COMP3308/3608, Lecture 8b & 9a

ARTIFICIAL INTELLIGENCE

Multilayer Neural Networks and Backpropagation Algorithm

Reference: Russell and Norvig, pp. 731-737

Witten, Frank and Hall, pp. 232-241

Outline

- **Multilayer perceptron NNs**
  - **XOR problem**
  - **neuron model**
- **Backpropagation algorithm**
  - **Derivation**
  - **Example**
  - **Universality - Cybenco’s theorems**
  - **Preventing overfitting**
- **Modifications**
  - **Momentum**
  - **Learning rate**
- **Limitations and capabilities**
- **Interesting applications**

COMMONWEALTH OF AUSTRALIA

Copyright Regulations 1969

WARNING

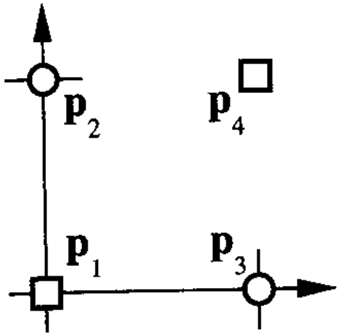
This material has been reproduced and communicated to you by or on behalf of the **University of Sydney** pursuant to Part VB of the Copyright Act 1968 (the Act).

The material in this communication may be subject to copyright under the Act. Any further reproduction or communication of this material by you may be the subject of copyright protection under the Act.

Do not remove this notice

XOR problem - Example

- **The XOR problem is <u>not</u> linearly separable and, hence, cannot be solved by a *single layer* perceptron network**



$$\left\{p_1=\begin{bmatrix}0\\0\end{bmatrix},t_1=0\right\}\quad\left\{p_2=\begin{bmatrix}0\\1\end{bmatrix},t_2=1\right\}\quad\left\{p_3=\begin{bmatrix}1\\0\end{bmatrix},t_3=1\right\}\quad\left\{p_4=\begin{bmatrix}1\\1\end{bmatrix},t_4=0\right\}$$

- **But it can be solved by a *multilayer* perceptron network:**

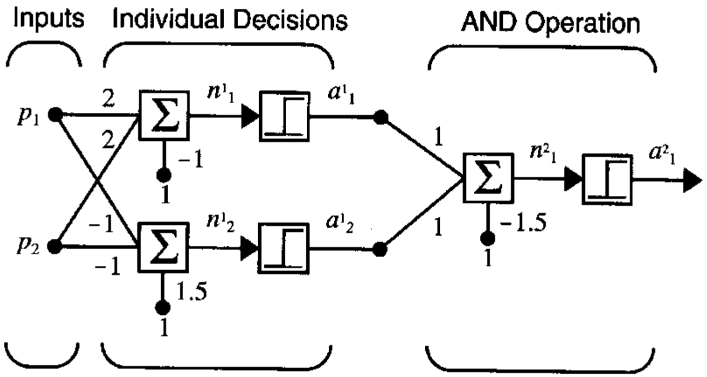


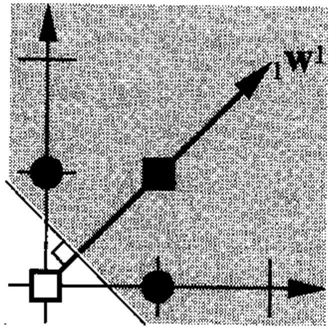
Image from Hagan, Demuth & Beale

- **The 2 perceptrons in the input layer identify linearly separable parts, and their outputs are combined by another perceptron to form the final solution**

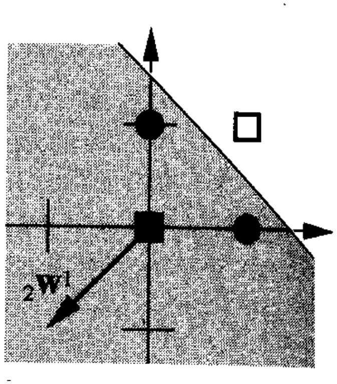
- ***Verify that the network with the weights shown in the diagram solves the XOR problem!***

Boundary Investigation for the XOR Problem

- 1st layer boundaries:

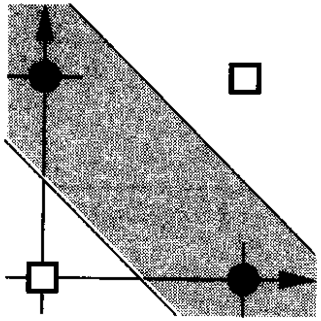


1st layer, 1st neuron;
1st decision boundary



1st layer, 2d neuron:
2nd decision boundary

- 2nd layer combines the two boundaries together:



2nd layer, 1st neuron:
combined boundary

Images from Hagan, Demuth & Beale

Can We Train Such a Network?

- **There exist a 2-layer perceptron network capable of solving the XOR task!**
  - **Rosenblatt and Widrow (who proposed the perceptron) were aware of this**
- **But how can we train such a network to learn from examples?**
  - **Rosenblatt and Widrow were not able to successfully modify the perceptron's rule to train these more complex networks**
  - **1969 – book "Perceptrons" by Minsky and Papert**
    - **Discusses the limitations of the perceptrons (can solve linearly separable problems only)**
    - ***"There is no reason to suppose that any of the virtues of perceptrons carry over to the many-layered version"***
  - **=> The majority of scientific community walked away from the field of NNs, funding was suspended**

Discovery of the Backropagation Algorithm

- **An algorithm for training multi-layer perceptron networks**
- **Proposed by Paul Werbos in 1974**
  - **His thesis presented the algorithm in the context of general networks, with NNs as a special case, and was not disseminated in the NN community**
- **Rediscovered by David Rumelhart, Geoffrey Hinton, Ronald Williams 1986; David Parker 1985; Yann Le Cun 1985**
  - **Book: Parallel Distributed Processing (1986) by Rumelhart and McClelland**
- **Multilayer NNs trained with the backpropagation algorithm - the most widely used NNs**
- **The backpropagation algorithm is also used to train deep NNs**

Problem Statement – Supervised Learning:

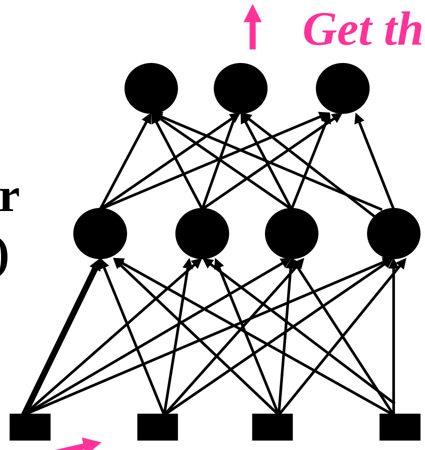
- Given: A labeled training dataset
  - Input vectors associated with target classes
- Task: Build a backpropagation NN that encapsulates this knowledge (i.e. maps the inputs to the target classes) and can be used predictively
  - We would like the NN to predict well new input vectors (unseen during training)

Training data

sepal length	sepal width	petal length	petal width	iris type
5.1	3.5	1.4	0.2	Iris setosa
4.9	3.0	1.4	0.2	Iris setosa
6.4	3.2	4.5	1.5	iris versicolor
6.9	3.1	4.9	1.5	iris versicolor
6.3	3.3	6.0	2.5	iris virginica
5.8	2.7	5.1	1.9	iris virginica
...				

Build the classifier (i.e. train the NN)

Get the prediction



New data

sepal length	sepal width	petal length	petal width	iris type
4.1	3.6	1.6	0.3	?
5.9	3.1	2.4	0.1	?
7.4	3.3	3.5	1.2	?

Learning in NN (review)

- **The “intelligence” of a NN is in the values of its weights**
- **The weights are the parameters of a NN**
- **We start from a random initialization and change them using a learning rule to learn the input-output mapping as specified by the training data**

- **How to learn the input-output mapping using the backpropagation algorithm?**
- **We will formulate an error function (e.g. sum of squared errors between the target and actual output) and use a learning rule that minimizes this error**

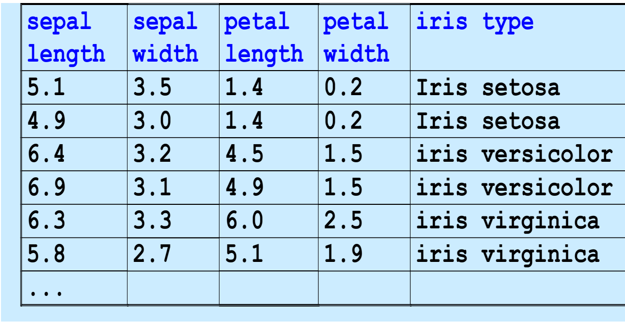
Neural Network Model

- **Computational model consisting of 3 parts:**
- **1) Architecture – neurons and connections**
  - **Input, hidden, output neurons**
  - **Fully or partially connected**
  - **Neuron model – computation performed by each neuron, type of transfer function**
  - **Initialization of the weights**
- **2) Learning algorithm**
  - **How are the weights of the connections changed in order to facilitate learning**
  - **Goal for classification tasks: mapping between the input vectors and their classes**
- **3) Recall technique – once the NN training is completed, how is the information obtained from the trained NN?**
  - **E.g. for classification tasks – how is the class of a new example determined?**

Multi-layer Perceptron Network - Architecture

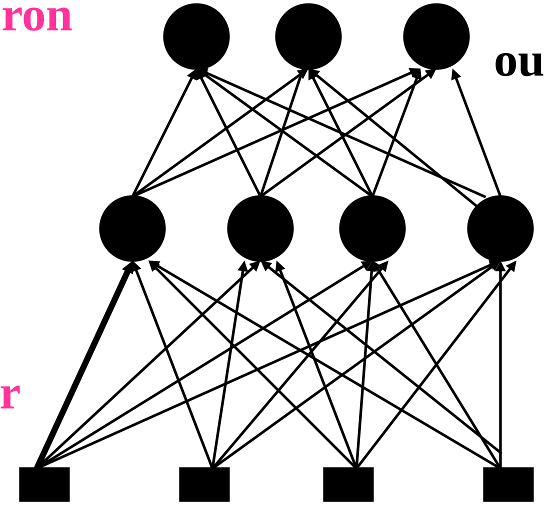
1) A network with 1 or more hidden layers

- e.g. a NN for the iris data:



sepal length	sepal width	petal length	petal width	iris type
5.1	3.5	1.4	0.2	Iris setosa
4.9	3.0	1.4	0.2	Iris setosa
6.4	3.2	4.5	1.5	iris versicolor
6.9	3.1	4.9	1.5	iris versicolor
6.3	3.3	6.0	2.5	iris virginica
5.8	2.7	5.1	1.9	iris virginica
...				

