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
(chapters 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

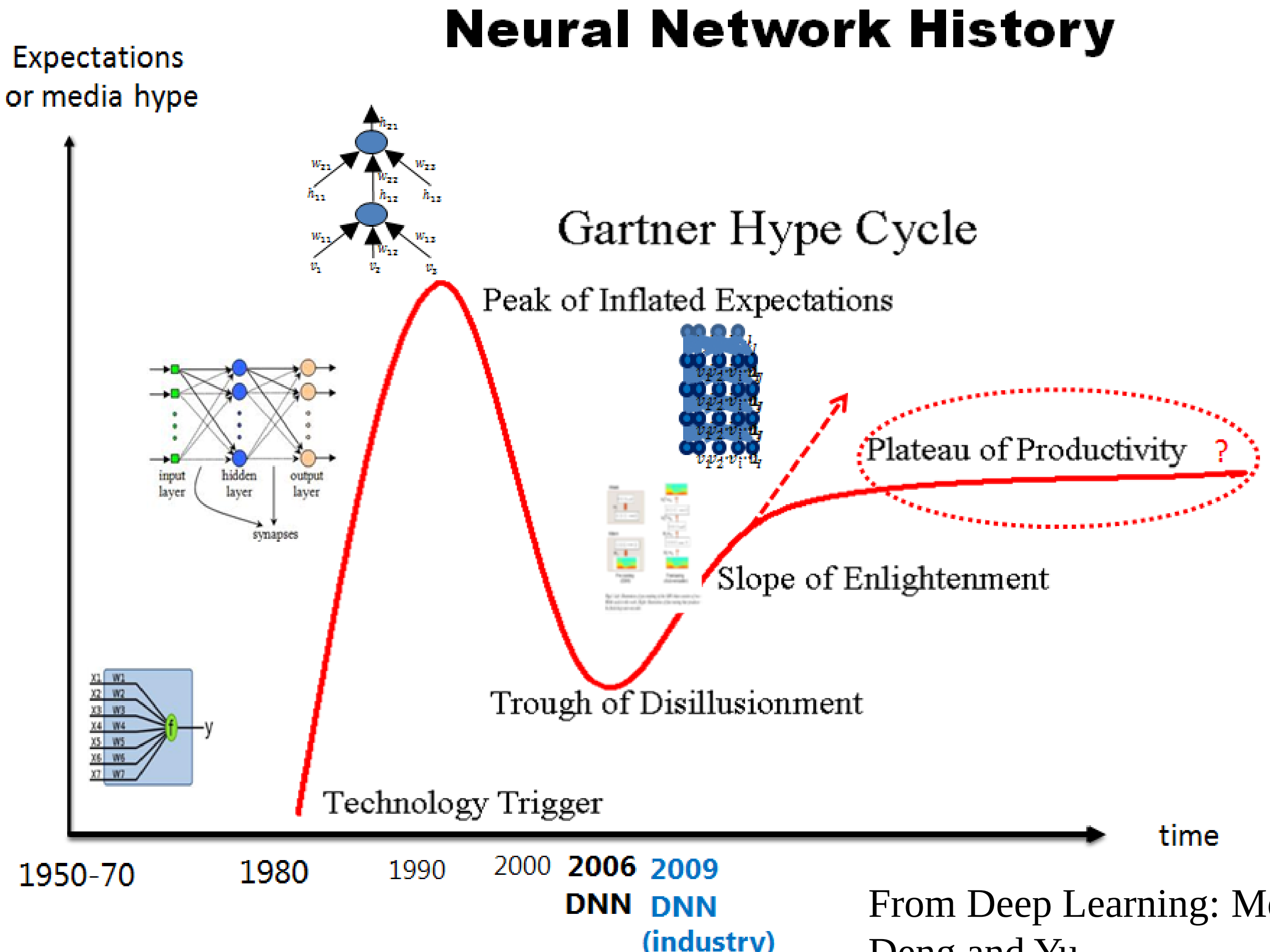
History

- 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

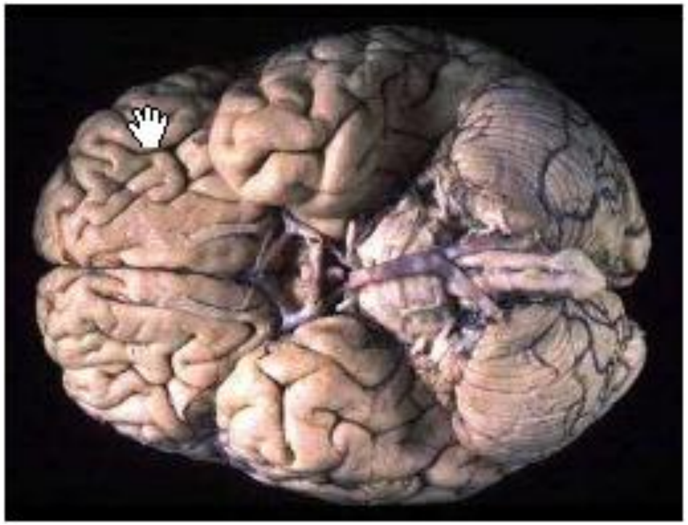
Hype Cycle



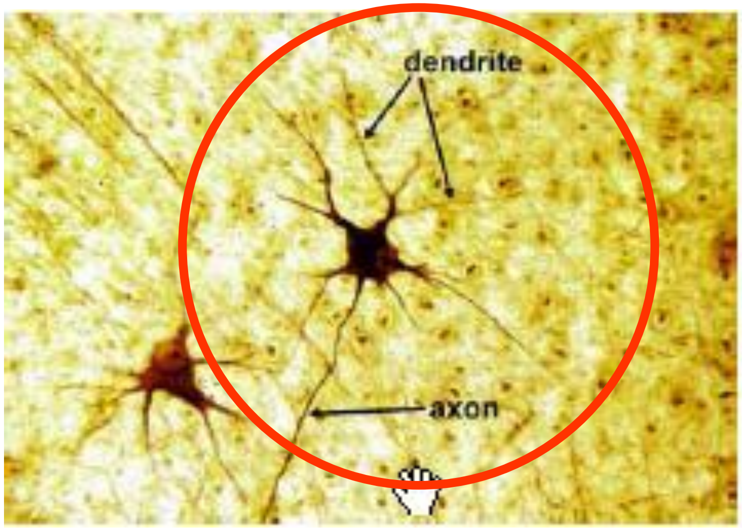
From Deep Learning: Methods and Applications, Deng and Yu

What are Neural Networks?

The real thing!



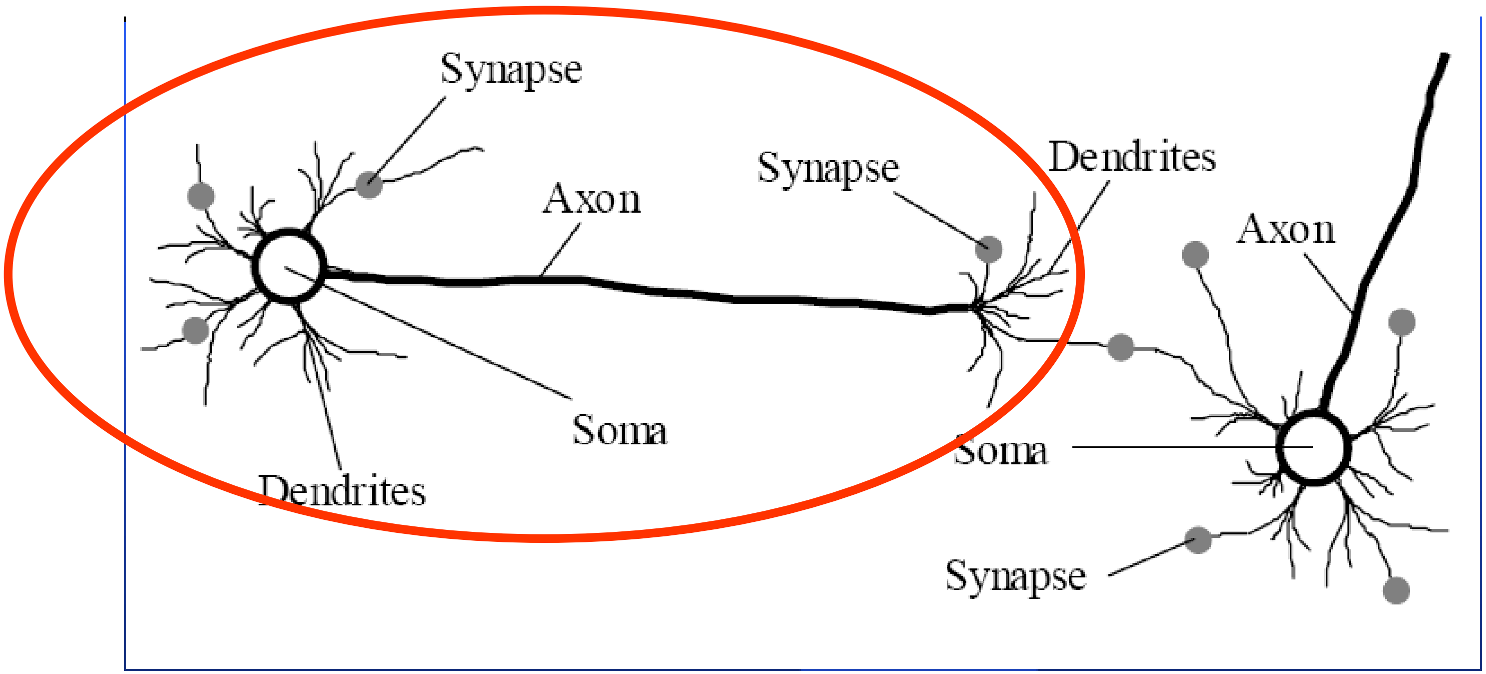
Billions of neurons



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

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

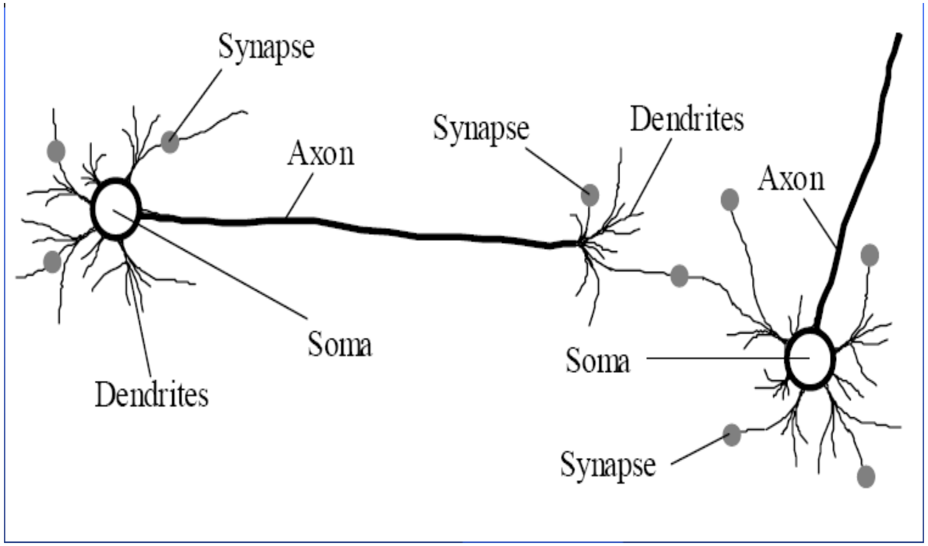
Biological Neural Networks

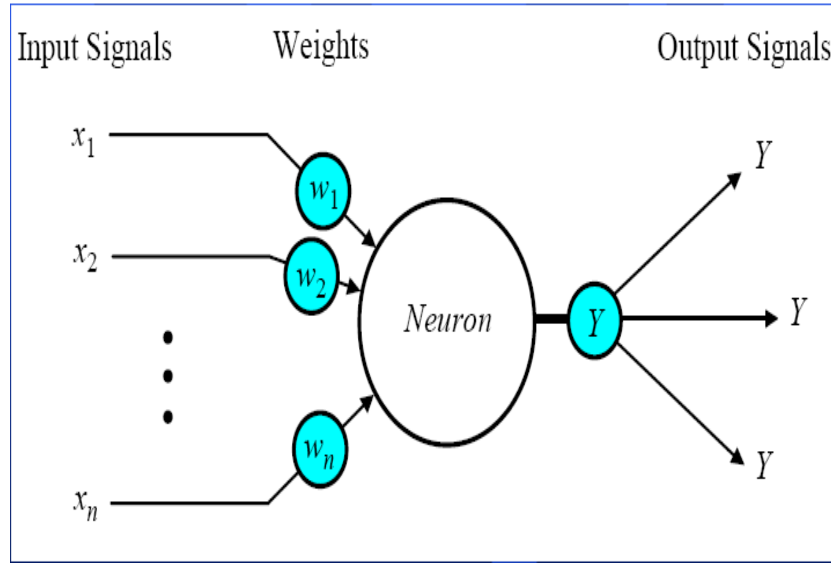


A network of interconnected biological neurons.

