


Course 395: Machine Learning - Lectures

- Lecture 1-2: Concept Learning (*M. Pantic*)
- Lecture 3-4: Decision Trees & CBC Intro (*M. Pantic*)
-  • Lecture 5-6: Artificial Neural Networks (*S. Petridis*)
- Lecture 7-8: Instance Based Learning (*M. Pantic*)
- Lecture 9-10: Genetic Algorithms (*M. Pantic*)
- Lecture 11-12: Evaluating Hypotheses (*THs*)
- Lecture 13-14: Invited Lecture
- Lecture 15-16: Inductive Logic Programming (*S. Muggleton*)
- Lecture 17-18: Inductive Logic Programming (*S. Muggleton*)

Neural Networks

Reading:

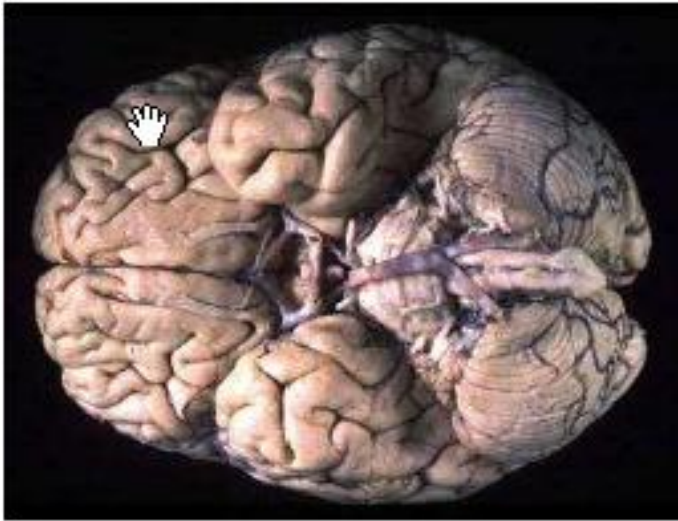
- *Machine Learning (Tom Mitchel) Chapter 4*
- *Pattern Classification (Duda, Hart, Stork) Chapter 6 (strongly to advised to read 6.1, 6.2, 6.3, 6.8)*

Further Reading:

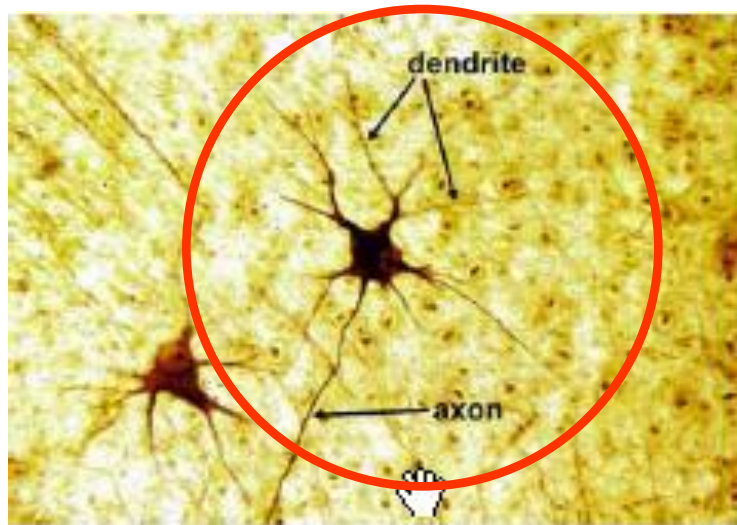
- *Pattern Recognition and Machine Learning by Christopher M. Bishop, Springer, 2006 (chapter: 5)*
- *Neural Networks (Haykin)*
- *Neural Networks for Pattern Recognition (Chris M. Bishop)*

What are Neural Networks?

The real thing!



10 billion neurons

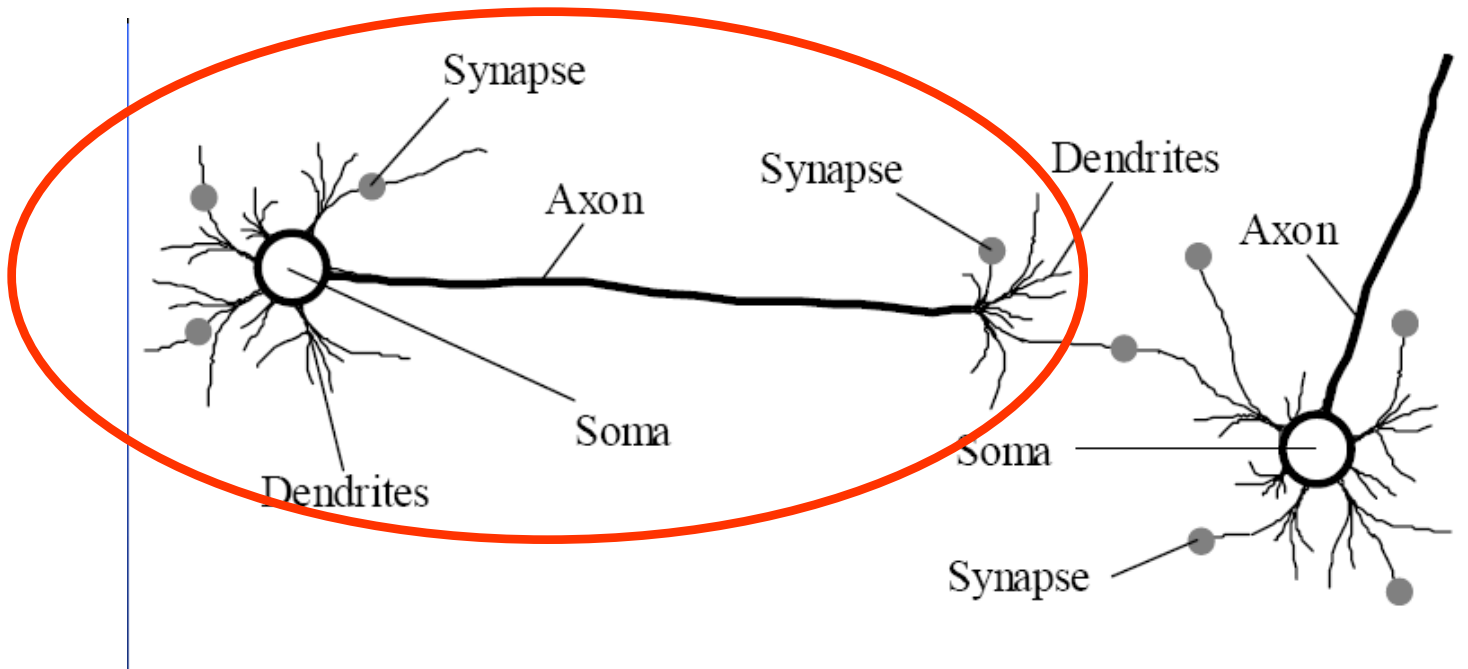


Local computations on **interconnected** elements (neurons)

Parallel computation

- neuron switch time 0.001sec
- recognition tasks performed in 0.1 sec.

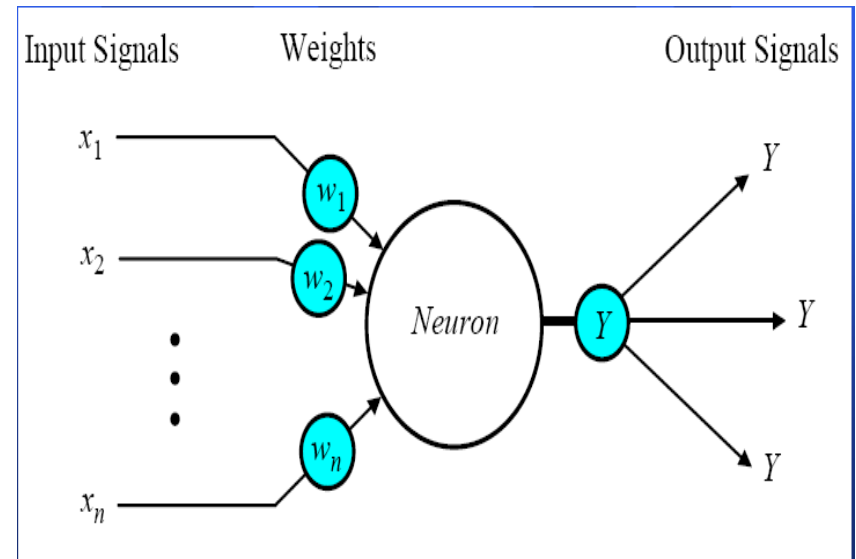
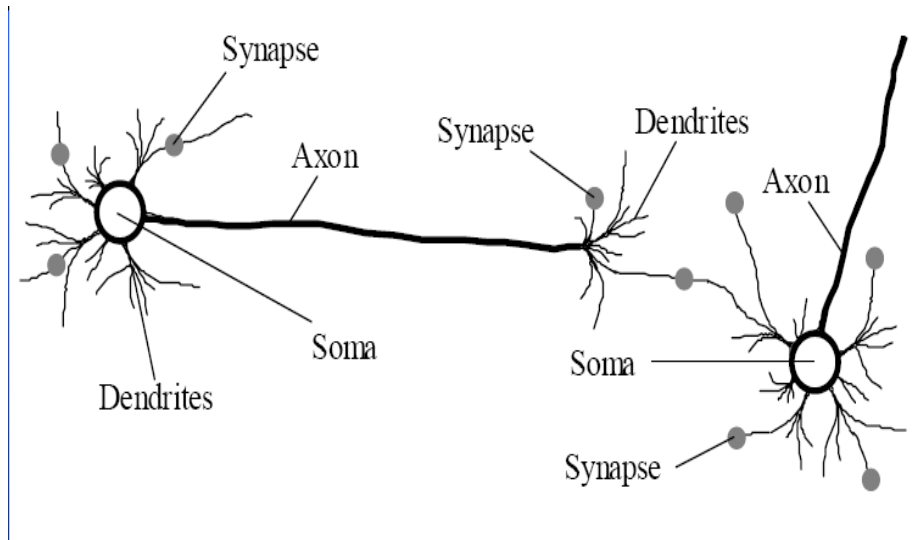
Biological Neural Networks



A network of interconnected biological neurons.