e.g. 1 output neuron for each class



output neurons

hidden neurons (1 hidden layer)

e.g. 1 input neuron for each attribute

inputs

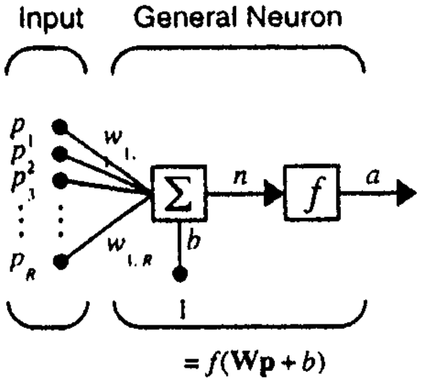
2) A feedforward network - each neuron receives input only from the neurons in the previous layer

3) A fully connected network (typically) - all neurons in a layer are connected with all neurons in the next layer

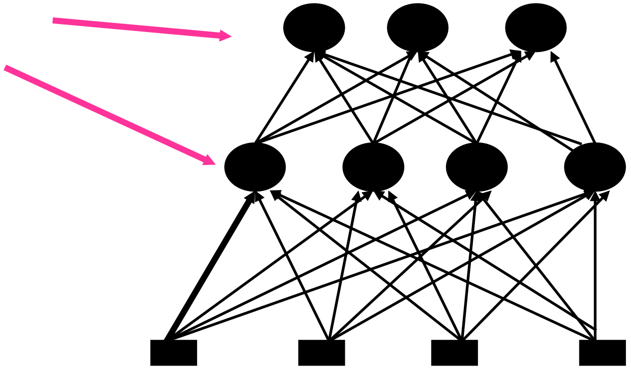
4) Its weights are initialized to small random values, e.g. [-1,1]

Multi-layer Perceptron Network – Architecture 2

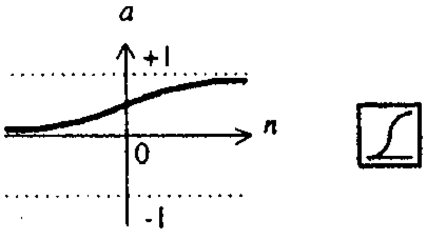
5) Each neuron (except the input neurons) computes the weighed sum of its inputs, and then applies a <u>differentiable</u> transfer function



$$a = f(\mathbf{wp} + b)$$



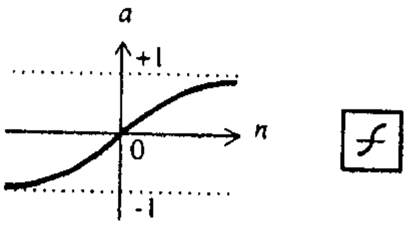
- **any *differentiable* transfer function *f* can be used, i.e. the derivative should exist for every point;**
- **most frequently used: sigmoid and tan-sigmoid (hyperbolic tangent sigmoid)**



$$a = \frac{1}{1 + e^{-n}}$$

a = logsig(n)

Log-Sigmoid Transfer Function



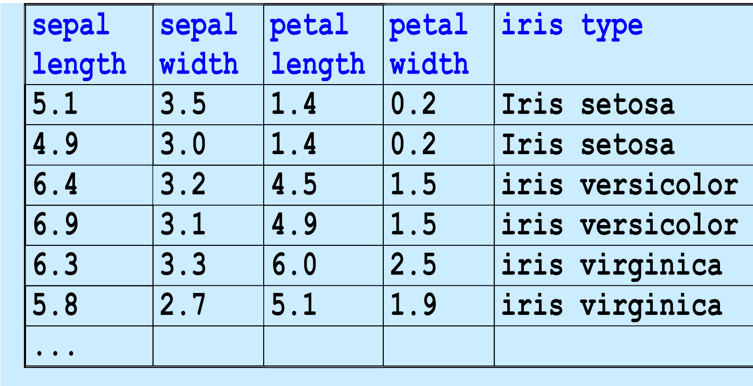
$$a = \frac{e^{n} - e^{-n}}{e^{n} + e^{-n}}$$

a = tansig(n)

Tan-Sigmoid Transfer Function

Architecture – Number of Input Neurons

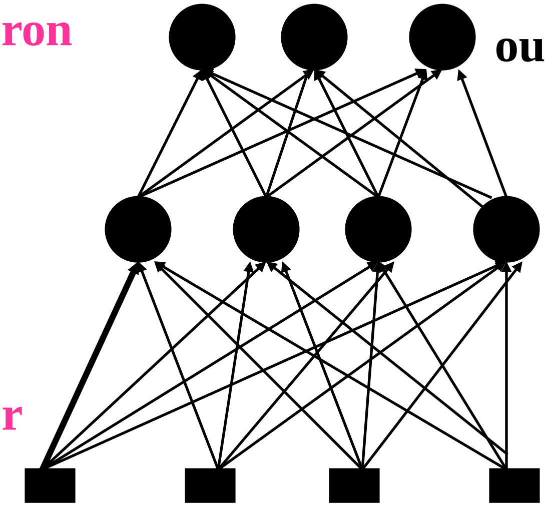
- Input neurons (=inputs) do not perform any computation (e.g. summation and transfer function); they just transmit data



sepal length	sepal width	petal length	petal width	iris type
5.1	3.5	1.4	0.2	Iris setosa
4.9	3.0	1.4	0.2	Iris setosa
6.4	3.2	4.5	1.5	iris versicolor
6.9	3.1	4.9	1.5	iris versicolor
6.3	3.3	6.0	2.5	iris virginica
5.8	2.7	5.1	1.9	iris virginica
...				

- Numerical data - 1 input neuron for each attribute

e.g. 1 output neuron for each class



output neurons

hidden neurons (1 hidden layer)

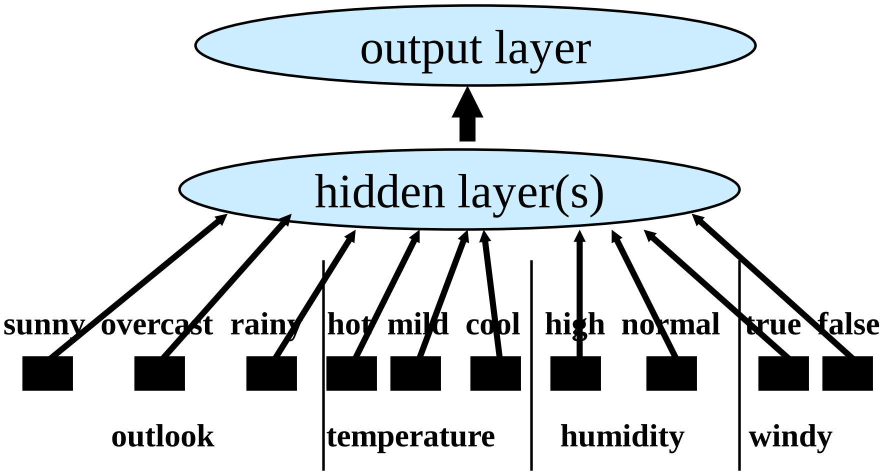
e.g. 1 input neuron for each attribute

inputs

- For numerical data, the input examples are directly fed into the network, e.g. 5.1, 3.5, 1.4 and 0.2 to each of the 4 input neurons (no need for encoding, see next slide)

Architecture – Number of Input Neurons (2)

- Categorical data – 1 input neuron for each attribute value
  - How many input neurons for the weather data?



outlook	temp.	humidity	windy	play
sunny	hot	high	false	no
overcast	hot	high	false	yes
rainy	mild	high	false	yes
rainy	cool	normal	false	yes
rainy	cool	normal	true	no
overcast	cool	normal	true	yes
…				

- Input examples cannot be directly fed; need to be encoded
- Encoding of the input examples – typically binary depending on the value of the attribute (on and off)

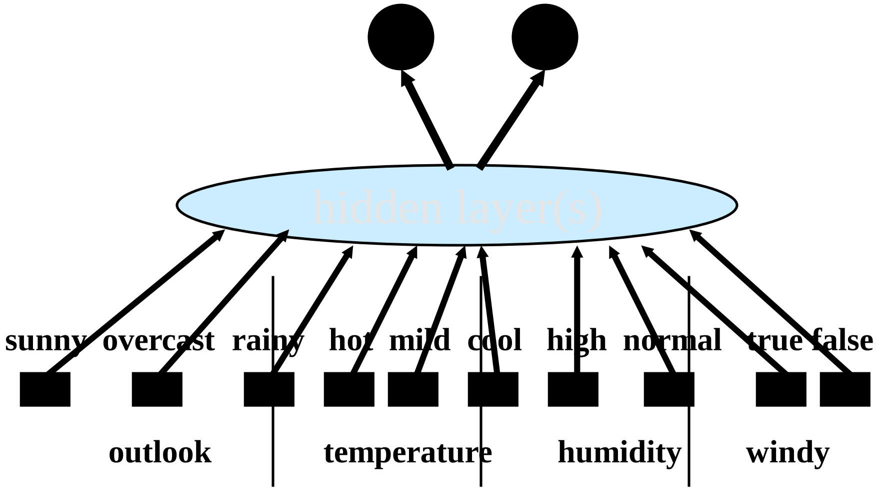
  e.g. ex 1: 100 100 10 01

Number of Output Neurons

outlook	temp.	humidity	windy	play
sunny	hot	high	false	no
overcast	hot	high	false	yes
rainy	mild	high	false	yes
rainy	cool	normal	false	yes
rainy	cool	normal	true	no
overcast	cool	normal	true	yes
…				

