COMP4302/COMP5322, Lecture 4, 5

NEURAL NETWORKS

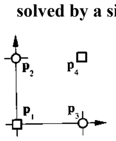
Backpropagation Algorithm

Backpropagation - Outline

- **Backpropagation**
  - **XOR problem**
  - **neuron model**
- **Backpropagation algorithm**
  - **Derivation**
  - **Example**
  - **Error space**
  - **Universality of backpropagation**
  - **Generalization and overfitting**
- **Heuristic modifications of backpropagation**
  - **Convergence example**
  - **Momentum**
  - **Learning rate**
- **Limitations and capabilities**
- **Interesting applications**

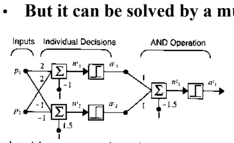
XOR problem - Example

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



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

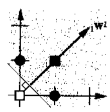
- **But it can be solved by a multilayer perceptron network:**



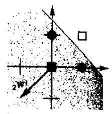
• The 2 perceptrons in the input layer identify linearly separable parts, and their outputs are combined by another perceptron to form the final solution

Boundary Investigation for the XOR Problem

- first layer boundaries:

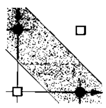


1st layer, 1st neuron;
1st decision boundary



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

- 2d layer combines the two boundaries together:



2d layer, 1st neuron:
combined boundary

Can We Train Such a Network?

- There exist a two layer perceptron network capable of solving the XOR problem!
  - Rosenblatt and Widrow were aware of this
- But how can we train such network to learn from examples?
  - Rosenblatt and others were not able to successfully modify the perceptron rule to train these more complex networks
  - 1969 – book "Perceptrons" by Minsky and Papert
    - "there is no reason to suppose that any of the virtues of perceptrons carry over to the many-layered version"
    - Mortal blow in the area – the majority of scientific community walked away from the field of NNs…
- Discovery of the backpropagation algorithm
  - 1974 by Paul Werbos
    - his thesis presented the algorithm in the context of general networks, with NNs as a special case, and was <u>not</u> disseminated in the NN community
  - Rediscovered by David Rumelhart, Geoffrey Hinton, Ronald Williams 1986; David Parker 1985; Yann Le Cun 1985

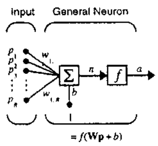
Can We Train Such a Network? – cont.

  - Book: Parallel Distributed Processing (1986) by Rumelhart and McClelland

- Multi-layer networks can be trained by the *backpropagation* algorithm (also called *generalized gradient descent* and *generalized delta rule*)

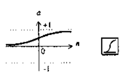
- Multi-layer networks trained with backpropagation are currently the most widely used NNs

Backpropagation – Neuron Model



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

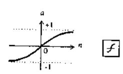
• any *differentiable* transfer function f can be used; most frequently the sigmoid and tan-sigmoid (hyperbolic tangent sigmoid) functions are used:



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

$a = logsig(n)$

Log-Sigmoid Transfer Function



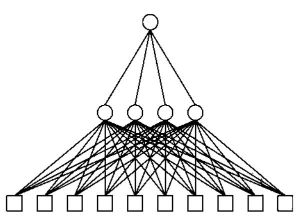
$$a = \frac{e^{n}-e^{-n}}{e^{n}+e^{-n}}$$

$a = tansig(n)$

Tan-Sigmoid Transfer Function

Backpropagation Network

- Example of a backpropagation network with one hidden layer



Backpropagation Learning

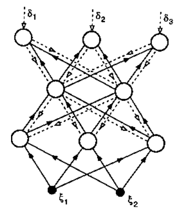
- Similar to ADALINE's learning
- Supervised learning
- We define an error function (based on the training set) and would like to minimize it by adjusting the weights using hill climbing algorithm
- Mean square error (mse) is the performance index
  - Error - difference between the target (*t*) and actual (*a*) network output
  - Mean square error of one output neuron over all *n* examples:

$$mse = \frac{1}{n}\sum_{k=1}^{n} e(k)^2 = \frac{1}{n}\sum_{k=1}^{n}(t(k)-a(k))^2$$

