Course 395: Machine Learning - Lectures

Lecture 1-2: Concept Learning (M. Pantic)

Lecture 3-4: Decision Trees & CBC Intro (M. Pantic & S. Petridis)

Lecture 5-6: Evaluating Hypotheses (S. Petridis)

➢ Lecture 7-8: Artificial Neural Networks I (S. Petridis)

Lecture 9-10: Artificial Neural Networks II (S. Petridis)

Lecture 11-12: Instance Based Learning (M. Pantic)

Lecture 13-14: Genetic Algorithms (M. Pantic)

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:

• http://neuralnetworksanddeeplearning.com/
(great online book)

•Coursera classes

- Machine Learning by Andrew Ng
- Neural Networks by Hinton

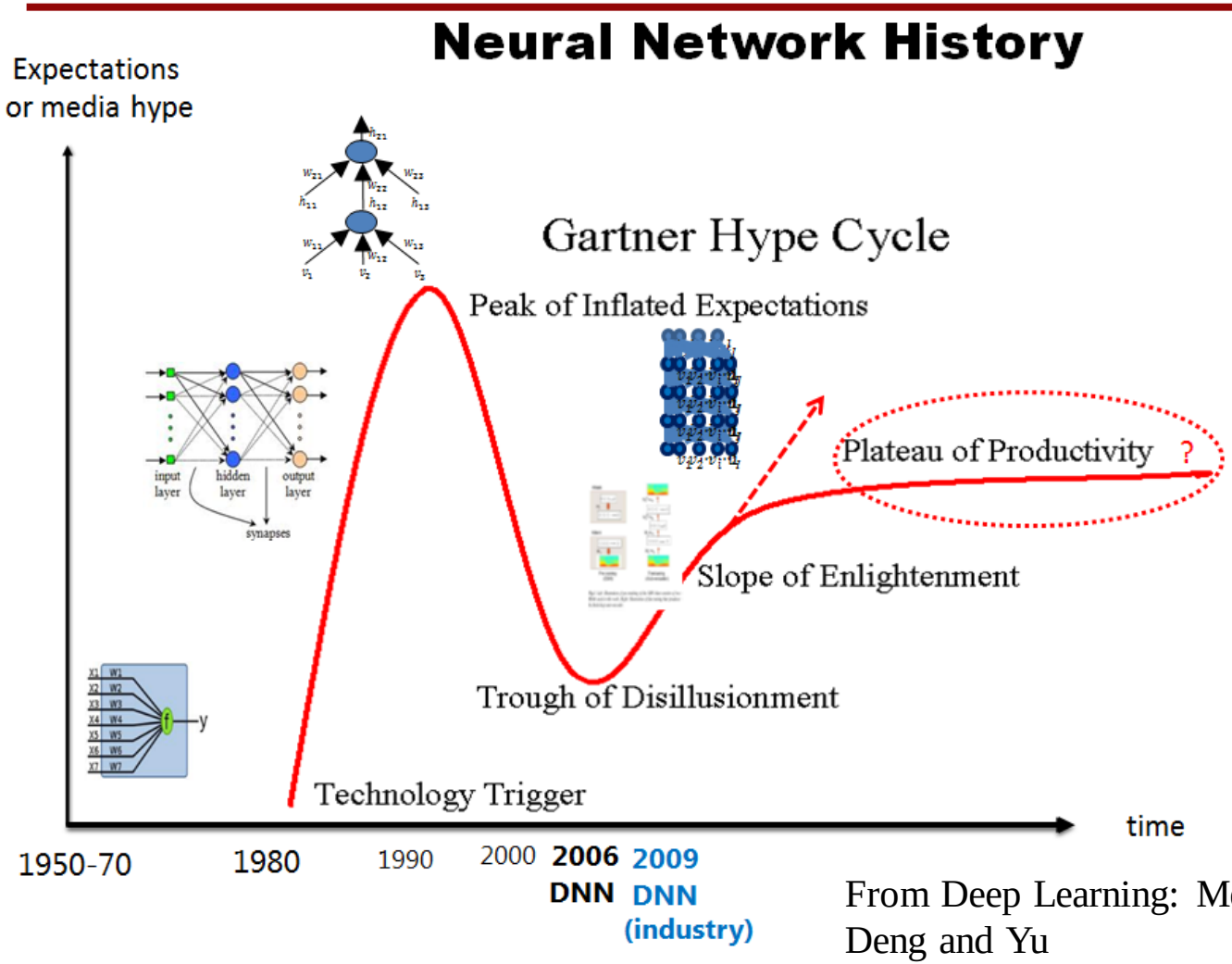
History

- 1st generation Networks: Perceptron 1957 – 1969
  - Perceptron is useful only for examples that are linearly separable

- 2nd generation Networks: Feedforward Networks and other variants, beginning of 1980s to middle/end of 1990s
  - Difficult to train, many parameters, similar performance to SVMs

- 3rd generation Networks: Deep Networks 2006 - ?
  - New approach to train networks with multiple layers
  - State of the art in object recognition / speech recognition

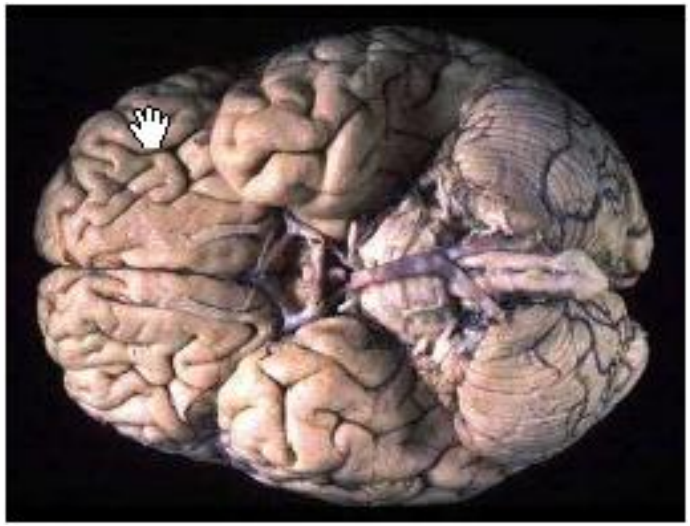
Hype Cycle



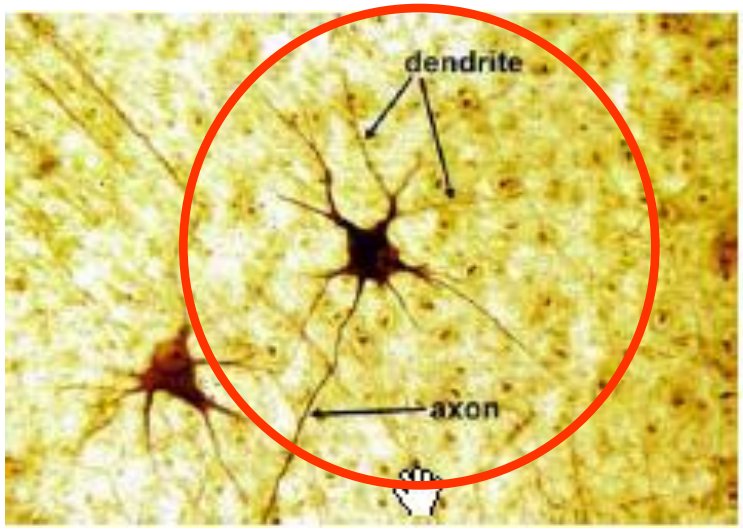
From Deep Learning: Methods and Applications, Deng and Yu

What are Neural Networks?

The real thing!



Billions of neurons

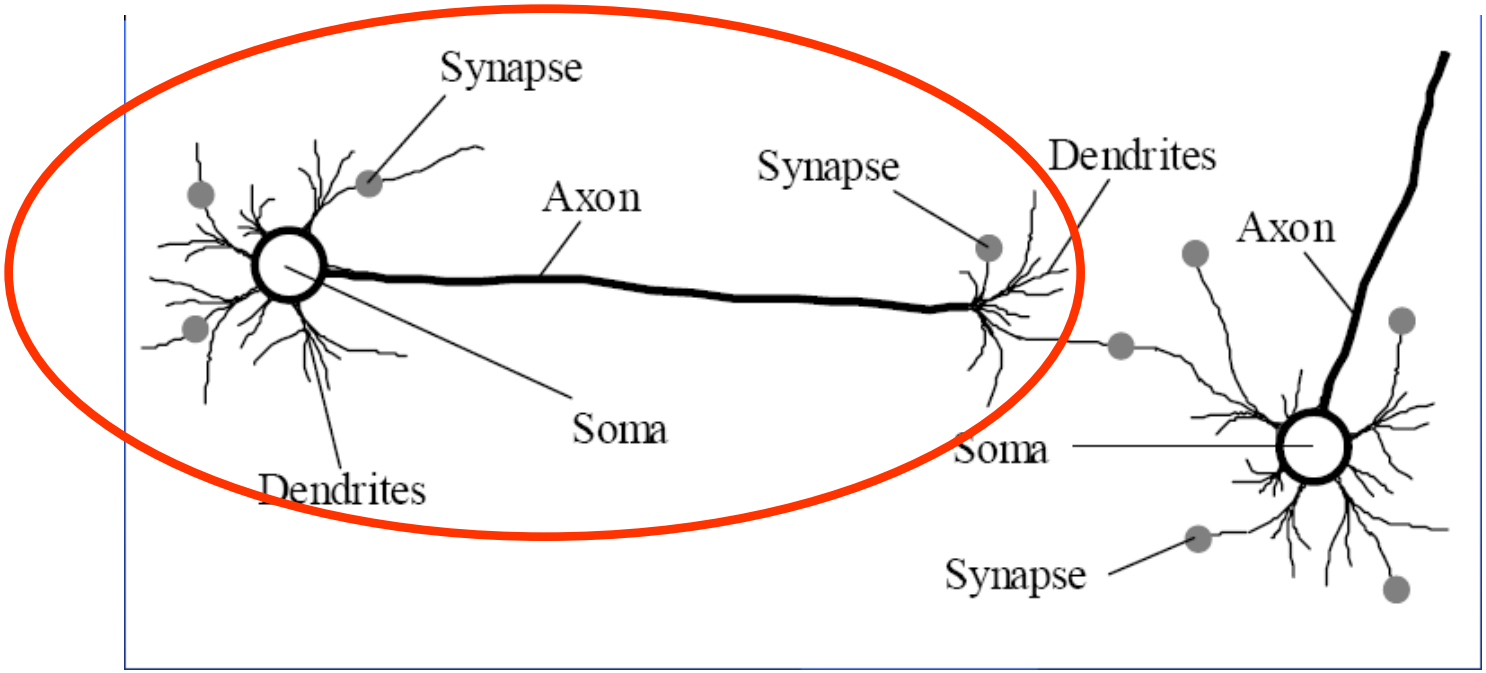


Local computations on **interconnected** elements (neurons)