Connections per neuron 10^4 - 10^5

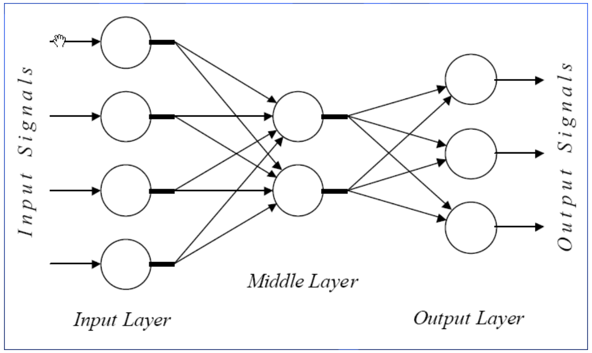
Biological vs Artificial Neural Networks





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

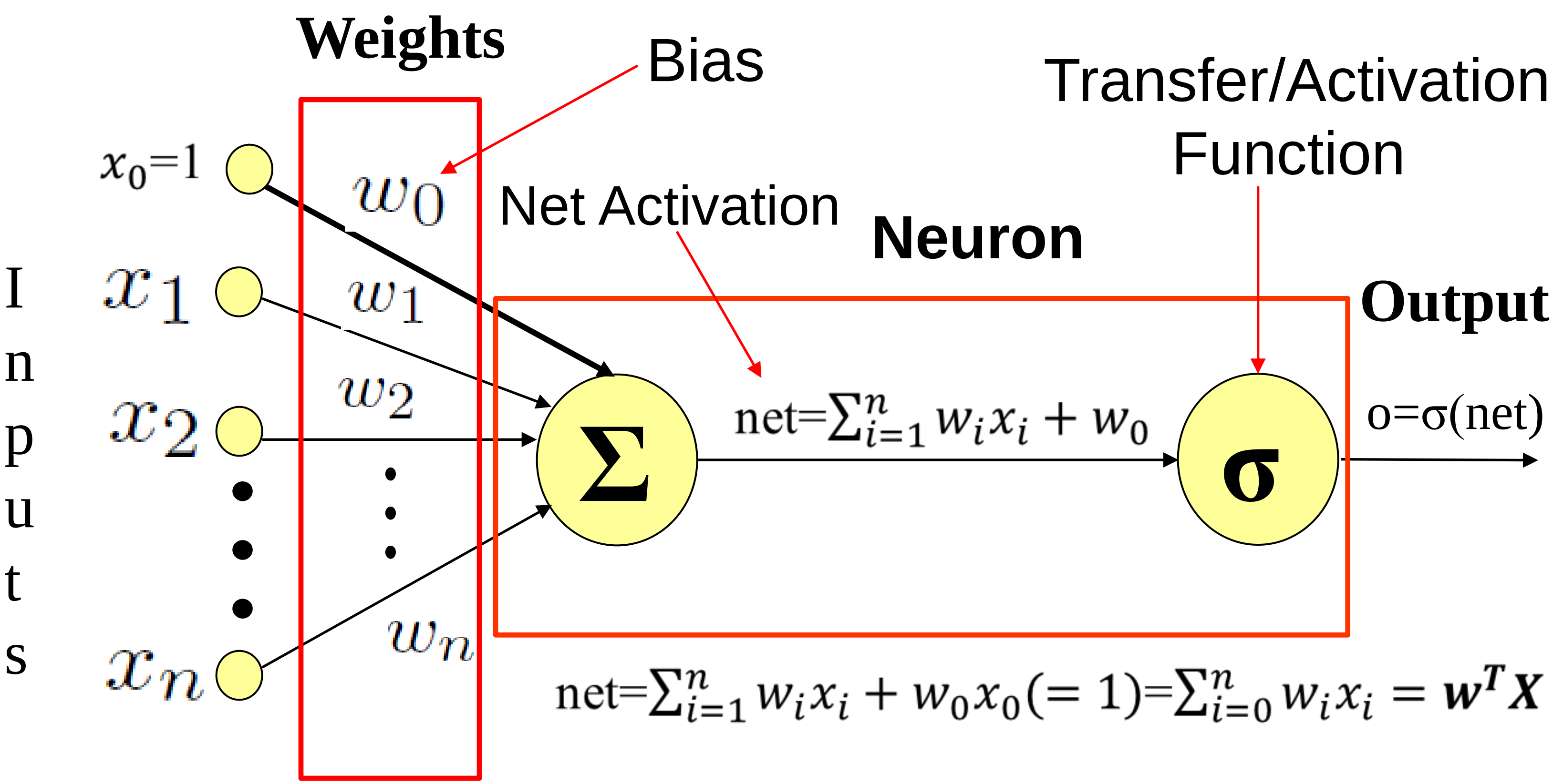
How information is processed in each unit. output = f(input)

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

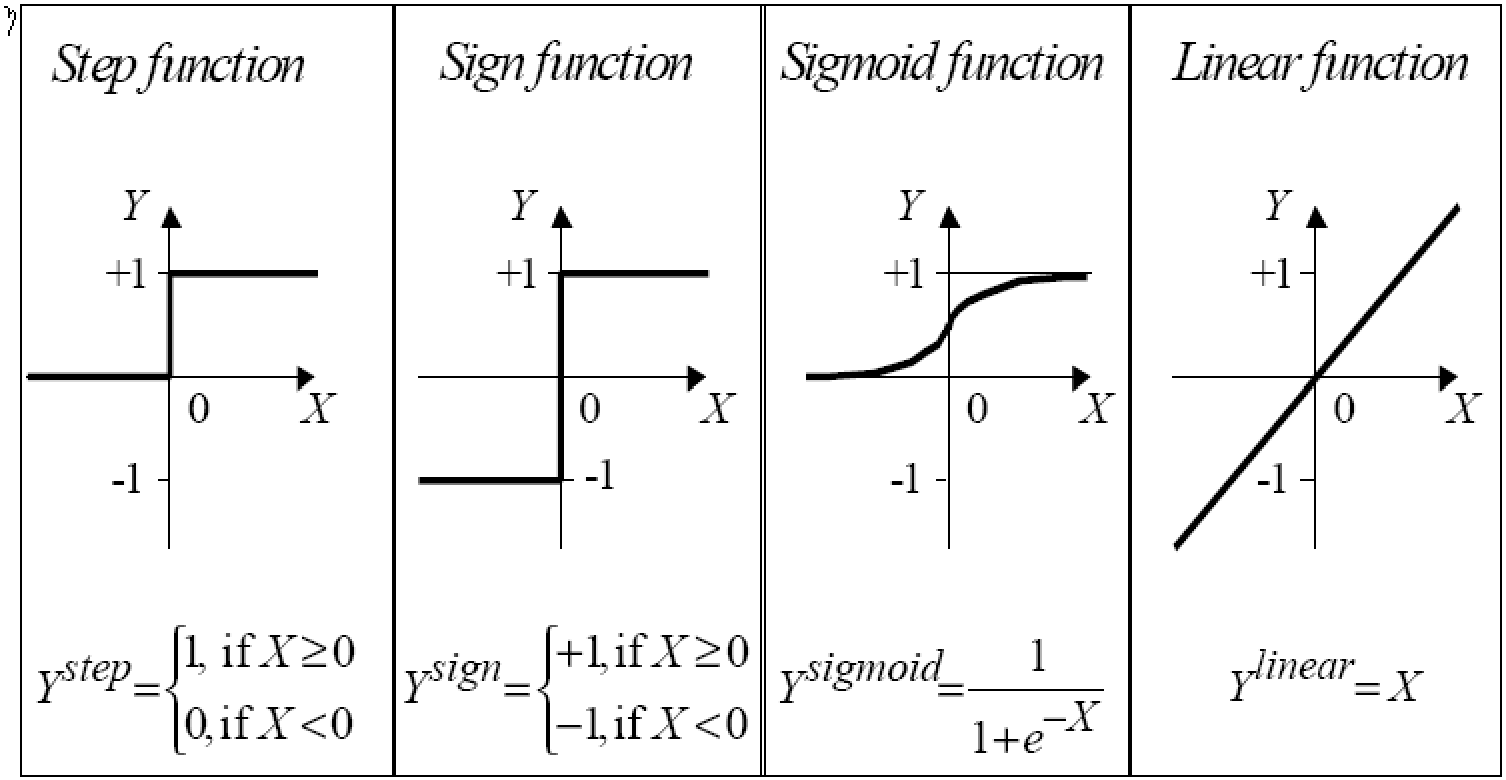


$$\text{net}=\sum_{i=1}^{n} w_i x_i + w_0 x_0 (= 1) = \sum_{i=0}^{n} w_i x_i = \boldsymbol{w}^{\boldsymbol{T}} \boldsymbol{X}$$

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

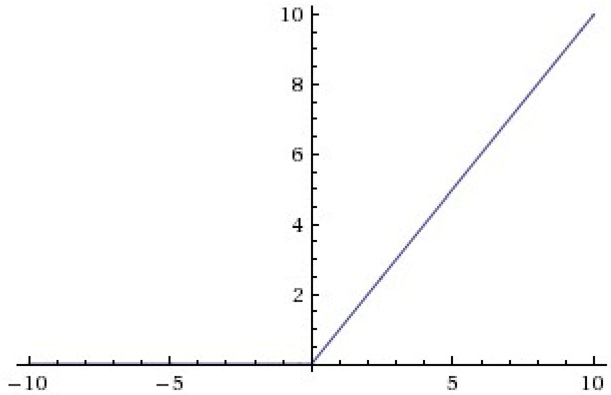
- Main building block of any neural network