- Typically 1 neuron for each class

target class ex1: 1 0



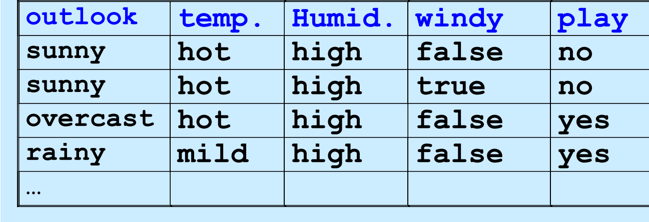
ex.1: 1 0 0 1 0 0 1 0 0 1

- The targets (i.e. classes of the examples need to be encoded) – typically binary encoding is used

  e.g. class1 (no): 1 0, class2 (yes): 0 1

For COMP3608 students only

More on Output Encoding

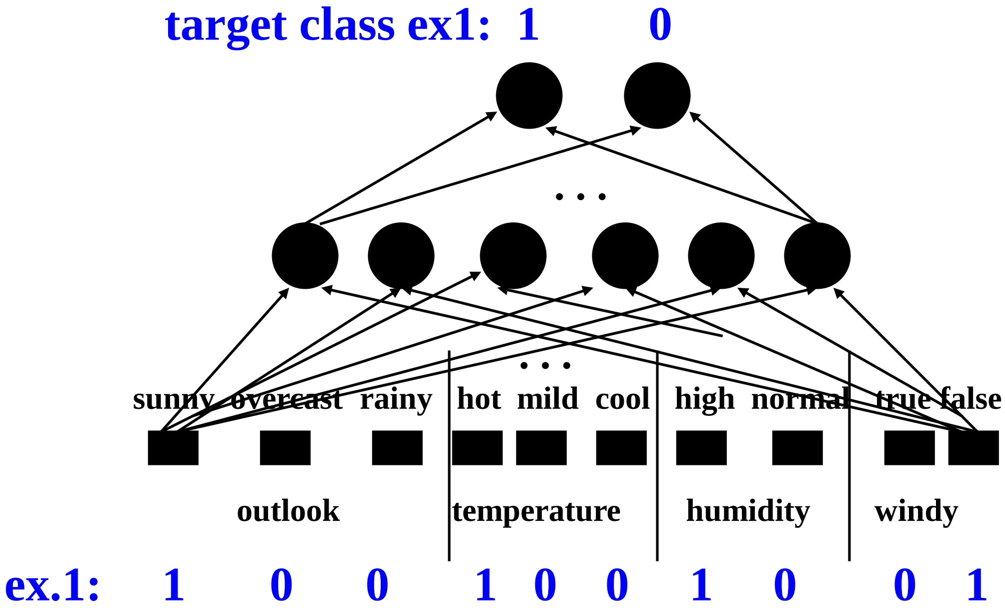


outlook	temp.	Humid.	windy	play
sunny	hot	high	false	no
sunny	hot	high	true	no
overcast	hot	high	false	yes
rainy	mild	high	false	yes
…				

- How to encode the outputs and represent targets?
  - Local encoding
    - 1 output neuron
    - different output values represent different classes, e.g. <0.2 – class 1, >0.8 – class 2, in between – ambiguous class (class 3)
  - Distributed (binary, 1-of-n) encoding - typically used in multi class problems
    - Number of outputs = number of classes
    - Example: 3 classes, 3 output neurons; class 1 is represented as 1 0 0, class 2 - as 0 1 0 and class 3 - as 0 0 1
    - Another representation of the targets: use 0.1 instead of 0 and 0.9 instead of 1
  - Motivation for choosing binary over local encoding
    - Provides more degree of freedom to represent the target function (n times as many weights available)
    - The difference between the the output with highest value and the second highest can be used as a measure how confident the prediction is (close values => ambiguous classification)

Number of Hidden Layers and Neurons in Them

- **An art! Typically - by trial and error**
- **The given task (data) constrains the number of inputs and output neurons but not the number of hidden layers and neurons in them**
  - **Too many hidden layers and neurons (i.e. too many weights) – overfitting**
  - **Too few – underfitting, i.e. the NN is not able to learn the input-output mapping**
  - **A heuristic to start with: 1 hidden layer with *n* hidden neurons, *n=(inputs+output_neurons)/2***



target class ex1: 1 0

. . .

. . .

sunny overcast rainy | hot mild cool | high normal | true false

outlook | temperature | humidity | windy

ex.1: 1 0 0 1 0 0 1 0 0 1

Learning in Multi-Layer Perceptron NNs

- **Supervised learning – the training data:**
  - **consists of labeled examples (p, *d*), i.e. the desired output *d* for them is given (p - input vector; *d* - desired output)**
  - **can be viewed as a *teacher* during the training process**
  - **error - difference between the desired *d* and actual *a* network output**

- **Idea of backpropagation learning**

  For each training example p

  - **Propagate p through the network and calculate the output *a* . Compare the desired *d* with the actual output *a* and calculate the error;**
  - **Update weights of the network to reduce the error;**

  Until error over all examples < threshold

- **Why "backpropagation"? Adjusts the weights *backwards* (from the output to the input neurons) by propagating the *weight change*** Δw

$$w_{pq}^{new} = w_{pq}^{old} + \Delta w_{pq}$$

How to calculate the weight change?

Error Function

- **Sum of Squared Errors (E) is a classical measure of error**
  - **e.g. E for a single training example over all output neurons *i*:**

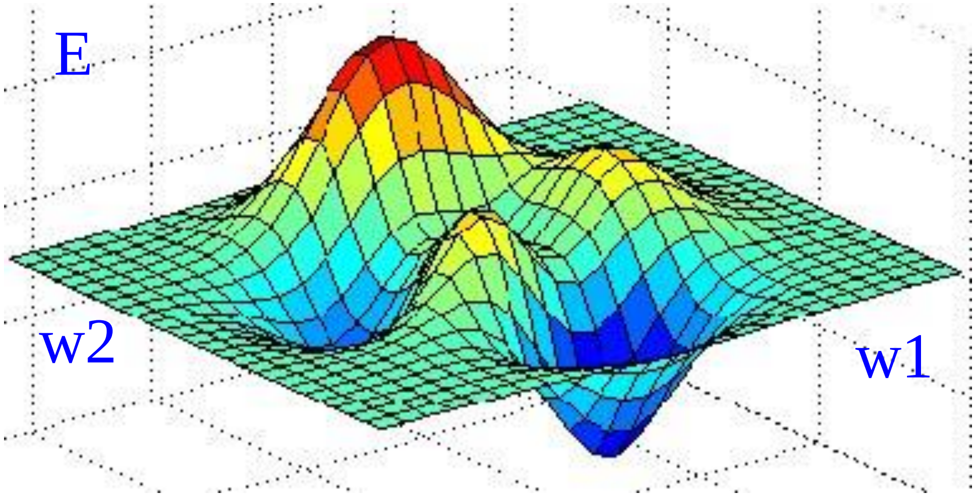
$$E = \frac{1}{2}\sum_i e_i^{\,2} = \frac{1}{2}\sum_i (d_i - a_i)^2$$

  - **d_i – desired value for the output neuron *i***
  - **a_i - actual NN value (i.e. predicted value by NN) for the output neuron *i***

- **Thus, backpropagation learning can be viewed as an optimization search in the weight space**
  - **Goal state – the set of weights for which the performance index (error E) is minimum**
  - **Search method – hill climbing**

Error Landscape in Weight Space

- E is a function of the weights (e.g. E is a function of w1 and w2)
- 1 state = 1 set of weights (w1 and w2 in this case)
- Several local minima and one global minimum



E as a function of w1 and w2

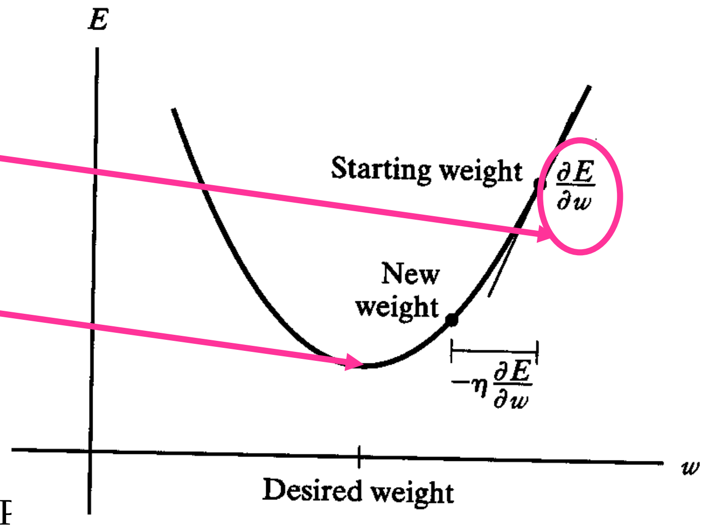
- How to minimize the error? Take steps downhill
  - Will find a local minimum (the one that is closest to the starting position), i.e. not guaranteed to find the global minimum (except when there is only one minimum)
  - How to get to the bottom as fast as possible? (i.e. we need to know what direction to move that will make the largest reduction in error)

Steepest Gradient Descent

- The direction of the *steepest descent* is called *gradient* and can be computed (dE/dw)
- A function *decreases* most rapidly when the direction of movement is *in the direction of the negative of the gradient*
- Hence, we want to adjust the weights so that the change moves the NN down the error surface in the direction of the locally steepest descent, given by the negative of the gradient
- η - learning rate, defines the step; typically in the range (0,1)

Image from Dunham

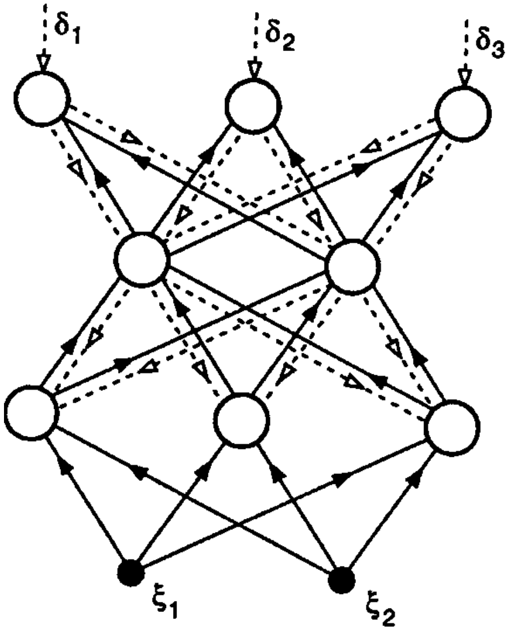
- Gives the slope (gradient) of the error function for one weight
- We want to find the weight where the slope (gradient) is 0


E
Starting weight
$\frac{\partial E}{\partial w}$
New weight
$-\eta\frac{\partial E}{\partial w}$
w
Desired weight

Backpropagation Algorithm - Idea

- **The backpropagation algorithm adjust weights by working backward from the output layer to the input layer**
- **Calculate the error and propagate this error from layer to layer**
- **2 approaches**
  - **Batch – weights are adjusted once after all training examples are applied and the total error is calculated**
  - **Incremental – the weights are adjusted after each training example is applied**
    - **Called also stochastic (or approximate) gradient descent**
    - **This is the preferred method – it is usually faster than batch learning and finds better solutions (can escape local optima)**

Image from Hertz, Krogh & Palmer



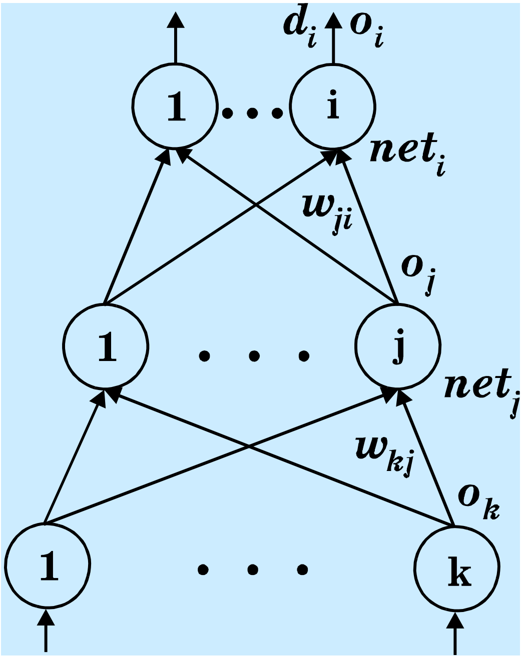
- **Solid lines - forward propagation of signals**
- **Dashed lines – backward propagation of error**

Backpropagation - Notation

- **A neural network with one hidden layer**
- **Indexes for the neurons: *k* over inputs, *j* over hidden, *i* over output neurons**
- **E – error function that we would like to minimize after each training example (i.e. incremental mode):**

$$E = \frac{1}{2}\sum_i (d_i - o_i)^2$$

d_i target value of neuron *i* for the current example p
o_i actual value of neuron *i* for the current example p



• Let's express E in terms of the weights of the network and the values of the current example!