Connections per neuron $10^4 - 10^5$

Biological vs Artificial Neural Networks



<i>Biological Neural Network</i>	<i>Artificial Neural Network</i>
Soma	Neuron
Dendrite	Input
Axon	Output
Synapse	Weight

Artificial Neural Networks: the dimensions

Architecture

How the neurons are connected

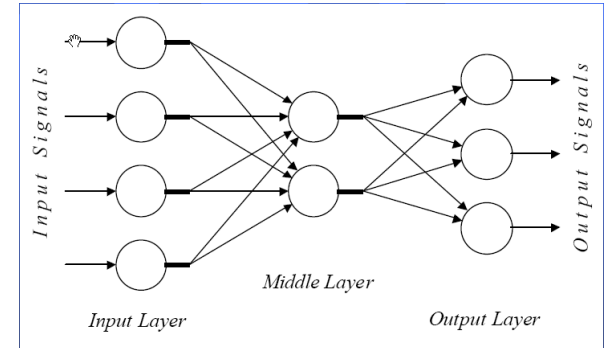
The Neuron

How information is processed in each unit. $\text{output} = f(\text{input})$

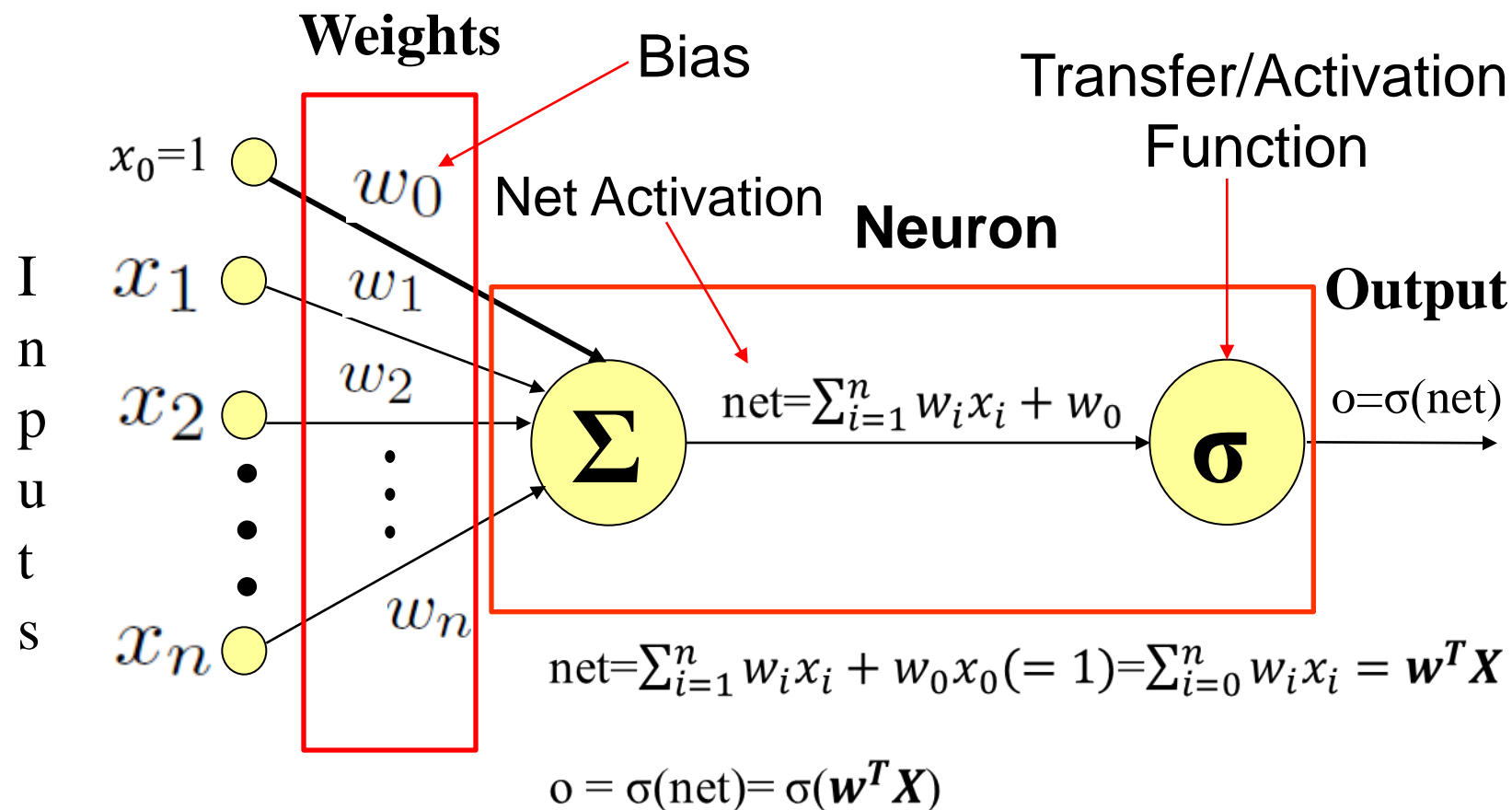
Learning algorithms

How a Neural Network modifies its **weights** in order to solve a particular **learning task** in a set of **training examples**

The goal is to have a Neural Network that **generalizes** well, that is, that it generates a 'correct' output on a set of **test/new examples/inputs**.

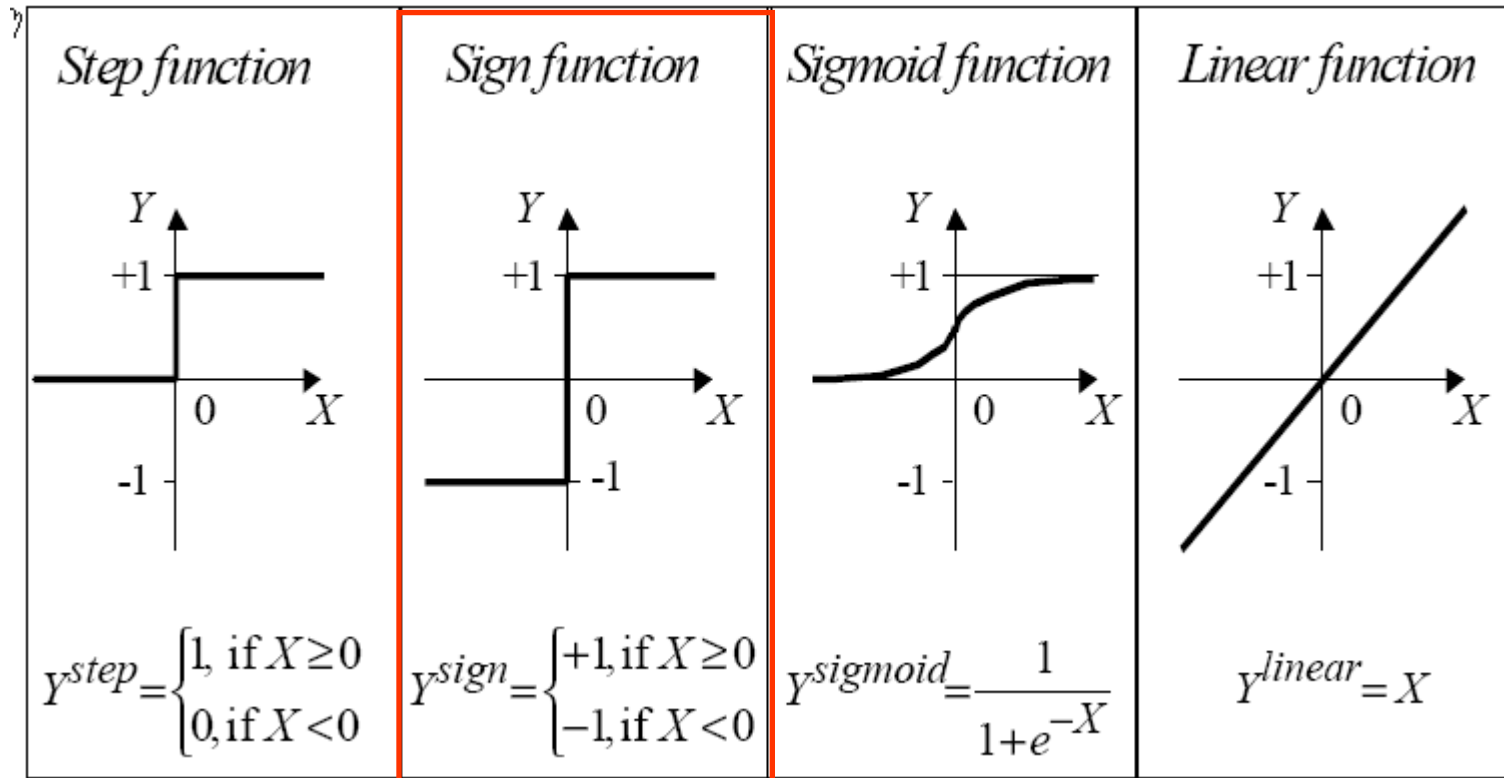


The Neuron



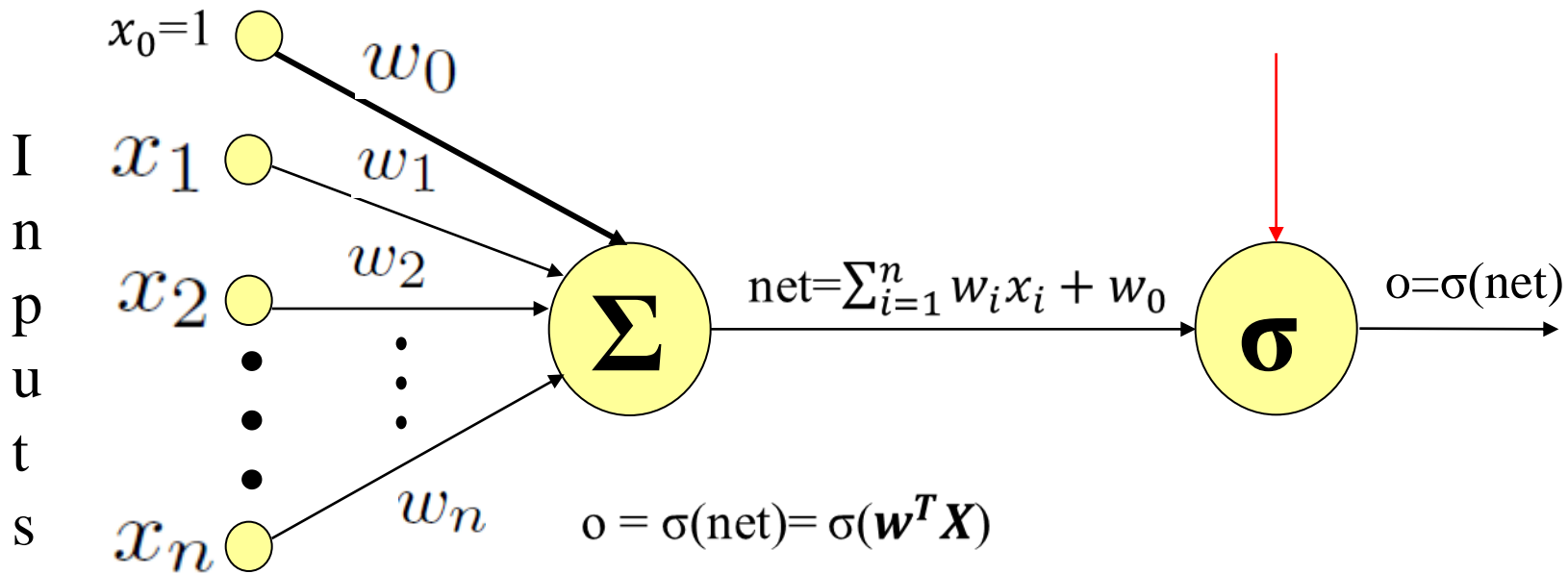
- Main building block of any neural network