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/

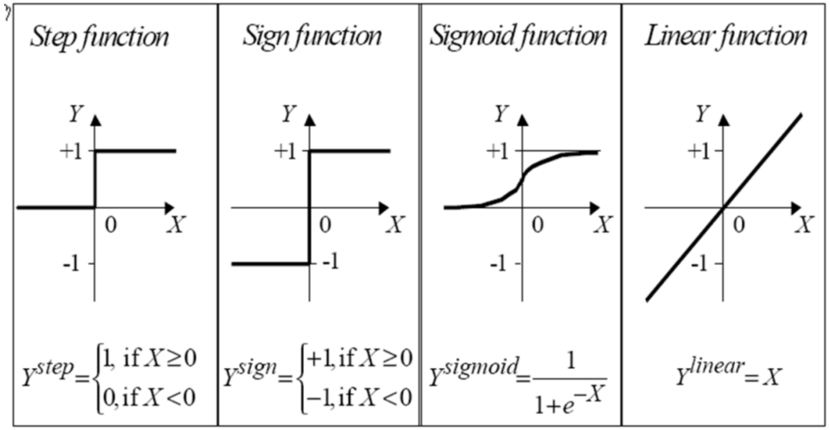
- Rectified Linear Unit (ReLu): max(0, x)
- Popular for deep networks
- Less computationally expensive than sigmoid
- Accelerates convergence during training
- Leaky ReLu: $output = \begin{cases} x & if\ x > 0 \\ 0.01x & otherwise \end{cases}$

Role of Bias

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

$$o = \sigma(net)$$

$$w_0 = -\theta$$

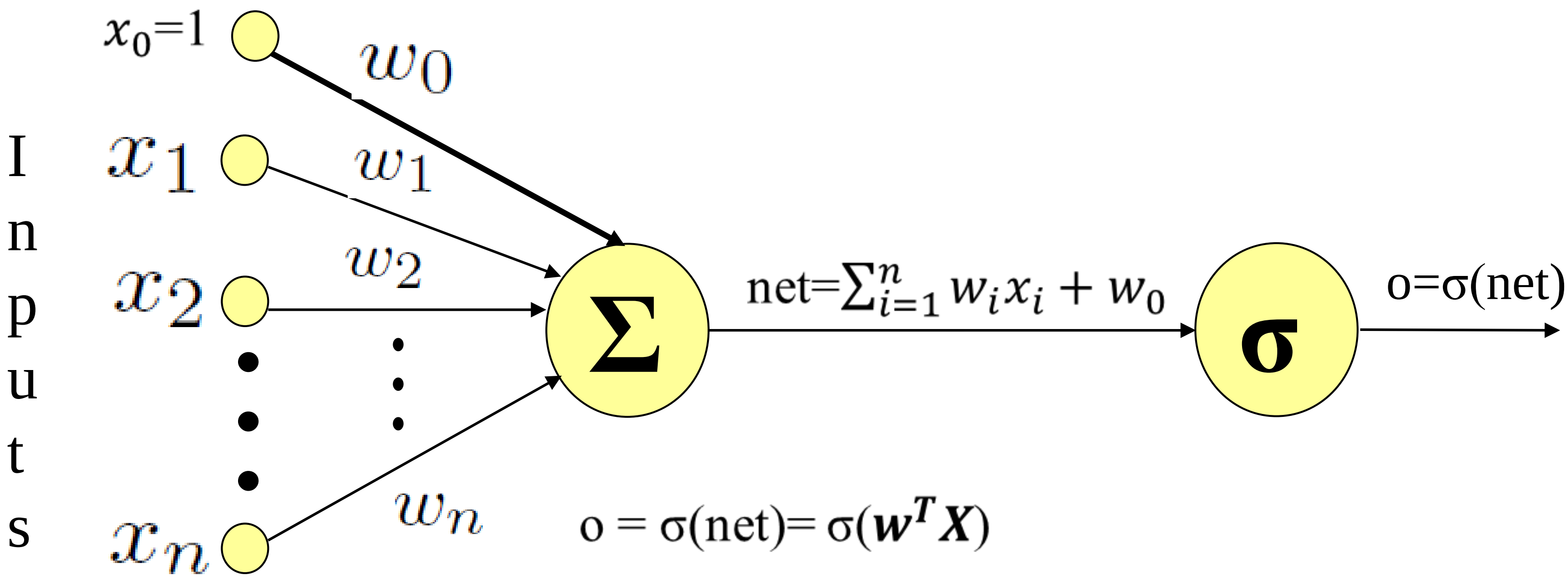


- The threshold where the neuron fires should be adjustable
- Instead of adjusting the threshold we add the bias term
- Defines how strong the neuron input should be before the neuron fires

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

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

Perceptron



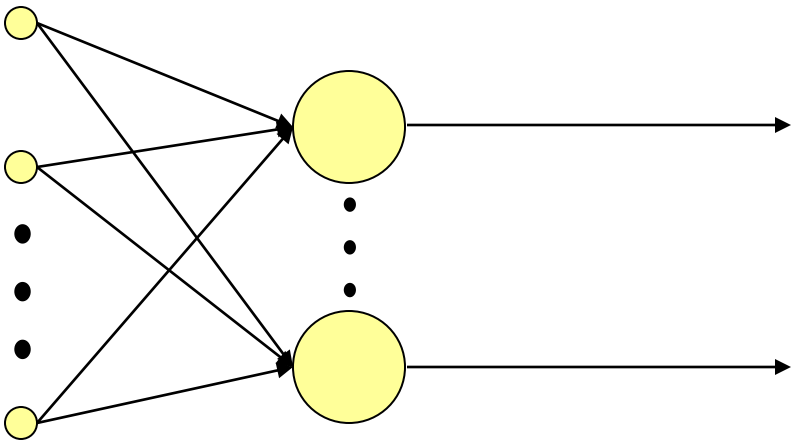
o = σ(net)= σ($\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{X}$)

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

- 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

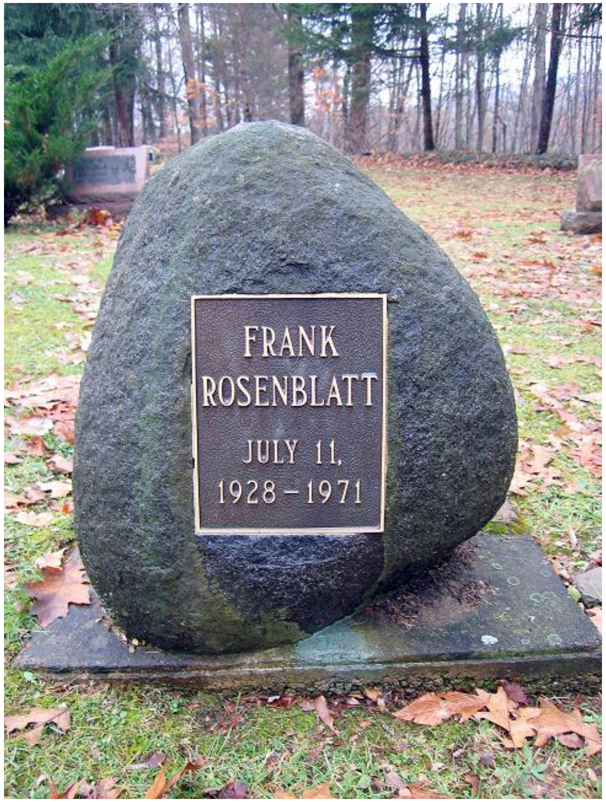
- Generalization to single layer perceptrons with more neurons is easy because:



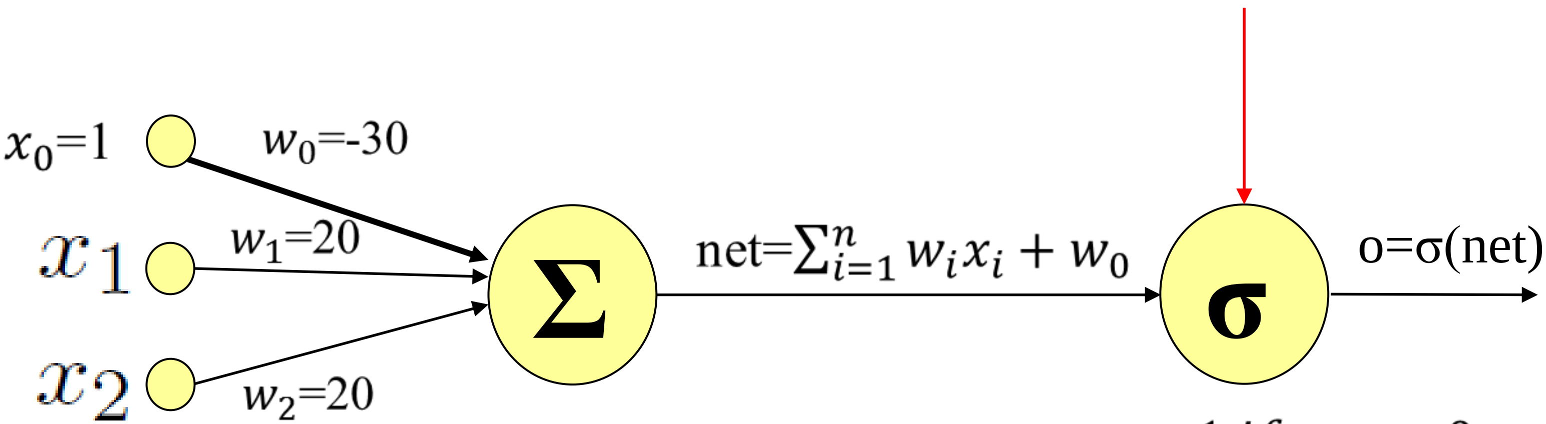
- The output units are mutually independent
- Each weight only affects one of the outputs