- Multilayer perceptrons used the *backpropagation* algorithm to adjusts the weights and biases of the network in order to minimize the mean square error (over all output and all examples)

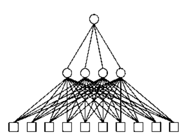
Backpropagation Algorithm

- Backpropagation (generalized gradient descent) is a generalization of the LMS algorithm
- We define an error function and would like to minimize it using the gradient descent
  - mean squared error is the performance index
- This is achieved by adjusting the weights
- The generalized delta rule does this by calculating the error for the current input example and then backpropagating this error from layer to layer



Backpropagation Algorithm – Intuitive Understanding

- How to adjust the weights?
- For output neuron the desired and target output is known, so the adjustment is simple
- For hidden neurons – it's not that obvious!
  - Intuitively: if a hidden neuron is connected to output with large error, adjust its weights a lot, otherwise don't alter the weights too much
  - Mathematically: weights of a hidden neuron are adjusted in direct proportion to the error in the neuron to which it is connected



Backpropagation - Derivation

- a neural network with one hidden layer; indexes: *i* over output neurons, *j* over hidden, *k* over inputs, ζ over input patterns
- mse (over all neurons, over all patterns):

$$E = \frac{1}{2}\sum_{\zeta i}(d_i^\zeta - o_i^\zeta)^2$$

d_i^ζ target output of neuron *i* for input pattern ζ

o_i^ζ actual output of neuron *i* for input pattern ζ

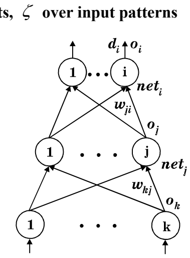
- Express E in terms of weights and input signals

1. Input for the hidden neuron *j* for ζ :

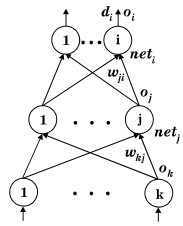
$$net_j^\zeta = \sum_k w_{kj}.o_k^\zeta + b_j$$

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

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



Backpropagation – Derivation - 2



3. Input for the output neuron *i* :

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

4. Output for the output neuron *i* :

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

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

5. Substituting 4 into E:

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

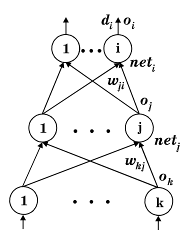
6. Steepest gradient descent: adjust the weights so that the change moves the system down the error surface in the direction of the locally steepest descent, given by the negative of the gradient: **where**

$$\Delta w_{ji} = -\eta \cdot \frac{\partial E}{\partial w_{ji}} = \eta \cdot \sum_\zeta \left(d_i^\zeta - o_i^\zeta\right) \cdot f'\left(net_i^\zeta\right) \cdot o_j^\zeta = \eta \cdot \sum_\zeta \delta_i^\zeta \cdot o_j^\zeta \qquad \delta_i^\zeta = \left(d_i^\zeta - o_i^\zeta\right) \cdot f'\left(net_i^\zeta\right)$$

for output neuron

Backpropagation – Derivation - 3

8. For hidden neuron - calculating the derivatives using the chain rule:



$$\Delta w_{kj} = -\eta \cdot \frac{\partial E}{\partial w_{kj}} = -\eta \cdot \frac{\partial E}{\partial o_j^\zeta} \cdot \frac{\partial o_j^\zeta}{\partial w_{kj}} =$$

$$= \eta \cdot \sum_{\zeta i} \left(d_i^\zeta - o_i^\zeta\right) \cdot f'\left(net_i^\zeta\right) \cdot w_{ji} \cdot f'\left(net_j^\zeta\right) \cdot o_k^\zeta =$$

$$= \eta \cdot \sum_{\zeta i} \delta_i^\zeta \cdot w_{ji} \cdot f'\left(net_j^k\right) \cdot o_k^\zeta = \eta \cdot \sum_\zeta \delta_j^\zeta \cdot o_k^\zeta$$

