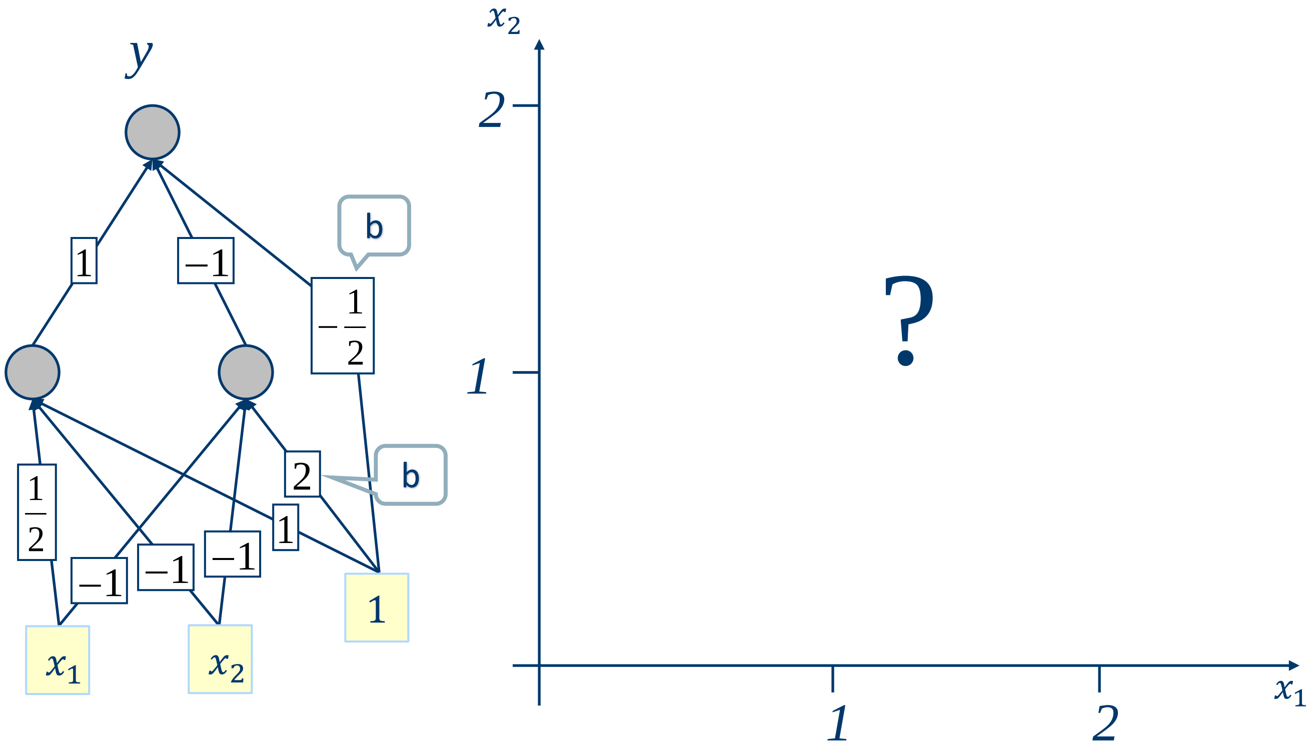
$$b = w_0$$

$$a(\boldsymbol{x}) = \sum_{i=0}^{n} x_i w_i$$

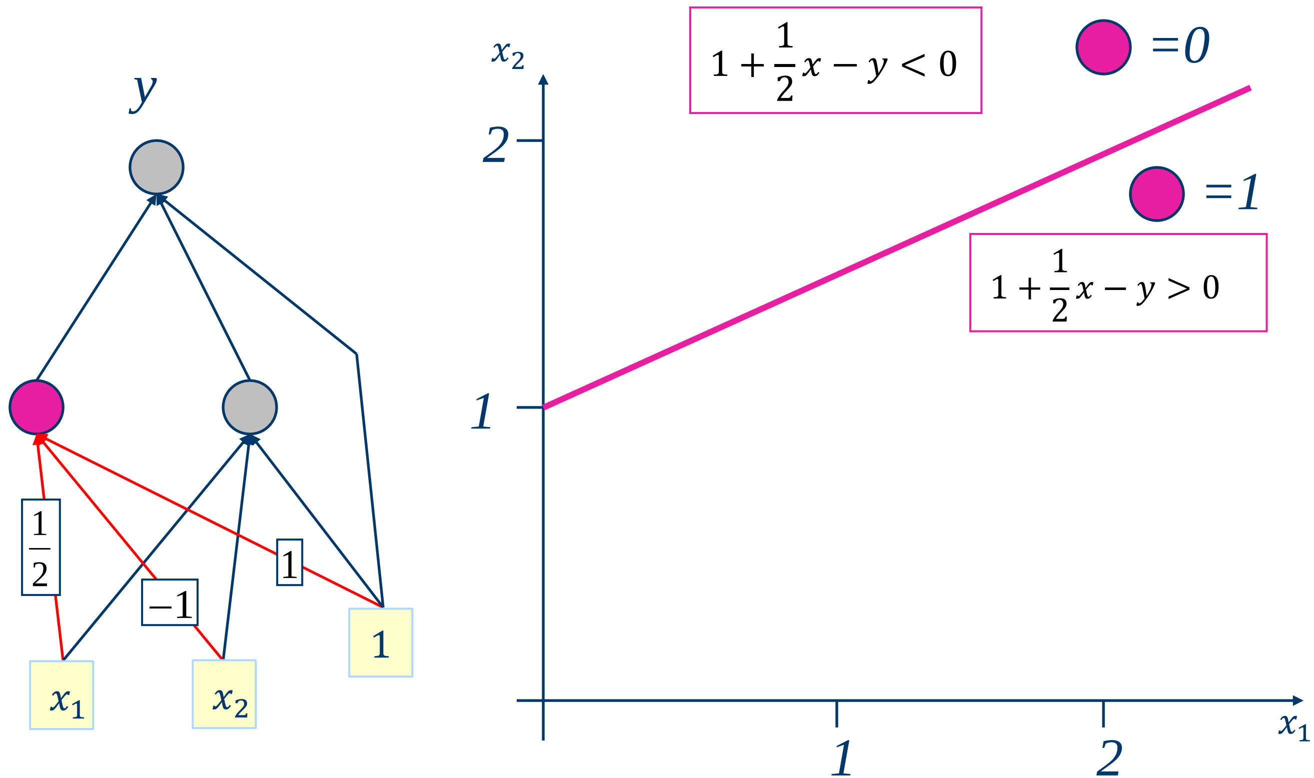
$$y(\boldsymbol{x}) = f(\sum_{i=0}^{n} x_i w_i)$$