Activation functions



$$X = net = \sum_{i=1}^n w_i x_i + w_0, \quad Y = o = \sigma(net)$$

Perceptron

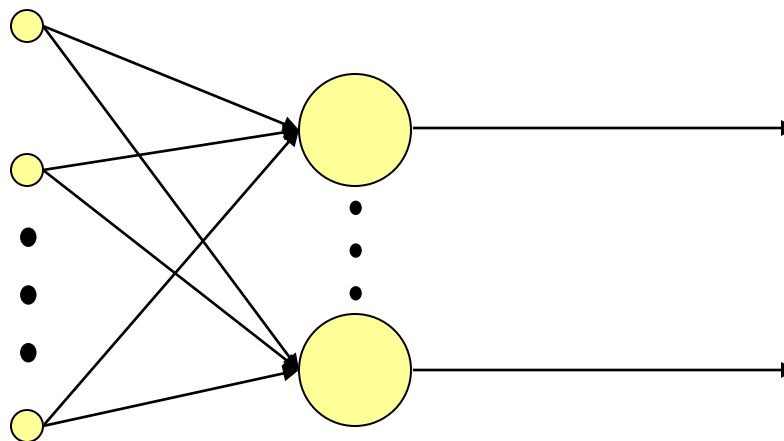


$$o = \sigma(\text{net}) = \begin{cases} 1 & \text{if } \text{net} > 0 \\ -1 & \text{otherwise} \end{cases}$$

- σ = sign/step/function
- Perceptron = a neuron that its input is the dot product of \mathbf{W} and \mathbf{X} and uses a step function as a transfer function

Perceptron: Architecture

- Generalization to single layer perceptrons with more neurons is easy because:



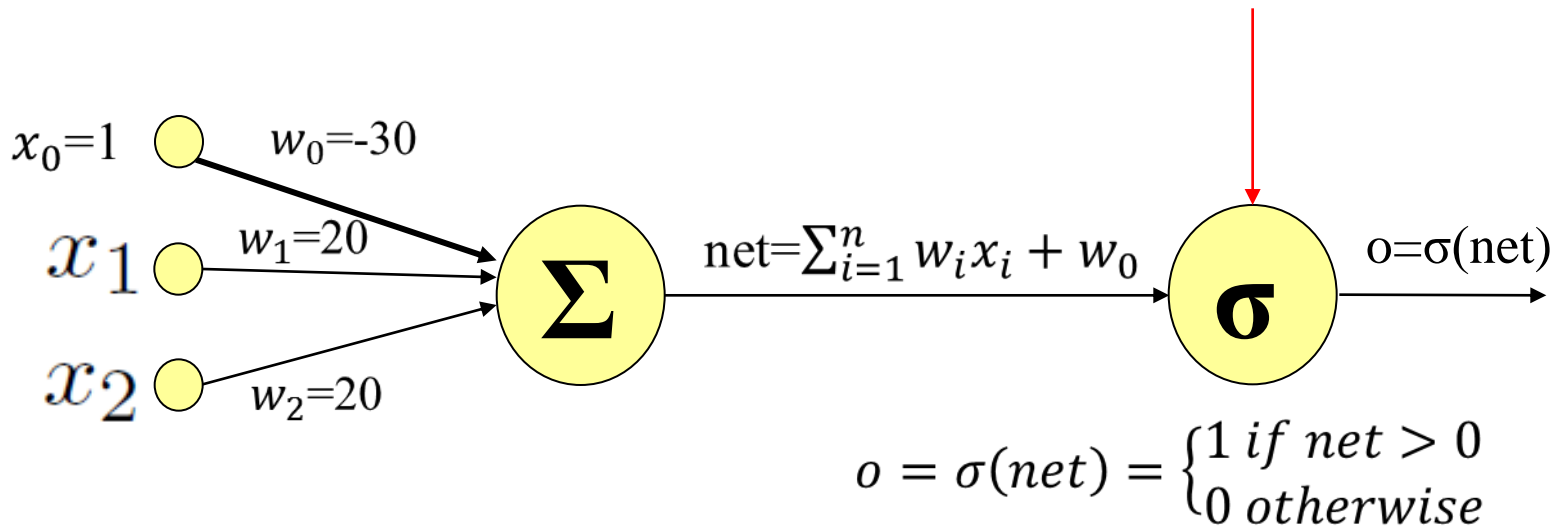
- The output units are mutually independent
- Each weight only affects one of the outputs

Perceptron

- Perceptron was invented by Rosenblatt
- *The Perceptron--a perceiving and recognizing automaton, 1957*

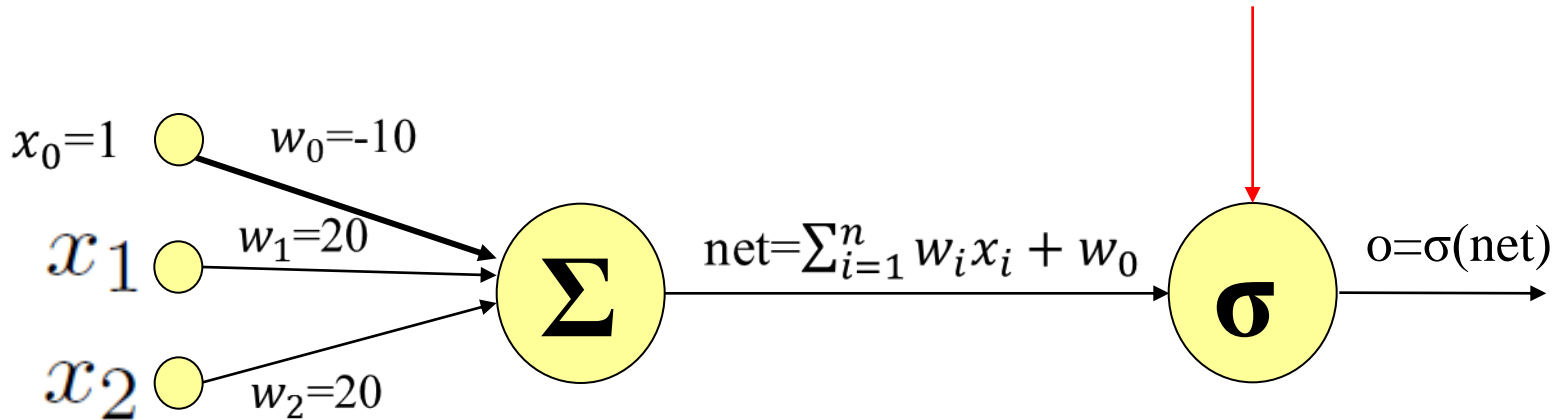


Perceptron: Example 1 - AND



- $x_1 = 1, x_2 = 1 \rightarrow \text{net} = 20 + 20 - 30 = 10 \rightarrow o = \sigma(10) = 1$
- $x_1 = 0, x_2 = 1 \rightarrow \text{net} = 0 + 20 - 30 = -10 \rightarrow o = \sigma(-10) = 0$
- $x_1 = 1, x_2 = 0 \rightarrow \text{net} = 20 + 0 - 30 = -10 \rightarrow o = \sigma(-10) = 0$
- $x_1 = 0, x_2 = 0 \rightarrow \text{net} = 0 + 0 - 30 = -30 \rightarrow o = \sigma(-30) = 0$

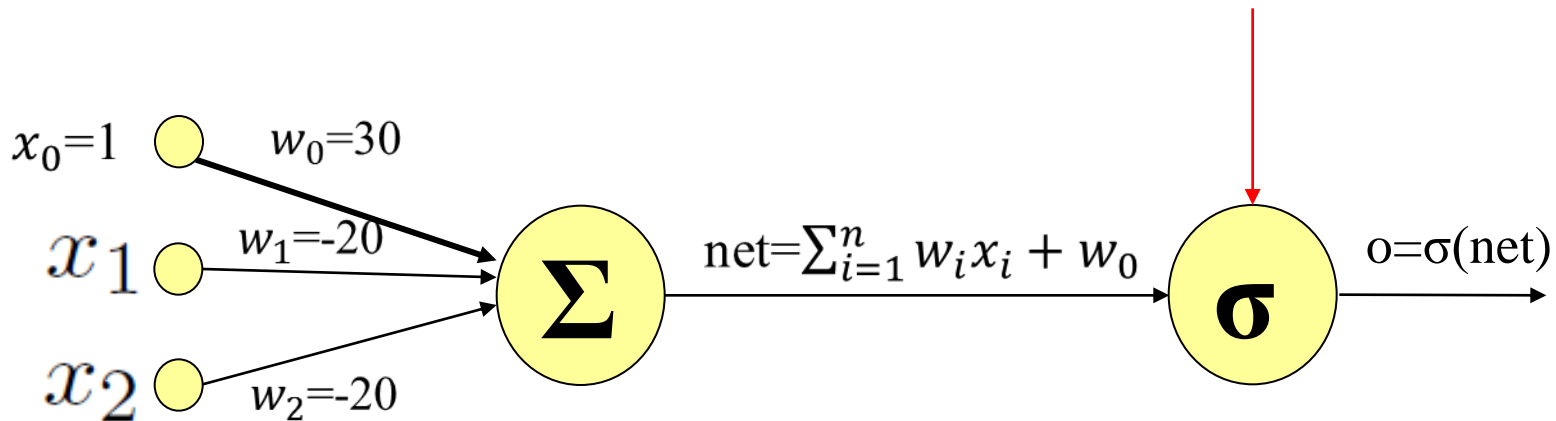
Perceptron: Example 2 - OR



$$o = \sigma(net) = \begin{cases} 1 & \text{if } net > 0 \\ 0 & \text{otherwise} \end{cases}$$

- $x_1 = 1, x_2 = 1 \rightarrow net = 20+20-10=30 \rightarrow o = \sigma(30) = 1$
- $x_1 = 0, x_2 = 1 \rightarrow net = 0+20-10 = 10 \rightarrow o = \sigma(10) = 1$
- $x_1 = 1, x_2 = 0 \rightarrow net = 20+0-10 = 10 \rightarrow o = \sigma(10) = 1$
- $x_1 = 0, x_2 = 0 \rightarrow net = 0+0-10 = -10 \rightarrow o = \sigma(-10) = 0$

Perceptron: Example 3 - NAND



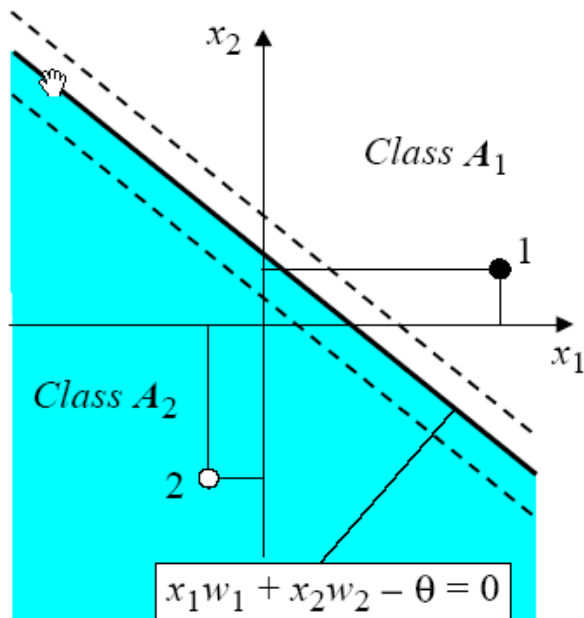
$$o = \sigma(net) = \begin{cases} 1 & \text{if } net > 0 \\ 0 & \text{otherwise} \end{cases}$$