Perceptron

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



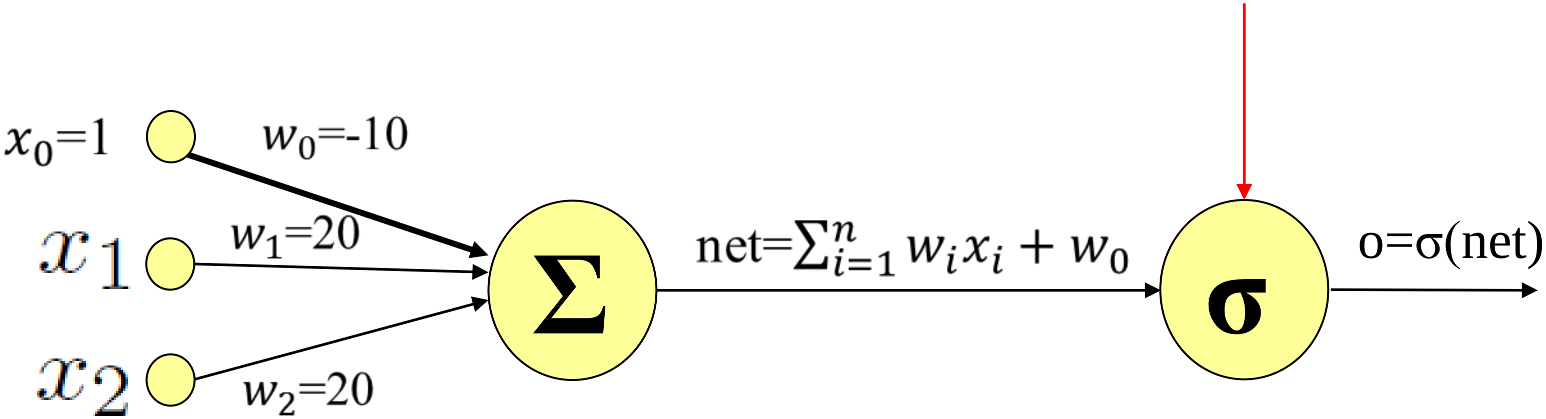
Perceptron: Example 1 - AND



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

- 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

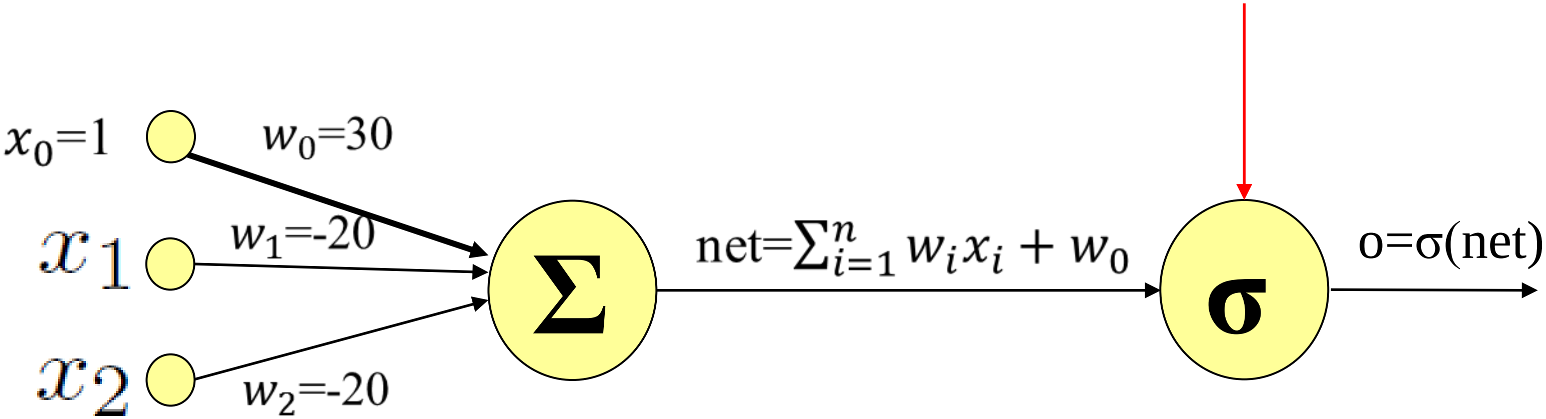
Perceptron: Example 2 - OR



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

- 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



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

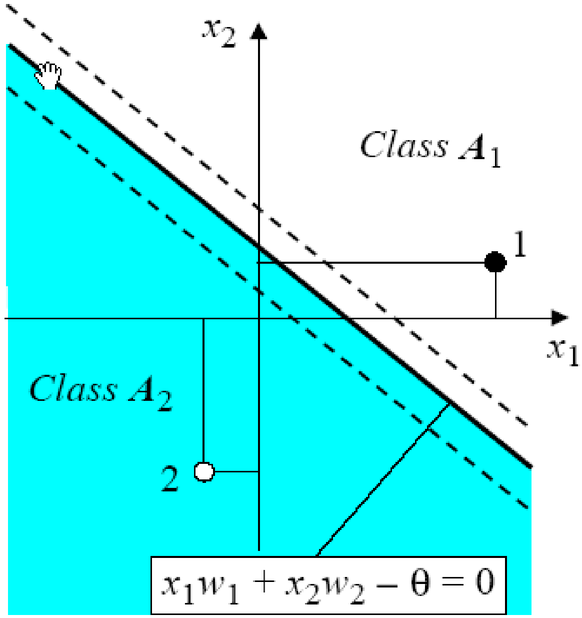
- x1 = 1, x2 = 1→ net = -20-20+30=-10 → o = σ(-10) = 0
- x1 = 0, x2 = 1→ net = 0-20+30 =10 → o = σ(10) = 1
- x1 = 1, x2 = 0→ net = -20+0+30 =10 → o = σ(10) = 1
- x1 = 0, x2 = 0→ net = 0+0+30 =30 → o = σ(30) = 1

Perceptron for classification

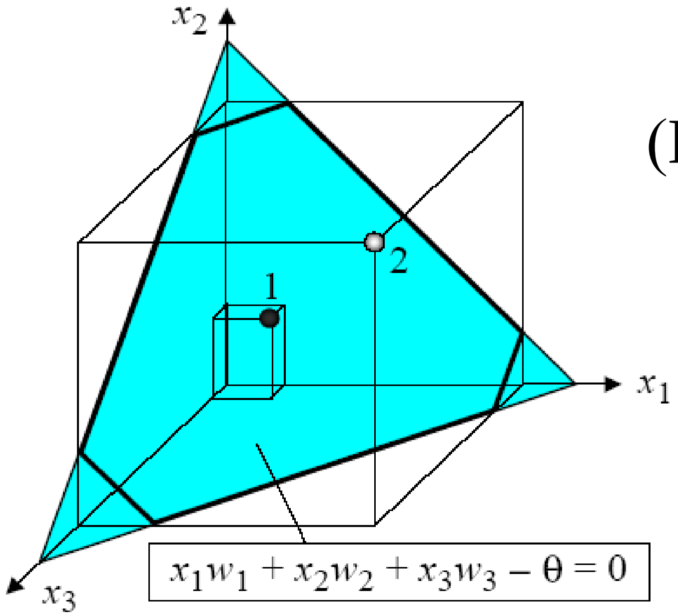
- Given training examples of classes A1, A2 train the perceptron in such a way that it classifies correctly the training examples:
  - ***If the output of the perceptron is 1 then the input is assigned to class A1 (i.e. if*** $\sigma(\mathbf{w}^T\mathbf{x}) = 1$ ***)***
  - ***If the output is 0 then 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.



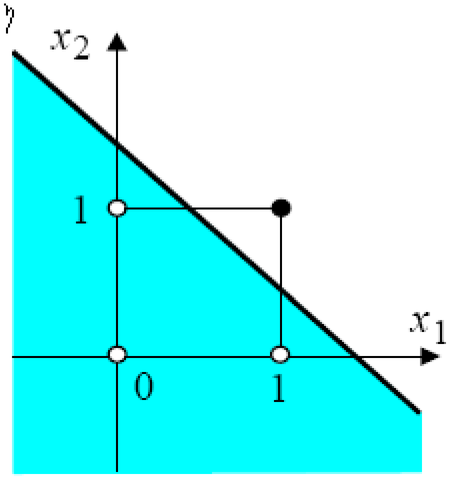
(b) Three-input perceptron.

(Note that $\theta = -w_0$)

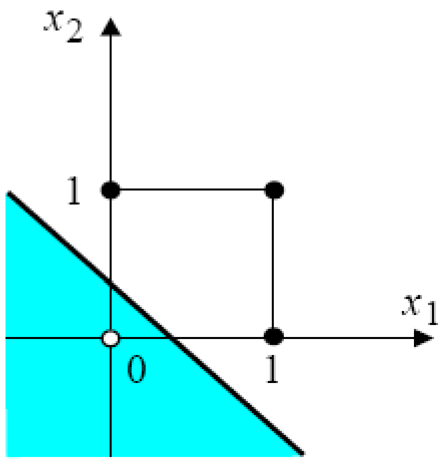
$$if \;\; w_1x_1 + w_2x_2 + w_0 > 0 \;\; then \;\; Class = A1$$
$$if \;\; w_1x_1 + w_2x_2 + w_0 < 0 \;\; then \;\; Class = A2$$

$$if \;\; w_1x_1 + w_2x_2 + w_0 = 0 \; then \; Class = A1 \; or \; A2$$
depends on our definition

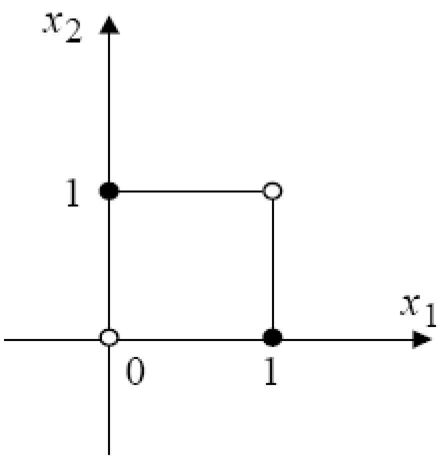
Perceptron: The limitations of perceptron



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



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



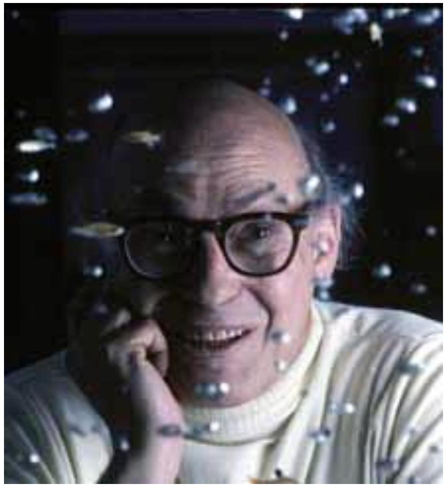
(c) Exclusive-OR ($x_1 \oplus x_2$)