- Neuron bias can be considered as a weight w_0 to a constant input $x_0 = +1$

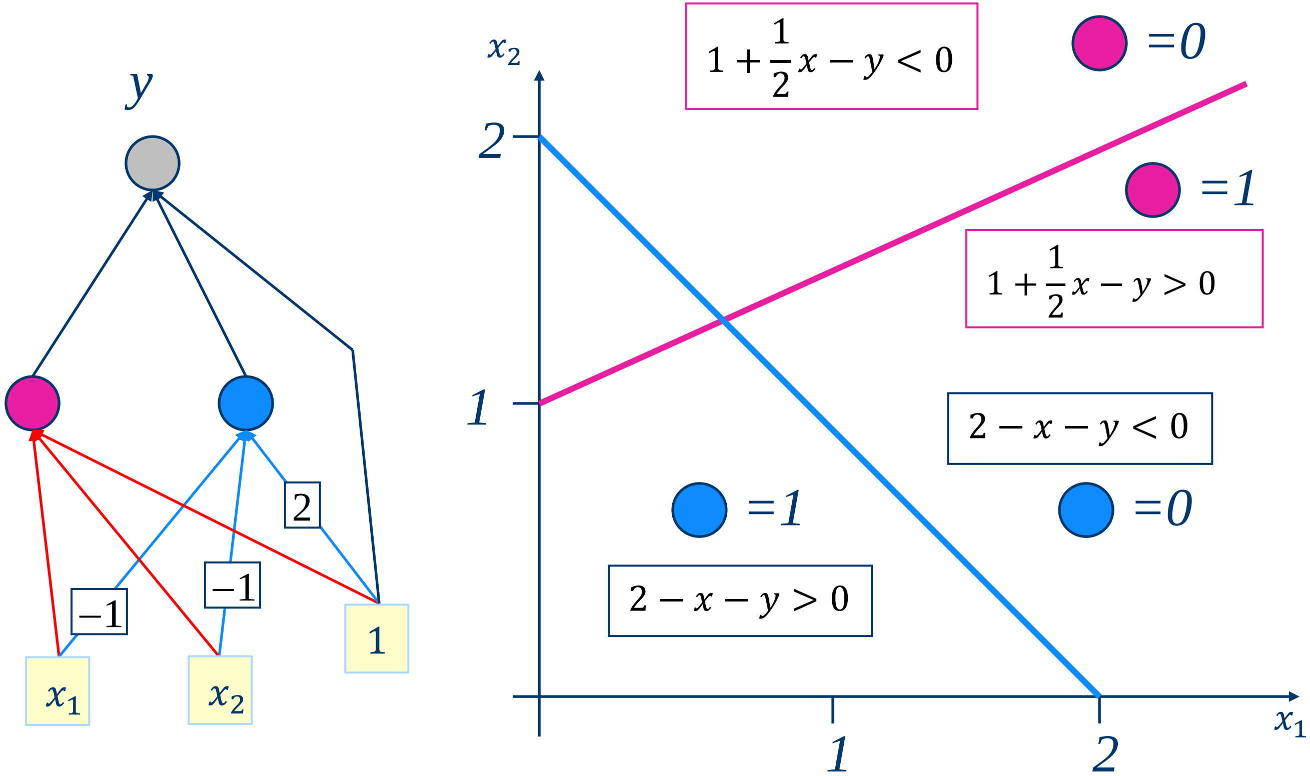
MLP: Example



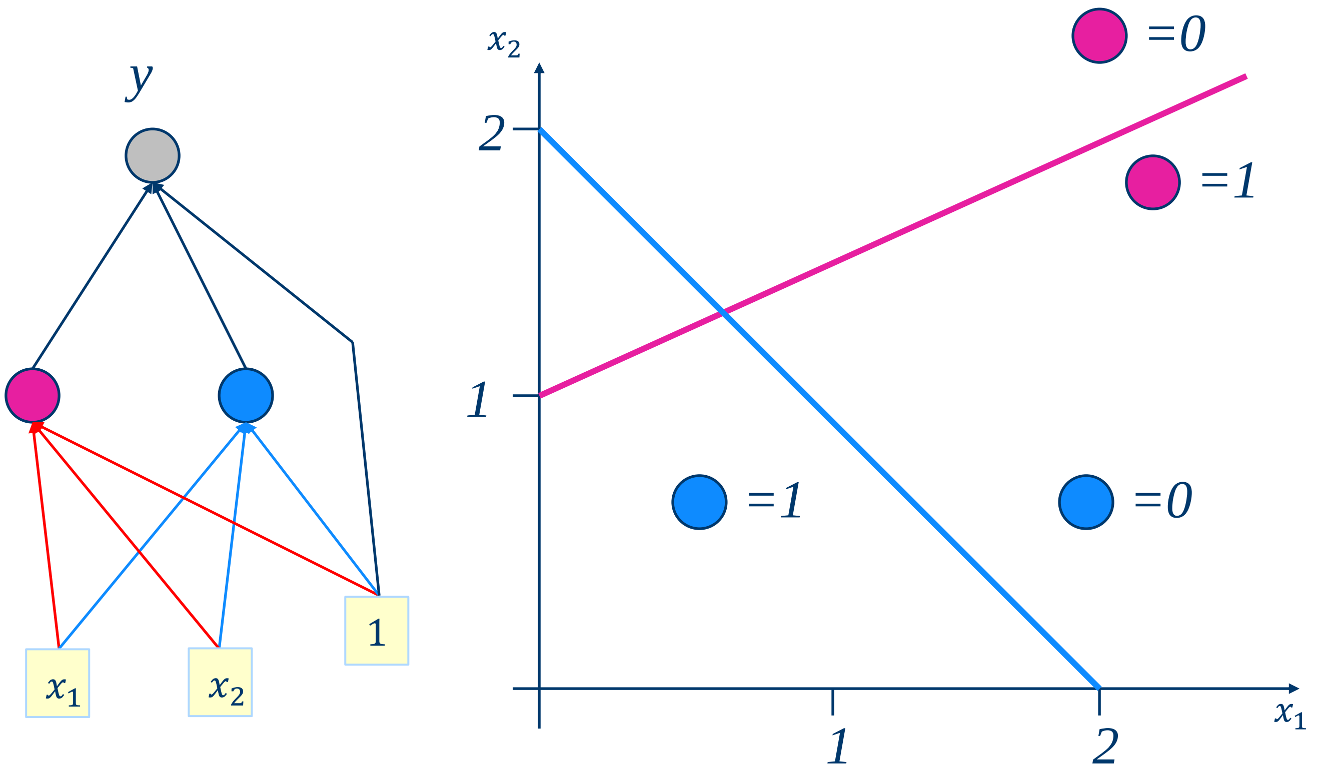
MLP: Example



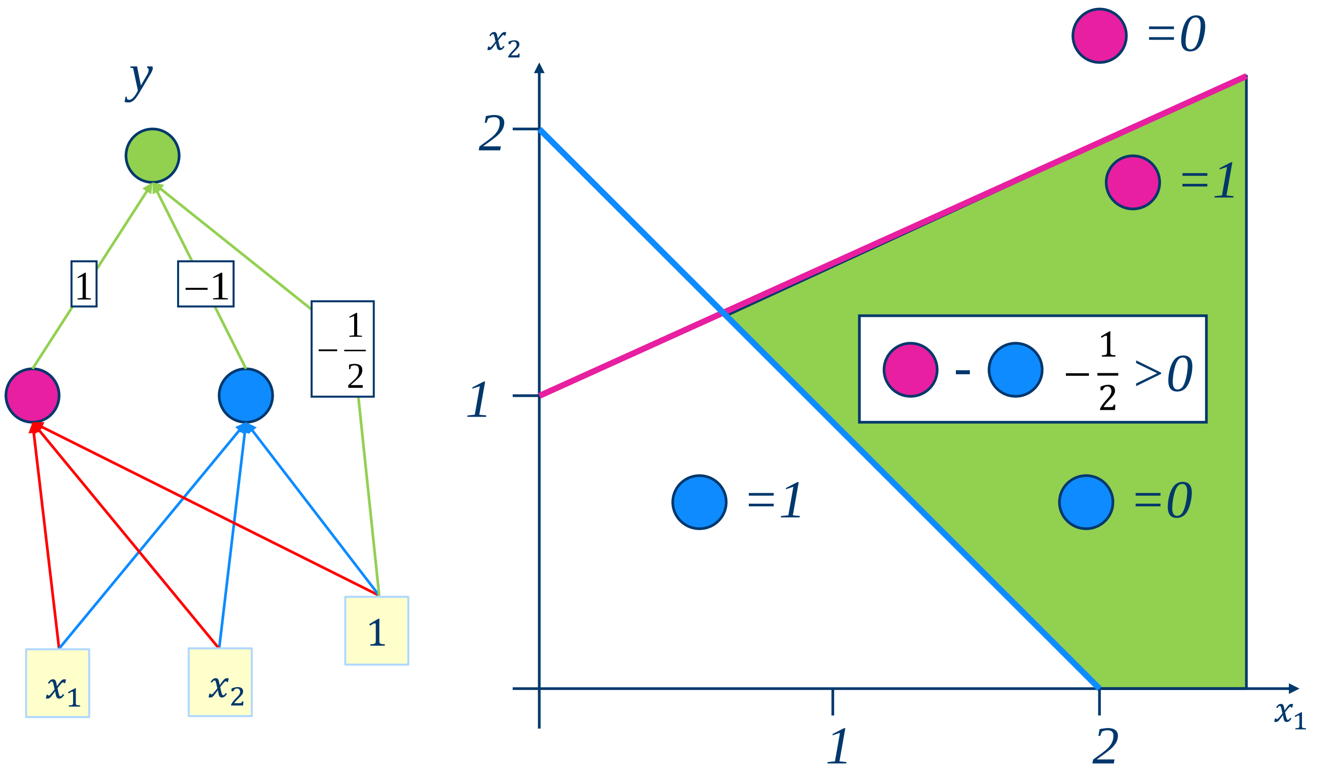
MLP: Example



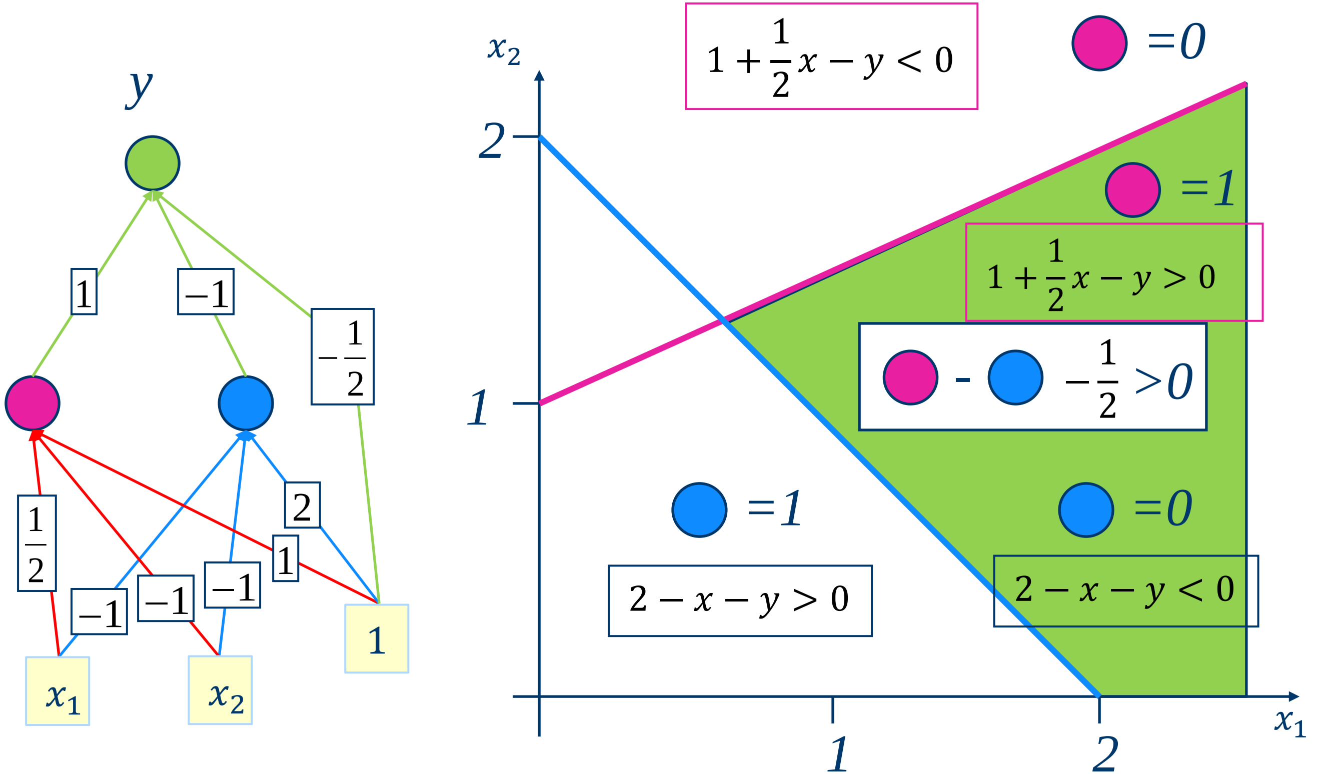
MLP: Example



MLP: Example

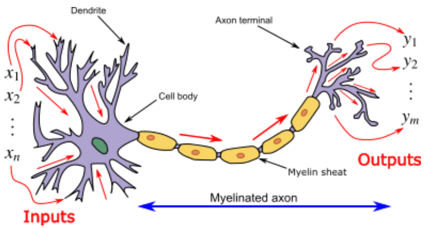


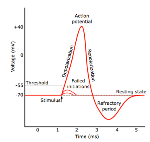
MLP: Example



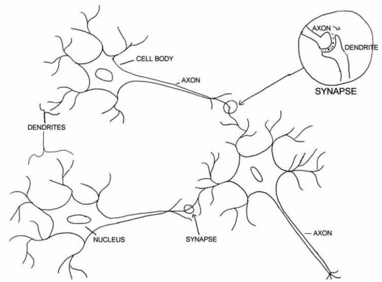
y
1
-1
$-\frac{1}{2}$
$\frac{1}{2}$
-1
-1
-1
2
1
x_1
x_2
1
x_2
2
1
x_1
1
2
$1 + \frac{1}{2}x - y < 0$
$=0$
$=1$
$1 + \frac{1}{2}x - y > 0$
$- \quad -\frac{1}{2} > 0$
$=1$
$=0$
$2 - x - y > 0$
$2 - x - y < 0$

Biological Neuron

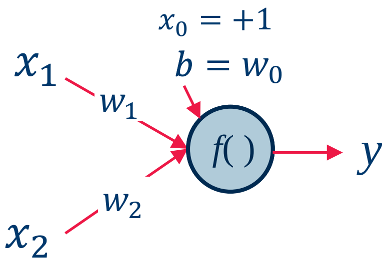




Biological Neural Networks



Artificial Neuron (**Perceptron**)



$$y = f(x_1 w_1 + x_2 w_2 + b)$$

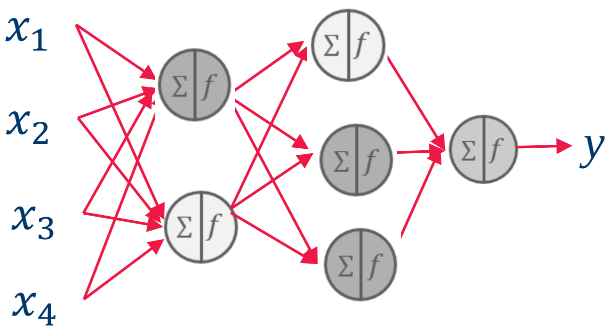
$$b = w_0$$

$$a(\boldsymbol{x}) = \sum_{i=0}^{n} x_i w_i$$

$$y(\boldsymbol{x}) = f(\sum_{i=0}^{n} x_i w_i)$$

Artificial Neural Networks

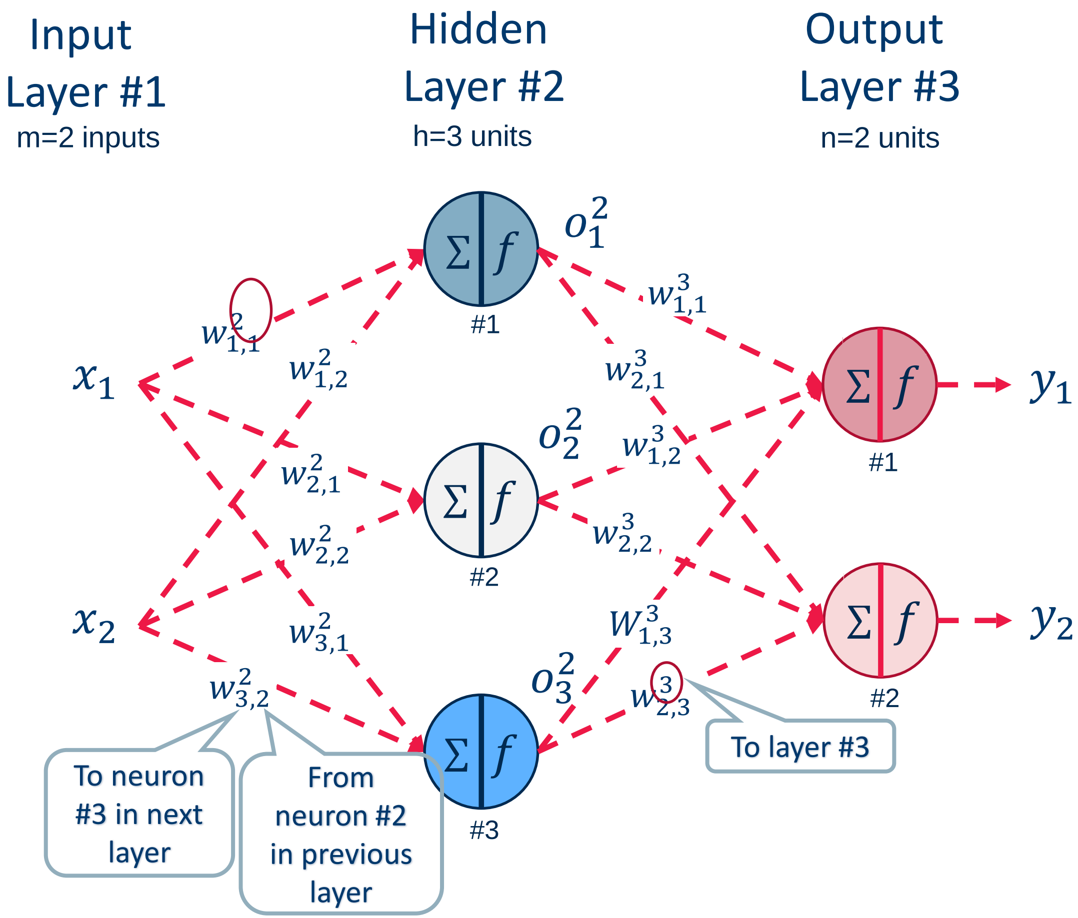
(Multilayer Perceptron, MLP)