- $x_1 = 1, x_2 = 1 \rightarrow net = -20 - 20 + 30 = -10 \rightarrow o = \sigma(-10) = 0$
- $x_1 = 0, x_2 = 1 \rightarrow net = 0 - 20 + 30 = 10 \rightarrow o = \sigma(10) = 1$
- $x_1 = 1, x_2 = 0 \rightarrow net = -20 + 0 + 30 = 10 \rightarrow o = \sigma(10) = 1$
- $x_1 = 0, x_2 = 0 \rightarrow net = 0 + 0 + 30 = 30 \rightarrow o = \sigma(30) = 1$

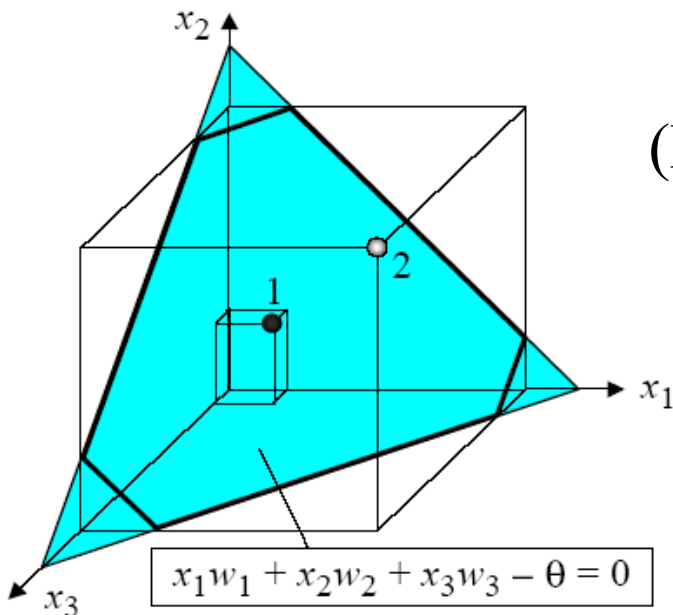
Perceptron for classification

- Given training examples of classes C_1, C_2 train the perceptron in such a way that it classifies correctly the training examples:
 - *If the output of the perceptron is 1 then the input is assigned to class C_1 (i.e. if $\sigma(\mathbf{w}^T \mathbf{x}) = 1$)*
 - *If the output is 0 then the input is assigned to class C_2*
- Geometrically, we try to find a hyper-plane that separates the examples of the two classes. The hyper-plane is defined by the linear function

Perceptron: Geometric view



(a) Two-input perceptron.



(b) Three-input perceptron.

(Note that $\theta = -w_0$)

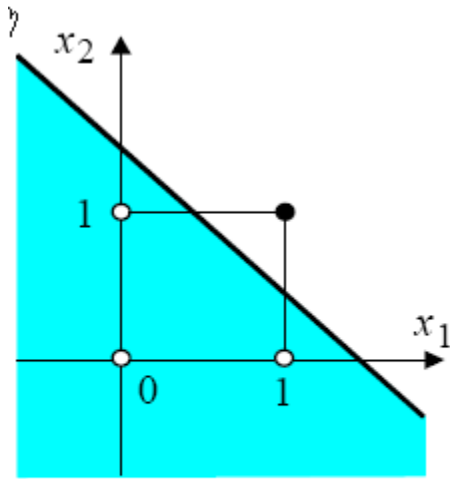
if $w_1x_1 + w_2x_2 + w_0 > 0$ then Class = A1

if $w_1x_1 + w_2x_2 + w_0 < 0$ then Class = A2

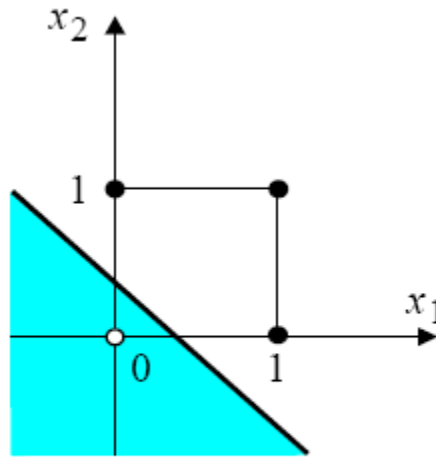
if $w_1x_1 + w_2x_2 + w_0 = 0$ then Class = A1 or A2

depends on our definition

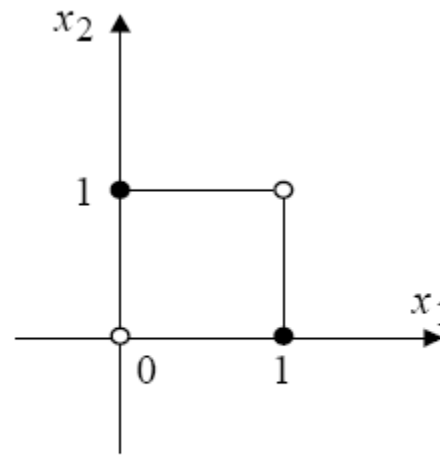
Perceptron: The limitations of perceptron



(a) *AND* ($x_1 \cap x_2$)



(b) *OR* ($x_1 \cup x_2$)



(c) *Exclusive-OR*
($x_1 \oplus x_2$)

- Perceptron can only classify examples that are linearly separable
- The XOR is not linearly separable.
- This was a terrible blow to the field

Perceptron

- A famous book was published in 1969: **Perceptrons**
- Caused a significant decline in interest and funding of neural network research
 - Marvin Minsky
 - Seymour Papert

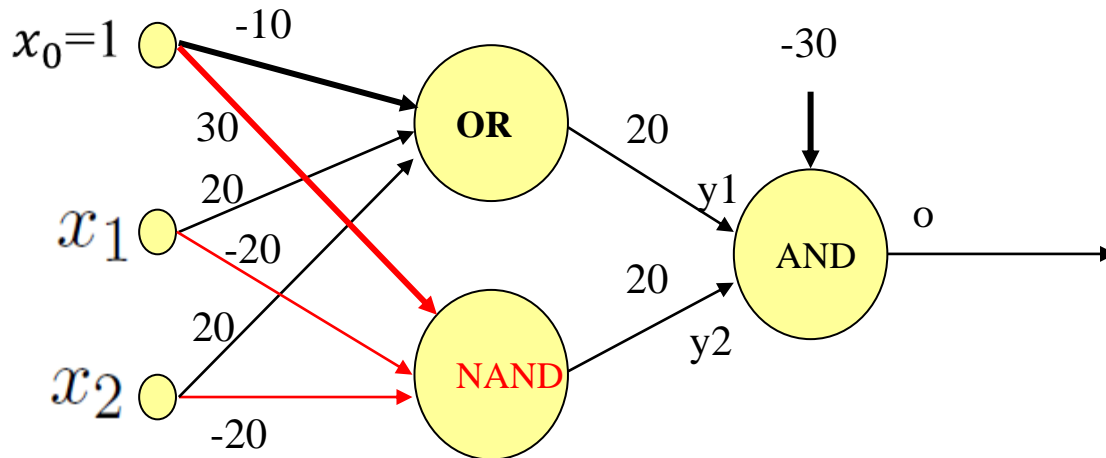


Perceptron XOR Solution

- XOR can be expressed in terms of AND, OR, NAND

Perceptron XOR Solution

- XOR can be expressed in terms of AND, OR, NAND
- $\text{XOR} = \text{NAND}(\text{AND}) \text{ OR}$



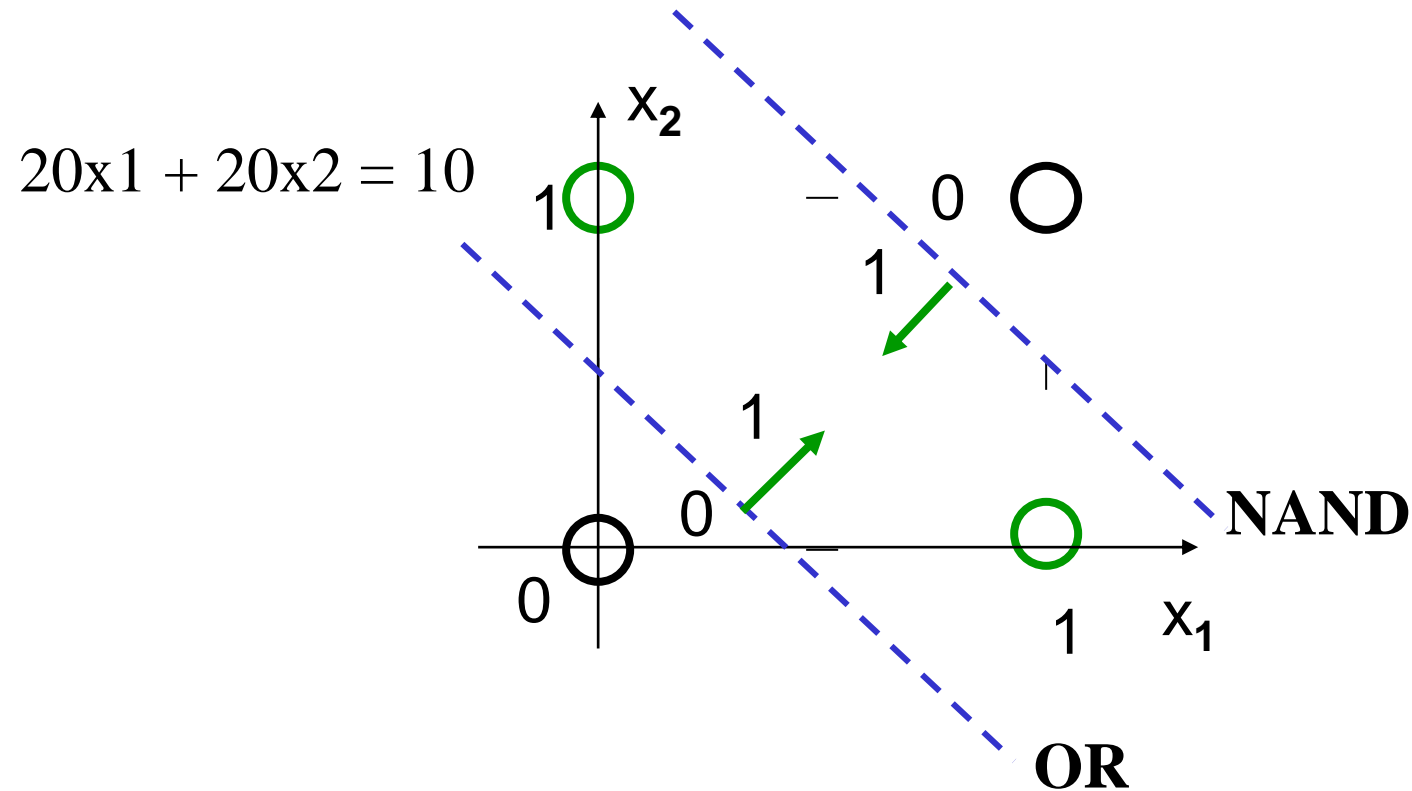
OR	NAND
$1\ 1 \rightarrow 1$	$1\ 1 \rightarrow 0$
$0\ 1 \rightarrow 1$	$0\ 1 \rightarrow 1$
$1\ 0 \rightarrow 1$	$1\ 0 \rightarrow 1$
$0\ 0 \rightarrow 0$	$0\ 0 \rightarrow 1$

AND
$1\ 1 \rightarrow 1$
$0\ 1 \rightarrow 0$
$1\ 0 \rightarrow 0$
$0\ 0 \rightarrow 0$