Parallel computation

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

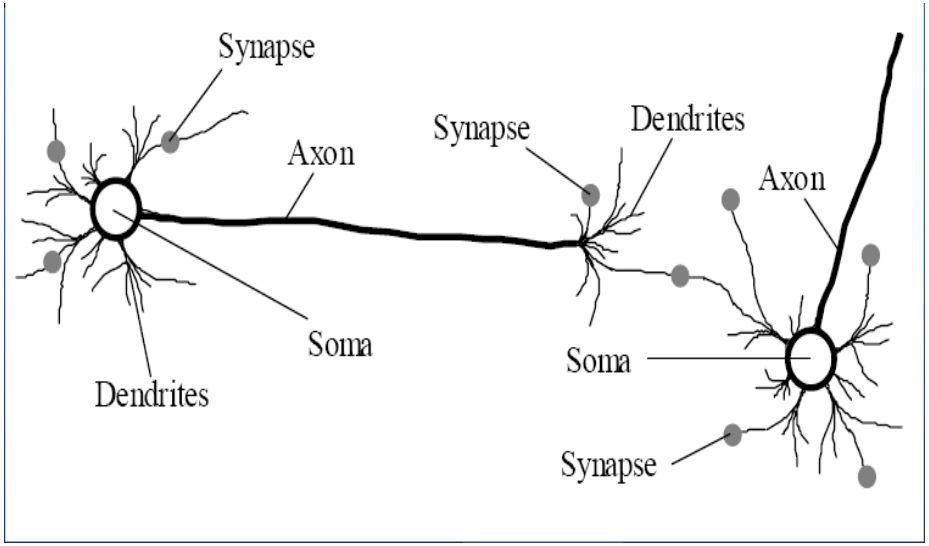
Biological Neural Networks

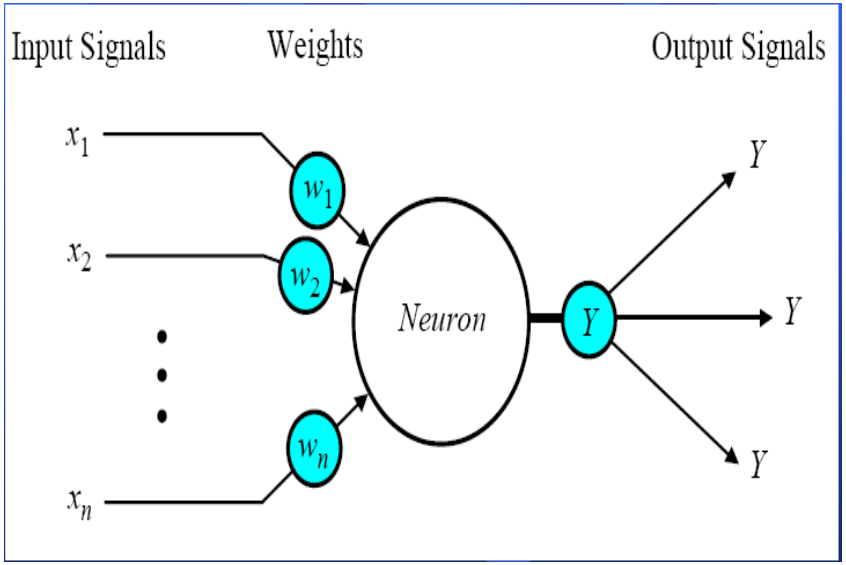


A network of interconnected biological neurons.

Connections per neuron 10^4 - 10^5

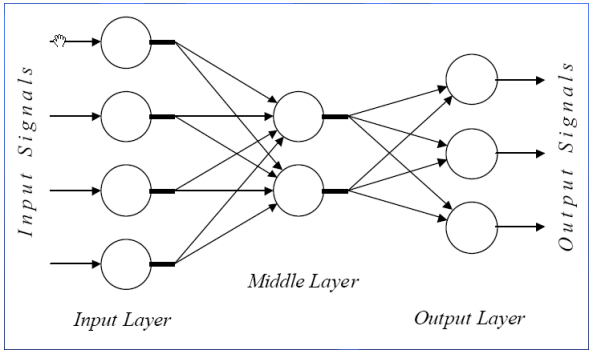
Biological vs Artificial Neural Networks





Biological Neural Network	*Artificial Neural Network*
Soma	Neuron
Dendrite	Input
Axon	Output
Synapse	Weight

Artificial Neural Networks: the dimensions



Architecture

How are the neurons connected

The Neuron

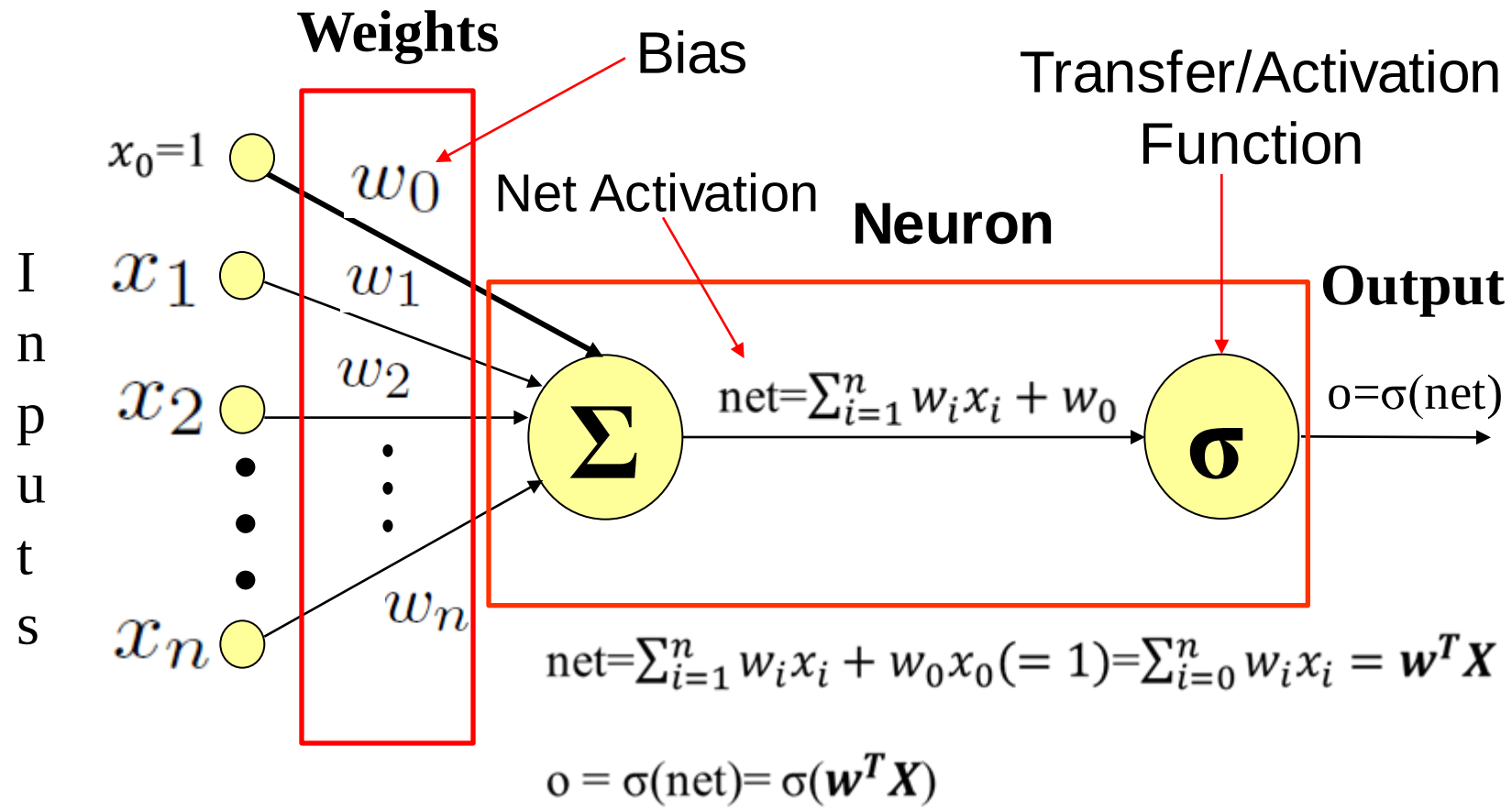
How information is processed in each unit. output = f(input)

Learning algorithms

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

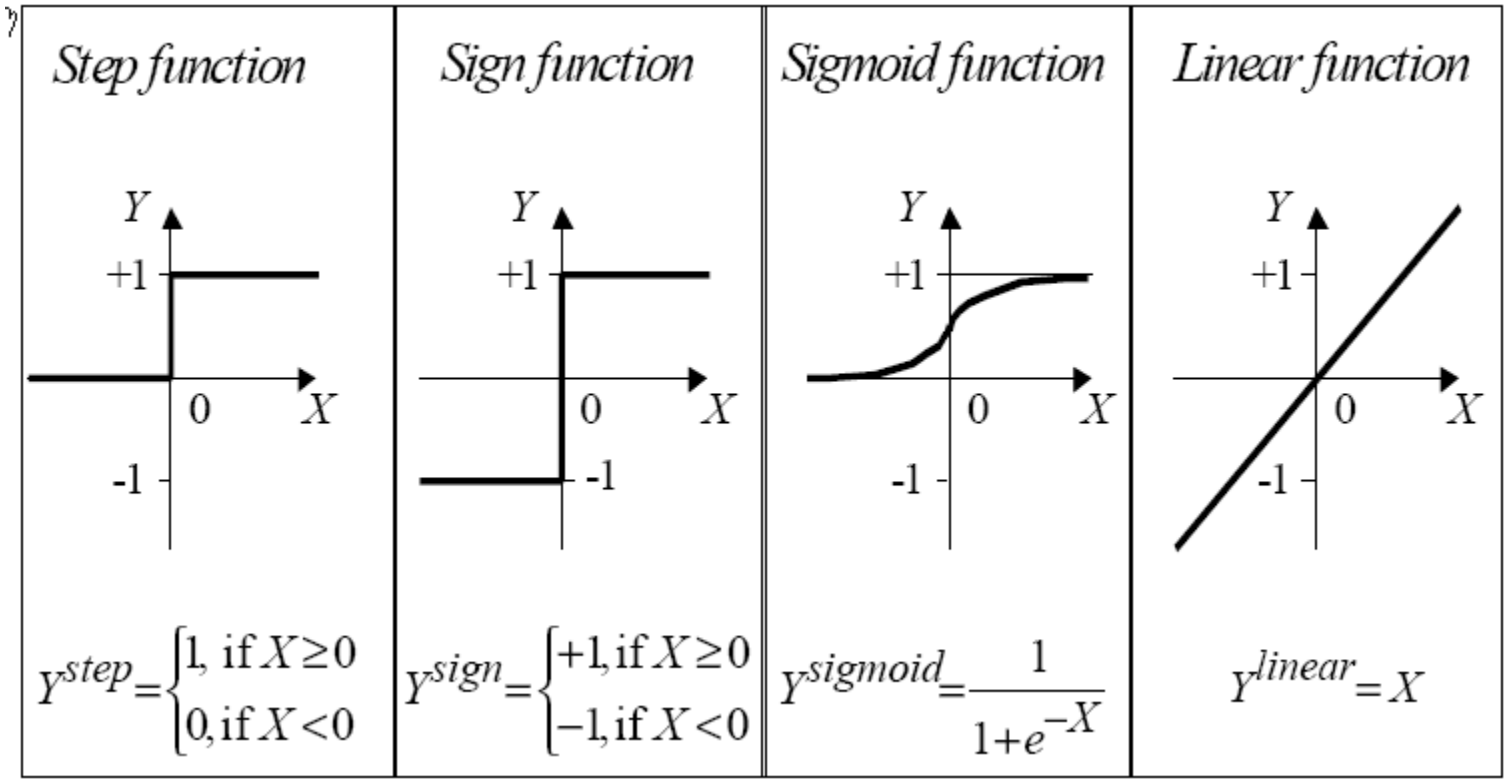
The goal is to have a Neural Network that **generalizes** well, that is, that it generates a 'correct' output on a set of **test/new examples/inputs**.