- Let‘s see an example of a MLP
- 3 layers:
  - 1 input layer with *m=2 inputs*
  - 1 hidden layer with *h=3 hidden neurons*
  - 1 output layer with *n=2 output neurons*
- All **feed-forward** connections: from a neuron only to neurons in the next layer
- **Fully-connected**: that is each neuron in one layer is connected to all neurons in the next layers

fully connected feed forward neural networks

Feed-Forward Neural Networks (FFNN)



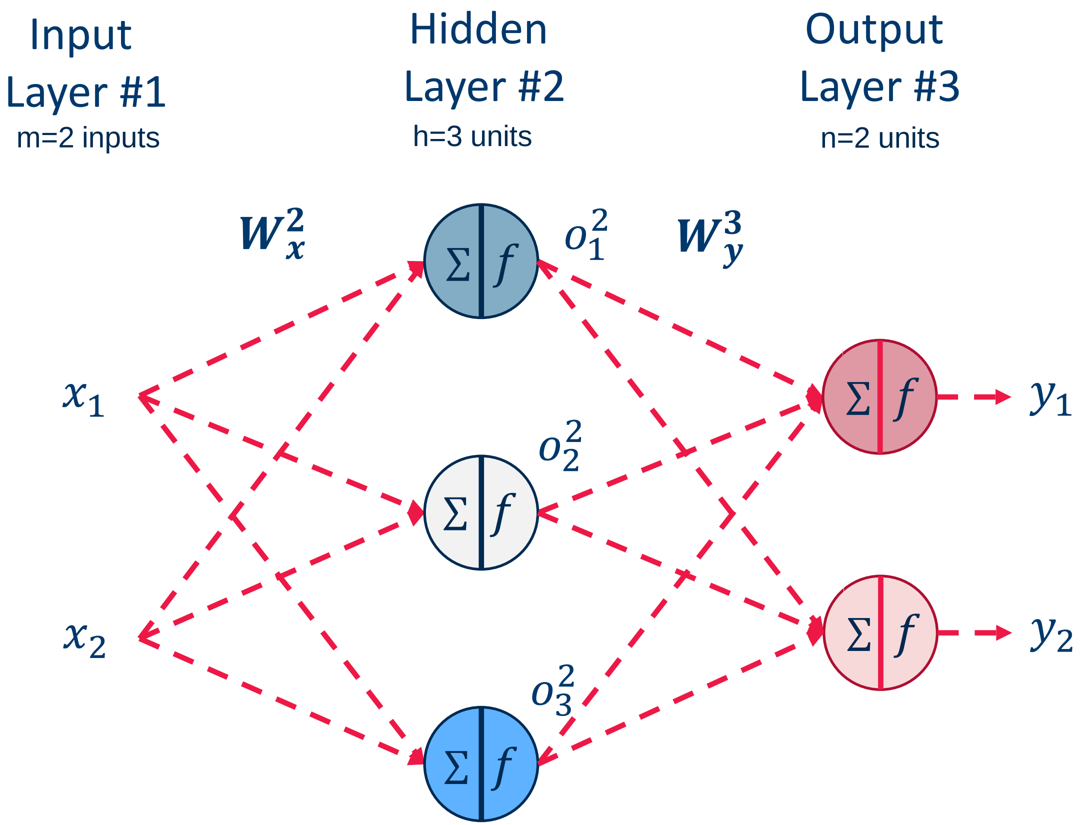
Forward pass:

$$o_j^2 = f(\sum_{i=1}^{n} w_{j,i}^2 x_i)$$

$$y_k = f(\sum_{j=1}^{h} w_{k,j}^3 o_j^2)$$

$$y_k = f(\sum_{j=1}^{h} w_{k,j}^3 f(\sum_{i=1}^{n} w_{j,i}^2 x_i))$$

Same with Matrix Notations



Forward pass:

$$\boldsymbol{o} = f(W_x^2 \boldsymbol{x})$$

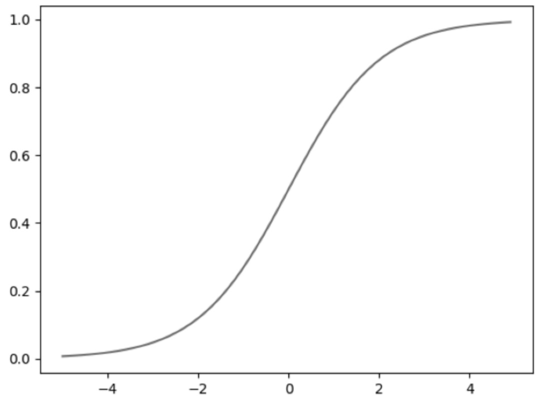
$$y = f(W_y^3 \boldsymbol{o})$$

$$y = f(W_y^3 f(W_x^2 \boldsymbol{x}))$$

$f(\ )$ = activation function

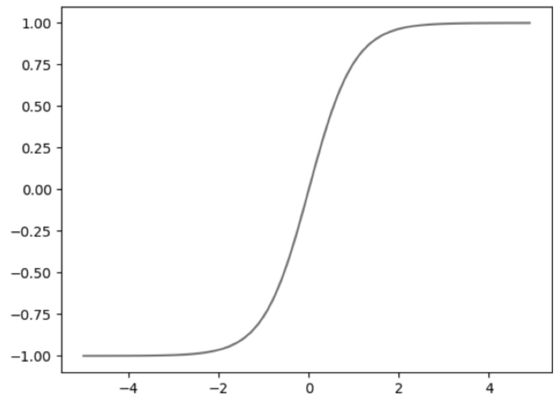
Frequently used activation functions

Sigmoid



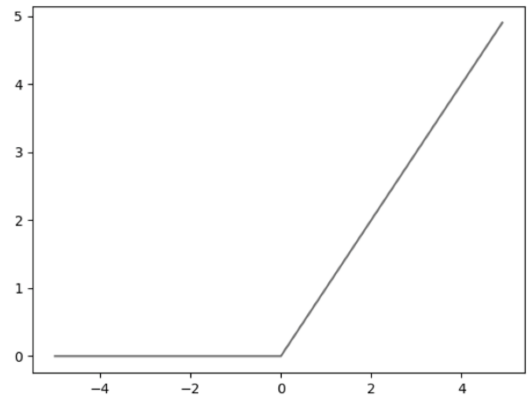
$$f(a) = \frac{1}{1 + e^{-ha}}$$

Tanh



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

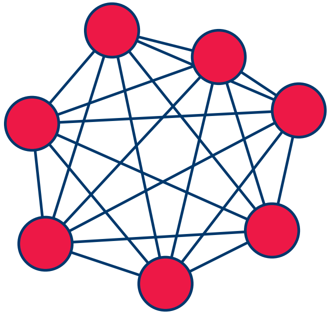
Rectified Linear Unit (ReLU)



$$f(a) = max\{0, ha\}$$

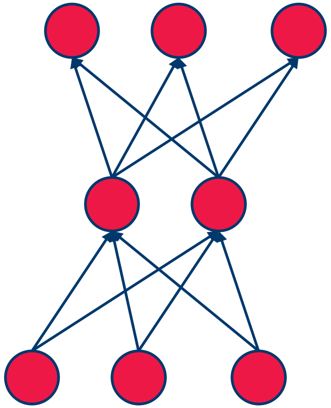
Other Neural Architectures

completely
connected



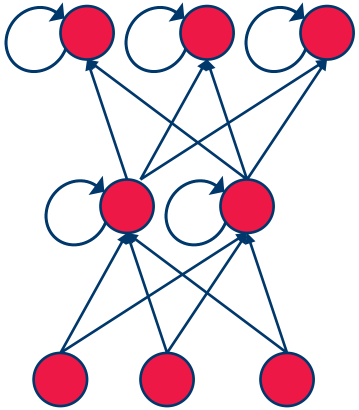
example:
- associative
  neural network
- Hopfield

feedforward
(directed, a-cyclic)



example:
- Multi Layer Perceptron
- Auto-encoder MLP

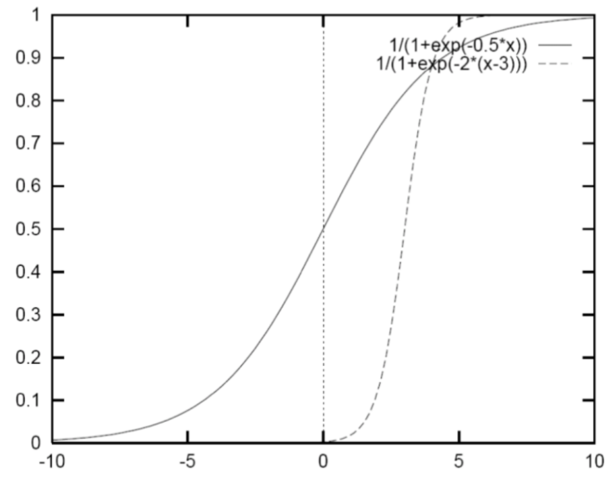
recurrent
(feedback connections)



example:
- Recurrent Neural
  Network (for time
  series recognition)

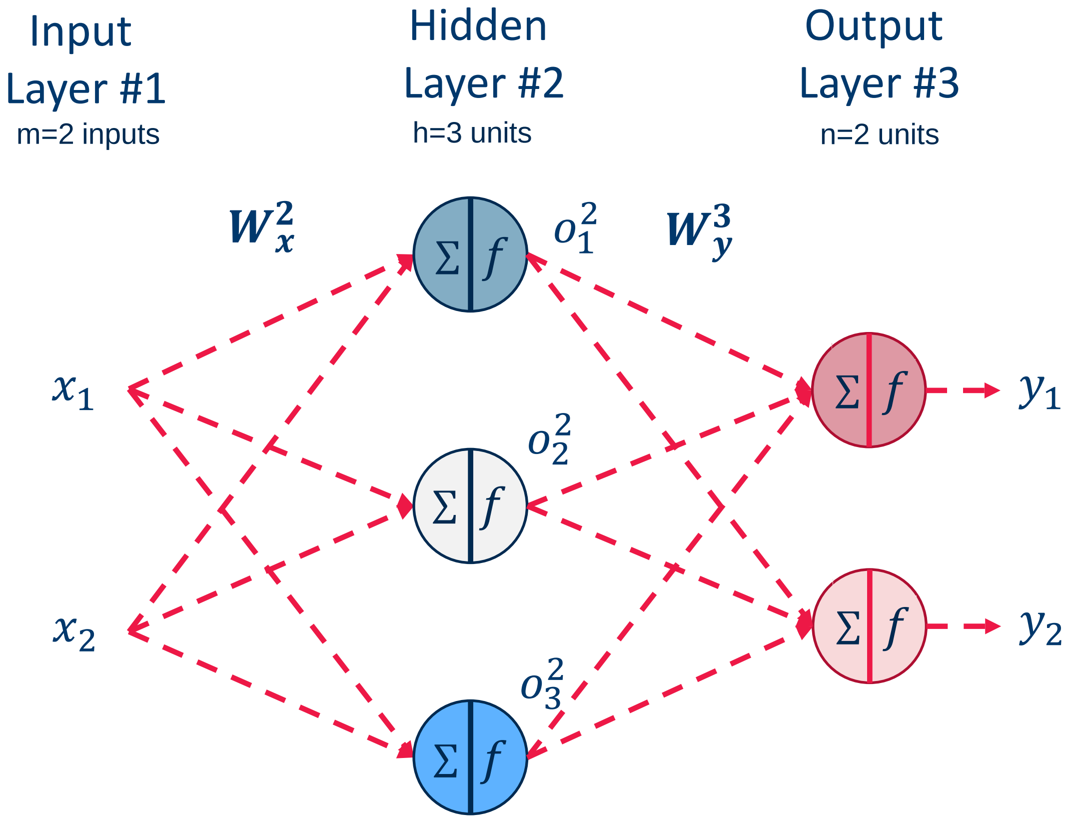
BackPropagation

- **Problem:** How do we automatically adjust the weights (and thresholds) for the neurons of the hidden layer?
- **Solution**: gradient descent
- Does not work with binary (non-differentiable) threshold function as activation function for the neurons.
- Activation function must be a **differentiable** function