- $x_1=1, x_2=1 \rightarrow y_1=1 \text{ AND } y_2=0 \rightarrow o=0$
- $x_1=1, x_2=0 \rightarrow y_1=1 \text{ AND } y_2=1 \rightarrow o=1$
- $x_1=0, x_2=1 \rightarrow y_1=1 \text{ AND } y_2=1 \rightarrow o=1$
- $x_1=0, x_2=0 \rightarrow y_1=0 \text{ AND } y_2=1 \rightarrow o=0$

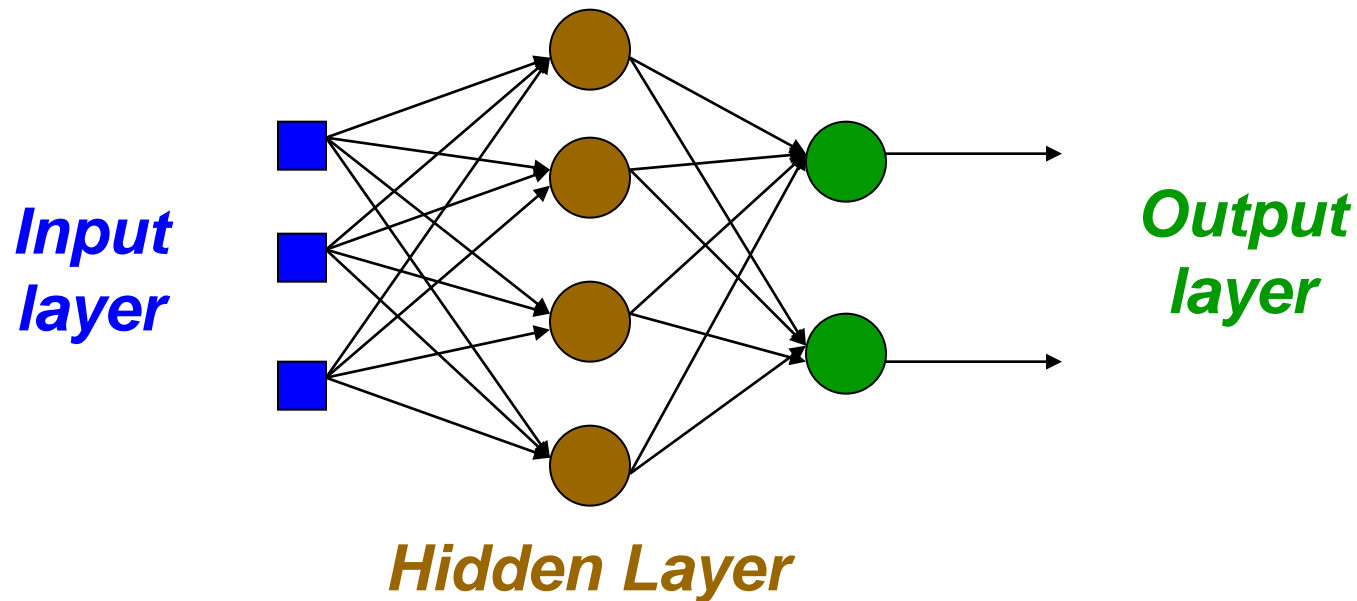
XOR

$$20x_1 + 20x_2 = 30$$

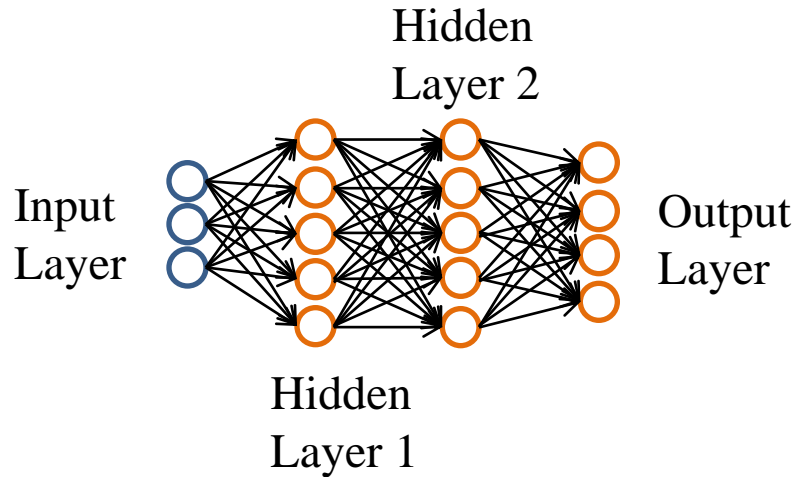


Multilayer Feed Forward Neural Network

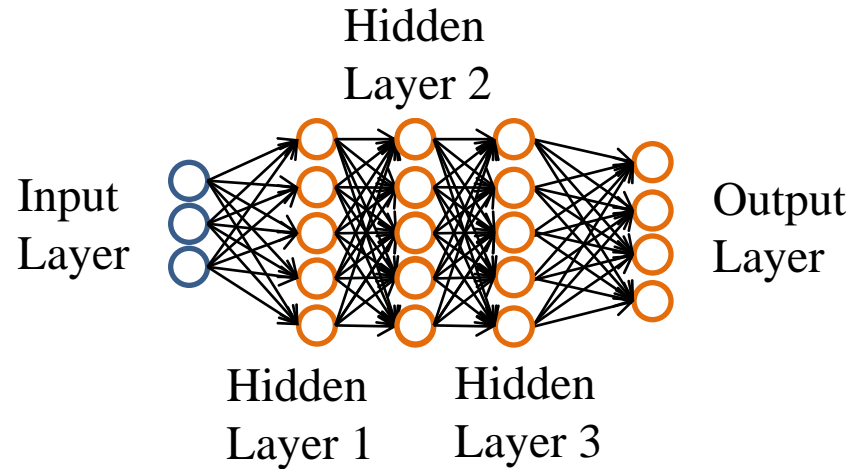
- We consider a more general network architecture: between the input and output layers there are hidden layers, as illustrated below.
- Hidden nodes do not directly receive inputs nor send outputs to the external environment.



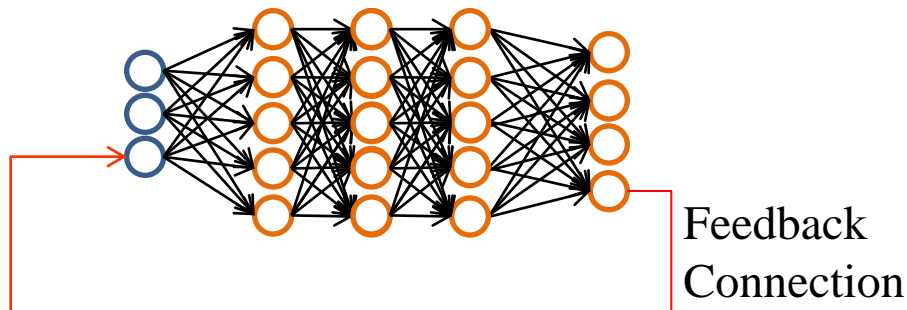
NNs: Architecture



3-layer feed-forward network



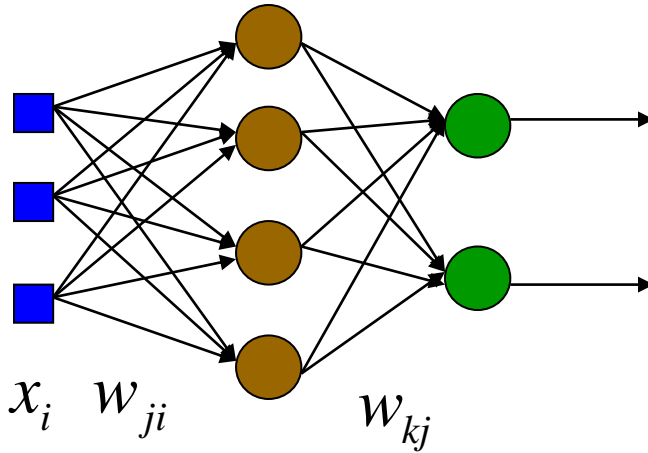
4-layer feed-forward network



4-layer recurrent network

- The input layer does not count as a layer

Multilayer Feed Forward Neural Network



w_{ji} = weight associated with i th input to hidden unit j

w_{kj} = weight associated with j th input to output unit k

y_j = output of j th hidden unit

o_k = output of k th output unit

n = number of inputs

nH = number of hidden neurons

$$y_j = \sigma\left(\sum_{i=0}^n x_i w_{ji}\right)$$

$$o_k = \sigma\left(\sum_{j=1}^{nH} y_j w_{kj}\right)$$

$$o_k = \sigma\left(\sum_{j=1}^{nH} \sigma\left(\sum_{i=0}^n x_i w_{ji}\right) w_{kj}\right)$$

Representational Power of Feedforward Neural Networks

- Boolean functions: Every boolean function can be represented **exactly** by some network with two layers
- Continuous functions: Every bounded continuous function can be **approximated** with arbitrarily small error by a network with 2 layers
- Arbitrary functions: Any function can be **approximated** to arbitrary accuracy by a network with 3 layers
- Catch: We do not know 1) what the appropriate number of hidden neurons is, 2) the proper weight values

$$o_k = \sigma\left(\sum_{j=1}^{n_H} \sigma\left(\sum_{i=0}^n x_i w_{ji}\right) w_{kj}\right)$$

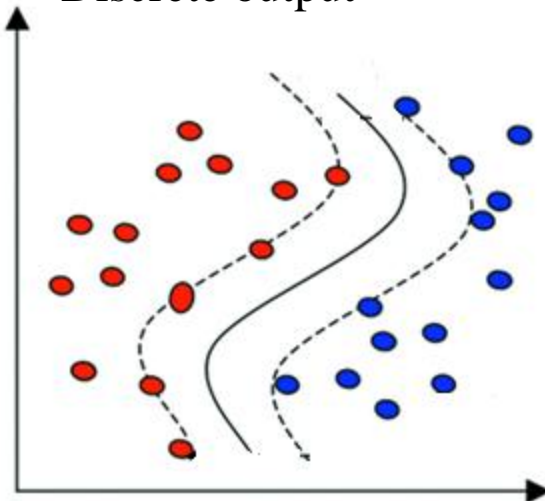
Classification /Regression with NNs

- You should think of neural networks as function approximators

$$o_k = \sigma\left(\sum_{j=1}^{nH} \sigma\left(\sum_{i=0}^n x_i w_{ji}\right) w_{kj}\right)$$