The Neuron



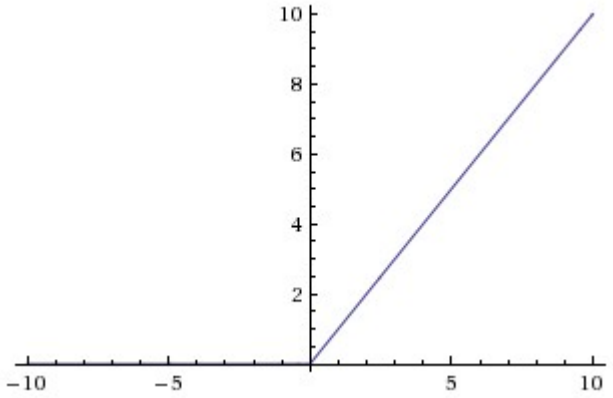
- Main building block of any neural network

Activation functions



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

Activation functions



From http://cs231n.github.io/neural-networks-1/

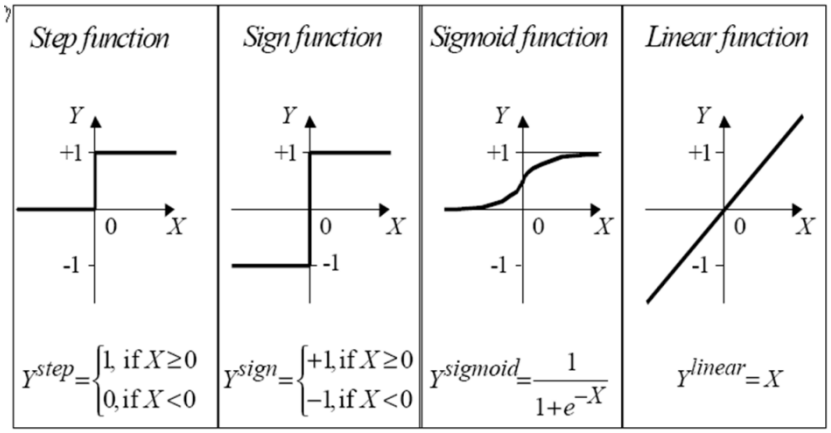
- Rectified Linear Unit (ReLu): max(0, x)
- Popular for deep networks
- Less computationally expensive than sigmoid
- Accelerates convergence during training
- Leaky ReLu: $output = \begin{cases} x & if\ x > 0 \\ 0.01x & otherwise \end{cases}$

Role of Bias

$$net = \sum_{i=1}^{n} w_i x_i + w_0 x_0 (= 1)$$

$$o = \sigma(net)$$

$$w_0 = -\theta$$

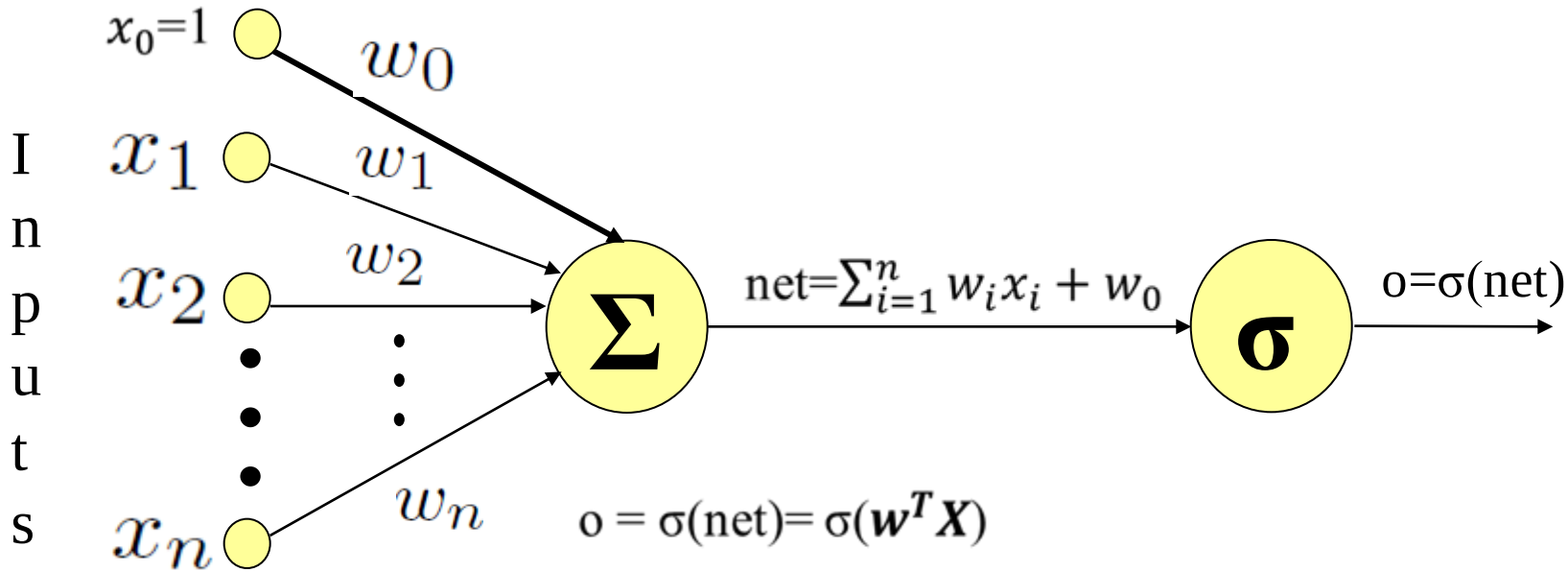


- The threshold where the neuron fires should be adjustable
- Instead of adjusting the threshold we add the bias term
- Defines how strong the neuron input should be before the neuron fires

$$o = \begin{cases} 1 \; if \sum_{i=1}^{n} w_i x_i \geq \theta \\ 0 \; if \sum_{i=1}^{n} w_i x_i < \theta \end{cases}$$

$$o = \begin{cases} 1 \; if \sum_{i=1}^{n} w_i x_i - \theta \geq 0 \\ 0 \; if \sum_{i=1}^{n} w_i x_i - \theta < 0 \end{cases}$$

Perceptron



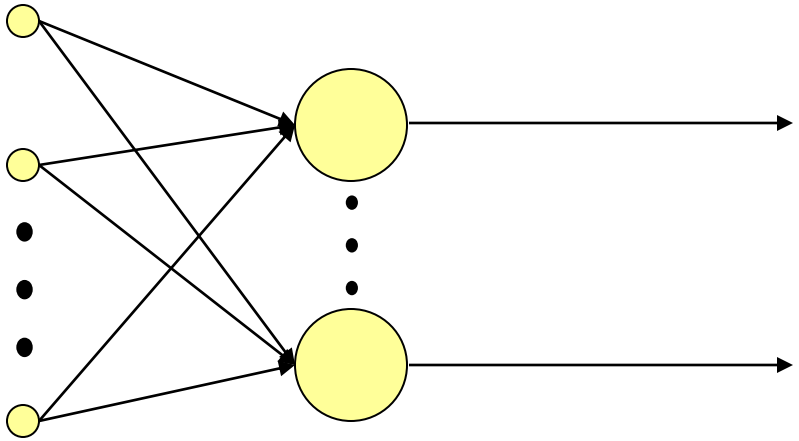
$$o = \sigma(net) = \sigma(\boldsymbol{w}^T \boldsymbol{X})$$

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

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

Perceptron: Architecture

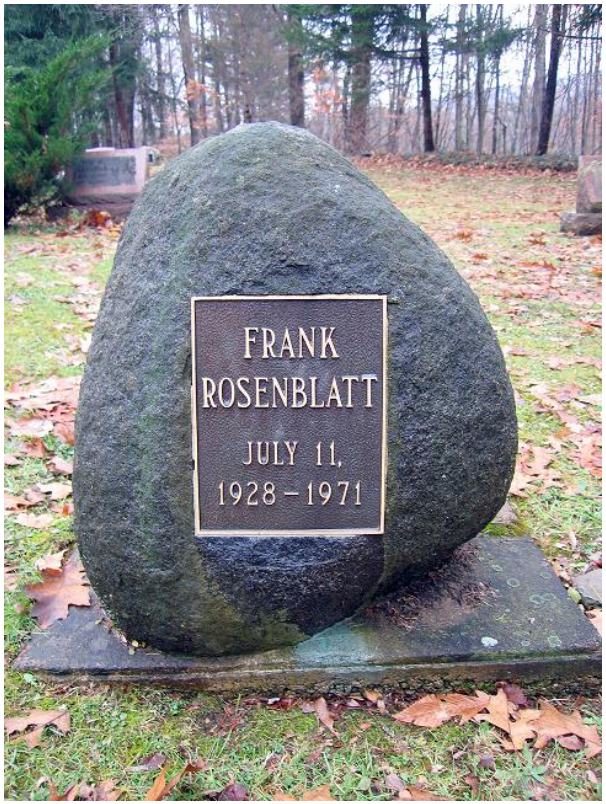
- Generalization to single layer perceptrons with more neurons is easy because:



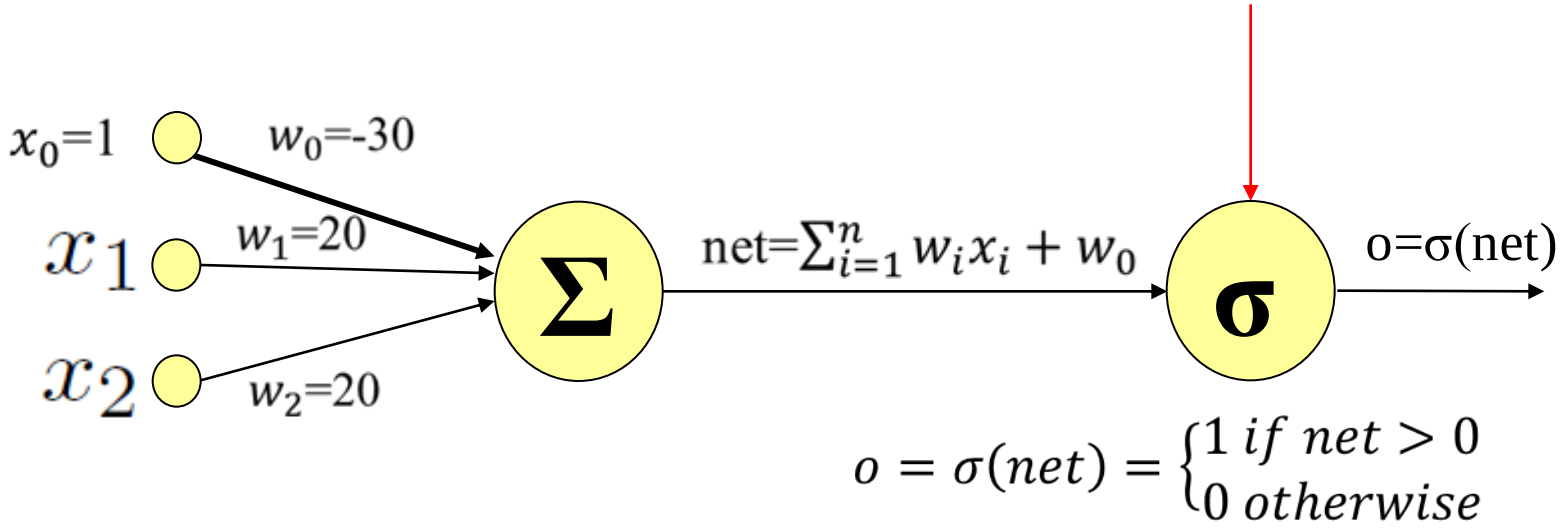
- The output units are mutually independent
- Each weight only affects one of the outputs