- Teach (ensemble of) neuron(s) a desired input-output behavior.
- Show examples from the training set repeatedly
- Networks adjusts parameters to fit underlying function
  - topology
  - weights
  - internal functional parameters

Error Function



Number of output units

$$E = \frac{1}{2}\sum_{x\in T}\sum_{k=1}^{n}(y_k - t_k)^2$$

On the whole Training set

Network output value k

Target value k

Forward pass:

$\boldsymbol{o} = f(W_x^2\boldsymbol{x})$

$y = f(W_y^3\boldsymbol{o})$

- Adjust the weights based on the gradient descent technique, i.e. proportionally to the gradient of the error function

$$\boldsymbol{w}(t+1) = \boldsymbol{w}(t) + \Delta\boldsymbol{w}(t)$$

- with

$$\Delta\boldsymbol{w}(t) = -\eta\,\nabla(E(\boldsymbol{w}(t)))$$

- with $\eta > 0$ a non-zero learning rate

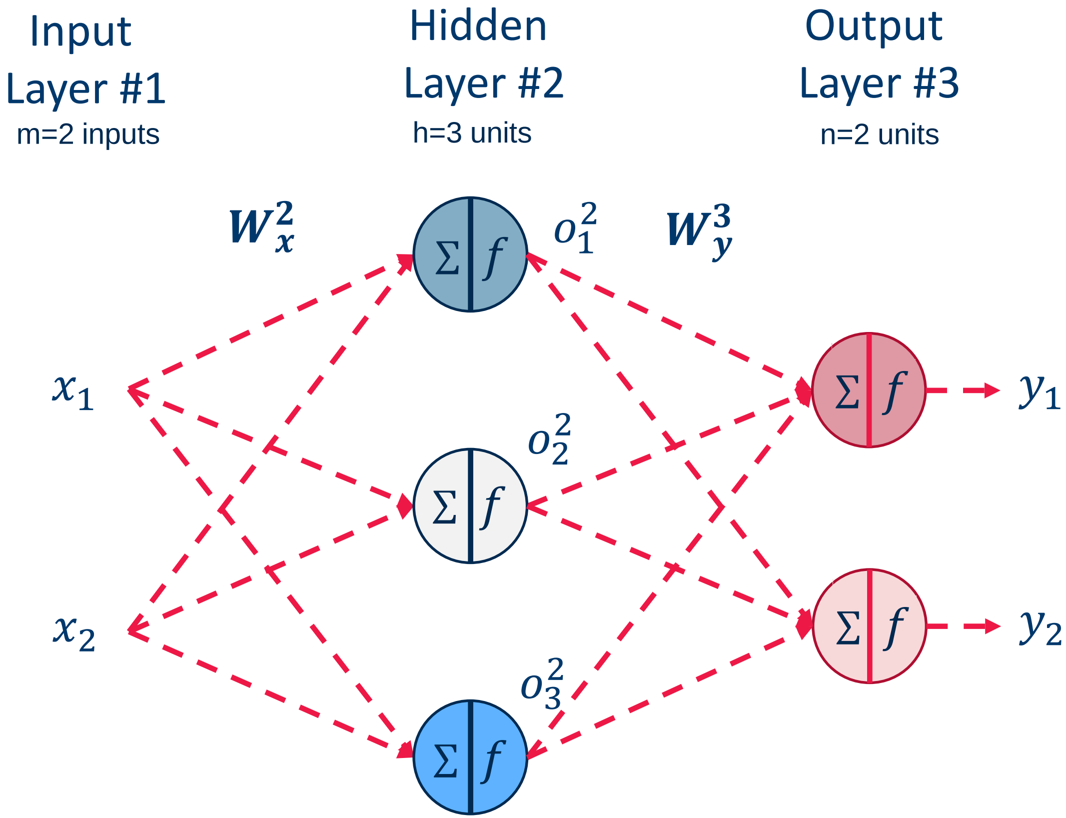
$$\Delta\boldsymbol{w}(t) = -\eta\,\nabla\Big(E\big(\boldsymbol{w}(t)\big)\Big) = -\eta\left(\frac{\partial E(\boldsymbol{w}(t))}{\partial w_1}, \dots, \frac{\partial E(\boldsymbol{w}(t))}{\partial w_m}\right)$$

- So we really need to determine only:

$$\Delta w_{u,v} = -\eta\frac{\partial E}{\partial w_{u,v}}$$

- For **each single weight** of the network.

Learning Rule from Gradient Descent



$$E = \frac{1}{2}\sum_{x \in T} \sum_{k=1}^{n} (y_k - t_k)^2$$

Gradient descent

$$\Delta w_{u,v} = -\eta \frac{\partial E}{\partial w_{u,v}}$$

For each weight in the output layer:

$$\Delta w_{ji}^{out} = -\eta \frac{\partial E(\mathbf{w}^{out})}{\partial w_{ji}^{out}}$$

To output neuron *j*
From hidden neuron *i*

$$\frac{\partial E}{\partial w_{ji}^{out}} = \frac{\partial \frac{1}{2} \sum_{x \in T} \sum_{k=1}^{n} (y_k - t_k)^2}{\partial w_{ji}^{out}} = \frac{1}{2} \sum_{x \in T} \frac{\partial (y_j - t_j)^2}{\partial w_{ji}^{out}} =$$

Net input to output neuron *j*

$$= \sum_{x \in T} (y_j - t_j) \frac{\partial y_j}{\partial w_{ji}^{out}} = \sum_{x \in T} (y_j - t_j) \frac{\partial f(net_j)}{\partial w_{ji}^{out}} =$$

$$= \sum_{x \in T} (y_j - t_j) \frac{\partial f(net_j)}{\partial net_j} \frac{\partial f(net_j)}{\partial w_{ji}^{out}} = \sum_{x \in T} (y_j - t_j) f'(net_j) \frac{\partial net_j}{\partial w_{ji}^{out}} =$$

For each weight in the output layer:

$$\Delta w_{ji}^{out} = -\eta \frac{\partial E(\boldsymbol{w}^{out})}{\partial w_{ji}^{out}}$$

Number of neurons in previous hidden layer

$$\frac{\partial E}{\partial w_{ji}^{out}} = \cdots = \sum_{x \in T} (y_j - t_j)\, f'(net_j) \frac{\partial \sum_{k'=1}^{h} w_{j,k'}^{out}\, o_{k'}^{hidden}}{\partial w_{ji}^{out}} =$$

$$= \sum_{x \in T} (y_j - t_j)\, f'(net_j)\, o_i^{hidden}$$

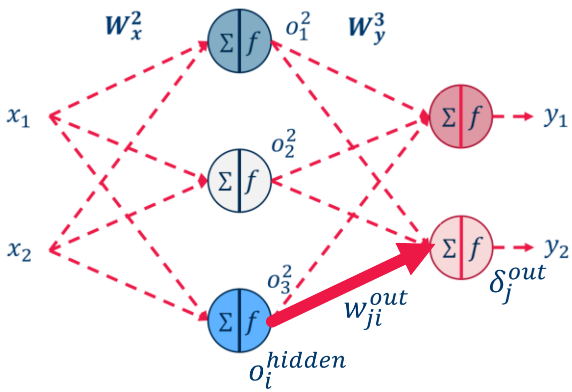
Output of neuron *i* in previous hidden layer

$$\Delta w_{ji}^{out} = -\eta \sum_{x \in T} \overbrace{(y_j - t_j)\, f'(net_j)}^{\delta_j^{out}}\; o_i^{hidden} = -\,\eta \sum_{x \in T} \delta_j^{out}\; o_i^{hidden}$$

– Final formula to update the weight w_{ji}^{out}, after all training samples in T have passed:

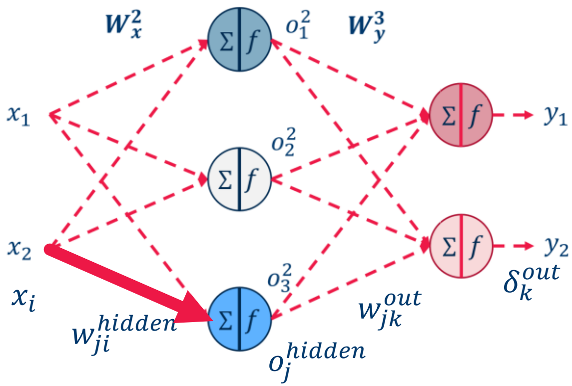
$$\Delta w_{ji}^{out} = -\eta \sum_{x \in T} \delta_j^{out} \, o_i^{hidden}$$

$$\delta_j^{out} = (y_j - t_j)\, f'(net_j)$$



- Now, where do we get the target values for the hidden neurons?
- Let's continue with gradient descent:
-

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



… some Calculations for the Hidden Layer …

$$\Delta w_{ji}^{hidden} = -\eta \frac{\partial \frac{1}{2}\sum_{x\in T}\sum_{k=1}^{n}(y_k - t_k)^2}{\partial w_{ij}^{hidden}} = -\frac{\eta}{2}\sum_{x\in T}\sum_{k=1}^{n}\frac{\partial(f(net_k^{out}) - t_k)^2}{\partial w_{ji}^{hidden}}$$

$$\ldots = -\frac{\eta}{2}\sum_{x\in T}\sum_{k=1}^{n} 2(f(net_k^{out}) - t_k)\frac{\partial(f(net_k^{out})-t_k)}{\partial w_{ji}^{hidden}}$$

$$\ldots = -\eta\sum_{x\in T}\sum_{k=1}^{n}(f(net_k^{out}) - t_k)f'(net_k^{out})\frac{\partial net_k^{out}}{\partial w_{ji}^{hidden}}$$

$$\ldots = -\eta\sum_{x\in T}\sum_{k=1}^{n}(f(net_k^{out}) - t_k)f'(net_k^{out})\frac{\partial \sum_{j'=1}^{h} w_{j'k}^{out} o_{j'}^{hidden}}{\partial w_{ji}^{hidden}}$$

If just one hidden layer

$$\ldots = -\eta\sum_{x\in T}\sum_{k=1}^{n}(f(net_k^{out}) - t_k)f'(net_k^{out})\frac{\partial \sum_{j'=1}^{h} w_{j'k}^{out} f\left(\sum_{i'=1}^{m} w_{i'j'}^{hidden} \cdot x_{i'}\right)}{\partial w_{ji}^{hidden}}$$

$$\ldots = -\eta\sum_{x\in T}\sum_{k=1}^{n} \delta_k^{out} \frac{\partial \sum_{j'=1}^{h} w_{j'k}^{out} f\left(\sum_{i'=1}^{m} w_{i'j'}^{hidden} \cdot x_{i'}\right)}{\partial w_{ji}^{hidden}}$$

$$\ldots = -\eta\sum_{x\in T}\sum_{k=1}^{n} \delta_k^{out} \; w_{jk}^{out} \frac{\partial f\left(\sum_{i'=1}^{m} w_{i'j}^{hidden} \cdot x_{i'}\right)}{\partial w_{ji}^{hidden}}$$

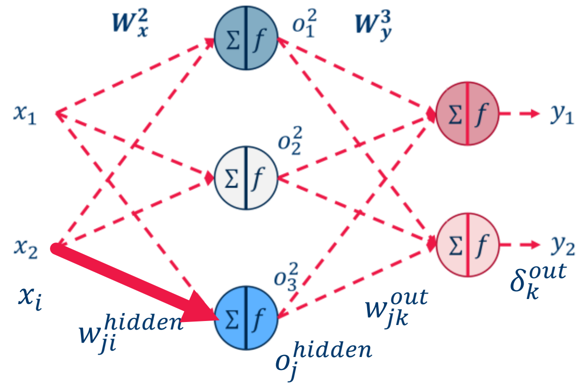
… some Calculations for the Hidden Layer …