where $\delta_j^\zeta = f'\left(net_j^\zeta\right) \cdot \sum_i w_{ji} \cdot \delta_i^\zeta$

for hidden neuron

9. In general, for a connection from *p* to *q* :

$$\Delta w_{pq} = \eta \cdot \sum_{inputpatterns} \delta_q \cdot o_p \qquad w_{pq}^{new} = w_{pq}^{old} + \Delta w_{pq}$$

where *o* is activation of an input or hidden neuron and δ is given either by eq. 7 (output neuron) or eq. 8 (hidden neuron)

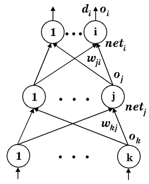
Backpropagation – Derivation - 4

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

$$f(net_l^\zeta) = o_l^\zeta = \frac{1}{1 + e^{-net_l^\zeta}}$$

$$\frac{\partial o_l^\zeta}{\partial net_l^\zeta} = \frac{\partial\left(\frac{1}{1+e^{-net_l^\zeta}}\right)}{\partial net_l^\zeta} =$$

$$= \frac{e^{-net_l^\zeta}}{\left(1 + e^{-net_l^\zeta}\right)^2} = o_l^\zeta \cdot \left(1 - o_l^\zeta\right)$$



11. Backpropagation rule for sigmoid transfer function:

output neuron

$$\delta_i^\zeta = \left(d_i^\zeta - o_i^\zeta\right) \cdot o_i^\zeta \cdot (1 - o_i^\zeta)$$

$$\Delta w_{ji} = \eta \cdot \sum_\zeta \left(d_i^\zeta - o_i^\zeta\right) \cdot o_i^\zeta \cdot (1 - o_i^\zeta) \cdot o_j^\zeta$$

hidden neuron

$$\delta_j^\zeta = o_j^\zeta \cdot \left(1 - o_j^\zeta\right) \cdot \sum_i w_{ji} \cdot \delta_i^\zeta$$

$$\Delta w_{kj} = \eta \cdot \sum_\zeta \delta_j^\zeta \cdot o_k^\zeta = \eta \cdot \sum_\zeta o_j^\zeta \cdot \left(1 - o_j^\zeta\right) \cdot \sum_i w_{ji} \cdot \delta_i^\zeta \cdot o_k^\zeta$$

Backpropagation - Summary

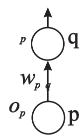
1. **Determine the architecture**
   - **how many input and output neurons; what output encoding**
   - **hidden neurons and layers**
2. **Initialize all weights and biases to small random values, typically $\in$[-1,1]**
3. **Repeat until termination criterion satisfied:**
   - **Present a training example and propagate it through the network *(forward pass)***
   - **Calculate the actual output**
   - **Adapt weights starting from the output layer and working backwards *(backward pass)***

$w_{pq}(t+1) = w_{pq}(t) + \Delta w_{pq}$ $\quad w_{pq}(t)$ **- weight from node *p* to node *q* at time *t***

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

$\delta_i = (d_i - o_i) \cdot o_i \cdot (1 - o_i)$ **- for output neuron *i***

$\delta_j = o_j \cdot (1 - o_j) \cdot \sum_i w_{ji} \cdot \delta_i$ **- for hidden neuron *j* (the sum is over the *i* nodes in the layer above the node *j*)**



Stopping Criteria

- **The stopping criteria is checked at the end of each epoch:**
  - **The error (mean absolute or mean square) at the end of an epoch is below a threshold**
    - **All training examples are propagated and the mean (absolute or square) error is calculated**
    - **The threshold is determined heuristicly – e.g. 0.3**
  - **Maximum number of epochs is reached**
  - **Early stopping using a validation set (TTS)**
-
- **It typically takes hundreds or thousands of epochs for an NN to converge**