- 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



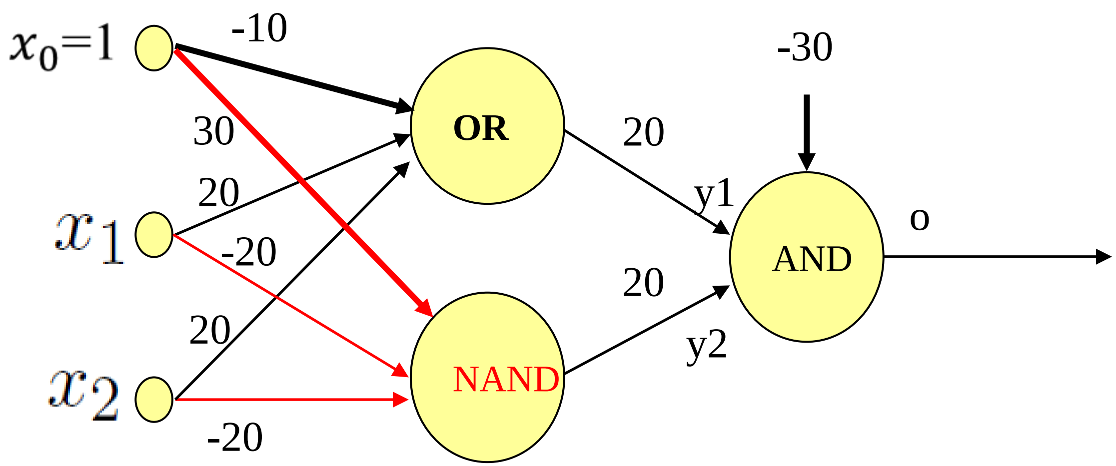
- Seymour Papert

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
- XOR = NAND (AND) OR



OR
1 1 → 1
0 1 → 1
1 0 → 1
0 0 → 0

NAND
1 1 → 0
0 1 → 1
1 0 → 1
0 0 → 1

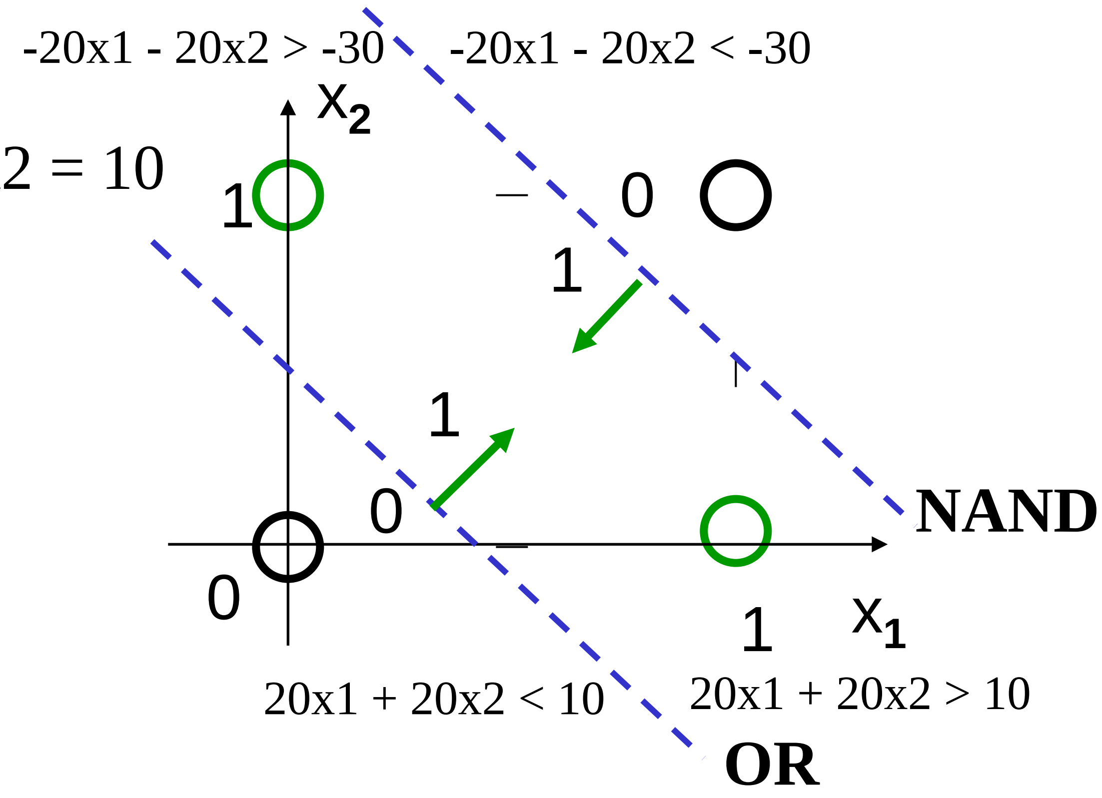
AND
1 1 → 1
0 1 → 0
1 0 → 0
0 0 → 0

- 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

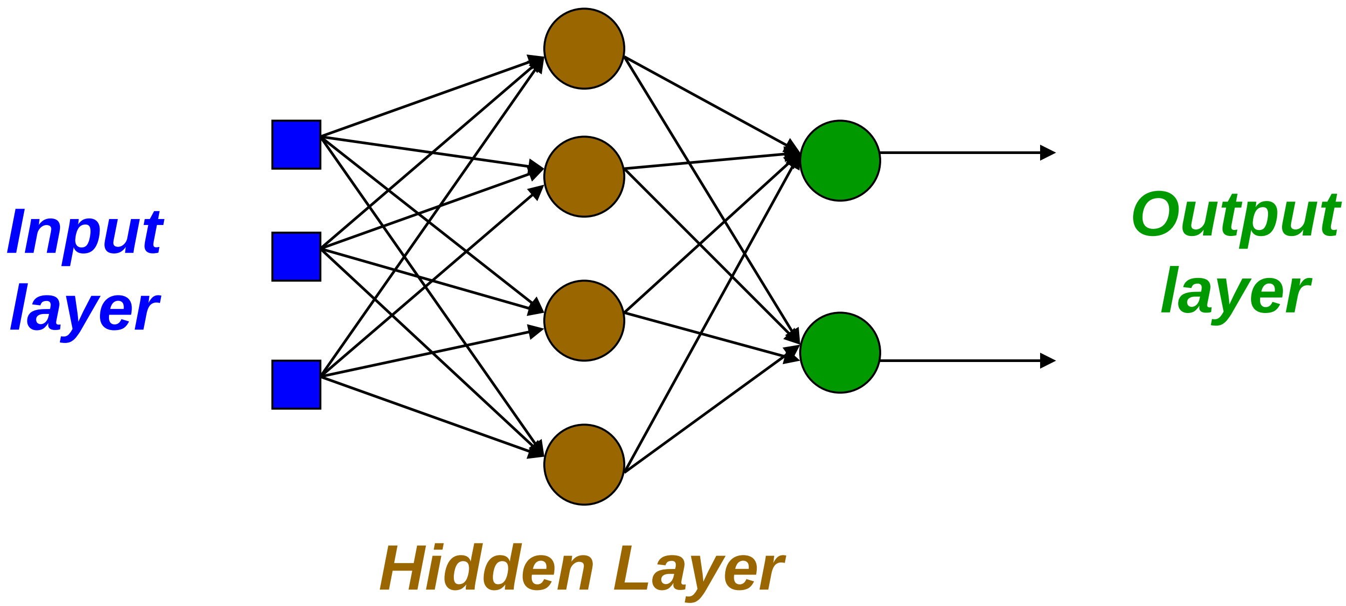
20x1 + 20x2 = 10



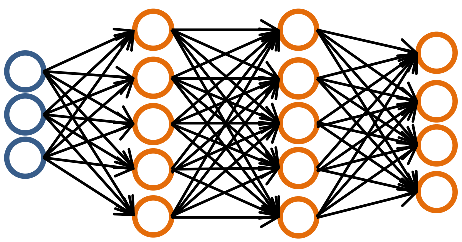
Multilayer Feed Forward Neural Network

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

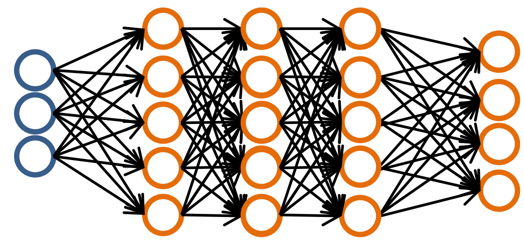
- Hidden nodes do not directly receive inputs nor send outputs to the external environment.



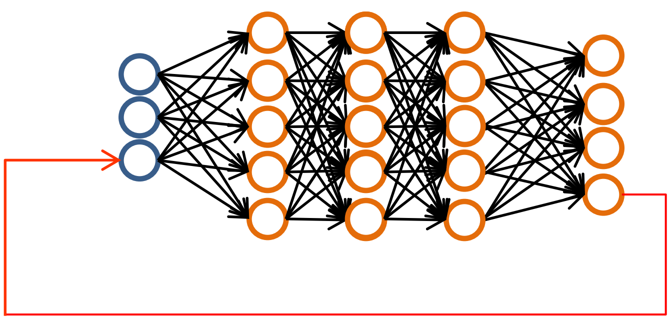
NNs: Architecture



3-layer feed-forward network



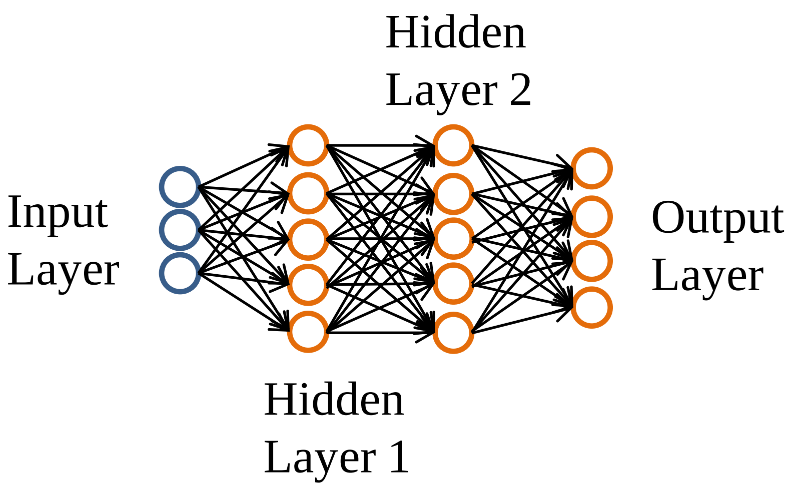
4-layer feed-forward network



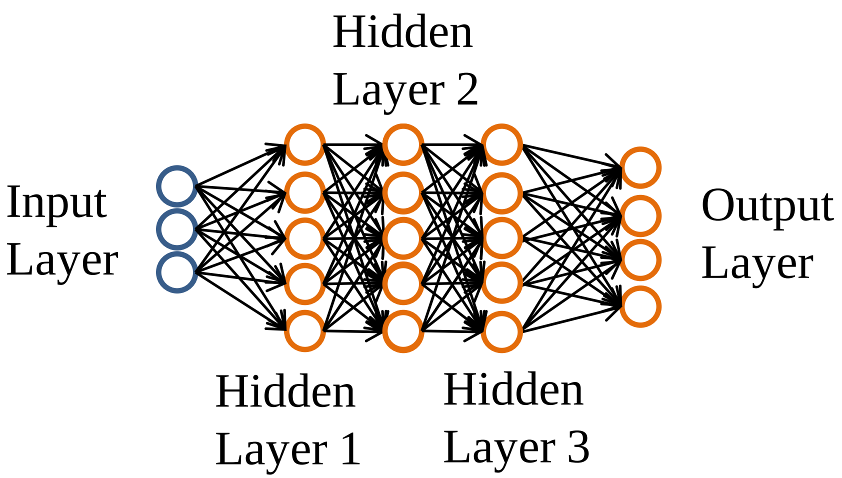
4-layer recurrent network – Difficult to train

- The input layer does not count as a layer

NNs: Architecture



3-layer feed-forward network

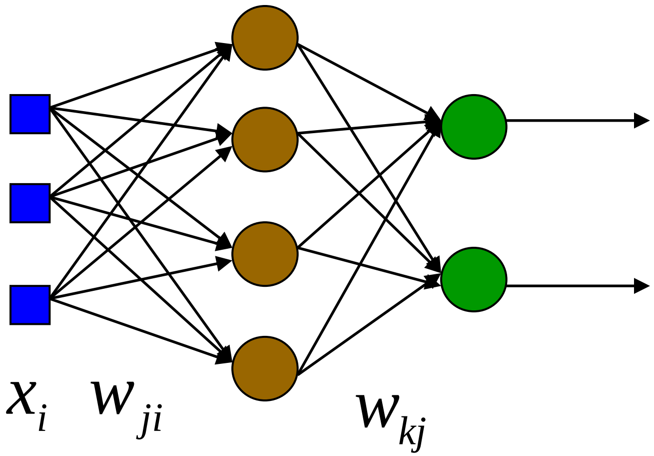


4-layer feed-forward network

- 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

K = number of output neurons

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

$$o_k = \sigma\left(\sum_{j=0}^{nH} y_j w_{kj}\right)$$

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

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

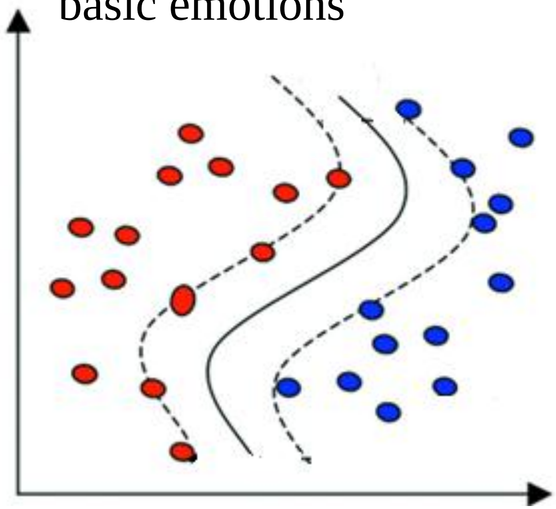
$$o_k = \sigma\left(\sum_{j=0}^{nH} \sigma\left(\sum_{i=0}^{n} x_i w_{ji}\right) w_{kj}\right)$$