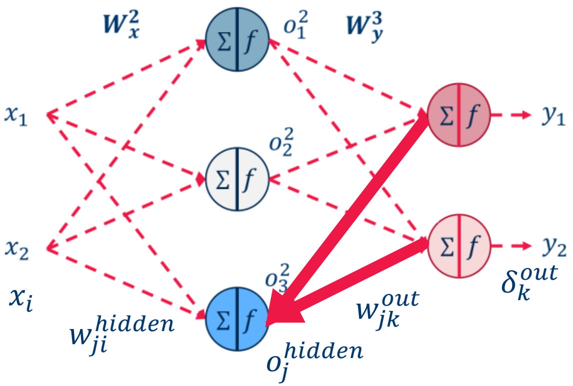
$$\ldots = -\eta \sum_{x\in T} \sum_{k=1}^{n} \delta_k^{out} \quad w_{jk}^{out} \frac{\partial f\left(\sum_{i'=1}^{m} w_{i'j}^{hidden} \cdot x_{i'}\right)}{\partial w_{ji}^{hidden}}$$

$$\ldots = -\eta \sum_{x\in T} \sum_{k=1}^{n} \delta_k^{out} \quad w_{jk}^{out} \; f'\left(\sum_{i'=1}^{m} w_{i'j}^{hidden} \cdot x_{i'}\right) \cdot x_i$$

$$\ldots = -\eta \sum_{x\in T} \sum_{k=1}^{n} \delta_k^{out} \quad w_{jk}^{out} \; f'\left(net_j^{hidden}\right) \cdot x_i$$

$$\ldots = -\eta \sum_{x\in T} \qquad \delta_j^{hidden} \qquad \cdot x_i$$





As for weights to output neurons

$$\Delta w_{ji}^{hidden} = -\eta \sum_{x\in T} \delta_j^{hidden} x_i$$

δ_k^{out} is back-propagated from output to input

$$\delta_j^{hidden} = \sum_{k=1}^{n} \delta_k^{out} \; w_{jk}^{out} \; f'\left(net_j^{hidden}\right)$$

Error BackPropagation or Generalized Delta Rule

Update of weights to the output layer:

$$\Delta w_{ji}^{out} = -\eta \sum_{x \in T} \delta_j^{out} \, o_i^{hidden}$$

with

$$\delta_j^{out} = \left(y_j - t_j\right) f'\left(net_j\right)$$

And update of weghts to hidden layers:

Previous layer

$$\Delta w_{ji}^{l} = -\,\eta \sum_{x \in T} \delta_j^{l} \, o_i^{l-1}$$

Current hidden layer

With

$$\delta_j^{hidden} = \sum_{k=1}^{n} \delta_k^{l+1} w_{jk}^{l+1} \, f'\left(net_j^l\right)$$

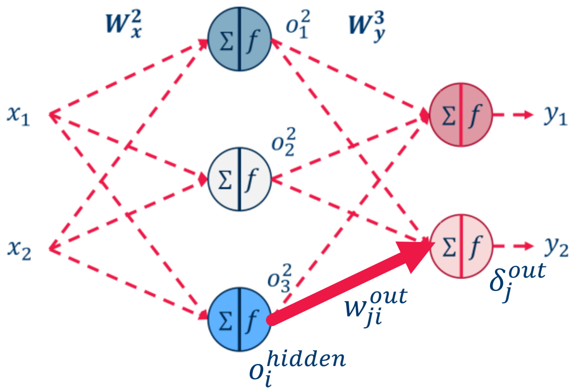
Next layer

Recursive equation for updating the weights:

Update the weights to the neuron in the output layer first and then go back layer by layer and update the corresponding weights.

Update formula for weights after each sample in T

– Final formula to update the weight w_{ji}^{out}, after **one single** training sample in T :



$$E = \frac{1}{2}\sum_{k=1}^{n}(y_k - t_k)^2$$

Error function after each training sample

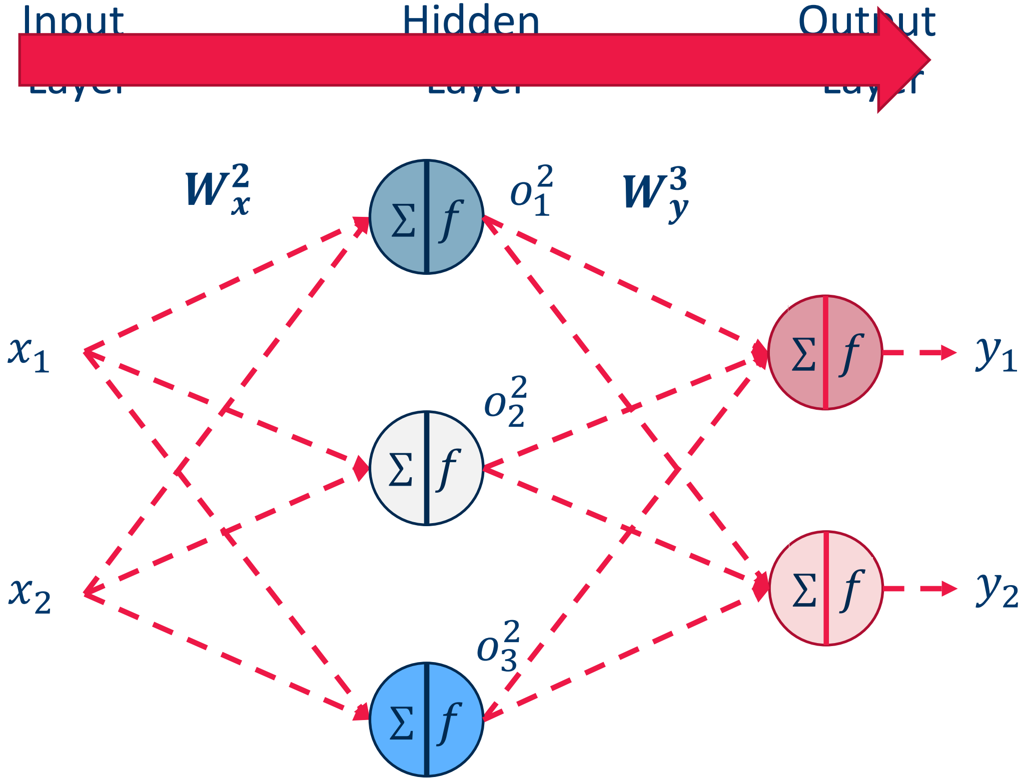
$$\Delta w_{ji}^{l} = -\eta\, \delta_j^{l} \;\; o_i^{l-1}$$

No sum on training set T

$$\delta_j^l = \begin{cases} (y_j - t_j)\; f'(net_j^l) & l = output\ layer \\ \sum_{k=1}^{n} \delta_k^{l+1}\; w_{jk}^{l+1}\; f'(net_j^l) & l = hidden\ layer \end{cases}$$

Same as before

Step 1. Forward Pass

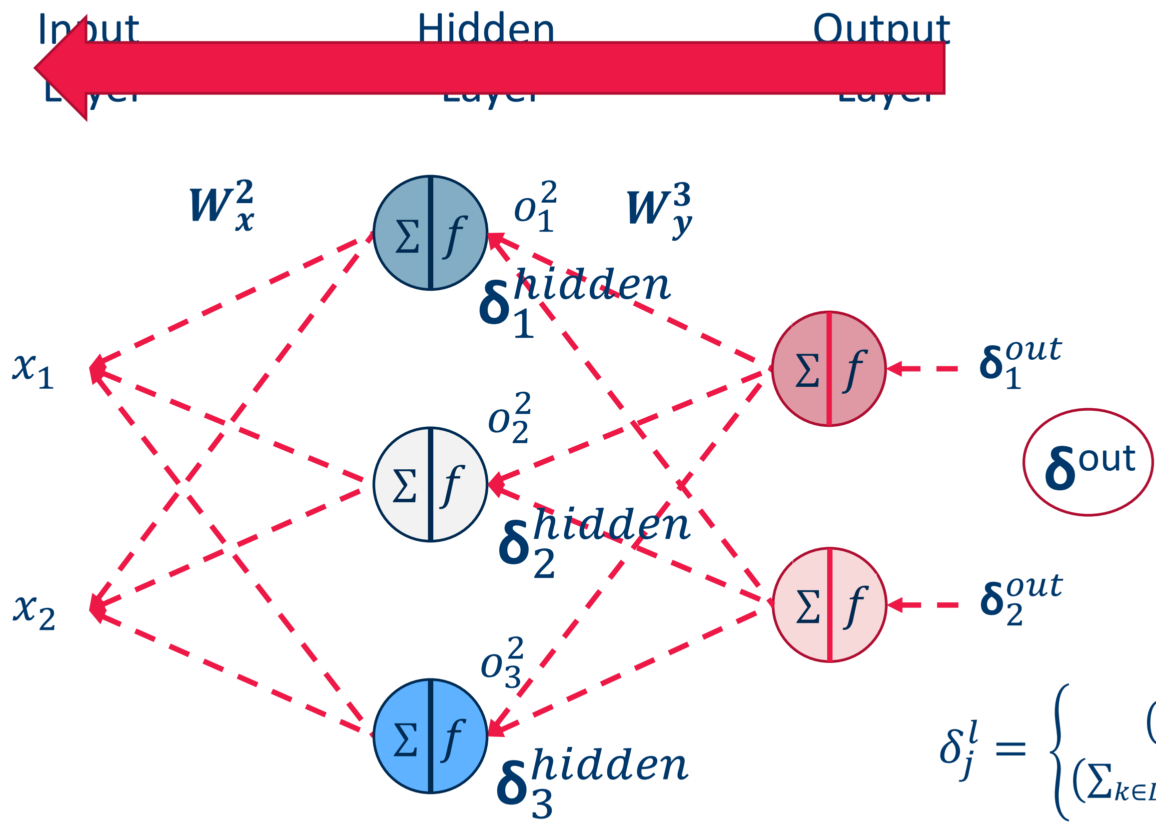


1. Forward pass:

$$o_j^2 = f\left(\sum_{i=1}^{n} w_{ji}^2\, x_i\right)$$

$$y_k = f\left(\sum_{j=1}^{h} w_{kj}^3\, o_j^2\right)$$

Step 2. Backward Pass - δ

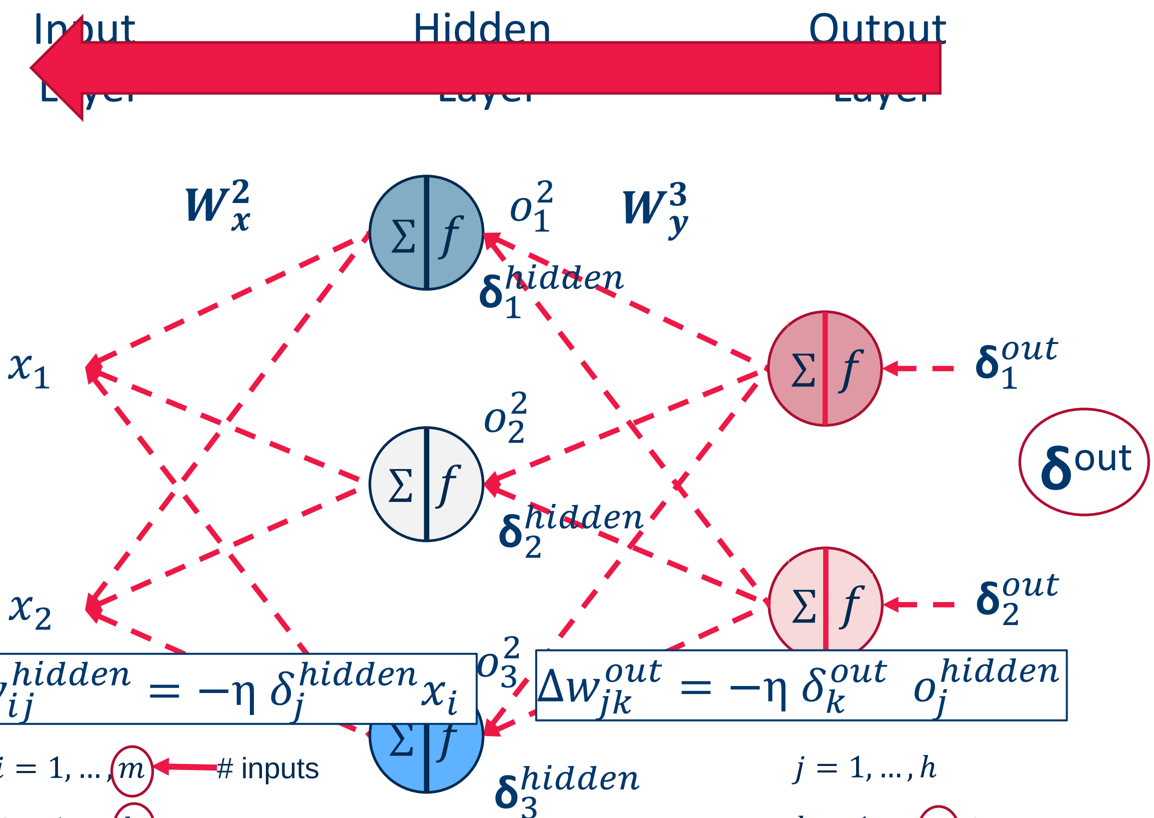


This is $o_j\,(1-o_j)$for a sigmoid activation function

2. Backward pass:

$$\delta_j^l = \begin{cases} (y_j - t_j)\boxed{f'(net_j)} & l = last\ layer \\ \left(\sum_{k\in L} w_{jk}\delta_k\right) f'(net_j) & l = hidden\ layer \end{cases}$$