Perceptron

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

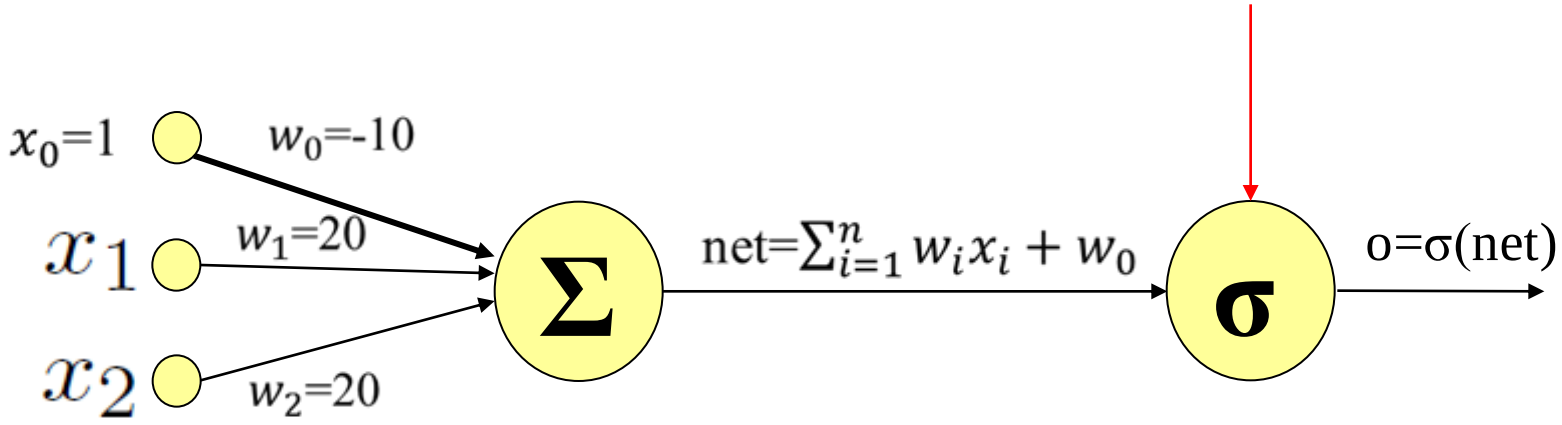


Perceptron: Example 1 - AND



- x1 = 1, x2 = 1→ net = 20+20-30=10 →o = σ(10) = 1
- x1 = 0, x2 = 1→ net = 0+20-30 =-10 →o = σ(-10) = 0
- x1 = 1, x2 = 0→ net = 20+0-30 =-10 →o = σ(-10) = 0
- x1 = 0, x2 = 0→ net = 0+0-30 =-30 →o = σ(-10) = 0

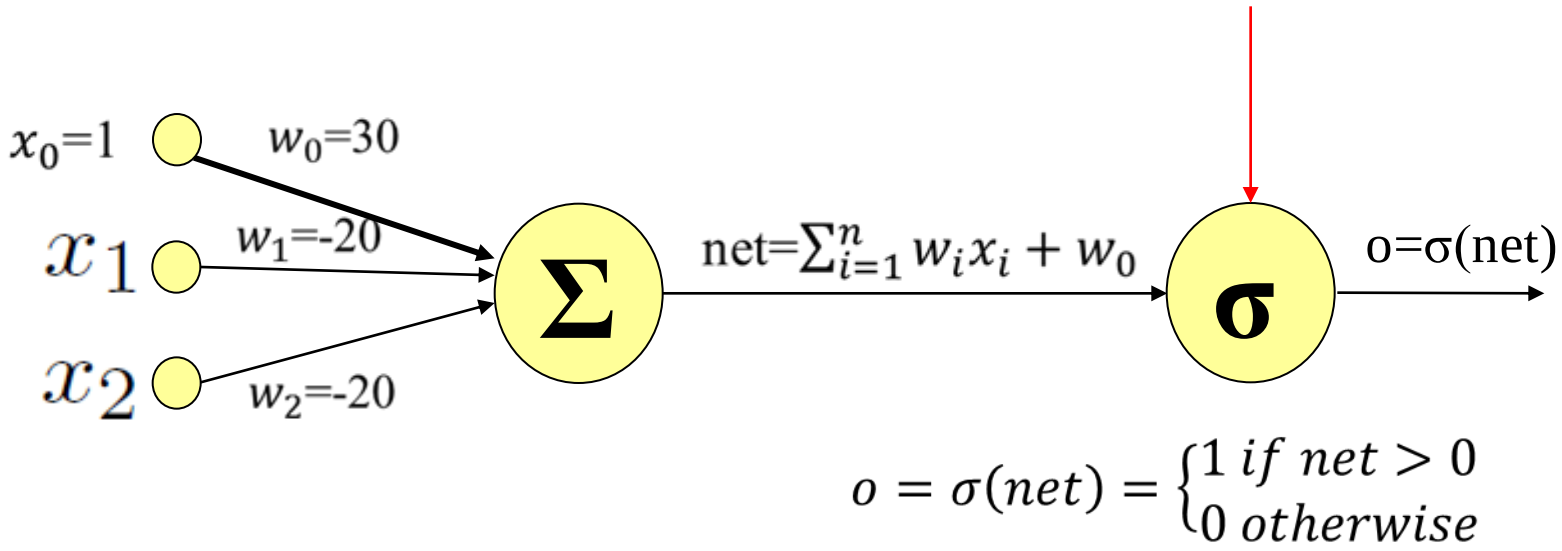
Perceptron: Example 2 - OR



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

- x1 = 1, x2 = 1→ net = 20+20-10=30 →o = σ(30) = 1
- x1 = 0, x2 = 1→ net = 0+20-10 =10 →o = σ(10) = 1
- x1 = 1, x2 = 0→ net = 20+0-10 =10 →o = σ(10) = 1
- x1 = 0, x2 = 0→ net = 0+0-10 =-10 →o = σ(-10) = 0

Perceptron: Example 3 - NAND



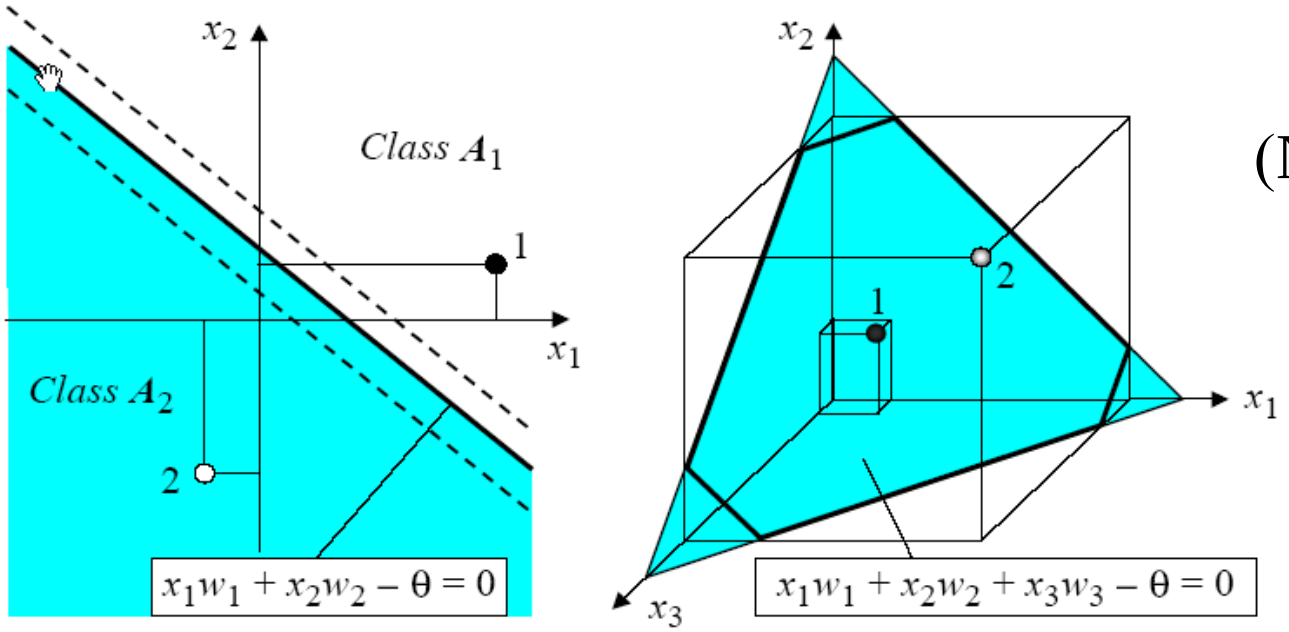
- x1 = 1, x2 = 1→ net = -20-20+30=-10 → o = σ(-10) = 0
- x1 = 0, x2 = 1→ net = 0-20+30 =10 → o = σ(10) = 1
- x1 = 1, x2 = 0→ net = -20+0+30 =10 → o = σ(10) = 1
- x1 = 0, x2 = 0→ net = 0+0+30 =30 → o = σ(30) = 1

Perceptron for classification

- Given training examples of classes A1, A2 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 A1 (i.e. if*** $\sigma(\mathbf{w}^T\mathbf{x}) = 1$ ***)***
  - ***If the output is 0 then the input is assigned to class A2***

• 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.

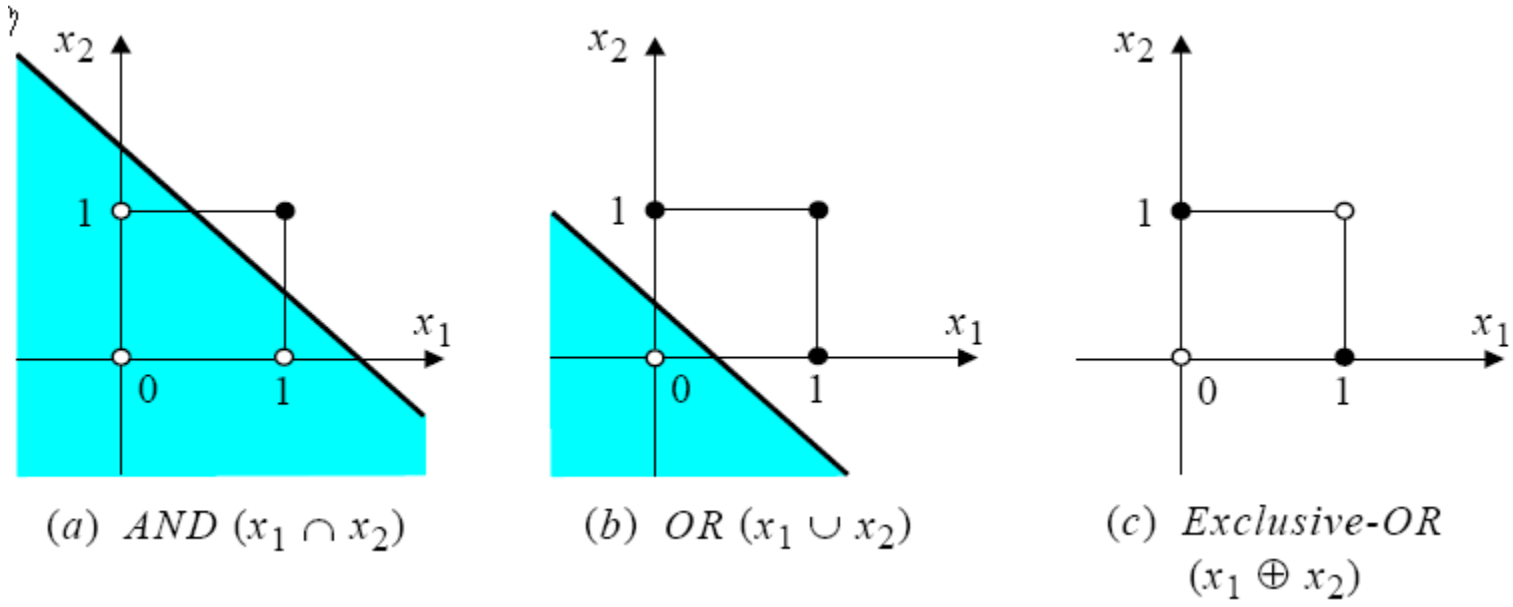
(Note that $\theta = -w_0$)

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

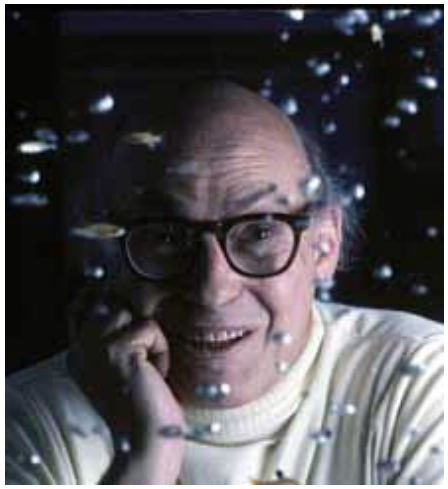


- Perceptron can only classify examples that are linearly separable
- The XOR is not linearly separable.
- This was a terrible blow to the field