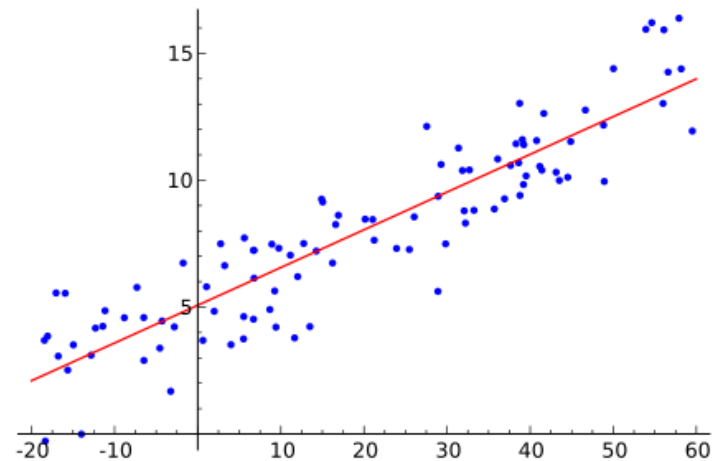
Classification

- Decision boundary approximation
- Discrete output



Regression

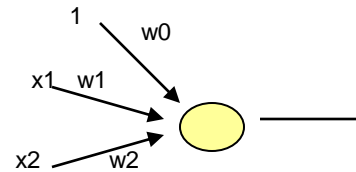
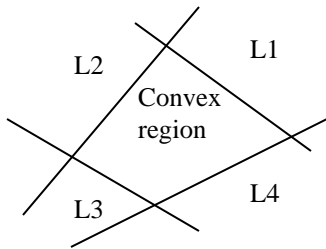
- Continuous output



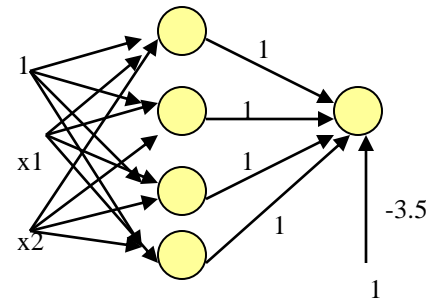
Decision boundaries

$$w_0 + w_1x_1 + w_2x_2 > 0$$

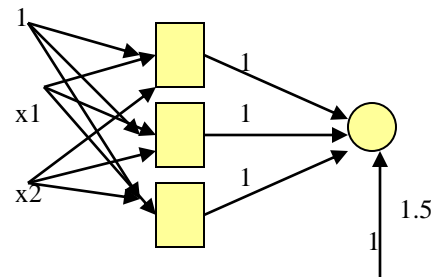
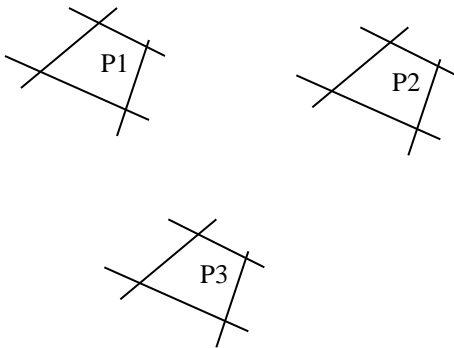
$$w_0 + w_1x_1 + w_2x_2 < 0$$



Network with a single node



One-hidden layer network that realizes the convex region: each hidden node realizes one of the lines bounding the convex region



two-hidden layer network that realizes the union of three convex regions: each box represents a one hidden layer network realizing one convex region

Output Representation

- Binary Classification

Target Values (t): 0 (negative) and 1 (positive)

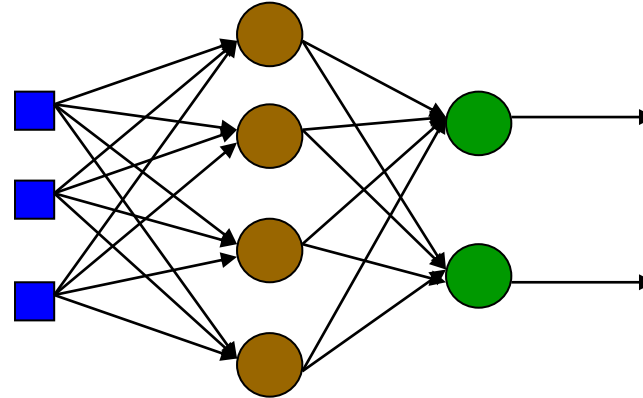
- Regression

Target values (t): continuous values $[-\infty, +\infty]$

- Ideally $o \approx t$

$$o_k = \sigma\left(\sum_{j=1}^{nH} \sigma\left(\sum_{i=0}^n x_i w_{ji}\right) w_{kj}\right)$$

Multiclass Classification



Target Values: vector (length=no. Classes)

Class1 Class2 Class3 Class4

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

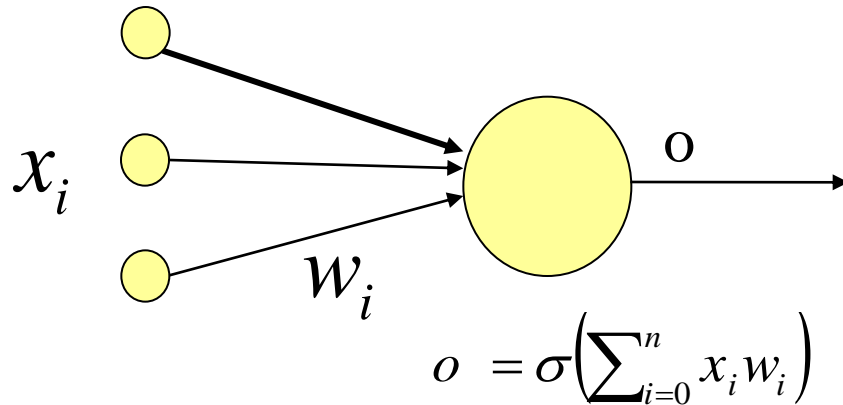
Training

- We have assumed so far that we know the weight values
- We are given a training set consisting of inputs and targets (**X**, **T**)
- Training: Tuning of the weights (**w**) so that for each input pattern (**x**) the output (**o**) of the network is close to the target values (**t**).

$$o \approx t$$

$$o = \sigma \left(\sum_{j=1}^{nH} \sigma \left(\sum_{i=0}^n x_i w_{ji} \right) w_{kj} \right)$$

Perceptron - Training



- Training Set: A set of input vectors

$x_i, i = 1 \dots n$ with the
corresponding targets t_i

- Begin with random weights
- Update the weights so $o \approx t$

Training – Gradient Descent

- Gradient Descent: A general, effective way for estimating parameters (e.g. \mathbf{w}) that minimise error functions
- We need to define an error function $E(\mathbf{w})$
- Update the weights in each iteration in a direction that reduces the error the order in order to minimize E

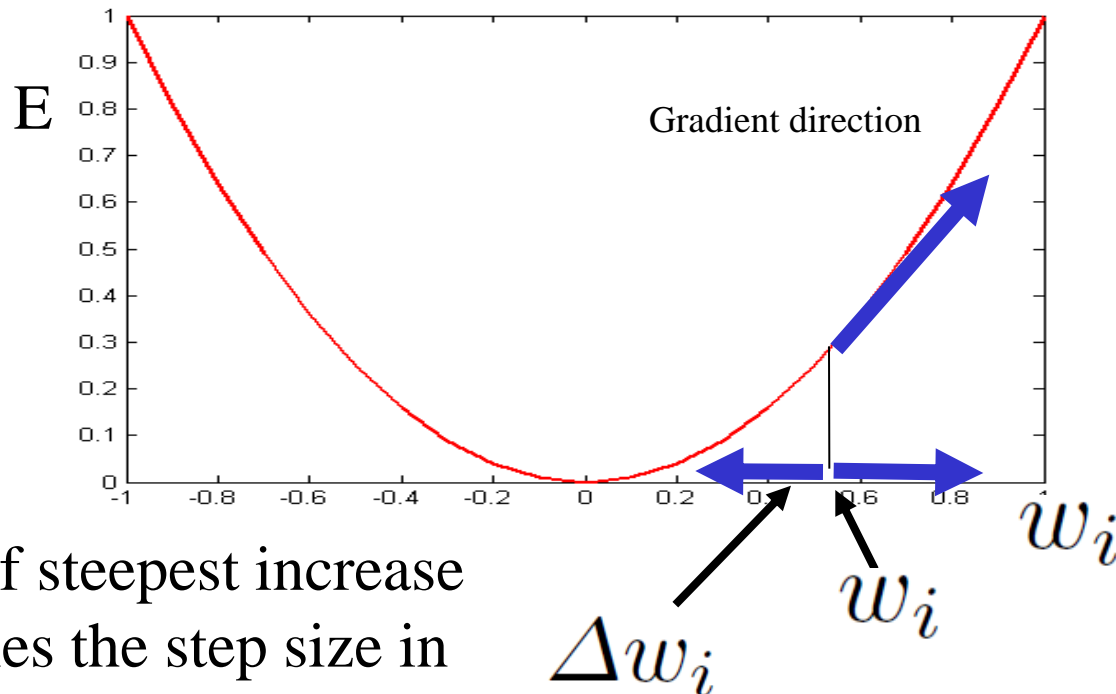
$$w_i \leftarrow w_i + \Delta w_i$$

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Gradient Descent

Gradient descent method: take a step in the direction that decreases the error E. This direction is the opposite of the derivative of E.

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$



- derivative: direction of steepest increase
- learning rate: determines the step size in the direction of steepest decrease

Gradient Descent

- We define our error function as follows (D = number of training examples):

$$E = \frac{1}{2} \sum_{d=1}^D (t_d - o_d)^2$$

- We assume linear activation transfer (i.e. its derivative = 1)
- E depends on the weights because $o_d = \sum_{i=0}^n x_i^d w_i$
- We wish to find the weight values that minimise E , i.e. the desired target t is very close to the actual output o .

Gradient Descent

$$E = \frac{1}{2} \sum_{d=1}^D (t_d - o_d)^2 \quad \Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

- The partial derivative can be computed as follows:

$$\begin{aligned} \frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\ &= \frac{1}{2} \sum_{d \in D} \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\ &= \frac{1}{2} \sum_{d \in D} 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\ &= \sum_{d \in D} (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x}_d) \\ \frac{\partial E}{\partial w_i} &= \sum_{d \in D} (t_d - o_d) (-x_{id}) \end{aligned}$$

- Therefore Δw is $\Delta w_i = \eta \sum_{d=1}^D (t_d - o_d) x_{id}$

Gradient Descent -Perceptron- Summary

1. Initialise weights randomly
2. Compute the output o for all the training examples
3. Compute the weight update for each weight

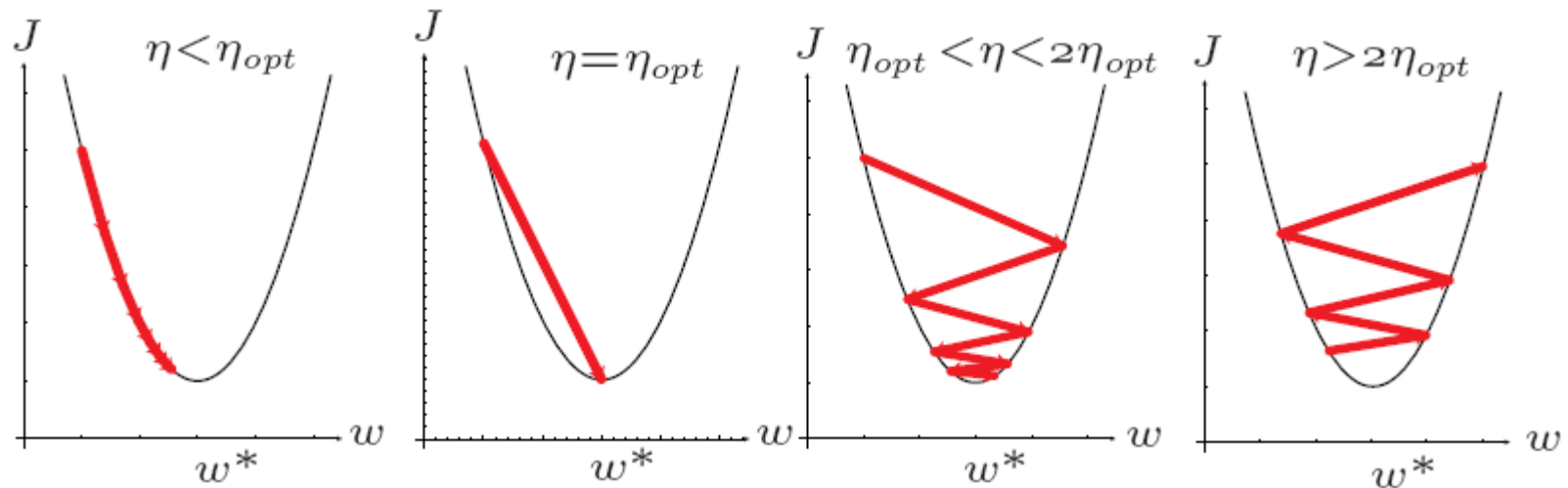
$$\Delta w_i = \eta \sum_{d=1}^D (t_d - o_d) x_{id}$$

4. Update the weights

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

- Repeat steps 2-4 until a termination condition is met
- The algorithm will converge to a weight vector with minimum error, given that the learning rate is sufficiently small

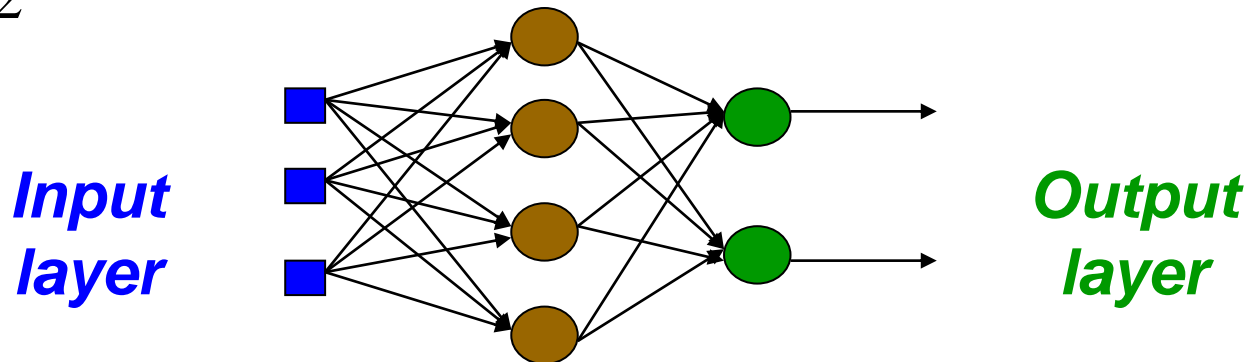
Gradient Descent – Learning Rate



Learning: The backpropagation algorithm

- The Backprop algorithm searches for weight values that minimize the total squared error of the network (K outputs) over the set of D training examples (training set).

$$E = \frac{1}{2} \sum_{k=1}^K \sum_{d=1}^D (t_{kd} - o_{kd})^2$$



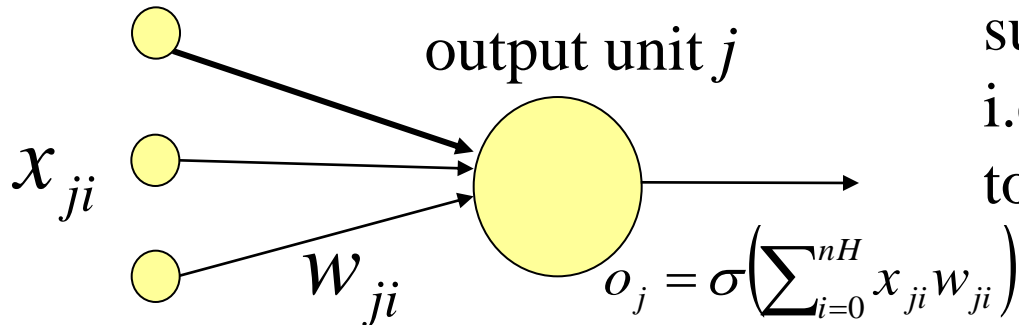
- Based on gradient descent algorithm

$$w_i \leftarrow w_i + \Delta w_i \quad \Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Learning: The backpropagation algorithm

- Backpropagation consists of the repeated application of the following two passes:
 - **Forward pass**: in this step the network is activated on one example and the error of (each neuron of) the output layer is computed.
 - **Backward pass**: in this step the network error is used for updating the weights (credit assignment problem). This process is complex because hidden nodes are linked to the error not directly but by means of the nodes of the next layer. Therefore, starting at the output layer, the error is propagated backwards through the network, layer by layer.

Learning: Output Layer Weights



subscript ji : i th input to unit j ,
i.e., input from hidden neuron i
to output unit j

- Consider the error of one pattern d : $E_d = \frac{1}{2} \sum_{k=1}^K (t_k - o_k)^2$
- Using exactly the same approach as in the perceptron

$$\Delta w_{ji} = \eta (t_j - o_j) x_{ji} \underbrace{\frac{\partial \sigma(\text{net}_j)}{\partial \text{net}_j}}_{-\partial E_d / \partial w_{ji}}$$

$$\Delta w_i = \eta \sum_{d=1}^D (t_d - o_d) x_{id}$$

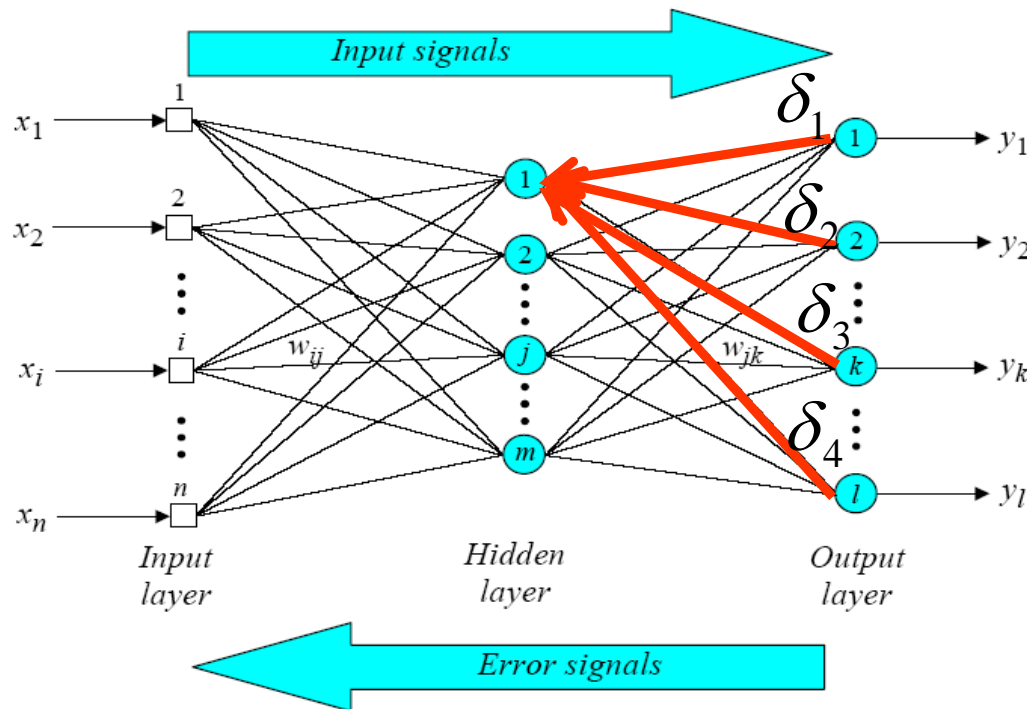
perceptron Δw

- The only difference is the partial derivative of the sigmoid activation function (we assumed linear activation \rightarrow derivative = 1)

Learning: Hidden Layer Weights

- We define the error term for the output k : $\delta_k = (t_k - o_k) \frac{\partial \sigma(net_k)}{\partial net_k}$
- Reminder: $\Delta w_{kj} = \eta(t_k - o_k) x_{kj} \frac{\partial \sigma(net_k)}{\partial net_k}$

Note that $j \rightarrow k$, $i \rightarrow j$, i.e.
 j : hidden unit
 k : output unit



- The error term for hidden unit j is:

$$\delta_j = \sum_{k=\text{outputNeuronsConnectedTo}j} \delta_k w_{kj} \frac{\partial \sigma(net_j)}{\partial net_j}$$

Learning: Hidden Layer Weights

- We define the error term for the output k : $\delta_k = (t_k - o_k) \frac{\partial \sigma(net_k)}{\partial net_k}$
- Reminder: $\Delta w_{kj} = \eta(t_k - o_k)x_{kj} \frac{\partial \sigma(net_k)}{\partial net_k} = \eta \delta_k x_{kj}$
- x is the input to output unit k from hidden unit j

- Similarly the update rule for weights in the input layer is

$$\Delta w_{ji} = \eta \delta_j x_{ji}$$

$$\delta_j = \sum_{k=\text{outputNeuronsConnectedTo}j} \delta_k w_{kj} \frac{\partial \sigma(net_j)}{\partial net_j}$$

- x is the input to hidden unit j from input unit i

Learning: Backpropagation Algorithm

- Finally for all training examples D

$$\Delta w_i^{\text{for all examples}} = \sum_{d=1}^D \Delta w_i$$

- This is called batch training because the weights are updated after all training examples have been presented to the network (=epoch)
- Incremental training: weights are updated after each training example is presented

Backpropagation Stopping Criteria

- When the gradient magnitude is small, i.e. the error does not change much between two epochs
- When the maximum number of epochs has been reached
- When the error on the validation set increases for n consecutive times (this implies that we monitor the error on the validation set)

Backpropagation Summary

1. Initialise weights randomly
2. For each input training example x compute the outputs (forward pass)
3. Compute the output neurons errors and then compute the update rule for output layer weights (backward pass)
$$\Delta w_{ki} = \eta \delta_k x_{ki}$$
4. Compute hidden neurons errors and then compute the update rule for hidden layer weights (backward pass)
$$\Delta w_{ji} = \eta \delta_j x_{ji}$$
5. Compute the sum of all Δw , once all training examples have been presented to the network

Backpropagation Summary

6. Update weights

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

7. Repeat steps 2-6 until the stopping criterion is met

Backpropagation: Variants

- Backpropagation with momentum

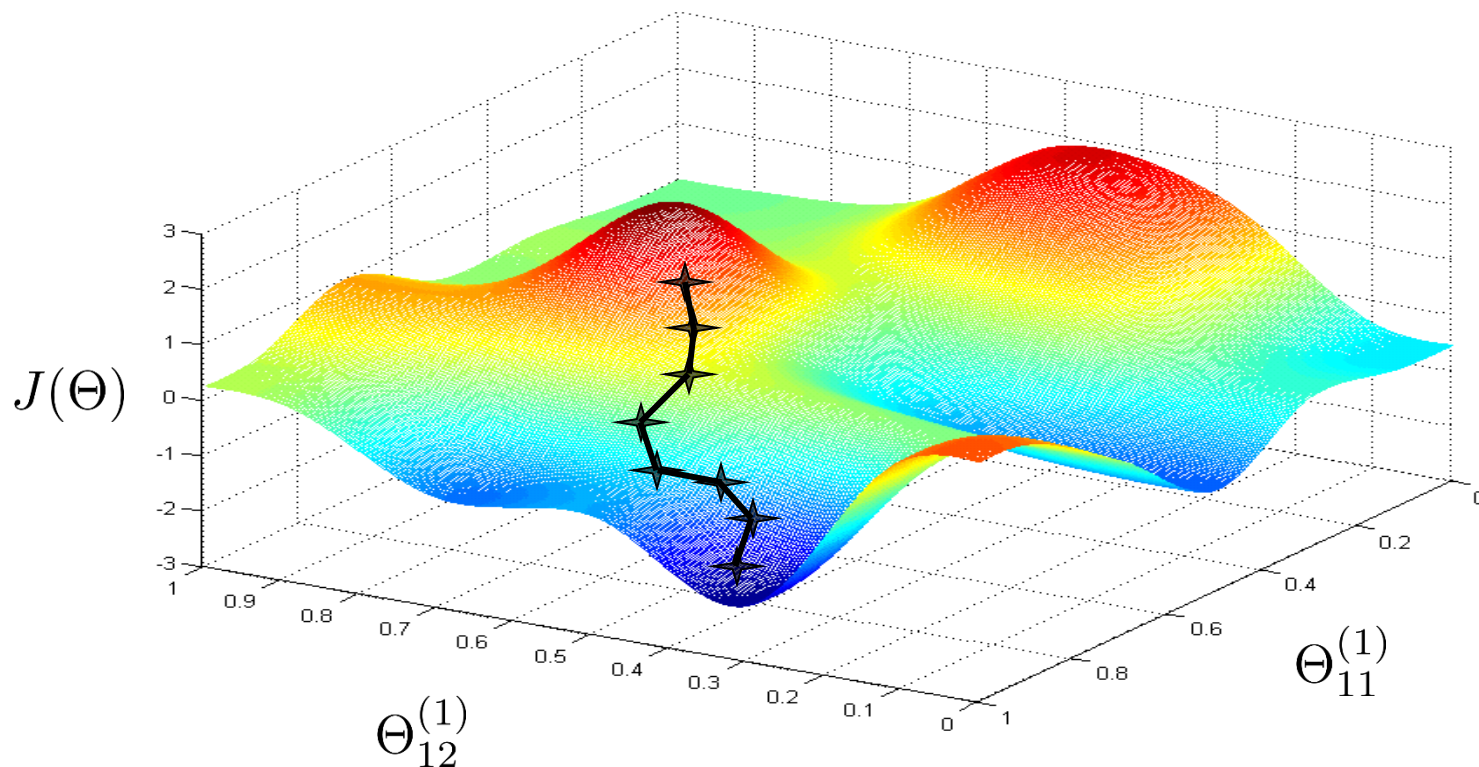
$$\Delta w_{ji}(n) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(n-1), \quad 0 \leq \alpha < 1$$

- The weight update depends also on the previous update
- Useful for flat regions where the update would be small
- Backpropagation with adaptive learning rate
- Learning rate changes during training
 - If error is decreased after weight update then lr is increased
 - If error is increased after weight update then lr is decreased

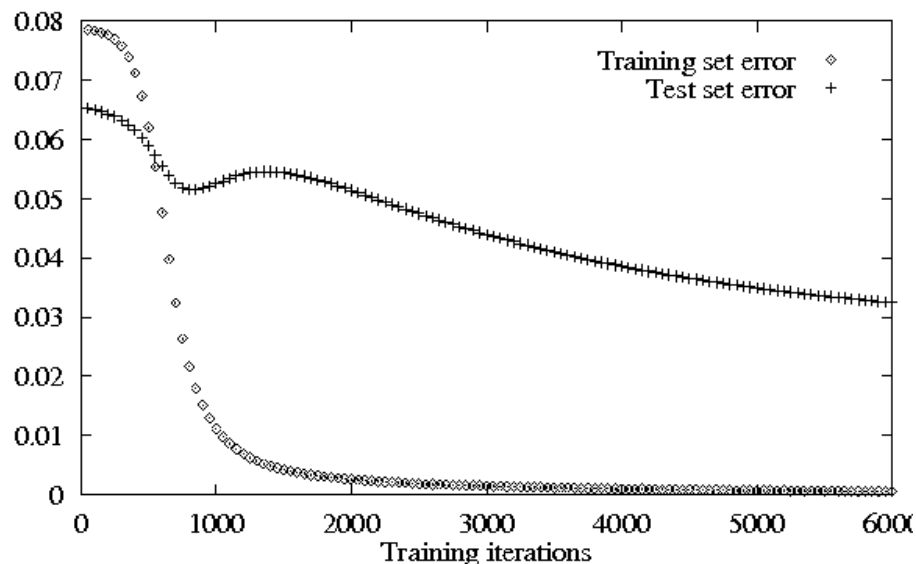
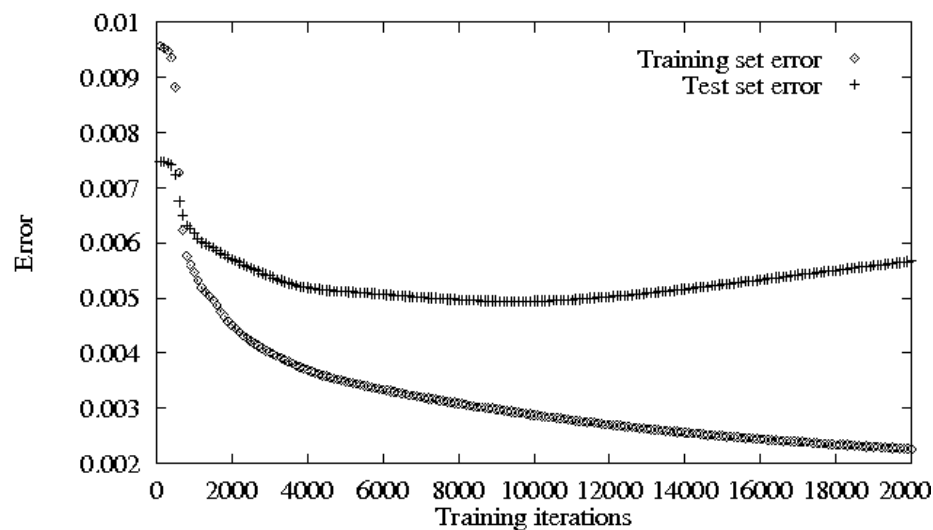
Backpropagation: Convergence

- Converges to a local minimum of the error function
 - ... can be retrained a number of times
- Minimises the error over the training examples
 - ...will it generalise well over unknown examples?
- Training requires thousands of iterations (slow)
 - ... but once trained it can rapidly evaluate output

Backpropagation: Error Surface



Backpropagation: Overfitting



- Error might decrease in the training set but increase in the ‘validation’ set (overfitting!)
- Stop when the error in the validation set increases (but not too soon!). This is called early stopping

Backpropagation: Overfitting

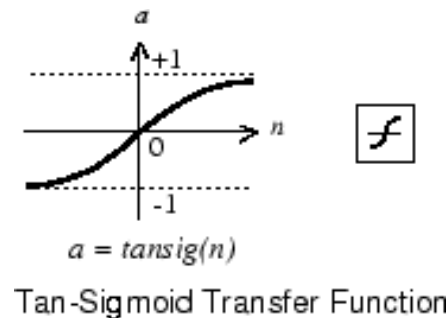
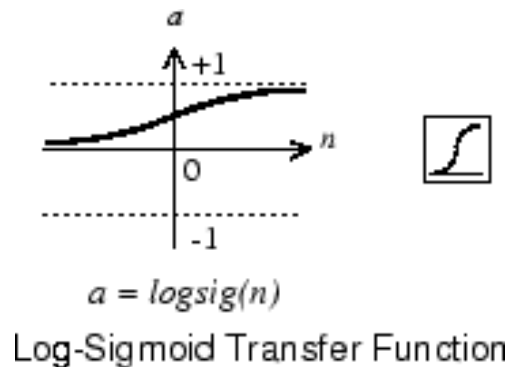
- Another approach to avoid overfitting is regularisation
- Add a term in the error function to penalise large weight values
- Having smaller weights and biases makes the network less likely to overfit
- Check matlab documentation for other approaches

Parameters / Weights

- Parameters are what the user specifies, e.g. number of hidden neurons, learning rate, number of epochs etc
- They need to be optimised
- Weights are the weights of the network
- They are also parameters but they are optimised automatically via gradient descent

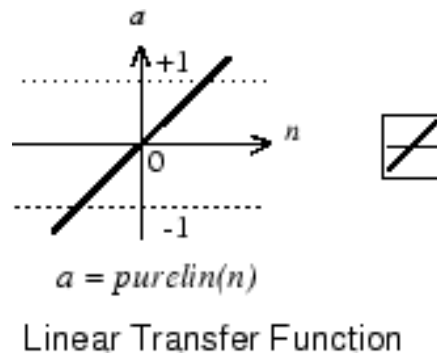
Practical Suggestions: Activation Functions

- Continuous, smooth (essential for backpropagation)
- Nonlinear
- Saturates, i.e. has a min and max value
- Monotonic (if not then additional local minima can be introduced)
- Sigmoids are good candidate (log-sig, tan-sig)



Practical Suggestions: Activation Functions

- In case of regression then output layer should have linear activation functions



Practical Suggestions: Scaling Inputs

- It is not desirable that some inputs are orders of magnitude larger than other inputs
- Map each input $x(i)$ to $[-1/0, +1]$
- Standardize inputs to 0 mean 1 std. dev.
 - Remove mean, divide by std. dev.

Practical Suggestions: Target Values

- Binary Classification
 - Target Values : -1/0 (negative) and 1 (positive)
 - 0 for log-sigmoid, -1 for tan-sigmoid
- Multiclass Classification
 - [0,0,1,0] or [-1, -1, 1, -1]
 - 0 for log-sigmoid, -1 for tan-sigmoid
- Regression

Target values: continuous values $[-\infty, +\infty]$

Practical Suggestions: Number of Hidden Layers

- Networks with many hidden layers are prone to overfitting and they are also harder to train
- For most problems one hidden layer should be enough
- 2 hidden layers can sometimes lead to improvement

Matlab Examples

- Create feedforward network
 - `net = feedforwardnet(hiddenSizes,trainFcn)`
 - `hiddenSizes = [10, 10, 10]` – 3 layer network with 10 hidden neurons in each layer
 - `trainFcn = 'trainlm', 'traingdm', 'trainrp'` etc
- Configure (set number of input/output neurons)
 - `net = configure(net,x,t)`
 - `x`: input data, `noFeatures x noExamples`
 - `t`: target data, `noClasses x noEx`

Matlab Examples

- Train network
 - `[net,tr] = train(net,P,T)`
 - P: input data
 - T: target data
- Simulate network
 - `[Y,Pf,Af,E,perf] = sim(net,P)`

Matlab Examples

- Batch training: Train
- Stochastic/Incremental Training: Adapt
- Use Train for the CBC