- ***Try nnd11bc!***

Inputs Encoding

- **How to encode the inputs for nominal attributes?**
  - **Example - nominal attribute *A* with values *none, some* and *many***
  - **Local encoding - use a single input neuron and use an appropriate number of distinct values to correspond to the attribute values, e.g. *none=0, some=0.5* and *many=1***
  - **Distributed (binary) encoding – use one neuron for each attribute value which is on or off (i.e. 1 or 0) depending on the correct value**

Output Encoding

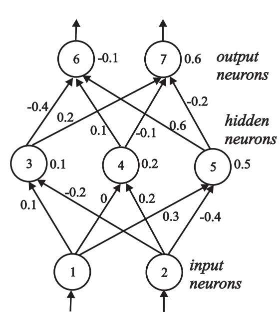
- **How to encode the outputs and represent targets?**
  - **Local encoding**
    - **1 output neuron**
    - **different output values represent different classes, e.g. <0.2 – class 1, >0.8 – class 2, in between – ambiguous class (class 3)**
  - **Distributed (binary, 1-of-n) encoding is typically used 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)**

How to Determine if an Example is Correctly Classified?

- **Accuracy may be used to evaluate performance once training has finished or as a stopping criteria checked at the end of each epoch**
- **Binary encoding**
  - **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**

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?**
    - **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**



Backpropagation – Example (cont. 1)

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

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

  - Activations of the hidden units:

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

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

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

  - Activations of the output units:

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

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

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 units ($\eta=0.1$)

  $\Delta w_{36}= \eta * \delta_6 * o_3 = 0.1*0.12*0.53=0.006$
  $w_{36}^{new} = w_{36}^{old} + \Delta w_{36} = -0.4+0.006=-0.394$

  $\Delta w_{37}= \eta * \delta_7 * o_3 = 0.1*-0.15*0.53=-0.008$
  $w_{37}^{new} = w_{37}^{old} + \Delta w_{37} = 0.2-0.008=-0.19$
  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 units δ_3, δ_4 and δ_5

  $\delta_3 = o_3 * (1-o_3) * (w_{36}* \delta_6 +w37 * \delta_7) =$
  $= 0.53*(1-0.53)(-0.4*0.12+0.2*(-0.15))=-0.019$
  Similarly for δ_4 and δ_5

- Calculate the new weights between the input and hidden units ($\eta=0.1$)

  $\Delta w_{13}= \eta * \delta_3 * o_1 = 0.1*(-0.019)*0.6=-0.0011$
  $w_{13}^{new} = w_{13}^{old} + \Delta w_{13} = 0.1-0.0011=0.0989$
  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

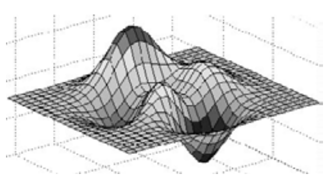
Backpropagation – Example (cont. 4)

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

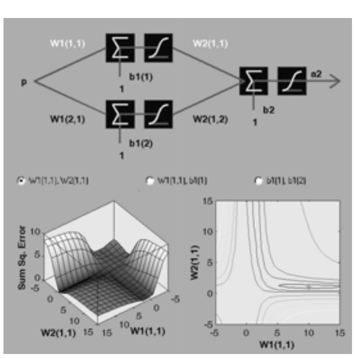
- **if yes: stop training**
- **if not, continue training:**
  - **epoch++**
  - **go to step 1**

Steepest Gradient Descent

- **Not optimal - is guaranteed to find a minimum but it might be a local minimum!**
- **Backpropagation's error space: many local and 1 global minimum => the generalized gradient descent may not find the global minimum**



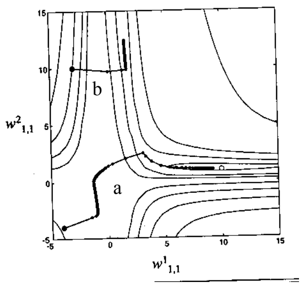
Error Surface - Example



w1(1,1) vs w2(1,1) vs error

Try nnd12sd1!