Perceptron

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

- Marvin Minsky



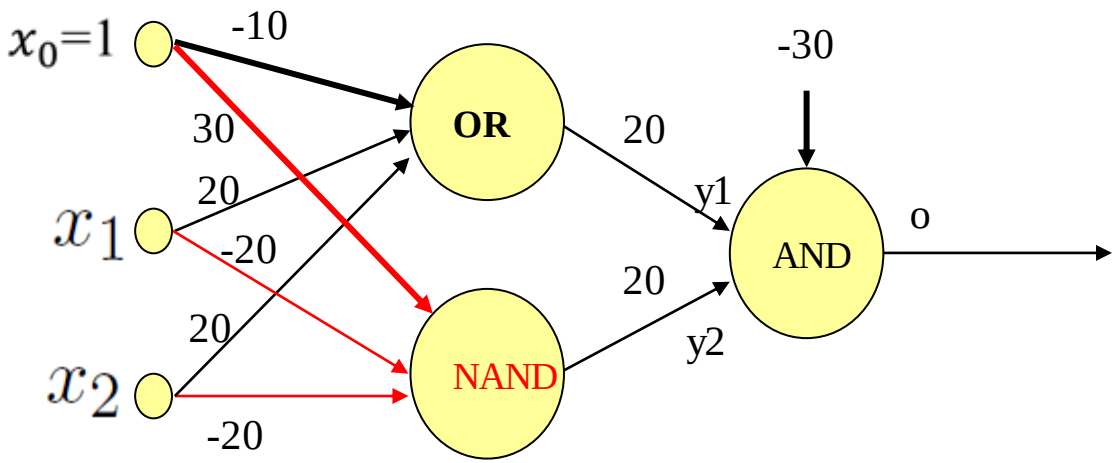
- Seymour Papert

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
- XOR = NAND (AND) OR

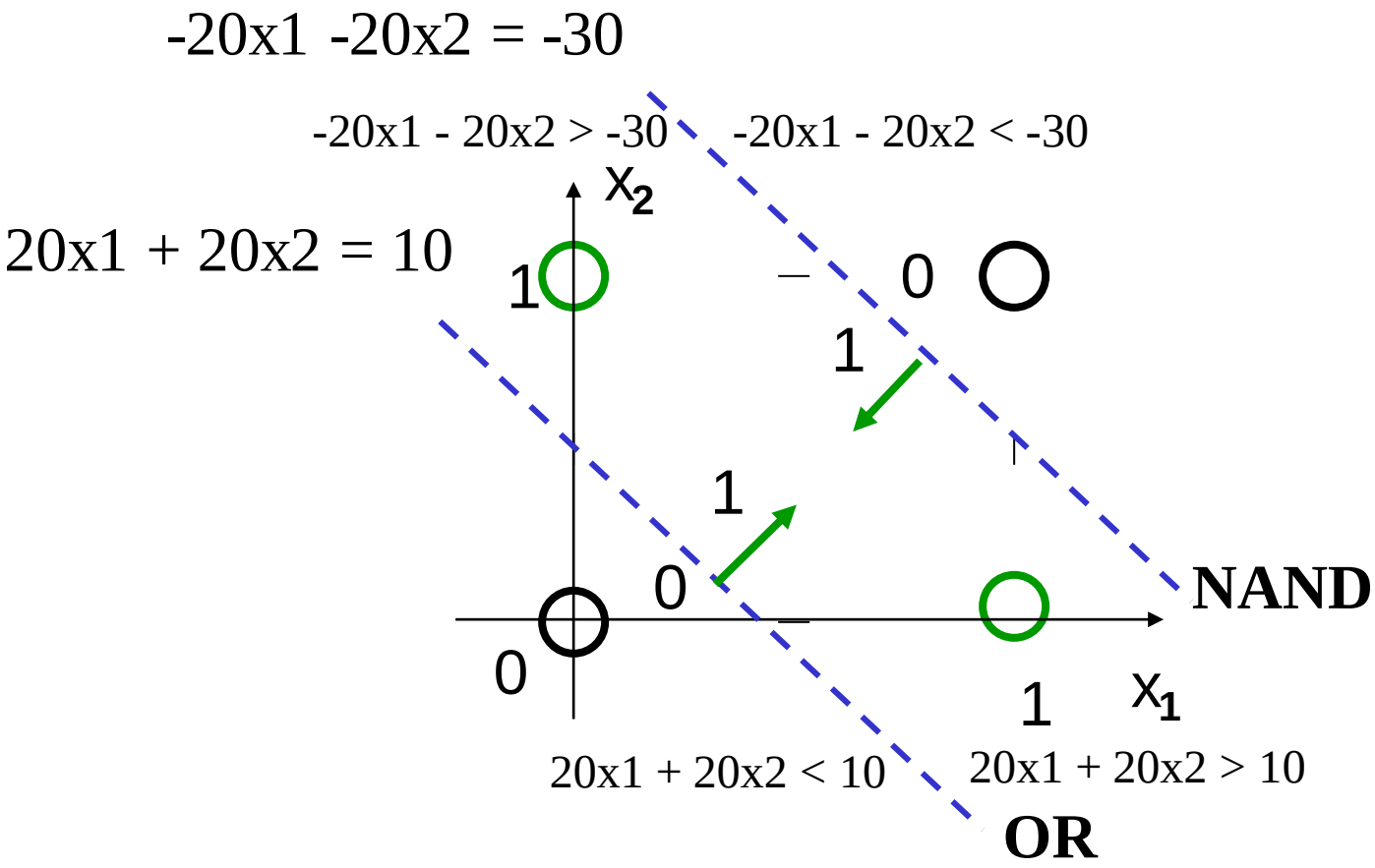


OR	NAND
1 1 → 1	1 1 → 0
0 1 → 1	0 1 → 1
1 0 → 1	1 0 → 1
0 0 → 0	0 0 → 1

AND
1 1 → 1
0 1 → 0
1 0 → 0
0 0 → 0

- x1=1, x2 =1→ y1=1 AND y2=0 → o = 0
- x1=1, x2 =0→ y1=1 AND y2=1 → o = 1
- x1=0, x2 =1→ y1=1 AND y2=1 → o = 1
- x1=0, x2 =0→ y1=0 AND y2=1→ o = 0

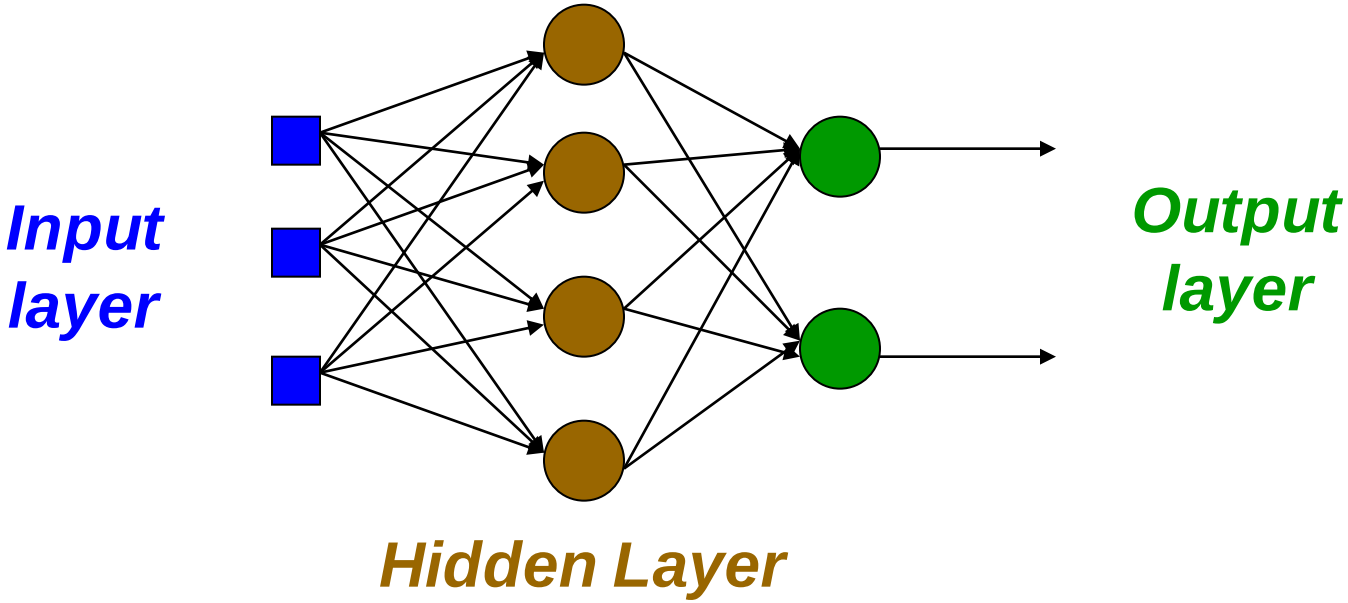
XOR


-20x1 -20x2 = -30
-20x1 - 20x2 > -30
-20x1 - 20x2 < -30
20x1 + 20x2 = 10
x2
x1
1
0
1
1
0
0
1
NAND
20x1 + 20x2 < 10
20x1 + 20x2 > 10
OR

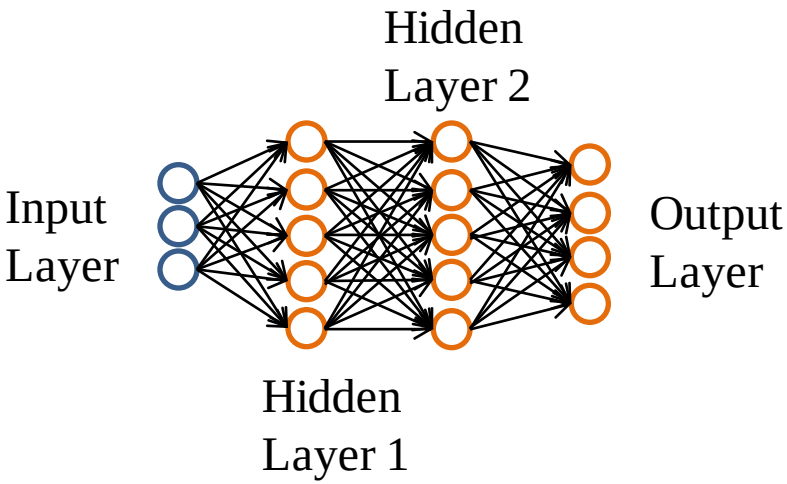
Multilayer Feed Forward Neural Network

- We consider a more general network architecture: between the input and output layers there are hidden layers, as illustrated below.

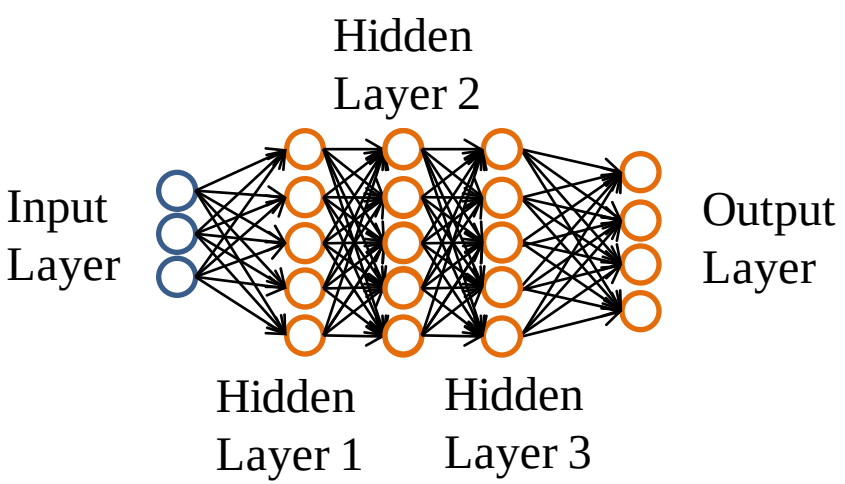
- Hidden nodes do not directly receive inputs nor send outputs to the external environment.