Step 3: Learning after **each** training pattern



$$\Delta w_{ij}^{hidden} = -\eta\ \delta_j^{hidden} x_i$$

$i = 1, \ldots, m$ # inputs

$j = 1, \ldots, h$ # hidden units

$$\Delta w_{jk}^{out} = -\eta\ \delta_k^{out}\ \ o_j^{hidden}$$

$j = 1, \ldots, h$

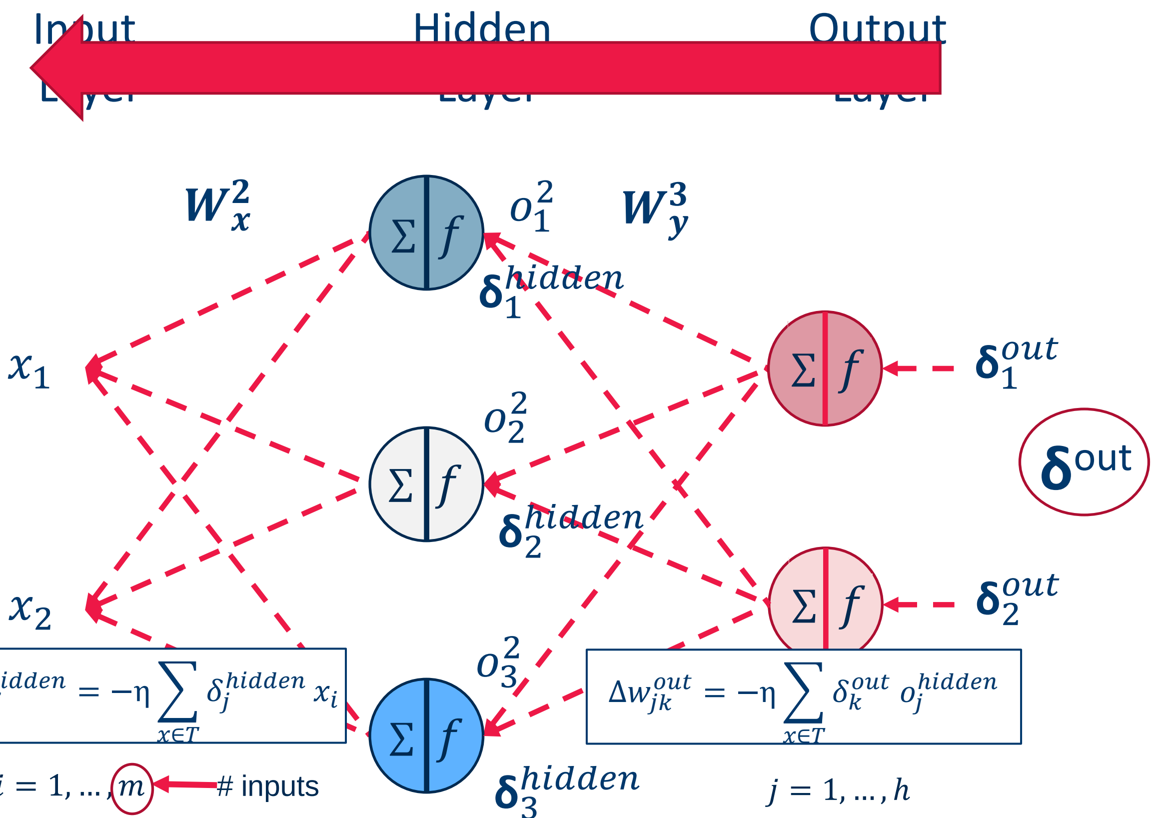
$k = 1, \ldots, n$ # outputs

3. Weight Update after **each** training example:

$$\Delta w_{ji}^{l} = -\eta\ \delta_j^{l}\ o_i^{l-1}$$

$$w_{ji}^{l}(t+1) = w_{ji}^{l}(t) + \Delta w_{ji}^{l}$$

Step 3: Learning after **all** training patterns



$$\Delta w_{ij}^{hidden} = -\eta \sum_{x \in T} \delta_j^{hidden}\, x_i$$

$i = 1, \ldots, m$ # inputs

$j = 1, \ldots, h$ # hidden units

$$\Delta w_{jk}^{out} = -\eta \sum_{x \in T} \delta_k^{out}\, o_j^{hidden}$$

$j = 1, \ldots, h$

$k = 1, \ldots, n$ # outputs

Learning rate

4. Weight Update after **all** training examples:

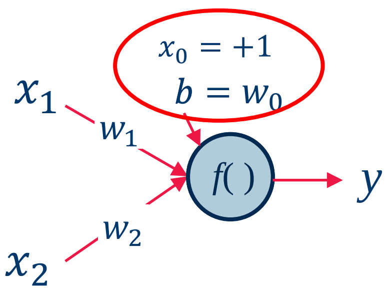
$$\Delta w_{ji}^{l} = -\boxed{\eta} \sum_{x \in T} \delta_j^{l}\, o_i^{l-1}$$

$$w_{ji}^{l}(t+1) = w_{ji}^{l}(t) + \Delta w_{ji}^{l}$$

Training the bias values

- Remember?
- Bias values can be considered as special weights to constant inputs +1
- Therefore biases are trained together with all other weights

Artificial Neuron (**Perceptron**)



$$y = f(x_1 w_1 + x_2 w_2 + b)$$

$$b = w_0$$

$$a(\boldsymbol{x}) = \sum_{i=0}^{n} x_i w_i$$

$$y(\boldsymbol{x}) = f\left(\sum_{i=0}^{n} x_i w_i\right)$$

- Neuron bias can be considered as a weight w_0 to a constant input $x_0 = +1$

Training: Batch vs. Online

- **Batch Training**: Weight update after **all** training patterns
  - correct
  - computationally expensive and slow
  - works with reasonably large learning rates (fewer updates!)

- **Online Training**: Weight update after **each** training pattern
  - Approximation
    - can (in theory) run into oscillations
  - faster (fewer epochs!)
  - smaller learning rates necessary

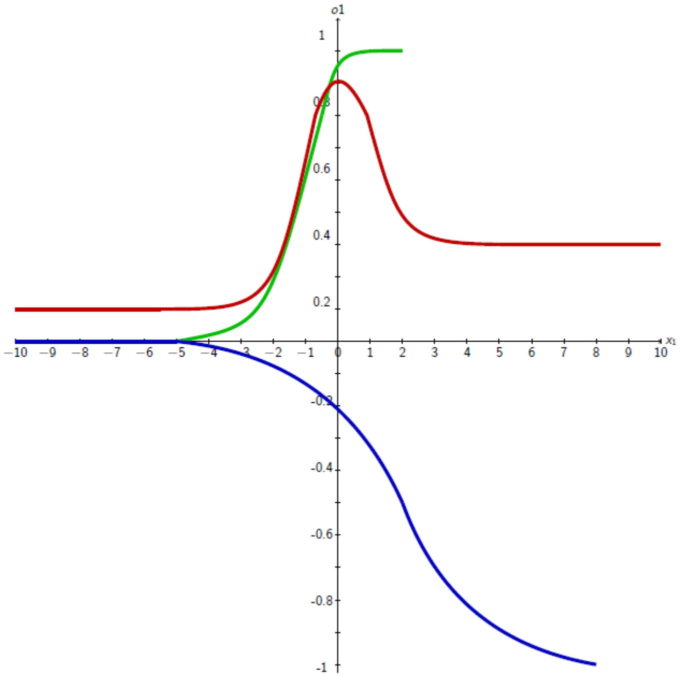
- The Sigmoid Activation Function has one really nice property (among others):

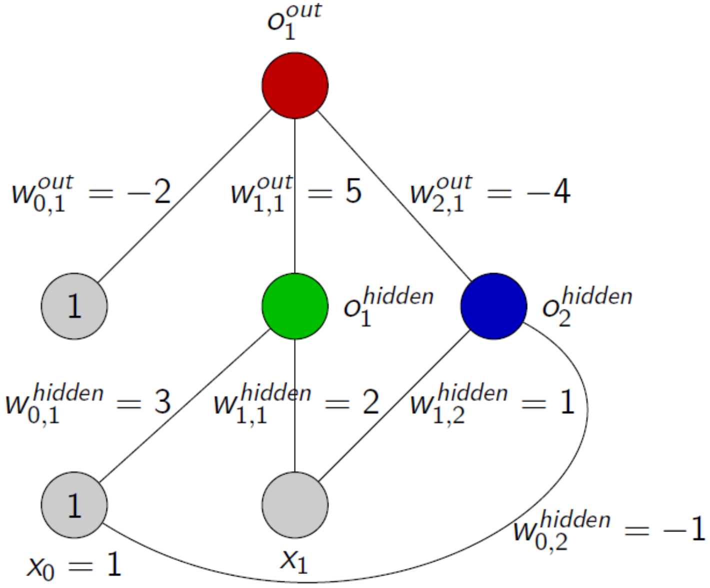
$$f'(a) = \frac{\partial}{\partial a}\left(\frac{1}{1+e^{-ha}}\right) = -\frac{e^{-ha}}{(1+e^{-ha})^2} = \ \dots = f(a)\big(1-f(a)\big)$$

- We can compute the derivative *f'*(*a*) simply from *f*(*a*) !

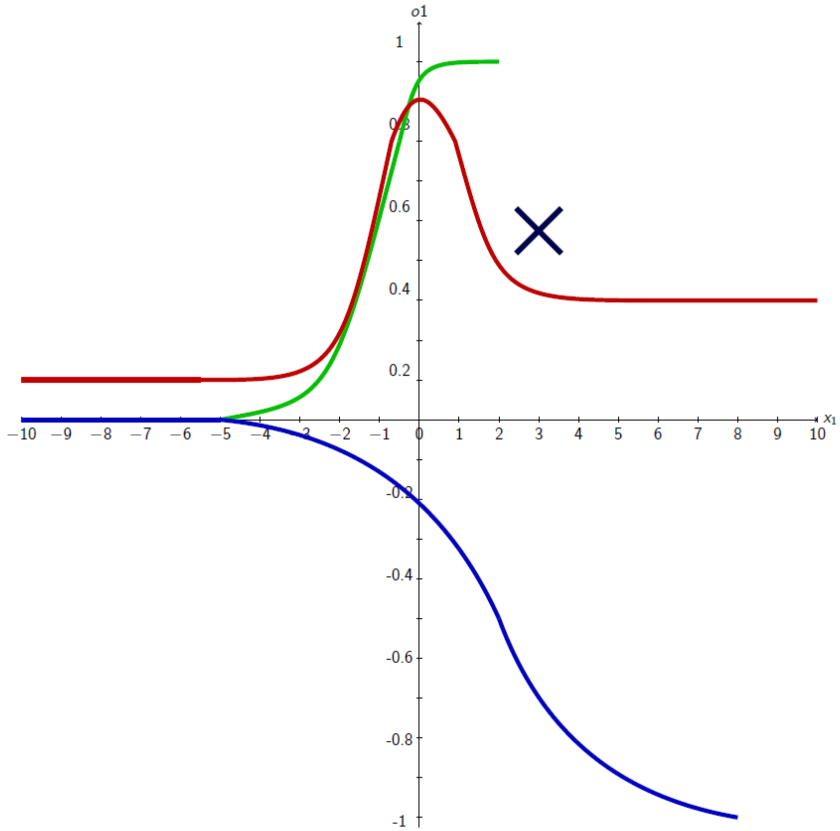
BackPropagation Example

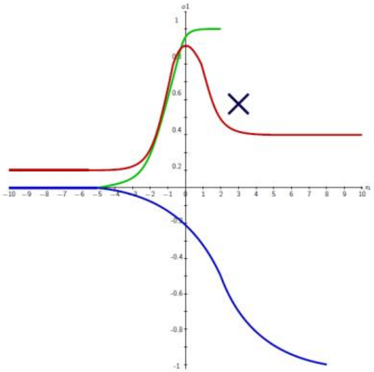
1D Example

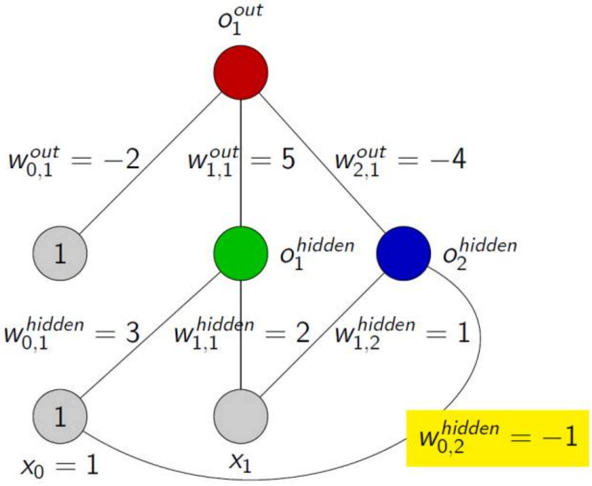

o1
1
0.8
0.6
0.4
0.2
-0.2
-0.4
-0.6
-0.8
-1
−10 −9 −8 −7 −6 −5 −4 −3 −2 −1 0 1 2 3 4 5 6 7 8 9 10
x1


o_1^{out}
$w_{0,1}^{out} = -2$
$w_{1,1}^{out} = 5$
$w_{2,1}^{out} = -4$
1
o_1^{hidden}
o_2^{hidden}
$w_{0,1}^{hidden} = 3$
$w_{1,1}^{hidden} = 2$
$w_{1,2}^{hidden} = 1$
1
$w_{0,2}^{hidden} = -1$
$x_0 = 1$
x_1