• Recall that the weights are the parameters of the NN

Expressing E in terms of the NN weights and input example

1. Input for the hidden neuron *j* for p

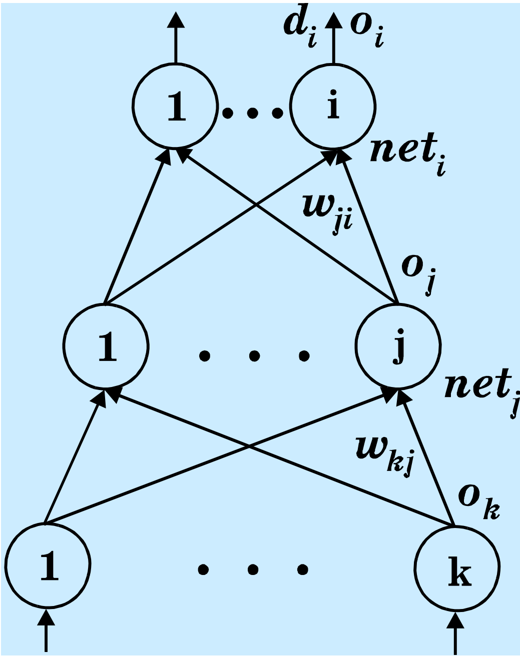
$$net_j = \sum_k w_{kj}.o_k + b_j$$

2. Activation of neuron *j* as function of its input:

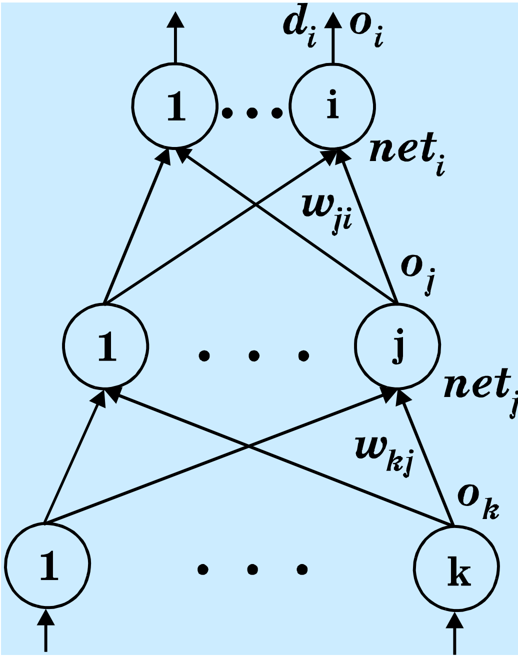
$$o_j = f(net_j) = f(\sum_k w_{kj}.o_k + b_j)$$

3. Input for the output neuron *i* :

$$net_i = \sum_j w_{ji}.o_j + b_i = \sum_j w_{ji} \cdot f(\sum_k w_{kj}.o_k + t_j) + b_i$$



Expressing E in terms of the NN weights and input example (2)



4. Output for the output neuron ***i*** **:**

$$o_i = f(net_i) = f(\sum_j w_{ji}.o_j + b_j) =$$

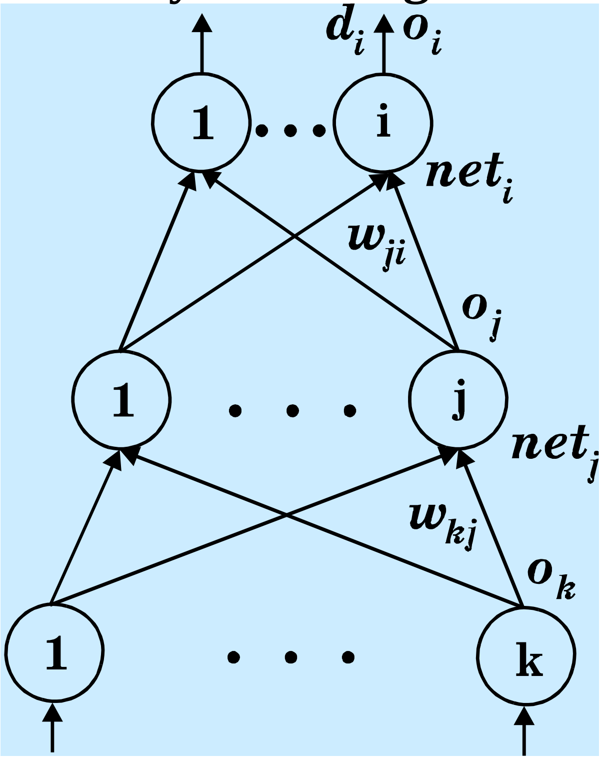
$$= f(\sum_j w_{ji}.f(\sum_k w_{kj}.o_k + b_j) + b_i)$$

5. Substituting 4 into E:

$$E = \frac{1}{2}\sum_i \left(d_i - o_i\right)^2 = \frac{1}{2}\sum_i \left[d_i - f(\sum_j w_{ji}.f(\sum_k w_{kj}.o_k + b_j) + b_i))\right]^2$$

Backpropagation – Derivation

- **Steepest gradient descent:** **adjust the weights proportionally to the negative of the error gradient**



For a weight w_{ji} to an *output neuron*:

$$\Delta w_{ji} = -\eta \frac{\partial E}{\partial w_{ji}} = -\eta \frac{\partial E}{\partial o_i} \frac{\partial o_i}{\partial net_i} \frac{\partial net_i}{\partial w_{ji}} =$$

<- chain rule

$$= -\eta 2 \frac{1}{2} (d_i - o_i)(-1) \frac{\partial o_i}{\partial net_i} o_j =$$

$$= \eta (d_i - o_i) f'(net_i) o_j = \eta \delta_i \cdot o_j,$$

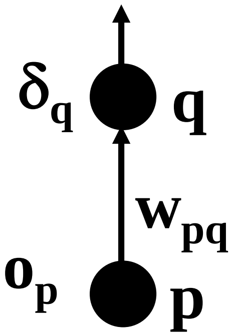
$$where \quad \delta_i = (d_i - o_i) f'(net_i)$$

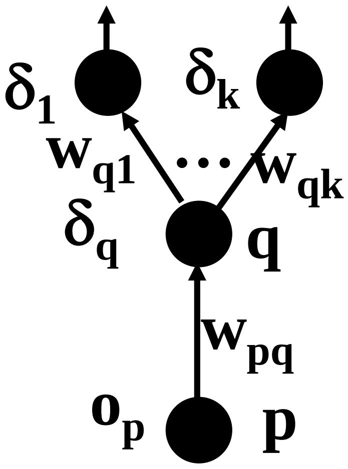
For a weight w_{kj} to a *hidden neuron:*

$$\Delta w_{kj} = -\eta \frac{\partial E}{\partial w_{kj}} = -\eta \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial w_{kj}} = \eta \sum_i (d_i - o_i) f'(net_i) w_{ji} f'(net_j) o_k$$

$$= \eta \sum_i \delta_i w_{ji} f'(net_j) o_k = \eta \delta_j o_k, \quad where\ \delta_j = f'(net_j) \sum_i \delta_i w_{ji}$$

Backpropagation Rule - Summary





$w_{pq}(t)$ **- weight from node *p* to node *q* at time *t***

$$w_{pq}(t+1) = w_{pq}(t) + \Delta w_{pq}$$

$\Delta w_{pq} = \eta \cdot \delta_q \cdot o_p$ **- weight change**

- **The weight change is proportional to the output activation of neuron p and the error δ of neuron q**
- δ **is calculated in 2 different ways:**
  - **q is an output neuron** $\delta_q = (d_q - o_q) \cdot f'(net_q)$
  - **q is a hidden neuron** $\delta_q = f'(net_q) \sum_i w_{qi}\, \delta_i$ **(*i* is over the nodes in the layer above *q*)**

Derivative of the activation function used in neuron q with respect to the input of q (netq)

Exercise

$$Given: \quad f(x) = \frac{1}{1+e^{-x}}$$

$$Show\ that:$$

$$f'(x) = f(x)(1 - f(x))$$

Solution

$$Given \quad f(x)=\frac{1}{1+e^{-x}}, \quad show\ that\ f'(x)=f(x)(1-f(x))$$

$$f'(x)=\frac{1'(1+e^{-x})-(1+e^{-x})'1}{(1+e^{-x})^2}=\frac{0-(-1)e^{-x}}{(1+e^{-x})^2}=$$

$$=\frac{e^{-x}}{(1+e^{-x})^2} \qquad (1)$$

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

$$=\frac{1}{1+e^{-x}}\frac{e^{-x}}{1+e^{-x}}=\frac{e^{-x}}{(1+e^{-x})^2} \qquad (2)$$

$$As\ (1)=(2) => f'(x)=f(x)(1-f(x))$$

Complete Rule for Sigmoid Neurons

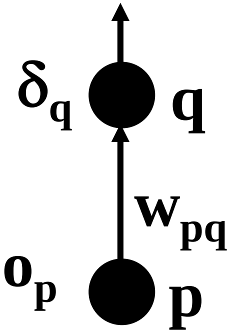
- From the formulas for δ => we must be able to calculate the derivatives for f. For a sigmoid transfer function:

$$f(net_m) = o_m = \frac{1}{1+e^{-net_m}}$$

$$f'(net_m) = \frac{\partial o_m}{\partial net_m} = \frac{\partial\left(\frac{1}{1+e^{-net_m}}\right)}{\partial net_m} =$$

$$= \frac{e^{-net_m}}{\left(1+e^{-net_m}\right)^2} = o_m \cdot (1-o_m)$$

The same derivation as in the example on the previous slide, just different notation