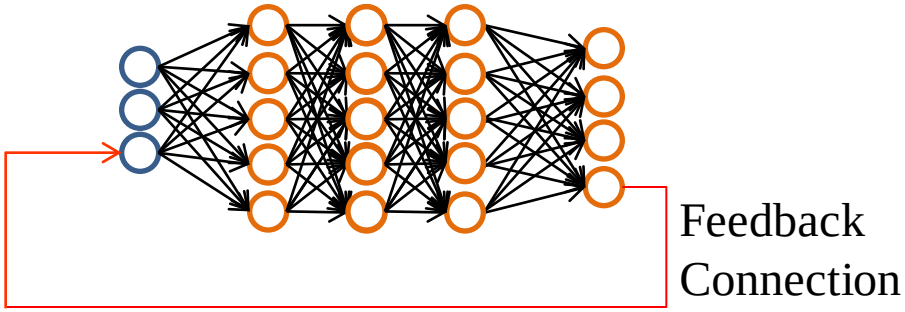
NNs: Architecture



3-layer feed-forward network



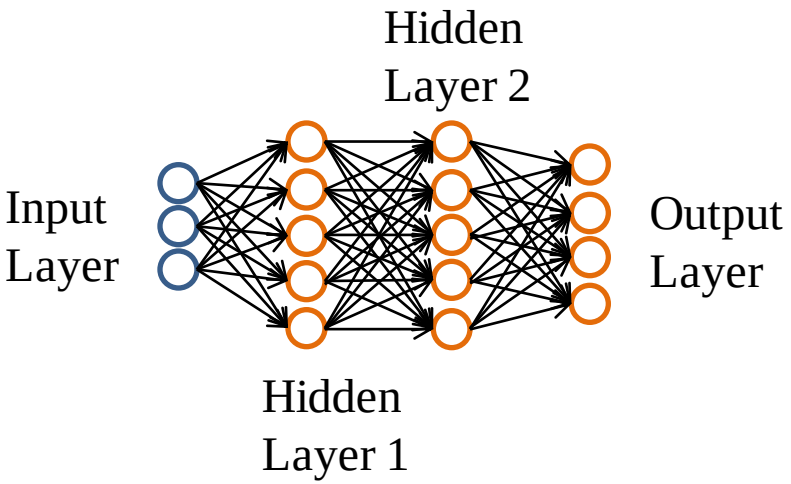
4-layer feed-forward network



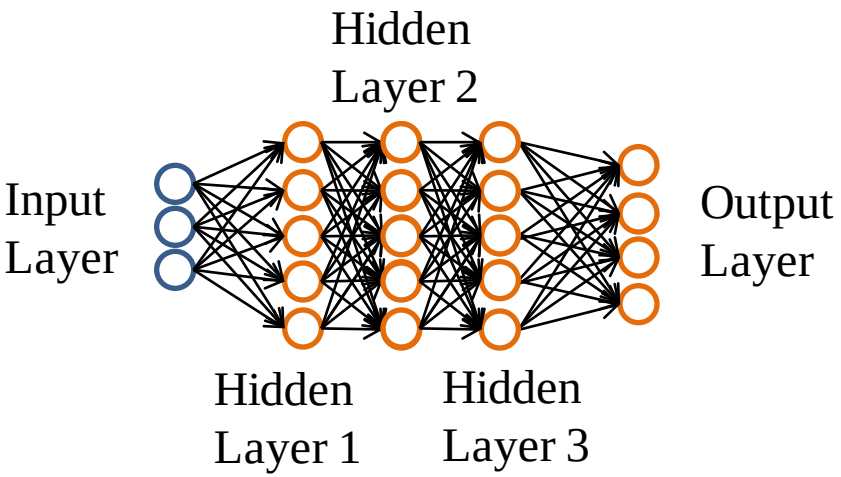
- The input layer does not count as a layer

4-layer recurrent network – Difficult to train

NNs: Architecture



3-layer feed-forward network

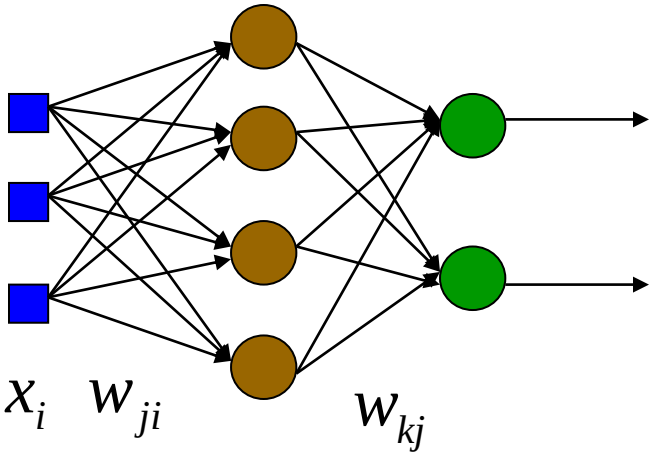


4-layer feed-forward network

- Deep networks are simply networks with many layers.

- They are trained in the same way as shallow networks but
  1) either weight initialisation is done in a different way.
  2) or we use a lot of data with strong regularisation

Multilayer Feed Forward Neural Network



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

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

y_j = output of jth hidden unit

o_k = output of kth 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

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

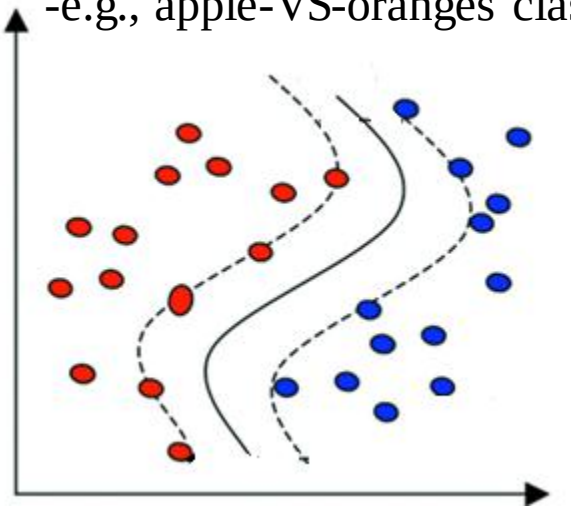
Classification / Regression with NNs

- You should think of neural networks as function approximators

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

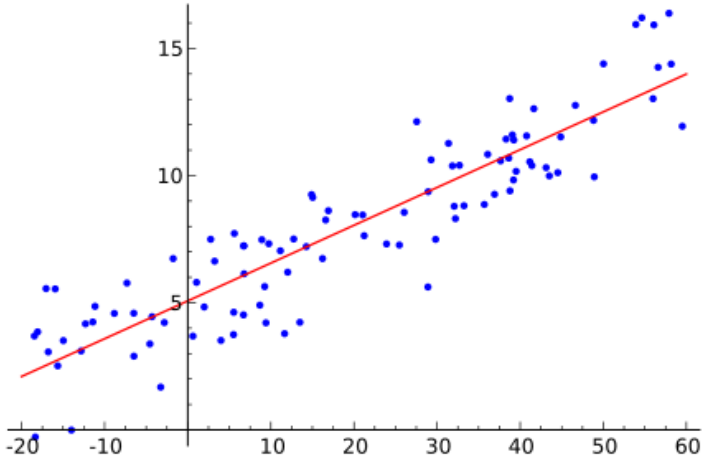
Classification

-Decision boundary approximation
-Discrete output
-e.g., apple-VS-oranges classifier

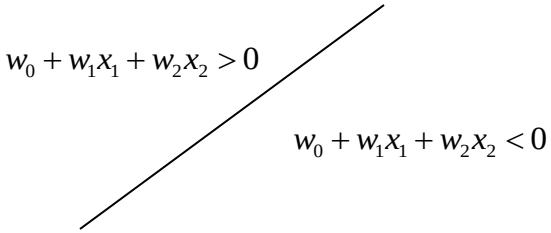


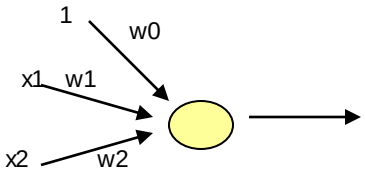
Regression

- Function approximation
- Continuous output
- e.g., house price estimation

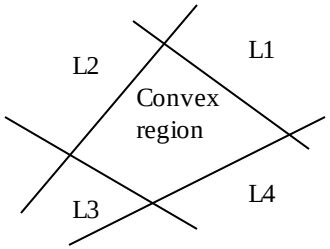


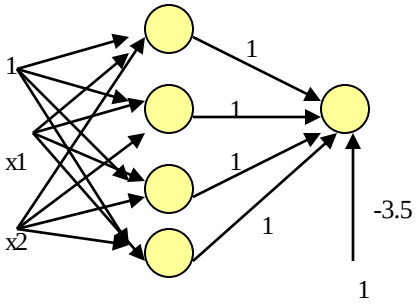
Decision boundaries



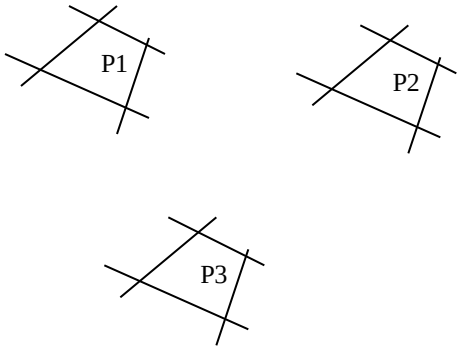


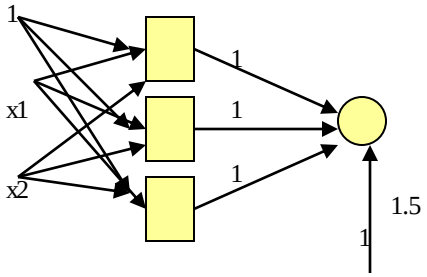
Network with a single node





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 or -1 (negative) and 1 (positive)

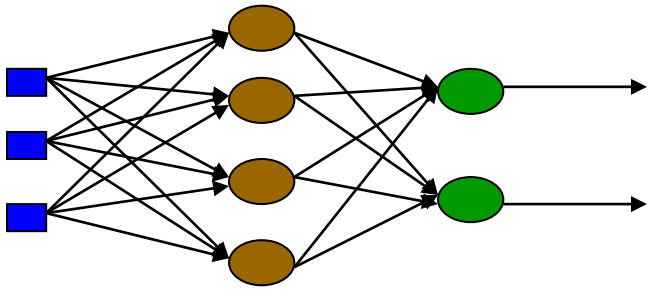
- Regression

  Target values (t): continuous values [-inf, +inf]

- 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)
e.g. for 4 classes the targets are the following:

Class1 Class2 Class3 Class4

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

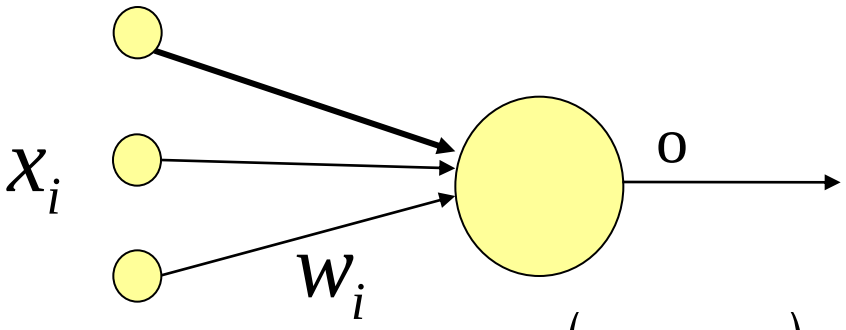
Training

- 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 Rule



- Training Set: A set of input vectors $x_i, i = 1 \ldots n$ with the corresponding targets t_i