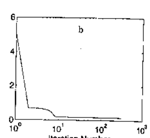
Convergence for the Example

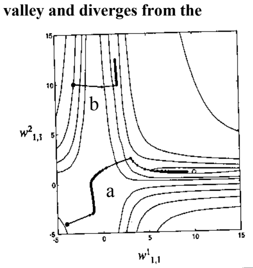


- Path a converges to the optimum solution but is very slow
- Path b gets trapped in a local minimum

Convergence – Case b

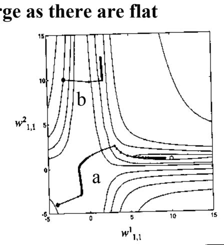
- The algorithm converges to a local minimum
  - The trajectory is trapped in a valley and diverges from the optimal solution
- What can be done?
  - Try different initializations

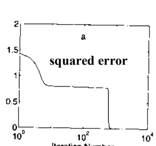




Convergence – Case a

- The algorithm is slow to converge as there are flat surfaces over the path
- What can we do?

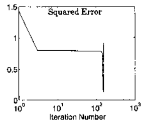


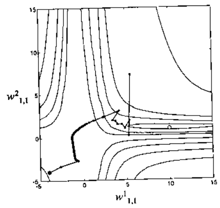


<- a typical learning curve:
- long periods of little progress
- short periods of rapid advance

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!





- **Solution 2: Smooth out the trajectory by averaging the updates to the parameters**
  - **The use of *momentum* might smooth out the oscillations and produce a stable trajectory**

Backpropagation with Momentum

- **The modified learning rule (μ - momentum):**

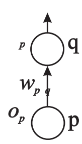
$$\Delta w_{pq}(t+1) = \mu\, \Delta w_{pq}(t) + \Delta w_{pq}$$
$$=> w_{pq}(t+1) = w_{pq}(t) + \mu\, (w_{pq}(t) - w_{pq}(t-1)) + \Delta w_{pq}$$

where

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

$\delta_i = (d_i - o_i) \cdot o_i \cdot (1 - o_i)$ **- for output neuron *i***

$\delta_j = o_j \cdot (1 - o_j) \cdot \sum_i w_{ji} \cdot \delta_i$ **- for hidden neuron *j* (the sum is over the *i* nodes in the layer above the node *j*)**

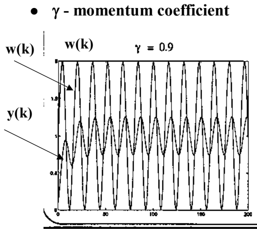


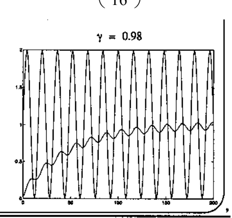
- **Typical values for momentum: 0.6 - 0.9**
- **The theory behind momentum comes from linear filters**

Momentum

- **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**

$$y(k) = \gamma y(k-1) + (1-\gamma) w(k) \quad 0 \le \gamma < 1$$
$$w(k) = 1 + \sin\left(\frac{2\pi k}{16}\right)$$





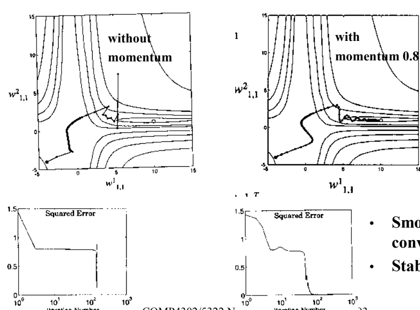
First Order Linear Filter

- 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

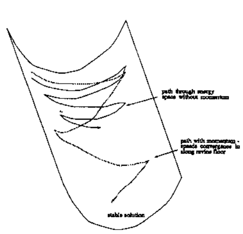
- Example – the same learning rate and initial position:



- Smooth and faster convergence
- Stable algorithm

Backpropagation with Momentum – cont.

- By the use of momentum we can use a larger learning rate while maintaining the stability of the algorithm
- Momentum also tends to accelerate convergence when the trajectory is moving in a consistent direction



- *Try nnd12mo!*

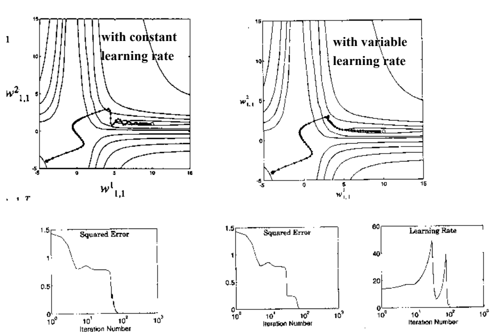
More on the Learning Rate

- **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**

Variable Learning Rate

- **Update the weights**
- **Calculate the squared error (over the entire training set)**
- **If the error increases by more than a predefined % θ:**
  - **Weight update is discarded**
  - **Learning rate is decreased by some factor (1> α>0)** [α=5% typically]
  - **Momentum is set to 0**
- **If the error increases by less than θ:**
  - **Weight update is accepted**
  - **Learning rate is increased by some factor β>1**
  - **If momentum has been set to 0, it is reset to its original value**
- **If the error decreases:**
  - **Weight update is accepted**
  - **Learning rate is unchanged**
  - **Momentum is unchanged**

Variable Learning Rate - Example



- ***Try nnd12vl!***

Universality of Backpropagation

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

Choice of Network Architecture

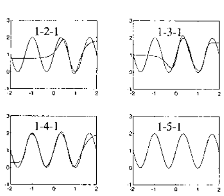
- **An art! Typically - by trial and error**
- **The task constrains the number of inputs and output units but not the number of hidden layers and neurons in them**
  - **Too many free parameters (weights) – overtraining**
  - **Too few – the network is not able to learn the input-output mapping**
  - **A heuristic to start with: 1 hidden layer with *n* hidden neurons, *n=(inputs+output_neurons)/2***

Choice of Network Architecture - Example

- **How many hidden neurons? (1 input, 1 output)**

$$f(x) = 1 + \sin\left(\frac{6\pi}{4}x\right)$$

- **Performance with different number of hidden units:**



- **Unless there are at least 5 hidden neurons, NN cannot represent the function**
- **=> backpropagation produces network which minimizes the error, but the capabilities of the NN are limited by the number of hidden neurons**
- ***Try nnd11fa!***

Generalization

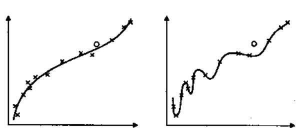
- Supervised learning – training with finite number of examples of proper behaviour: $\{p_1,t_1\}, \{p_2,t_2\},\ldots,\{p_n,t_n\}$
- Based on them the network should be able to *generalize* what it has learned to the total population of examples
- Overtraining (overfitting):
  - the error on the training set is very small but when a new data is presented to the network, the error is high
  - => the network has memorized the training examples but has not learned to generalize to new situations!

When Does Overfitting Occur?

- Training examples are noisy

Example: x- training set, o-testing set

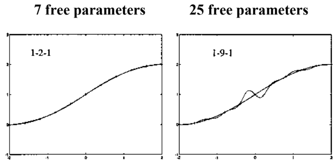
A good fit to noisy data   Overfitting



When Does Overfitting Occur? – cont.

- Number of the free parameters is bigger than the number of training examples
  - $f(x) = 1 + \sin\left(\frac{6\pi}{4}x\right)$ was sampled to create 11 training examples

7 free parameters   25 free parameters



•*Try nnd11gn!*

Preventing Overtraining

- Use network that is just large enough to provide an adequate fit
  - Ockham'Razor – don't use a bigger network when a smaller one will work
  - The network should not have more free parameters than there are training examples!
- However, it is difficult to know beforehand how large a network should be for a specific application!

Preventing Overtraining - Validation Set Approach

- Use an early stopping method
- Available data is divided into 3 subsets
  - Training set (TS)
    - Used for computing the gradient and updating the weights
  - Training test set (TTS), also called validation set
    - The error on the TTS 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 TTS will typically begin to rise
    - Training is stopped when the error on the TTS increases for a pres-specified number of iterations and the weights and biases at the minimum of the TTS 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 result in best performance on the respective test partition
  - Calculate the mean of *ep*, *ep_mean*
  - Final run: train the network on all examples for *ep_mean* epochs

Limitations and Capabilities

- MLPs trained with backpropagation can perform function approximation and pattern classification
- Theoretically they can
  - Perform any linear and non-linear computation
  - Can approximate any reasonable function arbitrary well
  - => are able to overcome the limitations of perceptrons and ADALINEs
- In practice:
  - May not always find a solution – can be trapped in a local minimum
  - Their performance is sensitive to the starting conditions (initialization of weights)
  - 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.

- 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
- => succesfful application of NNs requires time and experience

- *"NN training is an art. NN experts are artists; they are not mere handbook users"* P.H. Winston

Backpropagation Algorithm Summary

- Backpropagation
  - uses approximate steepest descent algorithm for minimizing the mean square error
- Gradient descent
  - The standard gradient descent is slow as it requires small learning rate for stable learning
  - Gradient descent with momentum is faster as it allows higher learning rate while maintaining stability
- There are several variations of the backpropagation algorithm

When to Consider NNs?

- Input is high dimensional discrete or continuous data
- Output is discrete or continuous
- Output is a vector of values
- Data might be noisy
- Long training times are acceptable
- Fast reply is needed
- Form of target function is unknown (but there are examples available)
- Explaining the result to humans is not important (NN are like black boxes)
- See the following examples

Some Interesting NN Applications

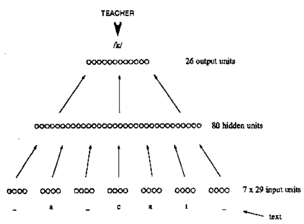
- A few examples of the many significant applications of NNs
- Network design was the result of several months trial and error experimentation
- Moral: NNs are widely applicable but they cannot magically solve problems; wrong choices lead to poor performance
- *"NNs are the second best way of doing just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

NETtalk

- Sejnowski and Rosenberg 87
- Pronunciation of written English
  - Fascinating problem in linguistics
  - Task with high commercial profit
  - How?
    - Mapping the text stream to phonemes
    - Passing the phonemes to speech generator
- Task for the NN: learning to map the text to phonemes
  - Good task for a NN as most of the rules are approximately correct
    - E.g. cat [k], century [s]

NETtalk -Architecture

- 203 input neurons – 7 (sliding window: the character to be pronounced and the 3 characters before and after it) x 29 possible characters (26 letters + blank, period, other punctuation)
- 80 hidden
- 26 output – corresponding to the phonemes



NETtalk - Performance

- Training set
  - 1024-words hand transcribed into phonemes
  - Accuracy on training set: 90% after 50 epochs
  - Why not 100%?
  - 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

Handwritten Character Recognition

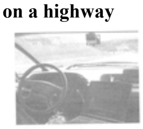
- Le Cun et al. 89
- Read zip code on hand-addressed envelopes
- Task for the NN:
  - A preprocessor is used to recognize the segments in the individual digits
  - Based on the segments, the network has to identify the digits
- Network architecture
  - 256 input neurons – 16x16 array of pixels
  - 3 hidden layers – 768, 192, 30 neurons respectively
  - 10 output neurons – digits 0-9
  - Not fully connected network
    - If it was a fully connected network 200 000 connections (impossible to train); instead only 9760 connections
    - Units in the hidden layer act as feature detectors – e.g. each unit in the 1st hidden layer is connected with 25 input neurons (5x5pixel region)

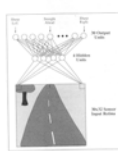
Handwritten Character Recognition – cont.

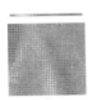
- **Training – 7300 examples**
- **Testing – 2000 examples**
- **Accuracy – 99%**
- **Hardware implementation (in VLSI)**
  - **enables letters to be sorted at high speed**
  - **zip codes**
- **One of the largest applications of NNs**

Driving Motor Vehicles

- **Pomerleau, 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**







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

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)**