- Thus, backpropagation errors for a network with sigmoid transfer function:

- q is an output neuron

$$\delta_q = (d_q - o_q)\, o_q\,(1-o_q)$$

- q is a hidden neuron

$$\delta_q = o_q(1-o_q)\sum_i w_{qi}\,\delta_i$$

Backpropagation Algorithm – Summary (1)

1. Determine the architecture of the network

- **how many input and output neurons; what output encoding**
- **how many hidden neurons and layers**

2. Initialize all weights (biases incl.) to small random values, typically $\in$[-1,1]

3. Repeat until termination criterion satisfied:

```
*(forward pass)*
--Present a training example and propagate it through the network to calculate the actual output of the network
*(backward pass)*
--Compute the error (the $\delta$ values for the output neurons)
--Starting with output layer, repeat for each layer in the network:
        --propagate the $\delta$ values back to the previous layer
        --update the weights between the two layers
```

Backpropagation Algorithm – Summary (2)

4. The stopping criterion is checked at the end of each epoch. Training stops when 1 of these 2 conditions is satisfied:

epoch - 1 pass through the training set

1) The error on the training data is below a threshold

--This requires all training examples to be propagated again and the total error to be calculated (as in the perceptron)

--Threshold is set heuristically, e.g. 0.3;

--Different type of errors may be used, e.g. mean square or mean absolute or even accuracy (% of correctly classified examples)

2) Maximum number of epochs is reached

Instead of 1) or 1)+2) we can use an alternative stopping criterion called *Early stopping using a validation set* to prevent overfiting, see next slides

- It typically takes hundreds or thousands of epochs for an NN to converge

- *Try Matlab's demo nnd11bc!*

How to Determine if an Example is Correctly Classified?

- Binary encoding of the target outputs
  - Apply each example and get the resulting output activations of the output neurons; the example will belong to the class corresponding to the output neuron with highest activation.
  - Example: 3 classes; the outputs for ex. X are 0.3, 0.7, 0.5 => ex. X belongs to class 2
  - i.e. each output value is regarded as the probability of the example to belong to the class corresponding to this output

Backpropagation - Example

- **2 classes, 2-dim. input data**

  training set:

  ex.1: 0.6 0.1 | class 1 (banana)

  ex.2: 0.2 0.3 | class 2 (orange)

  …

- **Network architecture**
  - **How many inputs?**
  - **How many hidden neurons?**

  Heuristic:

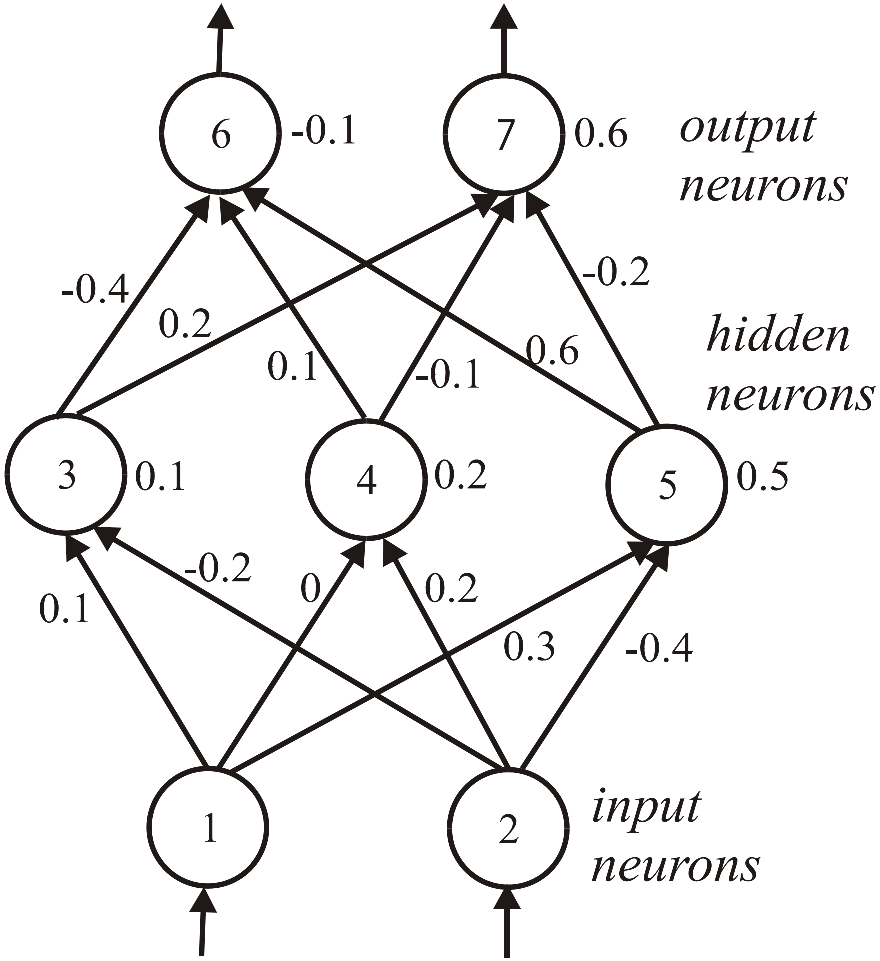
  n=(inputs+output_neurons)/2

  - **How many output neurons?**
  - **What encoding of the outputs?**

  e.g. 10 for class 1, 01 for class 0

- **Initial weights and learning rate**
  - **Let's η=0.1 and the weights are set as in the picture**

- **Sigmoid transfer function**



Backpropagation – Example (cont. 1)

- **1. Forward pass for ex. 1 - calculate the outputs o_6 and o_7**

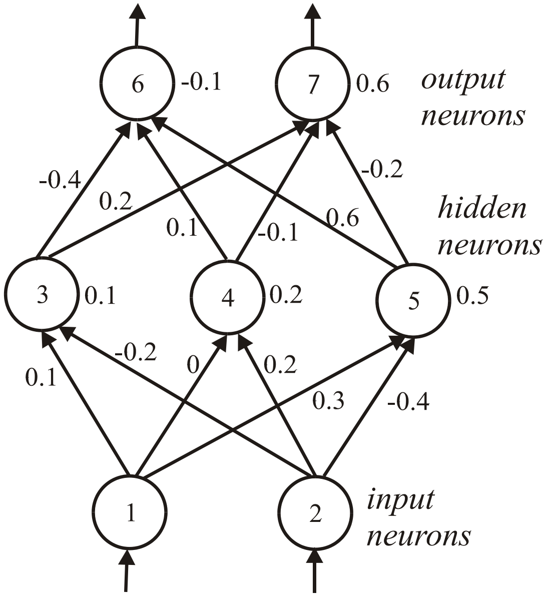
ex.1: o_1=0.6, o_2=0.1, target output 1 0, i.e. class 1

- **Activations of the hidden neurons:**

$net_3 = o_1 * w_{13} + o_2 * w_{23} + b_3 = 0.6*0.1+0.1*(-0.2)+0.1=0.14$
$o_3 = 1/(1+e^{-net3}) = 0.53$

$net_4 = o_1 * w_{14} + o_2 * w_{24} + b_4 = 0.6*0+0.1*0.2+0.2=0.22$
$o_4 = 1/(1+e^{-net4}) = 0.55$

$net_5 = o_1 * w_{15} + o_2 * w_{25} + b_5 = 0.6*0.3+0.1*(-0.4)+0.5=0.64$
$o_5 = 1/(1+e^{-net5}) = 0.65$



- **Activations of the output neurons:**

•$net_6 = o_3 * w_{36} + o_4 * w_{46} + o_5 * w_{56} + b_6 = 0.53*(-0.4)+0.55*0.1+0.65*0.6-0.1=0.13$
$o_6 = 1/(1+e^{-net6}) = 0.53$

$net_7 = o_3 * w_{37} + o_4 * w_{47} + o_5 * w_{57} + b_7 = 0.53*0.2+0.55*(-0.1)+0.65*(-0.2)+0.6=0.52$
$o_7 = 1/(1+e^{-net7}) = 0.63$

Backpropagation – Example (cont. 2)

- 2. Backward pass for ex. 1
  - Calculate the output errors δ_6 and δ_7 (note that $d_6=1$, $d_7=0$ for class 1)
  - $\delta_6 = (d_6-o_6) * o_6 * (1-o_6)=(1-0.53)*0.53*(1-0.53)=0.12$
  - $\delta_7 = (d_7-o_7) * o_7 * (1-o_7)=(0-0.63)*0.63*(1-0.63)=-0.15$
  - Calculate the new weights between the hidden and output neurons ($\eta=0.1$)

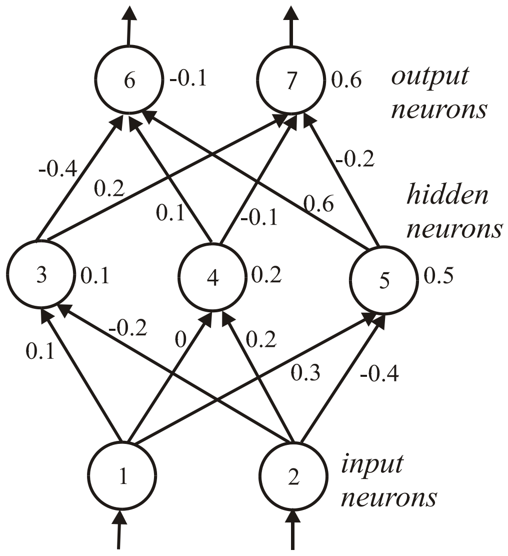
$\Delta w_{36}= \eta * \delta_6 * o_3 = 0.1*0.12*0.53=0.006$

$w_{36}^{new} = w_{36}^{old} + \Delta w_{36} = -0.4+0.006=-0.394$

$\Delta w_{37}= \eta * \delta_7 * o_3 = 0.1*-0.15*0.53=-0.008$

$w_{37}^{new} = w_{37}^{old} + \Delta w_{37} = 0.2-0.008=-0.19$

Similarly for w_{46}^{new} , w_{47}^{new} , w_{56}^{new} and w_{57}^{new}



For the biases b_6 and b_7 (remember: biases are weights with input 1):

$\Delta b_6= \eta * \delta_6 * 1 = 0.1*0.12=0.012$

$b_6^{new} = b_6^{old} + \Delta b_6 = -0.1+0.012=-0.012$

Similarly for b_7

Backpropagation – Example (cont. 3)

- Calculate the errors of the hidden neurons δ_3, δ_4 and δ_5

$\delta_3 = o_3 * (1-o_3) * (w_{36} * \delta_6 + w_{37} * \delta_7) =$

$= 0.53*(1-0.53)(-0.4*0.12+0.2*(-0.15))=-0.019$

Similarly for δ_4 and δ_5

- Calculate the new weights between the input and hidden neurons (η=0.1)