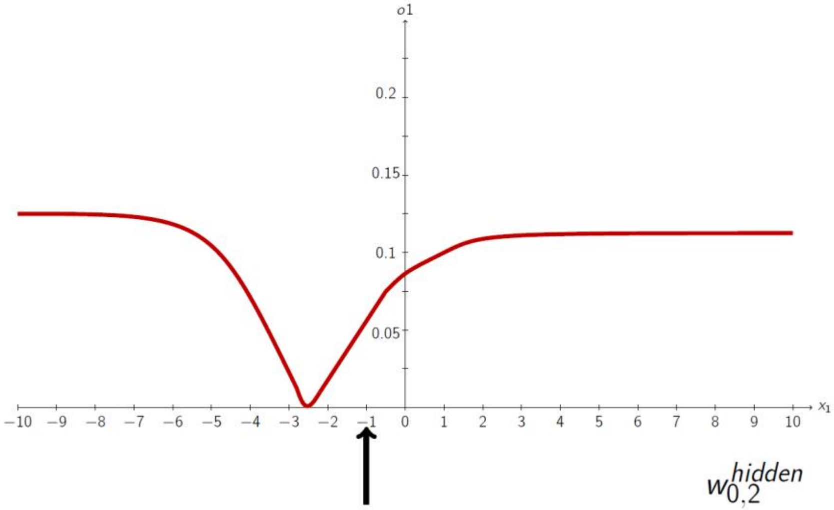
Training Pattern: $x = 3,\ y(x) = 0.6$



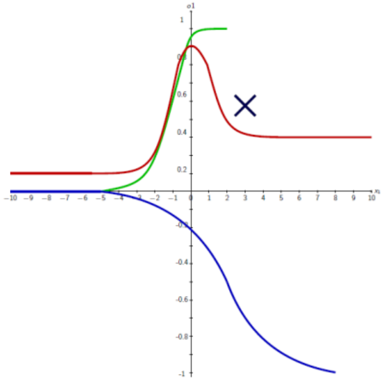


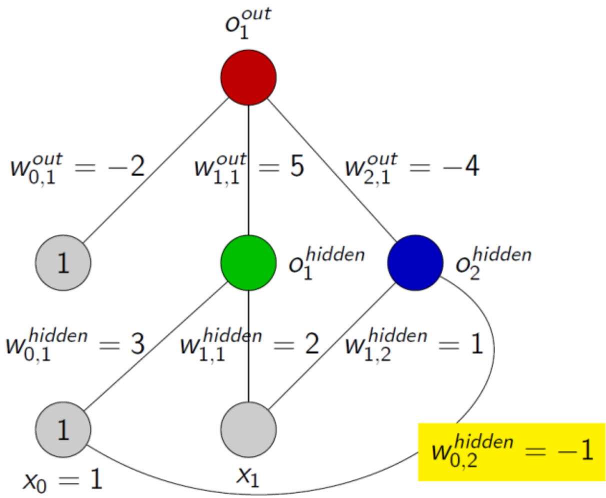


$$E\left(w_{0,2}^{\text{hidden}}\right)$$

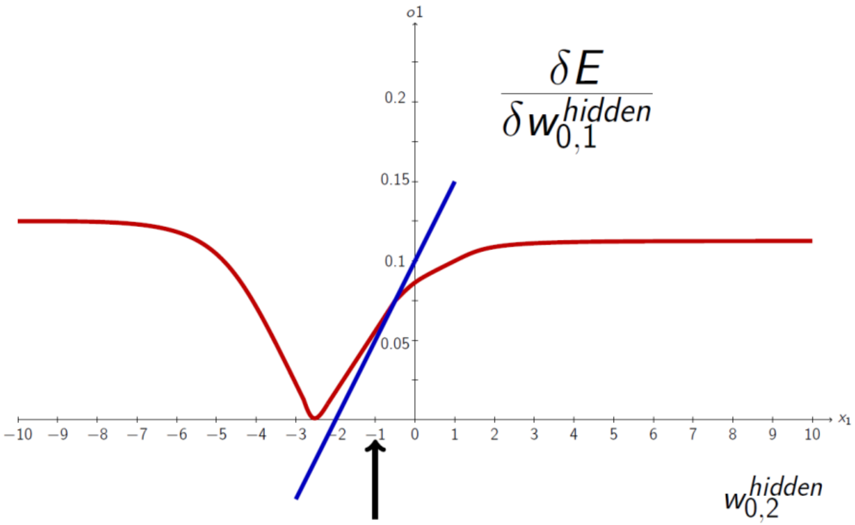


BackPropagation Example

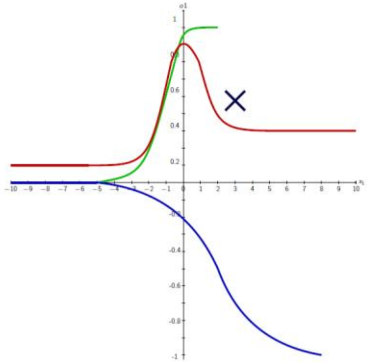


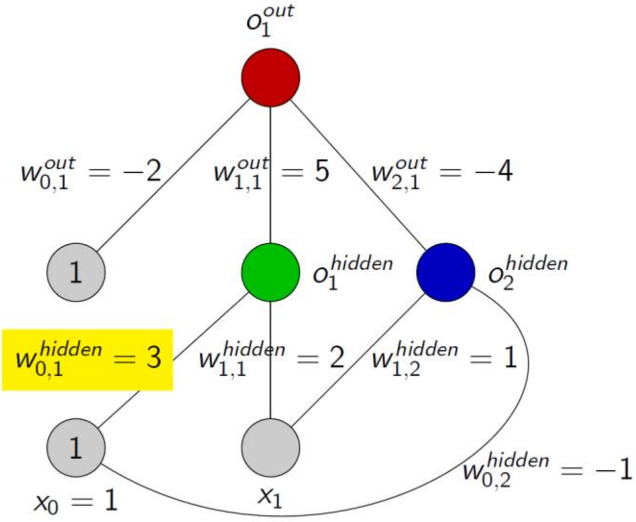


$$E\left(w_{0,2}^{\text{hidden}}\right)$$

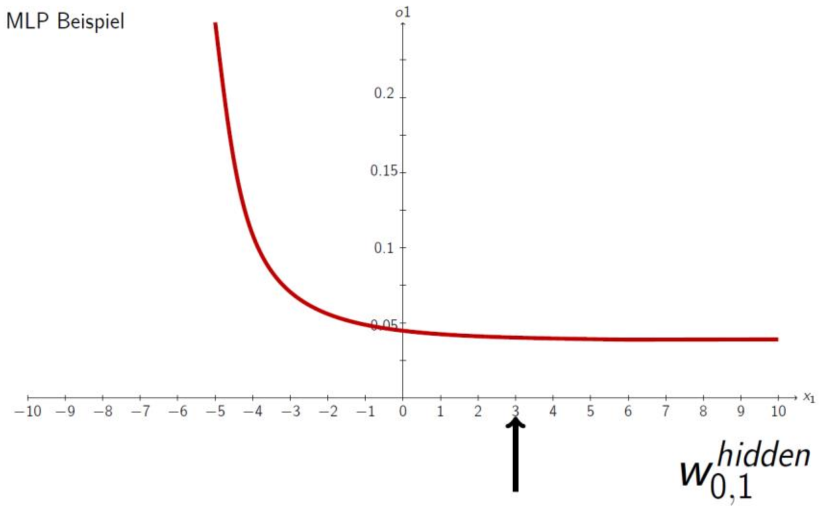


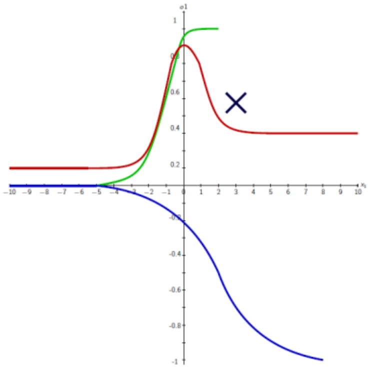
BackPropagation Example

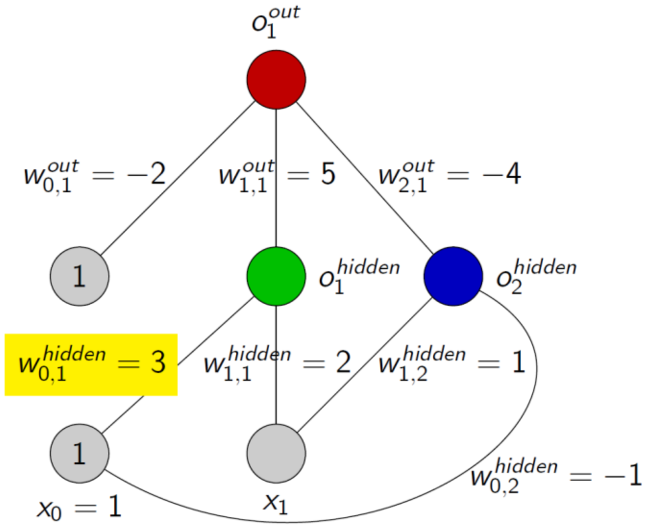




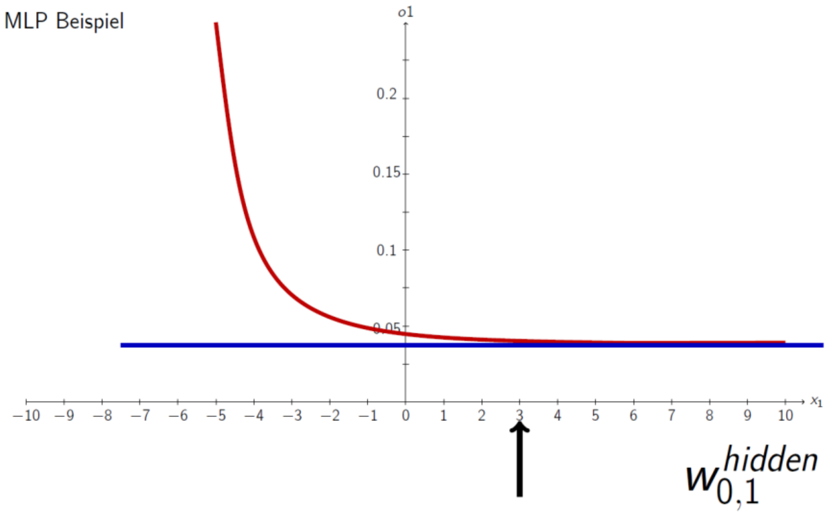
$$E\left(w_{0,1}^{\text{hidden}}\right)$$







$$E\left(w_{0,1}^{\text{hidden}}\right)$$



$$\frac{\delta E}{\delta w_{0,1}^{hidden}}$$

Variations of BackPropagation

Local Minima and Learning Rate

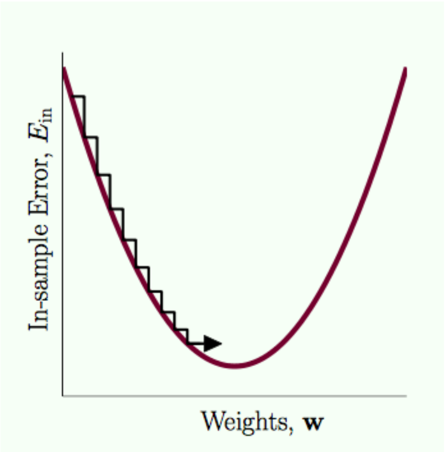
- Backpropagation as a gradient descent technique can only find a local minimum.