- η: learning rate, controls the change rate of the weights

- Begin with random weights
- Change the weights whenever an example is misclassified

$$w_i \leftarrow w_i + \Delta w_i$$
$$\Delta w_i = \eta (t_i - o) x_i$$

- This rule works if the examples are linearly seperable
  - for every input vector *x(i)* the output is the desired target o = t

Training – Gradient Descent

• In most problems the training examples are NOT linearly seperable. Therefore, we need a different approach to adjust the weights → Gradient Descent

• Gradient Descent: A general, effective way for estimating parameters (e.g. **w**) that minimise error functions

• We need to define an error function E(w)

• Update the weights in each iteration in a direction that reduces the error the order in order to minimize E

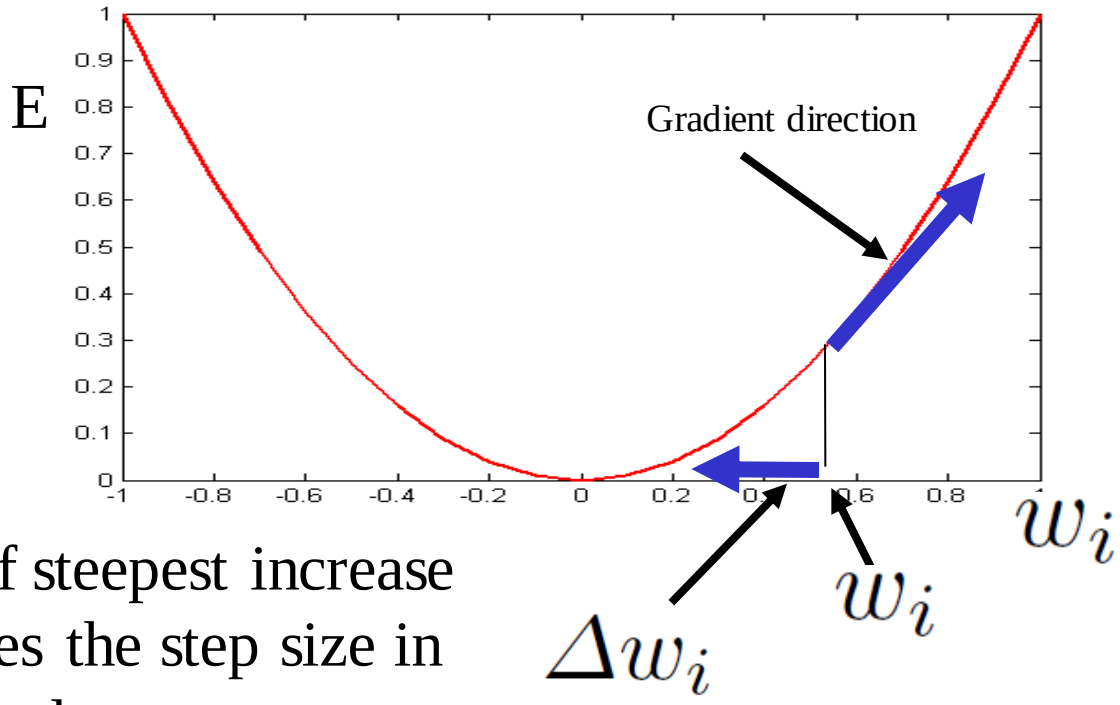
$$w_i \leftarrow w_i + \Delta w_i$$

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

Gradient Descent

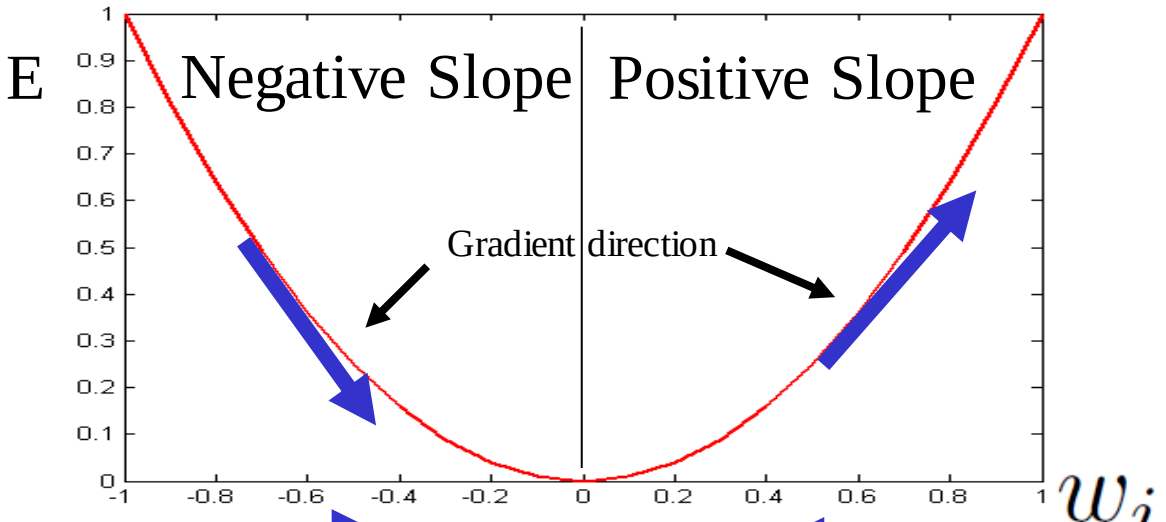
Gradient descent method: take a step in the direction that decreases the error E. This direction is the opposite of the derivative of E.

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



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

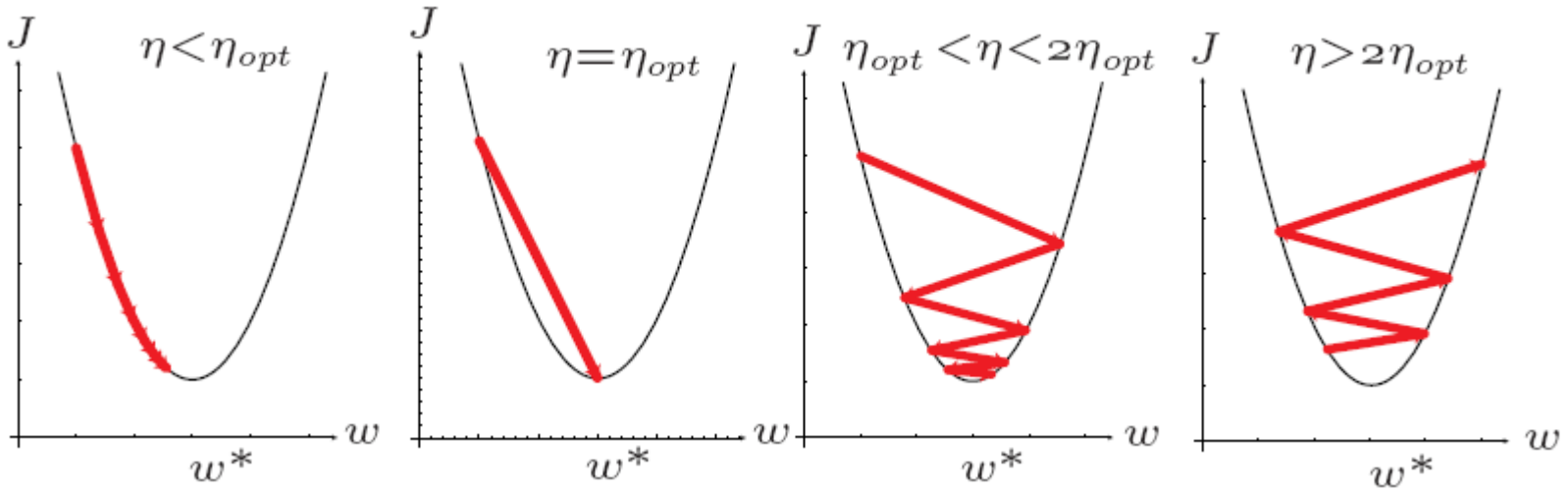
Gradient Descent – Learning Rate



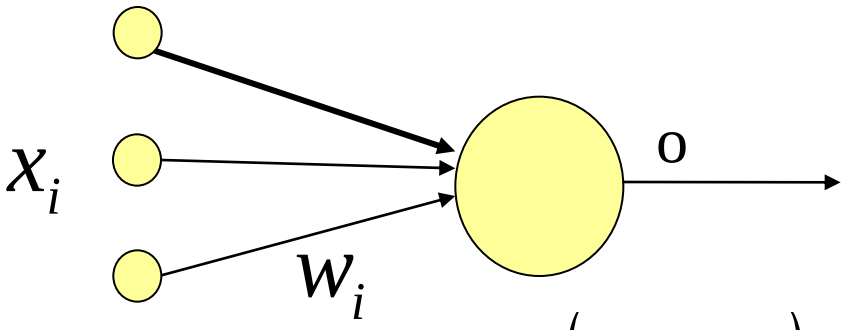
$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} > 0 \qquad \Delta w_i = -\eta \frac{\partial E}{\partial w_i} < 0$$

- Derivative: direction of steepest increase
- Learning rate: determines the step size in the direction of steepest decrease. It usually takes small values, e.g. 0.01, 0.1
- If it takes large values then the weights change a lot -> network unstable

Gradient Descent –Learning Rate



Gradient Descent – One neuron only



- Training Set: A set of input vectors $x_i, i = 1 \ldots n$ with the corresponding targets t_i

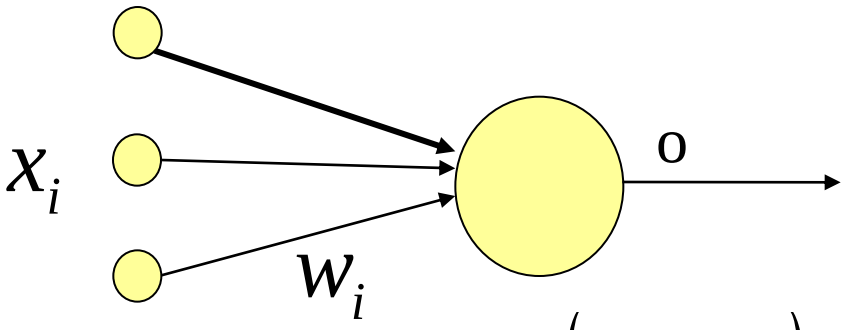
- η: learning rate, controls the change rate of the weights

- Begin with random weights
- We define our error function as follows (D = number of training examples)

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

- E depends on the weights because $o_d = \sum_{i=0}^{n} x_i^d w_i$

Gradient Descent – One neuron only



- Training Set: A set of input vectors $x_i, i = 1 \ldots n$ with the corresponding targets t_i

- η: learning rate, controls the change rate of the weights

• For simplicity we assume linear activation transfer

• We wish to find the weight values that minimise E, i.e. the desired target *t* is very close to the actual output *o*.

$$E = \frac{1}{2}\sum_{d=1}^{D}\left(t_d - o_d\right)^2 \qquad o_d = \sum_{i=0}^{n} x_i^d w_i$$

Gradient Descent

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

- 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 –One Neuron Only – 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

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

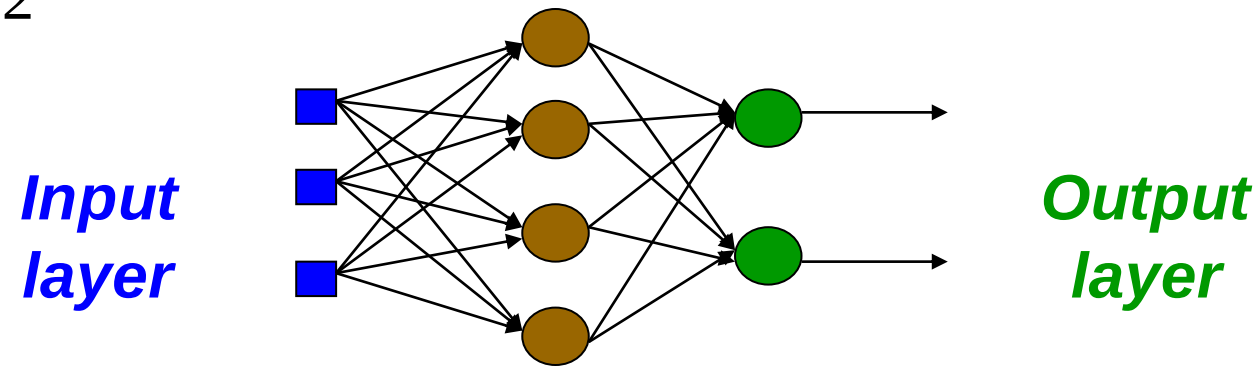
- Repeat steps 2-4 until a termination condition is met

- The algorithm will converge to a weight vector with minimum error, given that the learning rate is sufficiently small

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).