$\Delta w_{13} = \eta * \delta_3 * o_1 = 0.1*(-0.019)*0.6=-0.0011$

$w_{13}^{new} = w_{13}^{old} + \Delta w_{13} = 0.1-0.0011=0.0989$

Similarly for w_{23}^{new}, w_{14}^{new}, w_{24}^{new}, w_{15}^{new} and w_{25}^{new}; b_3, b_4 and b_6

3. Repeat the same procedure for the other training examples

- Forward pass for ex. 2…backward pass for ex.2…
- Forward pass for ex. 3…backward pass for ex. 3…
- …
- Note: it’s better to apply input examples in random order

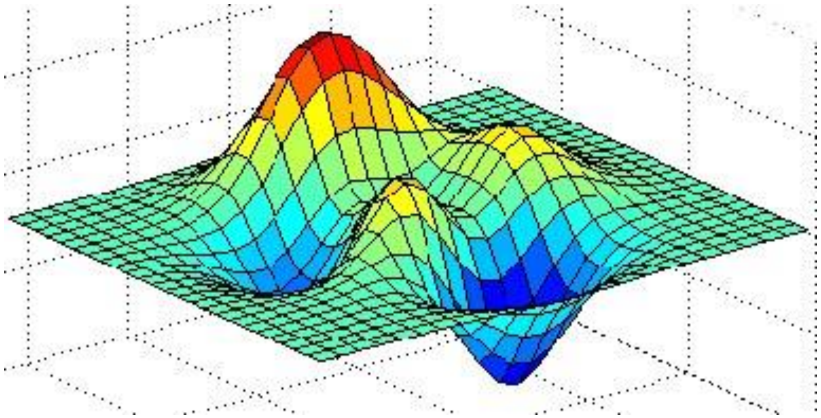
Backpropagation – Example (cont. 4)

4. At the end of the epoch – check if the stopping criterion is satisfied:

- **If yes: stop training**
- **If not, continue training:**
  - **epoch++**
  - **go to step 1**

Steepest Gradient Descent

- **Does the gradient descent guarantee that after each adjustment the error will be reduced?**
  - **No, if a local minimum is reached.**
- **Is the gradient descent optimal, i.e. will it find the global minimum?**
  - **No. It is guaranteed to find a minimum but it might be a local minimum not the global minimum.**
  - **However, a local minimum may be a good enough solution!**



Backpropagation’s error space: many local minima and 1 global minimum

Universality of Backpropagation

- **Boolean functions**
  - **Every boolean function of the inputs can be represented by network with a single hidden layer**
- **Continuous functions - universal approximation theorems**
  - **Any continuous function can be approximated with arbitrary small error by a network with one hidden layer (Cybenko 1989, Hornik et al. 1989):**
  - **Any function (including discontinuous) can be approximated to arbitrary small error by a network with two hidden layers (Cybenco 1988)**
- **These are existence theorems – they say the a solution (NN) exist but don't say how to choose the number of hidden neurons!**
  - **For a given NN it is hard to say exactly which functions can be represented and which can not**

Overfitting

- **Occurs when**
  - **Training examples are noisy**
  - **Small dataset set not representative of the whole data**
  - **Number of the free (trainable) parameters is bigger than the number of training examples**
    - **Which are the trainable parameters in a NN? The weights.**
- **Preventing overtraining**
  - **Use network that is just large enough to provide an adequate fit**
    - **Ockham'Razor – don't use a bigger net when a smaller one will work**
    - **The network should not have more free parameters than the number of training examples**
- **However, it is difficult to know beforehand how large a network should be for a specific application!**

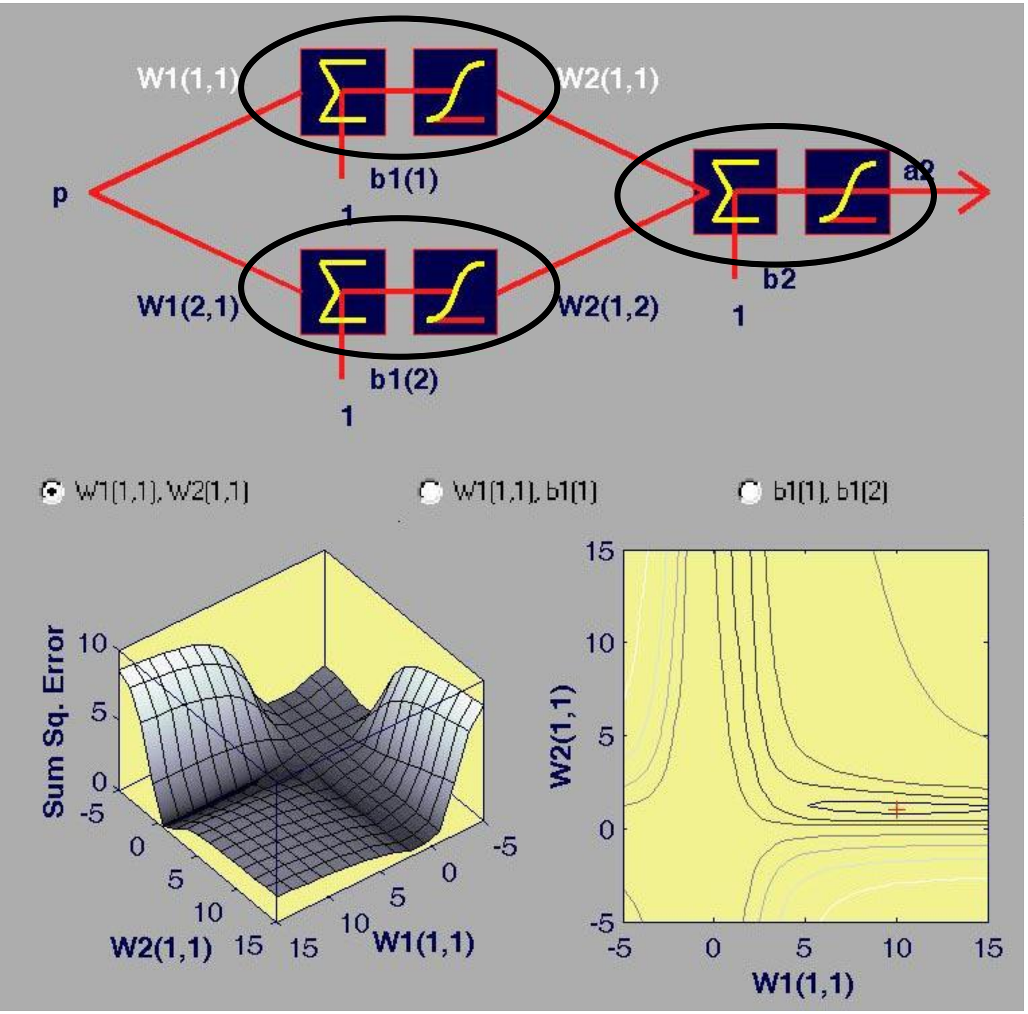
Preventing Overtraining - Validation Set Approach

- **Also called an *early stopping method***
- **Available data is divided into 3 subsets**
  - **Training set**
    - **Used for computing the gradient and updating the weights**
  - **Validation set**
    - **The error on the validation set is monitored during the training**
    - **This error will normally decrease during the initial phase of training (as does the training error)**
    - **However, when the network begins to overfit the data, the error on the validation set will typically begin to rise**
    - **Training is stopped when the error on the validation set increases for a pre-specified number of iterations and the weights and biases at the minimum of the validation set error are returned**
  - **Testing set**
    - **Not used during training but to compare different algorithms once training has completed**

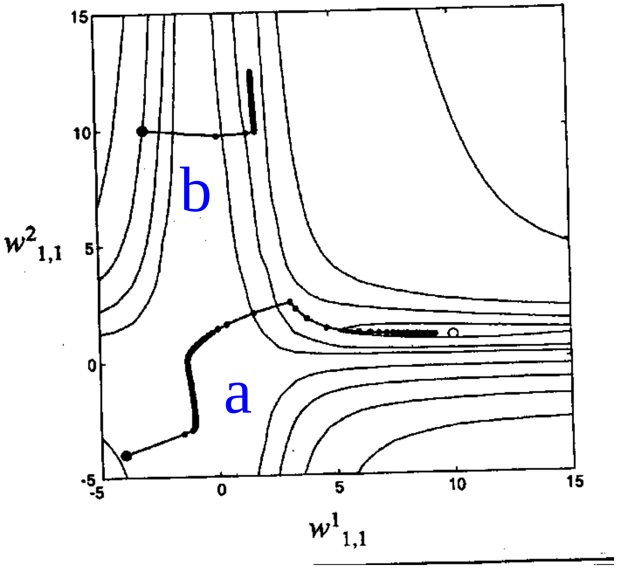
Preventing Overtraining – Cross Validation Approach

- **Problems with the validation set approach – small data sets**
  - **Not enough data may be available to provide a validation set**
  - **Overfitting is most severe for small data sets**
- **K-fold cross validation may be used**
  - **Perform k fold cross validation**
  - **Each time determine the number of epochs *ep* that achieved best performance on the respective test partition**
  - **Calculate *ep_mean,* the mean of *ep***
  - **Final run: train the network on all examples for *ep_mean* epochs**

Error Surface and Convergence - Example



● W1(1,1), W2(1,1) ○ W1(1,1), b1(1) ○ b1(1), b1(2)

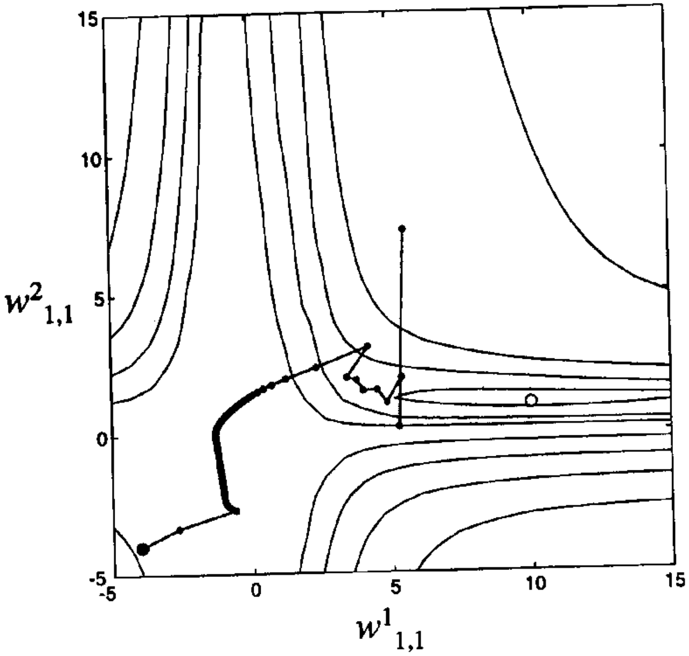


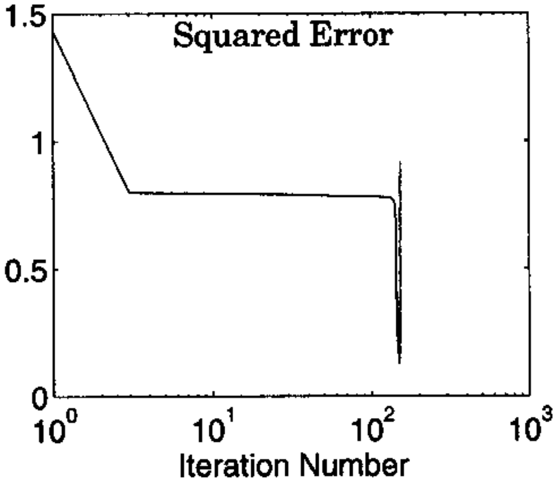
Try nnd12sd1!