- Training the networks with different random initialisations can lead to a different weight configuration on a different local minimum.

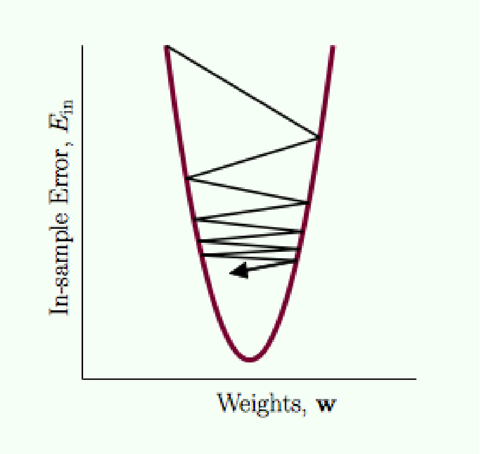
- The learning rate η defines the step width of the gradient descent technique.
  - A very large η leads to skipping minima or oscillations.
  - A very small η leads to starving, i.e. slow convergence or even convergence before the (local) minimum is reached.

Learning Rate η

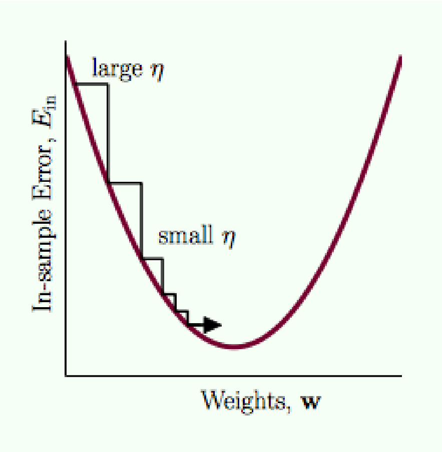
η too small



η too large



η just right



- Feed-Forward Neural Networks can potentially describe very complex relationships
- FFNNs are very simple but very flexible neural architectures
- It is easy to:
  - Expand the architecture by adding more units/layers
  - Experiment with new activation functions
- Too many parameters!
- Danger of fitting training data too well: Overfitting
  - Modeling of particularities in training data instead of underlying concept
    - $\Longrightarrow$ Modeling of artifacts or outliers

- Overfitting can be prevented by keeping the weights small

- Weight Decay:
  - Pushes all weights to zero; only those weights will “survive” that are really needed
- Momentum Term:
  - increase weight updates as long as they have the same sign
- Resilient Backpropagation (or RPROP):
  - estimate optimum for weights based on assumption that the error surface is a polynomial.

- Introduce a momentum term:
- For the weight update, the previous weight update is taken into account:

$$\Delta_p W(u,v) = \eta\, \delta_v^p\, o_u^p + \beta\, \Delta_q W(u,v)$$

- $\Delta_q W(u,v)$ is the weight update at the previous step q of the gradient descent algorithm.

- If weight is updated continuously in the same direction, the weight update increases, otherwise it decreases.

- Typical choices: $\eta = 0.2$, $\beta = 0.8$.

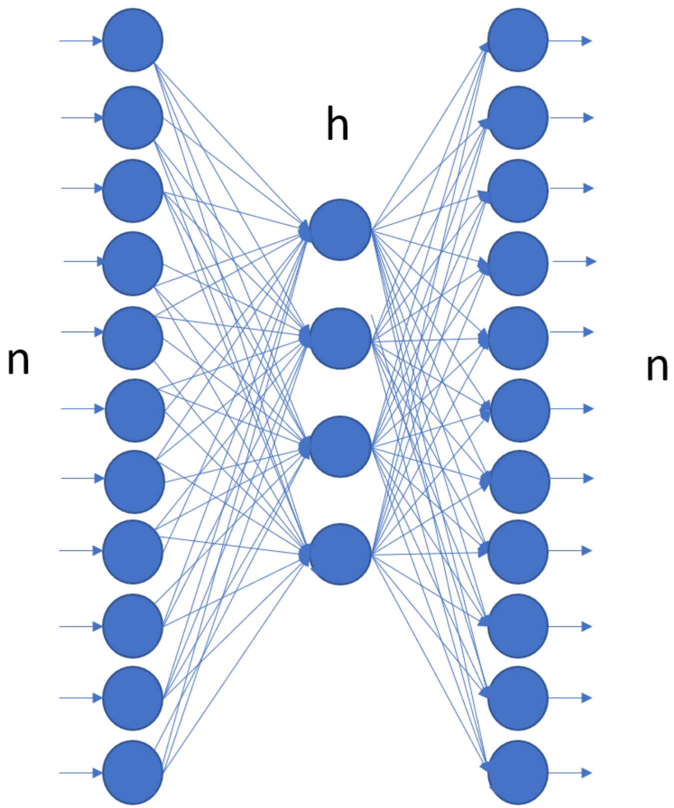
- MLPs are powerful but black boxes
- Rule extraction only possible in some cases
  - VI-Analysis (interval propagation)
  - extraction of decision trees
- Problems:
  - Global influence of each neuron
  - Interpretation of hidden layer(s) complicated

- Possible Solution:
  - Local activity of neurons in hidden layer: Local Basis Function Networks

BackPropagation Issues

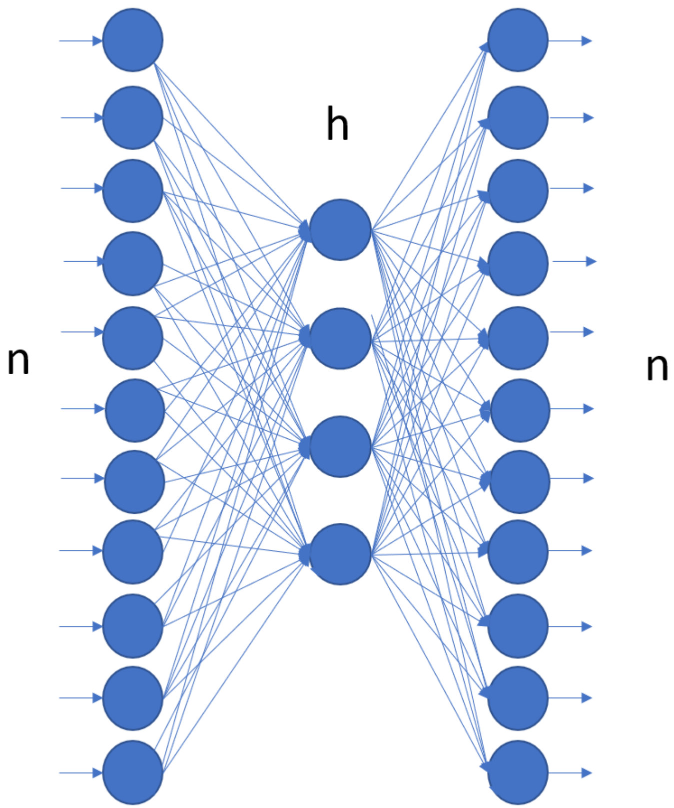
- Usually, weights are not updated after a whole epoch, i.e. after all patterns have been presented once (called offline training), but after the presentation of each input pattern (online training).
- This is usually faster, although not formally the same and potentially risky (oscillations).
- There is no general rule on how to choose the number of hidden layers and the size of the hidden layers.
  - Small neural networks might not be flexible enough to fit the data.
  - Large neural networks tend to overfit the data (note: Deep Learning...).
- The steepness of the activation function is usually fixed and is not adjusted.
- A perceptron learns only in those regions where the activation function is not close to zero or one, otherwise the derivative is almost zero.

The autoencoder architecture



- **Dimensionality reduction**
- Input and output are identical, i.e. the neural network should learn the identity function. (Auto-associative network)
- Introduce a hidden layer with only $h < n$ neurons: the bottleneck.
- Train the neural network with the data.
- After training, input the data into the network and use the outputs of the bottleneck neurons as a representation of the input data in a lower dimension
- If $h = 2$ then the outputs of the bottleneck neurons represent the two dimensions for the graphical representation of the data

The autoencoder architecture



- **Anomaly Detection**
- Input and output are identical, i.e. the neural network should learn the identity function. (Auto-associative network)
- Train the neural network with the data.
- After training, input the data into the network and calculate the distance between input and output layer.
- If input data is similar to training data, then distance is small.
- If input data is an anomaly not present in the training data, then distance is large

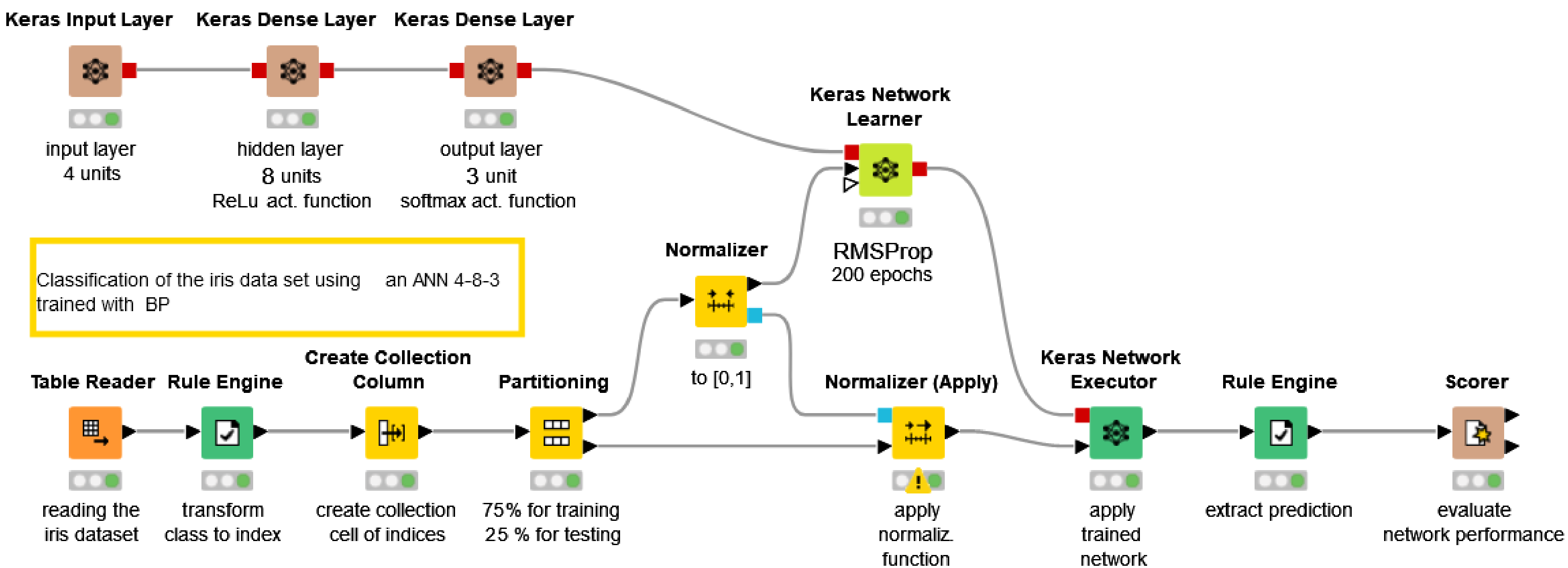
- (Hard/Soft) Competitive Learning
- Learning Vector Quantization
- Self Organizing Maps
- Radial (and other) Basis Function Networks
- Many connections to Kernel Methods and Support Vector Machines...

What you should remember from this lesson

- What is a the Perceptron
- And why we need MLPs
- The BackPropagation algorithm to train the hidden layers
- Issues with MLPs and BackPropgation
- The autoencoder architecture

Practical Example

– A multilayer perceptron with layers (4–8–3) is trained to classify the iris data set using the backpropagation algorithm, as set in the Keras Network Learner node



For any questions please contact: email@email.com