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



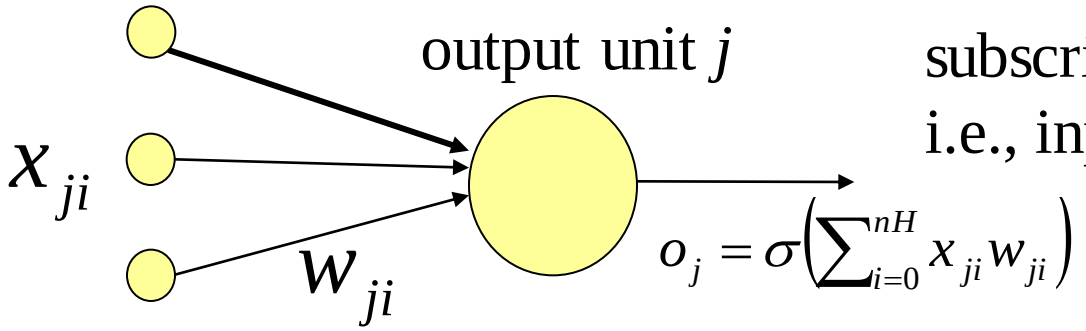
- Based on gradient descent algorithm

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

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: ith input to unit j, i.e., input from hidden neuron i

- 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

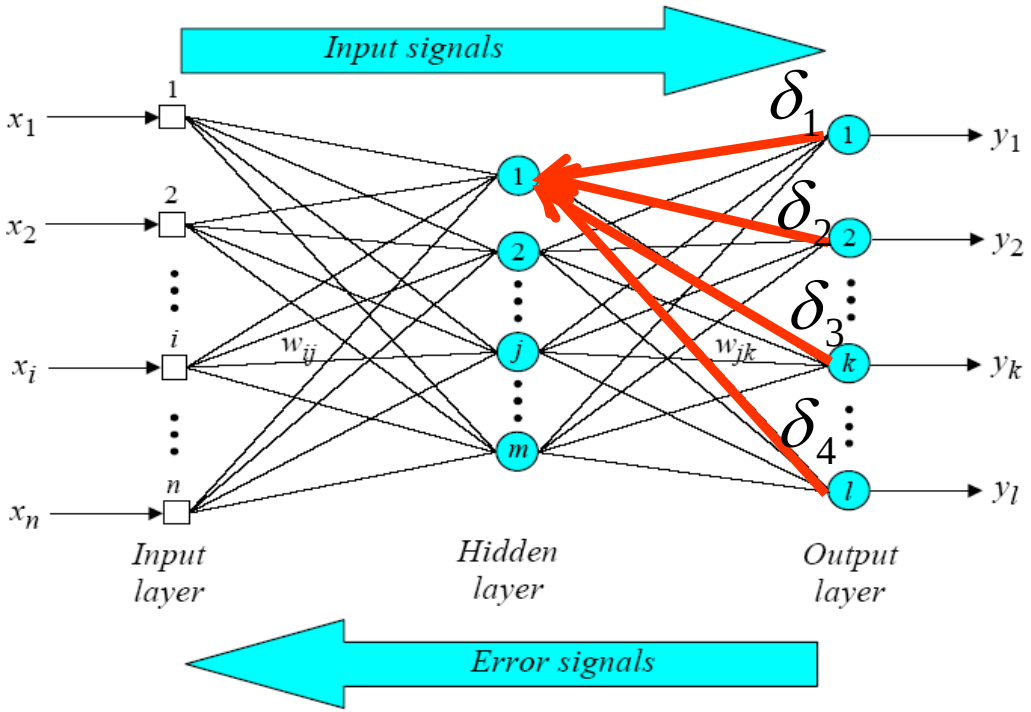
$$\Delta w_{ji} = \underbrace{\eta(t_j - o_j)x_{ji}\frac{\partial\sigma(net_j)}{\partial net_j}}_{-\partial E_d/\partial w_{ji}} \qquad \Delta w_i = \eta\sum_{i=1}^{d}(t_d - o_d)x_{id}$$

perceptron Δw

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

Learning: Hidden Layer Weights

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



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

- The error term for hidden unit j is:

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

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: $\Delta w_{ki} = \eta(t_k - o_k)x_{ki}\frac{\partial \sigma(net_k)}{\partial net_k} = \eta \delta_k x_{ki}$

- x is the input to output unit k from hidden unit i

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

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

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

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

Example

- http://galaxy.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html

Learning: Backpropagation Algorithm

- Finally for all training examples D

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

- This is called batch training because the weights are updated after all training examples have been presented to the network (=epoch)
- Matlab function: **traingd**

- Incremental training: weights are updated after each training example is presented

Backpropagation: Fundamental Equations

- So far we have considered the MSE as our error function

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

- What if we wish to use another error function?

- We simply compute the derivatives again $\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$

- For output units: $\frac{\partial E}{\partial w_{ki}} = \frac{\partial E}{\partial o_k}\frac{\partial o_k}{\partial net_k}\frac{\partial net_k}{\partial w_{ki}} = \frac{\partial E}{\partial o_k}\frac{\partial \sigma(net_k)}{\partial net_k}x_{ki}$

- We can redefine $\delta_k = \frac{\partial E}{\partial o_k}\frac{\partial \sigma(net_k)}{\partial net_k}$

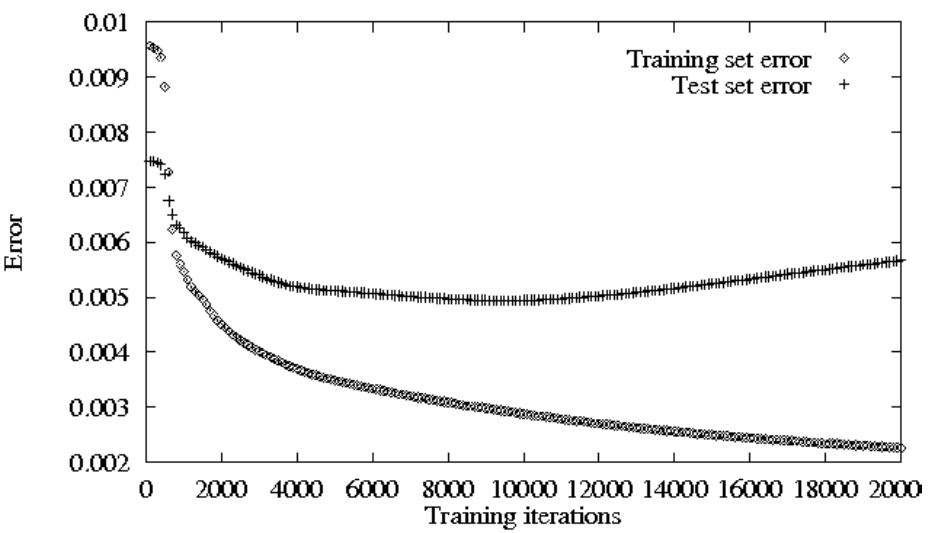
Backpropagation: Fundamental Equations

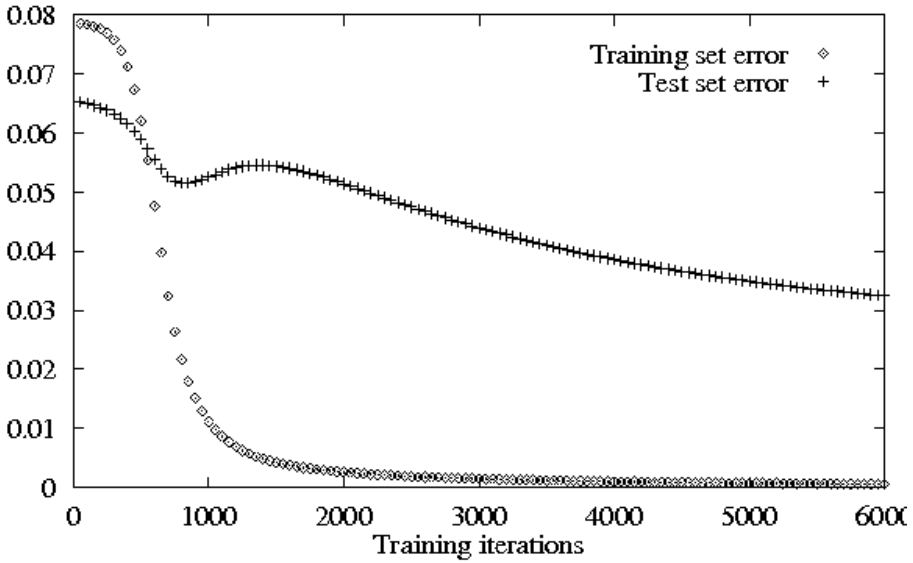
- For output units: $\Delta w_{ki} = -\eta \frac{\partial E}{\partial w_{ki}} = -\eta \delta_k x_{ki}$
- $\delta_k = \frac{\partial E}{\partial o_k} \frac{\partial \sigma(net_k)}{\partial net_k}$
- x is the input to output unit k from hidden unit I
- For hidden units: $\Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}} = -\eta \delta_j x_{ji}$
- $\delta_j = \sum_{k=outputNeuronsConnectedToj} \delta_k w_{kj} \frac{\partial \sigma(net_j)}{\partial net_j}$
- x is the input to hidden unit j from hidden unit i from the previous hidden layer (or input layer if it's the first hidden layer)

Backpropagation Stopping Criteria

- When the gradient magnitude (or Δw_i) is small, i.e. $\frac{\partial E}{\partial w_i} < \delta \; or \; \Delta w_i < \delta$
- 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). This is called early stopping.

Early stopping





- Stop when the error in the validation set increases (but not too soon!)
- Error might decrease in the training set but increase in the 'validation' set (overfitting!)
- It is also a way to avoid overfitting.

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)

$$\Delta w_{ki} = -\eta \frac{\partial E}{\partial w_{ki}} = -\eta \delta_k x_{ki} \quad where \quad \delta_k = \frac{\partial E}{\partial o_k} \frac{\partial \sigma(net_k)}{\partial net_k}$$

4. Compute hidden neurons errors and then compute the update rule for hidden layer weights (backward pass)

$$\Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}} = -\eta \delta_j x_{ji} \; where \; \delta_j = \sum_{k=outputNeuronsConnectedToj} \delta_k w_{kj} \frac{\partial \sigma(net_j)}{\partial net_j}$$

Backpropagation Summary

5. Compute the sum of all Δw, once all training examples have been presented to the network

6. Update weights $w_i \leftarrow w_i + \Delta w_i$

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

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