- **Problem 1:** **Path b gets trapped in a local minimum**
  - **What can be done? Try different initializations!**
- **Problem 2:** **Path a converges to the optimum solution but is very slow**
  - **What can we do?**

Speeding up the Convergence

- **Solution 1: Increase the learning rate**
  - **Faster on the flat part but unstable when falling into the steep valley that contains the minimum point – overshooting the minimum**





Try nnd12sd2!

Speeding up the Convergence (2)

- **Solution 2: Smooth out the trajectory by averaging the weight update**
- **How? Make the current update dependent on the previous by introducing an extra term (called *momentum term*) in the weight adaptation equation**
- **Before:**

$$w_{pq}(t+1) = w_{pq}(t) + \Delta w_{pq}(t), \text{ where } \quad \Delta w_{pq}(t) = \eta \delta_q o_p$$

- **Now:**

$$\Delta w_{pq}(t) = \eta \delta_q o_p + \overbrace{\mu\left(w_{pq}(t) - w_{pq}(t-1)\right)}^{\text{momentum term}}$$

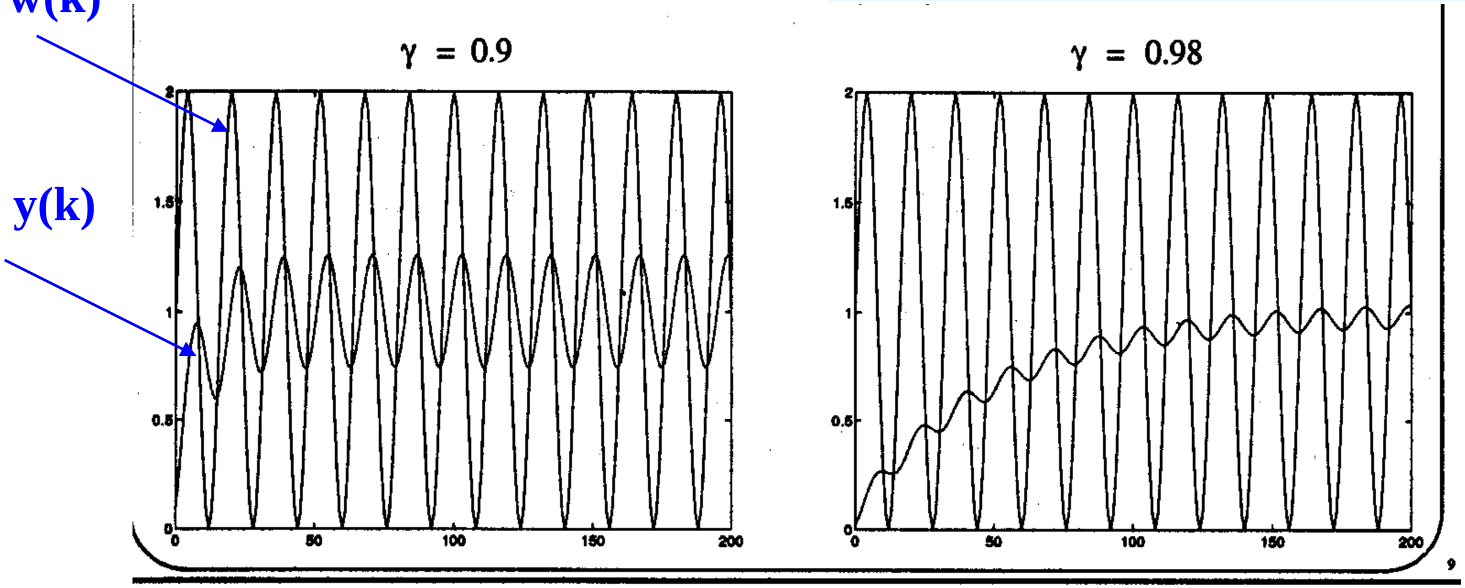
- **If previous change was large -> current change will be large and vice versa**
- **Speeds up convergence along shallow gradient valleys (convergence is slow as the direction that has to be followed has a very small gradient -> oscillations)**
- **The use of momentum typically smooths out the oscillations and produces a more stable trajectory**

Momentum

- The theory behind momentum comes from linear filters
- Observation
  - convergence might be improved if by smoothing out the trajectory by averaging the updates to the parameters
- First order filter:
  - w(k) – input, y(k) – output
  - γ - momentum coefficient

$$y(k) = \gamma y(k-1) + (1-\gamma) w(k) \quad 0 \le \gamma < 1$$

$$w(k) = 1 + \sin\left(\frac{2\pi k}{16}\right)$$



Based on Hagan, Demuth & Beale

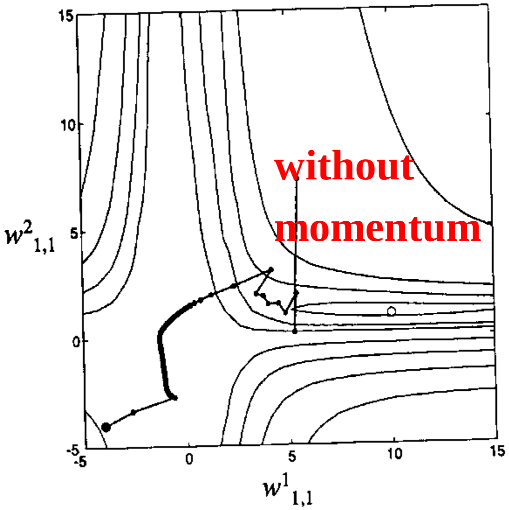
First Order Linear Filter

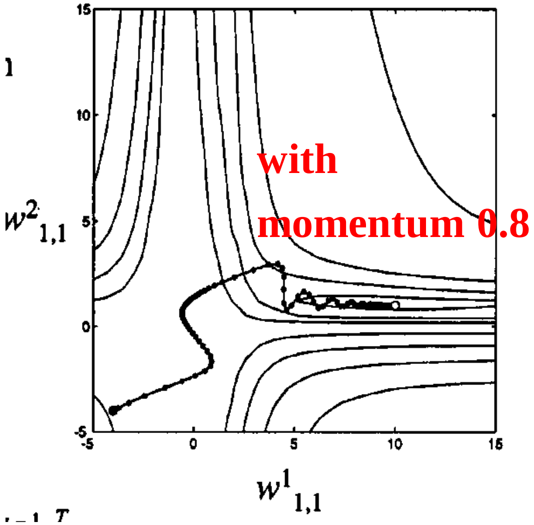
- Oscillation in the filter output *y(k)* is less than the oscillation in the filter input *w(k)*
- As the momentum coefficient increases, the oscillation in the output is reduced
- The average filter output is the same as the average filter input
  - Although as the momentum increases the filter output is slow to respond

⇒ The filter tends to reduce the amount of oscillation, while still tracking the average value

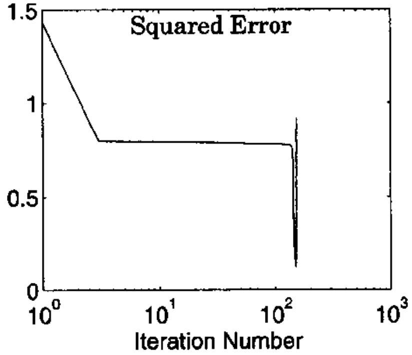
Backpropagation with Momentum - Example

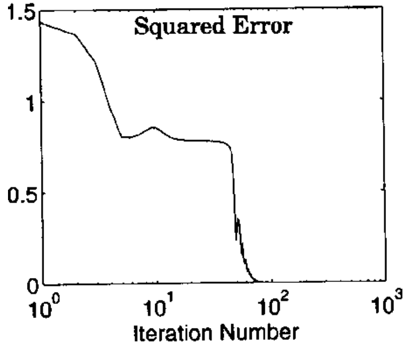
- Example – the same learning rate and initial position:





- Smooth and faster convergence
- Stable algorithm





- By the use of momentum we can use a larger learning rate while maintaining the stability of the algorithm

- ***Try nnd12mo!***

- Typical momentum values used in practice: 0.6-0.9

More on the Learning Rate

- Constant throughout training (standard steepest descent)
- The performance is very sensitive to the proper setting of the learning rate
  - Too small – slow convergence
  - Too big – oscillation, overshooting of the minimum
  - It is not possible to determine the optimum learning rate before training as it changes during training and depends on the error surface
- Variable learning rate
  - goal: keep the learning rate as large as possible while keeping learning stable
  - Several algorithms have been proposed

Some Practical Tricks

Y. LeCun, L. Bottou, G.B. Orr and K.-R Mueller, Efficient BackProp, http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf

1. **Stochastic (incremental) or batch learning**
   - **Use stochastic - it is faster and often finds better solutions**
2. **Input examples**
   - **Shuffle the training set so that successive training examples never (rarely) belong to the same class**
   - **Present input examples that produce a large error more often that examples that produce a small error**
   - **Normalise the input variables to a standard deviation of 1**
   - **If possible, use de-correlated input variables**
3. **Transfer function**
   - **Use a tangent sigmoid not the standard one – faster convergence**

Some Practical Tricks (2)

4. **Learning rate – a separate learning rate for each neuron so that all weights converge roughly with the same speed**
   - **Learning rate should be proportional to the square root of the number of inputs to the neuron**
   - **Learning rates in the lower layers should be higher than in the higher layers**

5. **Training - useful variations and extensions of the gradient descent method – Lavenberg-Marquardt method and conjugate gradient**
   - **If the training set is large (> few hundred examples) and the task is classification, use stochastic gradient descent with careful tuning or the Lavenberg-Marquardt method**
   - **If the training set is not too large or the task is regression, use the conjugate gradient method**

Limitations and Capabilities

- **Backpropagation NNs are used for supervised learning (classification or regression)**
- **Theoretically they can**
  - **Form arbitrary decision boundaries (i.e. both linear and non-linear)**
  - **Are universal approximators – can approximate any function arbitrary well**
- **In practice:**
  - **May not always find a good solution – can be trapped in a local minimum**
  - **Performance is sensitive to the starting conditions (weights initialization)**
  - **Sensitive to the number of hidden layers and neurons**
    - **Too few neurons – underfitting, unable to learn what you want it to learn**
    - **Too many – overfitting, learns slowly**
    - **=> the architecture of a MLP network is not completely constrained by the problem to be solved as the number of hidden layers and neurons are left to the designer**

Limitations and Capabilities – cont.