Classification / Regression with NNs

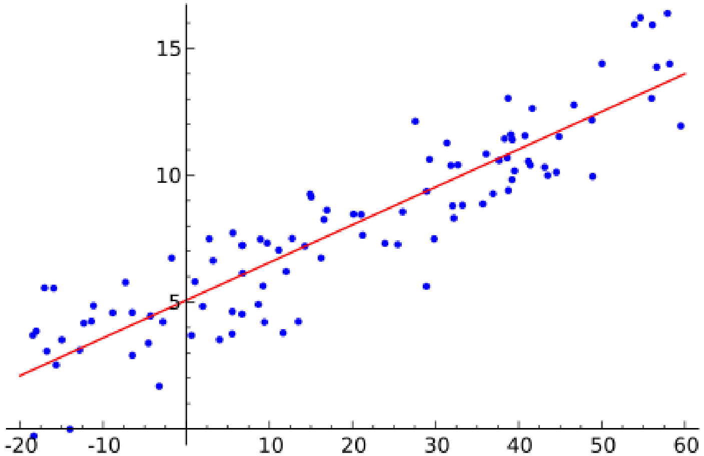
- You should think of neural networks as function approximators

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

Classification
- Discrete output
-e.g., recognise one of the six basic emotions



Regression
- Continuous output
- e.g., house price estimation



Output Representation

- Binary Classification

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

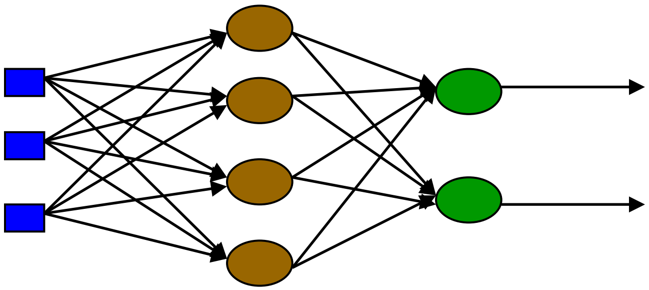
- Regression

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

- Ideally o ≈ t

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

Multiclass Classification



Target Values: vector (length=no. Classes)
e.g. for 4 classes the targets are the following:

$$\begin{array}{cccc} \text{Class1} & \text{Class2} & \text{Class3} & \text{Class4} \\ \begin{bmatrix}1\\0\\0\\0\end{bmatrix} & \begin{bmatrix}0\\1\\0\\0\end{bmatrix} & \begin{bmatrix}0\\0\\1\\0\end{bmatrix} & \begin{bmatrix}0\\0\\0\\1\end{bmatrix} \end{array}$$

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).

$$\mathrm{o} \approx \mathrm{t}$$

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

Training – 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

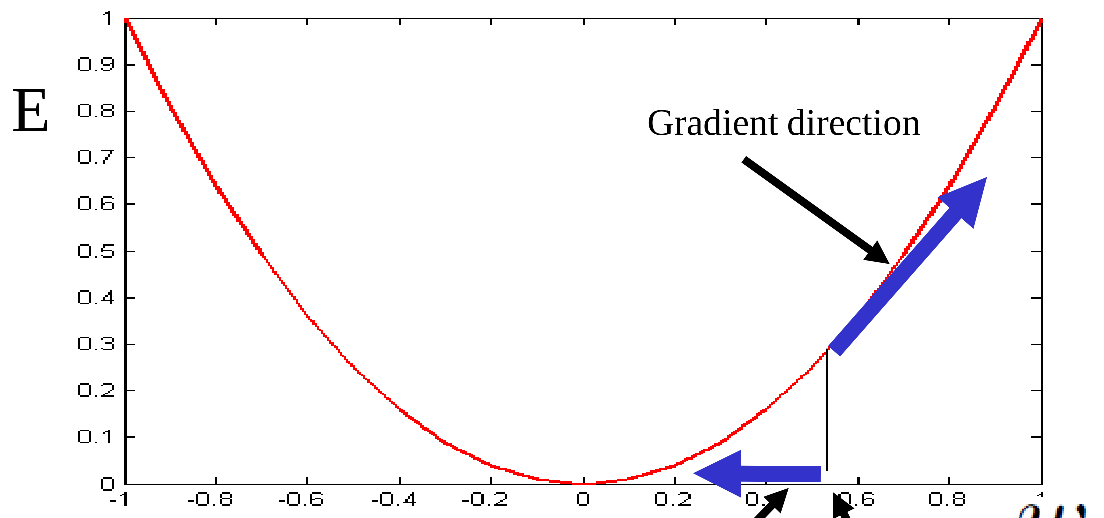
$$w_i \leftarrow w_i + \Delta w_i$$

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

Gradient Descent

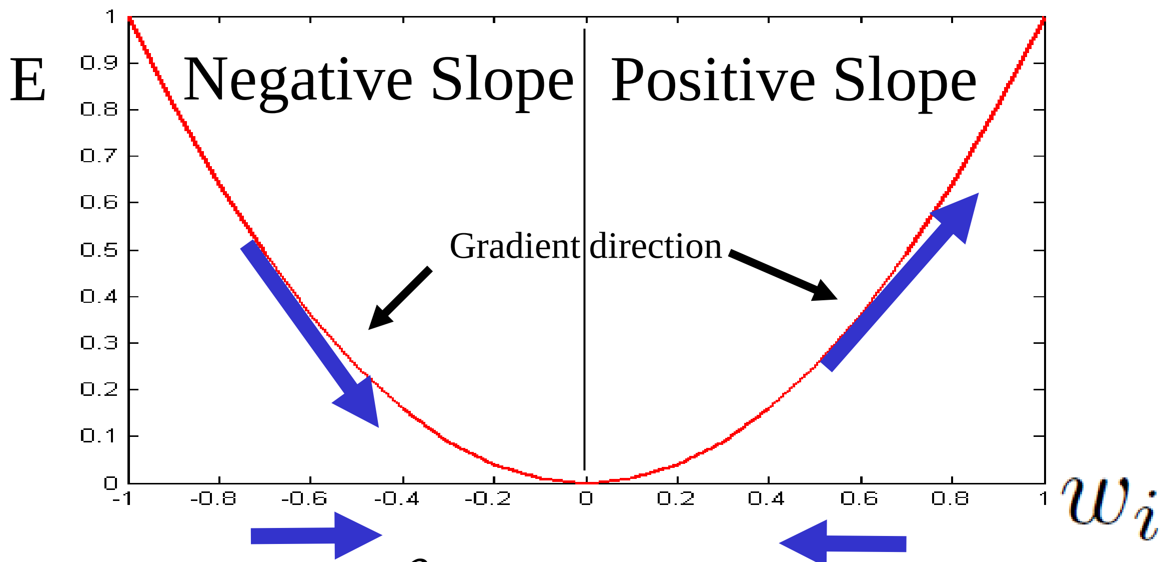
Gradient descent method: take a step in the direction that decreases the error E. This direction is the opposite of the derivative of E.

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



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

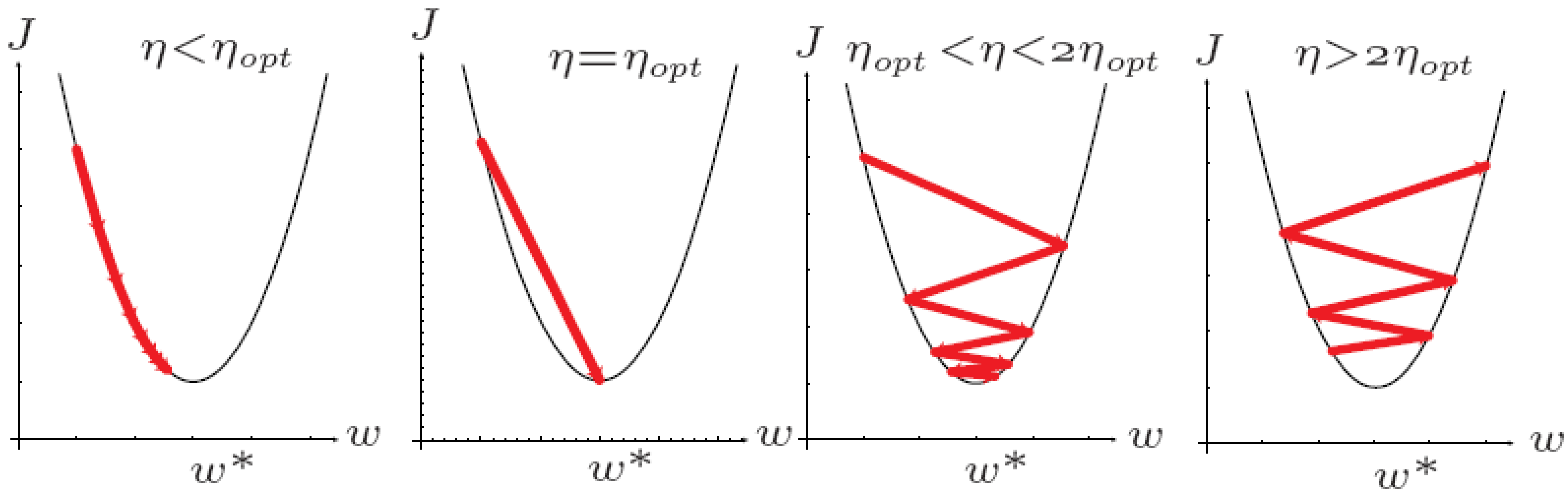
Gradient Descent – Learning Rate



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

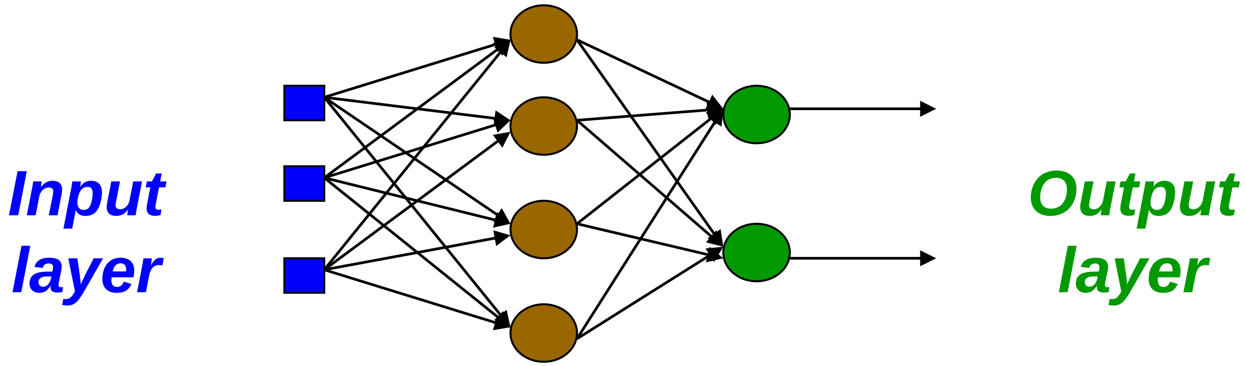
- Derivative: direction of steepest increase
- Learning rate: determines the step size in the direction of steepest decrease. It usually takes small values, e.g. 0.01, 0.1
- If it takes large values then the weights change a lot -> network unstable

Gradient Descent –Learning Rate



Learning: The backpropagation algorithm

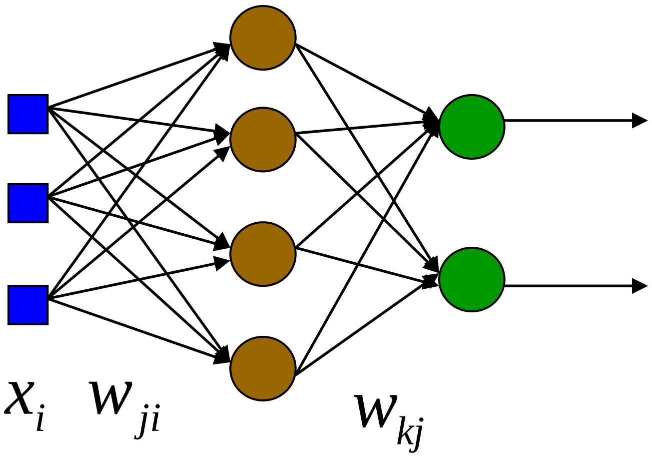
- The Backprop algorithm searches for weight values that minimize the error function of the network (K outputs) over the set of training examples (training set).



- Based on gradient descent algorithm

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

Reminder: Multilayer Feed Forward Neural Network



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

$$o_k = \sigma\left(\sum_{j=0}^{nH} y_j w_{kj}\right) = \sigma\left(net_k\right)$$

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

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

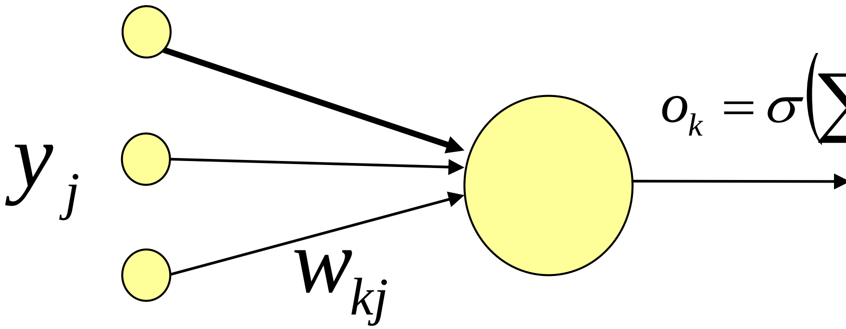
nH = number of hidden neurons

K = number of output neurons

Backpropagation: Initial Steps

- Training Set: A set of input vectors $x_i, i = 1 \dots D$ with the corresponding targets t_i
- η: learning rate, controls the change rate of the weights
- Begin with random weights (use one of the initialisation strategies discussed later)

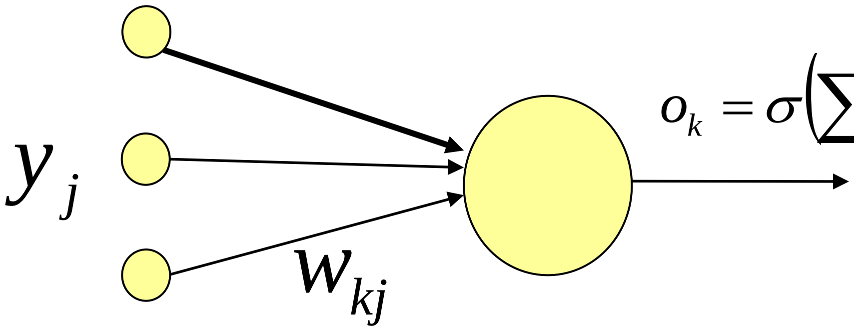
Backpropagation: Output Neurons



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

- We define our error function, for example $E = \frac{1}{2}\sum_{k=1}^{K}(t_k - o_k)^2$
- E depends on the weights because $o_d = \sum_{i=0}^{n} x_i^d w_i$
- For simplicity we assume the error of one training example

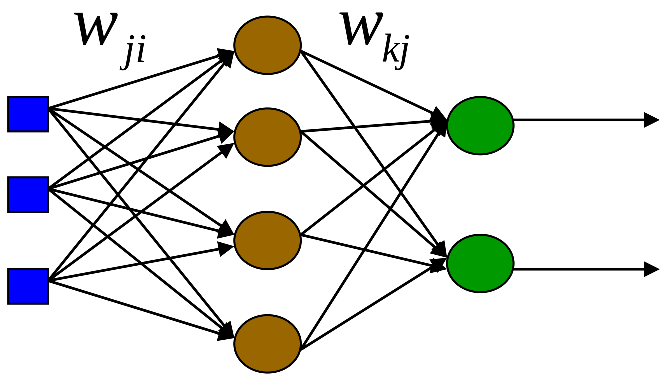
Backpropagation: Output Neurons



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

- $\frac{\partial E_k}{\partial w_{kj}} = \frac{\partial E_k}{\partial o_k}\frac{\partial o_k}{\partial net_k}\frac{\partial net_k}{\partial w_{kj}} = \frac{\partial E_k}{\partial o_k}\frac{\partial \sigma(net_k)}{\partial net_k} y_j$
- We define $\delta_k = \frac{\partial E_k}{\partial o_k}\frac{\partial \sigma(net_k)}{\partial net_k}$
- Update: $\Delta w_{kj} = -\eta \frac{\partial E_k}{\partial w_{kj}} = -\eta \delta_k y_j$

Backpropagation: Output/Hidden Neurons

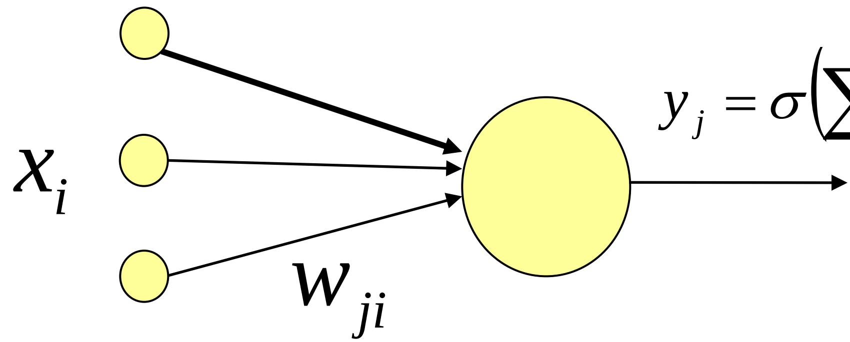


- Weights connected to output neuron k can influence the error of that particular neuron only.
- That's why $\frac{\partial E}{\partial w_{kj}} = \frac{\partial}{\partial w_{kj}}(E_1 + E_2 + \cdots + E_k + \cdots + E_K) = \frac{\partial E_k}{\partial w_{kj}}$

- Weights connected to hidden neuron j can influence the error of all output neurons.
- That's why $\frac{\partial E}{\partial w_{ji}} = \frac{\partial}{\partial w_{ji}}(E_1 + E_2 + \cdots + E_k + \cdots + E_K)$

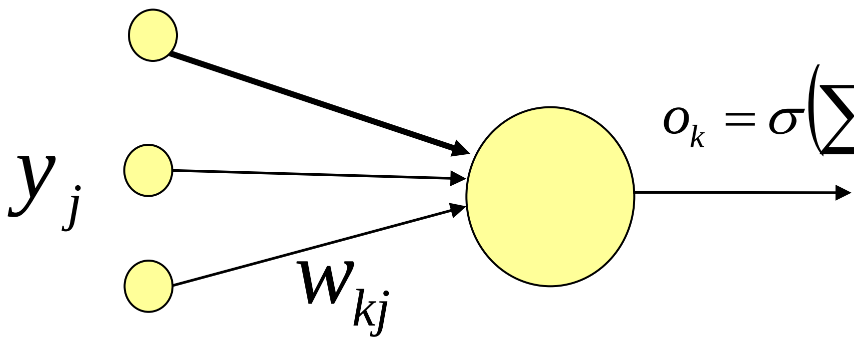
Backpropagation: Hidden Neurons

Hidden Neuron



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

Output Neuron



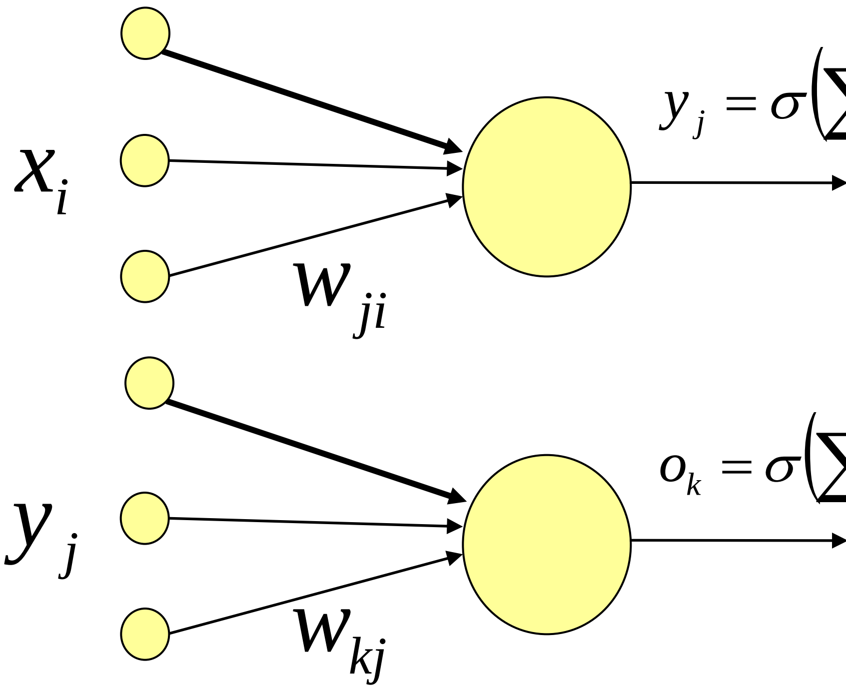
$$o_k = \sigma\left(\sum_{j=1}^{nH} y_j w_{kj}\right) = \sigma(net_k)$$

- $\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}} = \frac{\partial E}{\partial y_j} \frac{\partial \sigma(net_j)}{\partial net_j} x_i$
- $\frac{\partial E}{\partial y_j} = \sum_{k=1}^{K} \frac{\partial E_k}{\partial y_j} = \sum_{k=1}^{K} \frac{\partial E_k}{\partial o_k} \frac{\partial o_k}{\partial y_j} = \sum_{k=1}^{K} \frac{\partial E_k}{\partial o_k} \frac{\partial o_k}{\partial net_k} \frac{\partial net_k}{\partial y_j}$

Backpropagation: Hidden Neurons

Hidden Neuron

Output Neuron



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

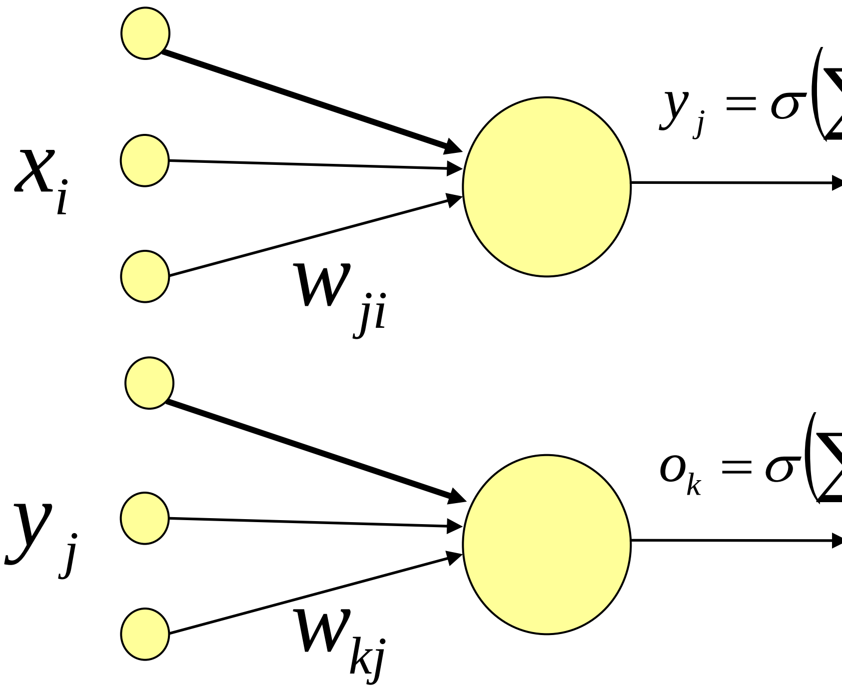
$$o_k = \sigma\left(\sum_{j=1}^{nH} y_j w_{kj}\right) = \sigma(net_k)$$

- $\frac{\partial E}{\partial y_j} = \sum_{k=1}^{K} \frac{\partial E_k}{\partial o_k} \frac{\partial o_k}{\partial net_k} \frac{\partial net_k}{\partial y_j} = \sum_{k=1}^{K} \delta_k w_{kj}$
- $\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial y_j} \frac{\partial y_j}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}} = \sum_{k=1}^{K} (\delta_k w_{kj}) \frac{\partial \sigma(net_j)}{\partial net_j} x_i$

Backpropagation: Hidden Neurons

Hidden Neuron

Output Neuron



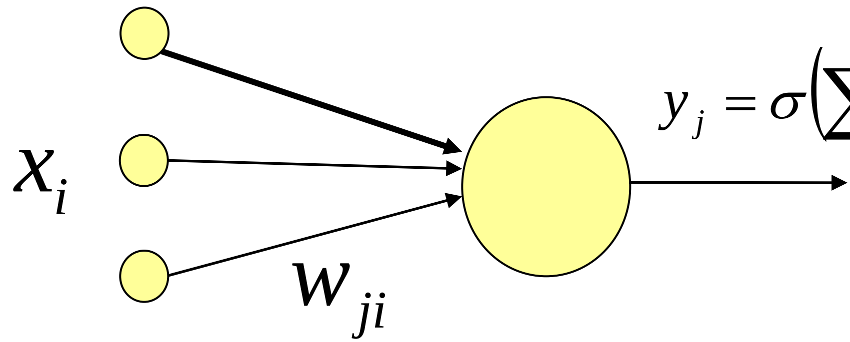
$$y_j = \sigma\left(\sum_{i=0}^{n} x_i w_{ji}\right) = \sigma\left(net_j\right)$$

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

- $\frac{\partial E}{\partial w_{ji}} = \sum_{k=1}^{K}(\delta_k w_{kj}) \frac{\partial \sigma(net_j)}{\partial net_j} x_i$
- We define $\delta_j = \sum_{k=1}^{K}(\delta_k w_{kj}) \frac{\partial \sigma(net_j)}{\partial net_j}$

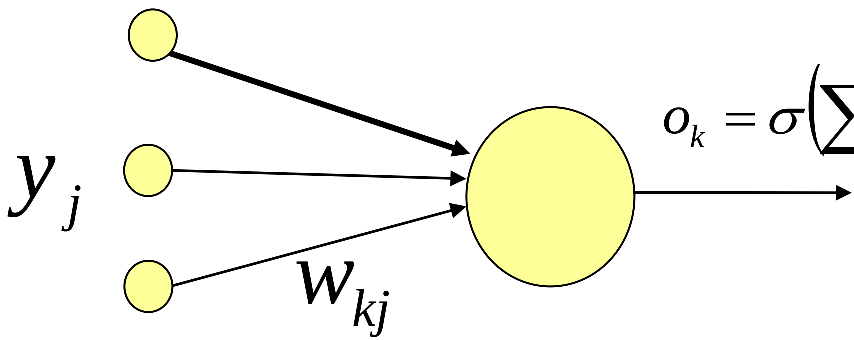
Backpropagation: Hidden Neurons

Hidden Neuron



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

Output Neuron



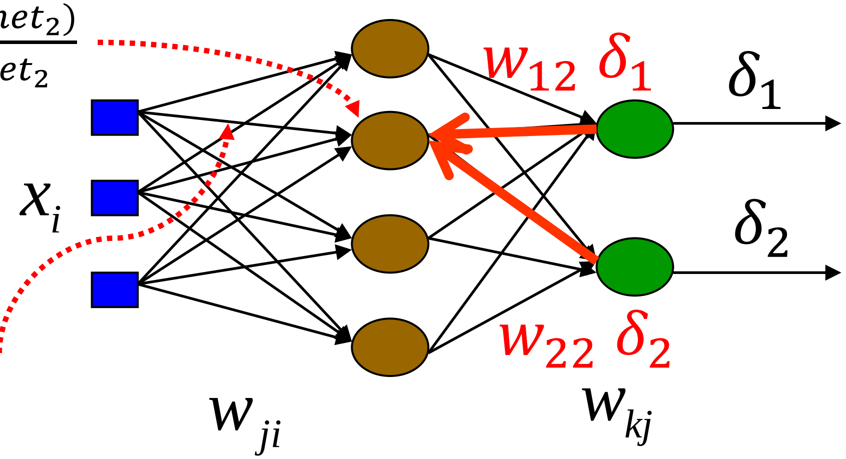
$$o_k = \sigma\left(\sum_{j=1}^{nH} y_j w_{kj}\right) = \sigma(net_k)$$

- $\frac{\partial E}{\partial w_{ji}} = \delta_j x_i$
- Update: $\Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}} = -\eta \delta_j x_i$

Backpropagation: Hidden Neurons

$$\delta_{2hid} = (w_{12}\ \delta_1 + w_{22}\ \delta_2)\frac{\partial\sigma(net_2)}{\partial net_2}$$

Input layer



Output layer

$$\Delta w_{21} = -\eta\ \delta_{2hid}\ x_1$$

- Update: $\Delta w_{ji} = -\eta\frac{\partial E}{\partial w_{ji}} = -\eta\delta_j x_i$
- $\delta_j = \sum_{k=1}^{K}(\delta_k w_{kj})\frac{\partial\sigma(net_j)}{\partial net_j}$
- $\delta_k = \frac{\partial E_k}{\partial o_k}\frac{\partial\sigma(net_k)}{\partial net_k}$

Example

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

Stochastic Gradient Descent

- Stochastic/Incremental/On-line: One example at a time is fed to the network.
- Weights are updated after each example is presented to the network

Batch Gradient Descent

- Batch: All examples are fed to the network. Weights are updated only after all examples have been presented to the network

- For each weight the corresponding gradient (or Δw) is computed (for each example).

- The weights are updated based on the average gradient over all examples.Type equation here.

- $\Delta w_{allExamples} = \frac{1}{D} \sum_{d=1}^{D} \Delta w_{oneExample}$

Mini-batch Gradient Descent

- Mini-Batch: M randomly examples are fed to the network.
  - M = 32…128 (typical value 100)
- For each weight the corresponding gradient (or Δw) is computed (for each example).
- The weights are updated based on the average gradient over all M examples.
- Set of M examples is called mini-batch.
- Popular approach in deep neural networks.
- Sometimes called stochastic gradient descent (NOT to be confused with online/incremental gradient descent).

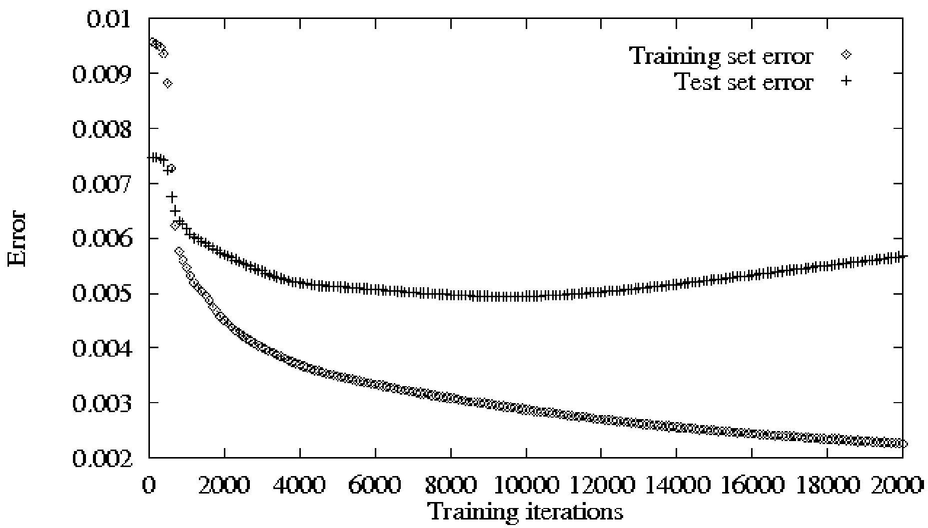
Backpropagation Stopping Criteria

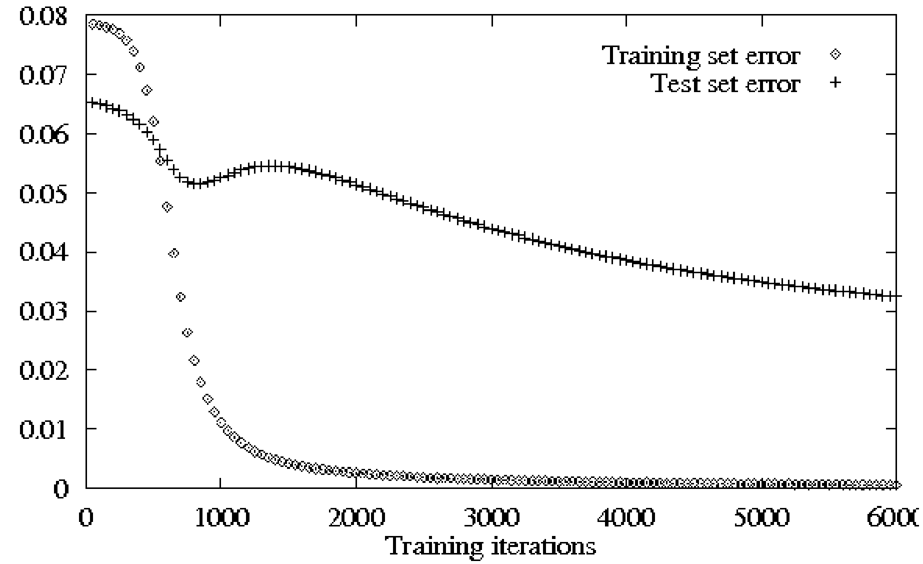
- When the gradient magnitude (or Δw_i) is small, i.e. $\frac{\partial E}{\partial w_i} < \delta \; or \; \Delta w_i < \delta$

- 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**)

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

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

$$\Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}} = -\eta \delta_j x_i \; where \quad \delta_j = \sum_{k=1}^{K} (\delta_k w_{kj}) \frac{\partial \sigma(net_j)}{\partial net_j}$$

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

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

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