- **In practice (continuation):**
  - **Sensitive to the value of the learning rate**
    - **Too small – slow learning**
    - **Too big – instability or poor performance**
  - **The proper choices depends on the nature of examples**
    - **Trial and error**
    - **Refer to the choices that have worked well in similar problems**
    - **=> successful application of NNs requires time and experience**

- **Backpropagation algorithm – summary:**
  - **uses steepest Gradient Descent (GD) for minimizing the mean square error**
  - **Standard GD is slow as it requires small learning rate for stable learning**
  - **GD with momentum is faster as it allows higher learning rate while maintaining stability**
  - **There are several variations of the backpropagation algorithm**

Interesting Classical NN Applications

- **A few examples of the many significant applications of NNs (before deep learning)**
- **Network design was the result of several months trial and error experimentation**
- **Moral: NNs are widely applicable but they cannot magically solve problems; wrong choices lead to poor performance**
- ***"NNs are the second best way of doing just about anything" John Denker***
    - **i.e. NNs provide passable performance on many tasks that would be difficult to solve explicitly with other techniques**

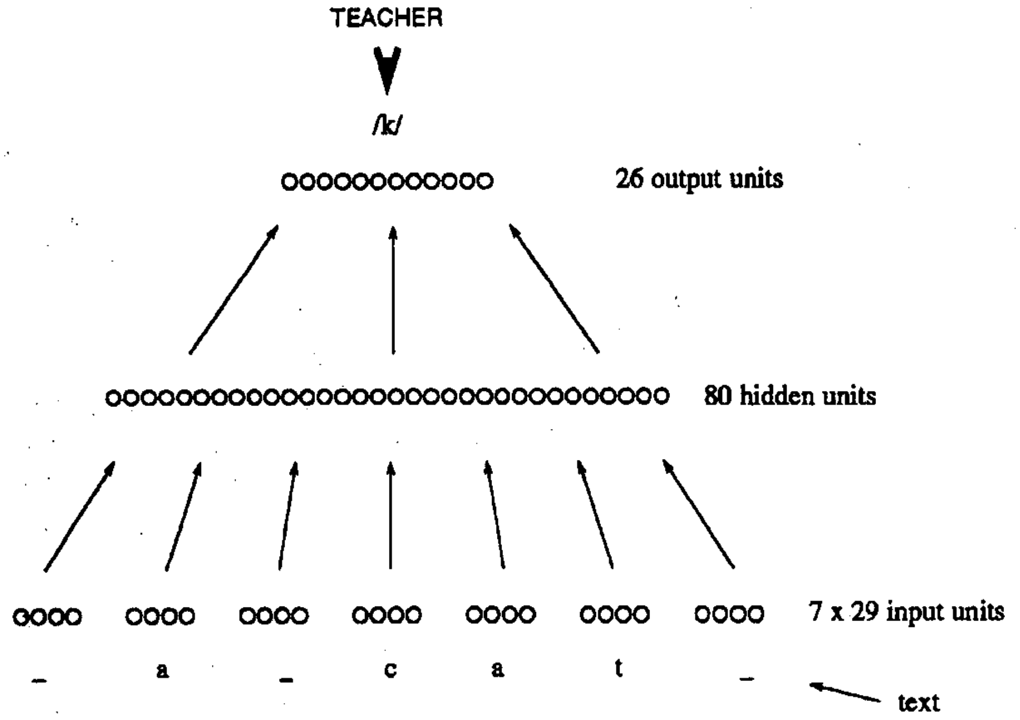
NETtalk

- Sejnowski and Rosenberg 1987
- Pronunciation of written English
  - Fascinating problem in linguistics as English is not a phonetic language
  - Task with high commercial profit
  - How?
    - Task 1: Mapping the text stream to phonemes

    e.g. cat -> [kat], believe->[bi'li:v]
    - Task 2: Passing the phonemes to speech generator
- Task for the NN: learning to map the text to phonemes (task 1)
  - Good task for a NN as most of the mapping rules are approximately correct
  - e.g. cat [k], century [s]

NETtalk -Architecture

- **203 input neurons – 7x29**
  - **7 – size of the sliding window (the character to be pronounced and the 3 characters before and after it)**
  - **29 possible characters (26 letters + blank, period, other punctuation symbols)**
- **80 hidden neurons**
- **26 output neurons**
  - **1 for each phoneme**

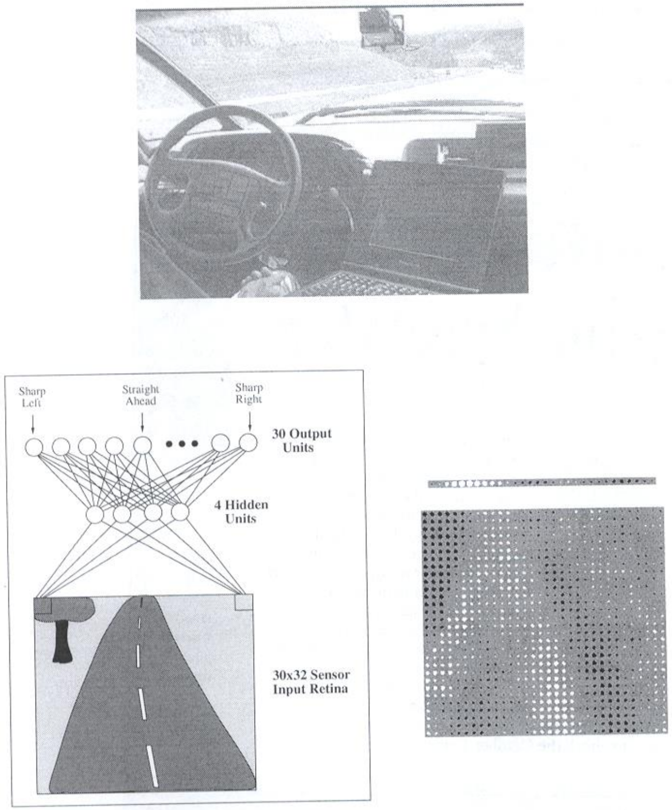


NETtalk - Performance

- **Training set**
  - **1024 pairs of word-phonemes (hand transcribed)**
  - **Accuracy on training set: 90% after 50 epochs**
  - **Why not 100% - possible reasons?**
  - **A few dozen hours of training time + a few months of experimentation with different architectures**
- **Testing**
  - **Accuracy 78%**
- **Importance**
  - **A good showpiece for the philosophy of NNs**
  - **The network appears to mimic the speech patterns of young children – incorrect bubble at first (as the weights are random), then gradually improving to become understandable**

Driving Motor Vehicles

- Pomerleau et al., 1993
- ALVIN (Autonomous Land Vehicle In a Neural Network)
- Learns to drive a van along a single lane on a highway
  - Once trained on a particular road, ALVIN can drive at speed > 40 miles per hour
  - Chevy van and US Army HMMWV personnel carrier
    - computer-controlled steering, acceleration and braking
    - sensors: color stereo video camera, radar, positioning system, scanning laser finders



Pictures from Mitchell

ALVINN - Architecture

- **Fully connected backpropagation NN with 1 hidden layer**
  - **960 input neurons – the signal from the camera is preprocessed to yield 30x32 image intensity grid**
  - **5 hidden neurons**
  - **32 output neurons corresponding to directions**
    - **If the output node with the highest activation is**
      - **The <u>left most</u> , than ALVINN turns sharply <u>left</u>**
      - **The <u>right most</u>, than ALVINN turns sharply <u>right</u>**
      - **A node <u>between</u> them, than ALVINN directs the van in a proportionally <u>intermediate</u> direction**
    - **Smoothing the direction – it is calculated as average suggested not only by the output node with highest activation but also by the node's immediate neighbours**

ALVINN - Training

- Training examples (image-direction pairs)
  - Recording such pairs when human drives the vehicle
  - After collecting 5 min such data and 10 min training, ALVINN can drive on its own

  - Potential problem: as the human is too good and (typically) does not stray from the lane, there are no training examples that show how to recover when you are misaligned with the road
  - Solution: ALVINN corrects this by creating synthetic training examples – it rotates each video image to create additional views of what the road would look like if the van were a little off course to the left or right

ALVINN - Results

- Impressive results
  - ALVINN has driven at speeds up to 70 miles per hour for up to 90 miles on public highways near Pittsburgh
  - Also at normal speeds on single lane dirt roads, paved bike paths, and two lane suburban streets
- Limitations
  - Unable to drive on a road type for which it hasn't been trained
  - Not very robust to changes in lighting conditions and presence of other vehicles
- Comparison with traditional vision algorithms
  - Use image processing to analyse the scene and find the road and then follow it
  - Most of them achieve 3-4 miles per hour

ALVINN - Discussion

- **Why is ALVINN so successful?**
  - **Fast computation - once trained, the NN is able to compute a new steering direction 10 times a second => the computed direction can be off by 10% from the ideal as long as the system is able to make a correction in a few tenths of a second**
  - **Learning from examples is very appropriate**
    - **No good theory of driving but it is easy to collect examples**
    - **Motivated the use of learning algorithm (but not necessary NNs)**
  - **Driving is continuous, noisy domain, in which almost all features contribute some information => NNs are better choice than some other learning algorithms (e.g. DTs)**

Acknowledgements

- **J. Hertz, A. Krogh, R.G. Palmer, *Introduction to the Theory of Neural Computation*, Addison-Wesley**
- **T. Mitchell, *Machine Learning***
- **M. Dunham, *Data Mining*, Pearson Education**
- **Hagan, Demuth, Beale, *Neural Network Design*, PWS, Thomson**
- ***Matlab NN toolbox***
- **Y. LeCun, L. Bottou, G.B. Orr and K.-R Mueller, *Efficient BackProp, Neural networks: Tricks of the Trade*, pp.9-48, http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf**