Chapter 2

Introduction to Neural network

2.1 Introduction to Artificial Neural Network

Artificial Neural Networks (ANN:s) are an attempt to model the information processing of neurons systems.

We want to use the Artificial Intelligence (AI) concept for solving problems which are hard (or impossible) to solve otherwise. Some examples of problems can be:

2.1.1 Approximation

We want to approximate an unknown function

\[ F : \mathbb{R}^n \rightarrow \mathbb{R}^m \]

by only knowing the behavior of the function for some input-output sequences.

Example: We are given a number of input-output pairs.

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
</tr>
</tbody>
</table>

find the function \( y = f(x) : \mathbb{R} \rightarrow \mathbb{R} \). (Could it be \( y = 2x \)?) □
Many signal processing problems can be transformed to the approximation problem. We will deal with arbitrary number of input and output for any nonlinear function.

### 2.1.2 Association

The idea is to put a finite number of items in memory (e.g. the Latin characters) and by presenting distorted versions, we want the item to be restored.

*Example:*

\[
\begin{array}{c}
\text{Input} \\
\text{ANN} \\
\text{Output}
\end{array}
\]

### 2.1.3 Pattern classification

A number of inputs should be classified into categories.

*Example:*

\[
\begin{array}{c}
90^\circ C \\
\text{ANN} \\
\text{OK}
\end{array} \\
\begin{array}{c}
100^\circ C \\
\text{ANN} \\
\text{Error}
\end{array}
\]

### 2.1.4 Prediction

Given information up to present time predict the behavior in the future.

### 2.1.5 Automatic control (Reglerteknik)

We want to simulate the behavior of a process so that we can control it to fit our purposes.
2.1.6 Beamforming

We want to create directional hearing by the use of multiple sensors.

The model

Regard the ANN as a mapping box. An input $x$ gives an output $y$

\[ F : \mathbb{R}^n \mapsto \mathbb{R}^m \]

The box can be feed forward (i.e. no recursions) or a recurrent (with recursion) network. The complexity of its interior can vary depending on the task.

The box have parameters (weights) which can be modified to suite different tasks.

2.2 A Neuron model

Given an input signal $x$ it create a single output value

\[ y = f(x^T w), f : \mathbb{R} \rightarrow \mathbb{R}, f \in C^1, \] any function. The vector \( w = [w_1, w_2, \cdots, w_n]^T \) is called the weights of the neuron. Often \( w \in \mathbb{R}^n \).
2.3 A feedforward Network of Neurons (3-layers)

\[ y = H \left[ W^{[3]} \cdot G \left[ W^{[2]} \cdot F \left[ W^{[1]} \cdot x \right] \right] \right] \]

where

\[ F = [f_1 \ f_2 \ \cdots \ f_m]^T, \ \ G = [g_1 \ g_2 \ \cdots \ g_k]^T, \ \ H = [h_1 \ h_2 \ \cdots \ h_i]^T \]

and

\[ W^{[1]} = [w_{ij}]_{m \times n}, \ \ W^{[2]} = [w_{ij}^{[2]}]_{k \times m}, \ \ W^{[3]} = [w_{ij}^{[3]}]_{i \times k} \]

2.4 A recurrent Network

\[ Z^{-1} \text{-delay operator. } \text{"The Hopfield Network"} \]
2.5 Learning

Depending on the task to solve different learning paradigms (strategies) are used.

Supervised - during learning we tell the ANN what we want as output (desired output).

corrective learning - desired signals are realvalued

reinforced learning - desired signals are true/false

Unsupervised - Only input signals are available to the ANN during learning (e.g. signal separation)

One drawback with learning system are that it is totally lost when facing a scenario which it has never faced during training.
Chapter 3

Threshold logic

These units are neurons where the function is a shifted step function and where the inputs are Boolean numbers (1/0). All weights are equal to \( \pm 1 \)

\[
y = \begin{cases} 
1 & \text{if } x^T w \geq \theta \\
0 & \text{if } x^T w < \theta 
\end{cases}
\]

*Proposition* All logical functions can be implemented with a network of these units, since they can implement, AND, OR and NOT function.
3.1 Hadamard-Walsh transform

For bipolar coding (-1/1) the Hadamard-Walsh transform converts every Boolean function of $n$ variables into a sum of $2^n$ products.

**Example:**

<table>
<thead>
<tr>
<th>$X_1$</th>
<th>$X_2$</th>
<th>$X_1 \text{ OR } X_2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>$-1$</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>$f_1$</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>$1$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>$f_3$</td>
</tr>
</tbody>
</table>

The hadamard-Walsh matrix $H_n$ is defined recursively as

$$H_n = \frac{1}{2} \begin{bmatrix} H_{n-1} & H_{n-1} \\ -H_{n-1} & H_{n-1} \end{bmatrix}$$

whereby

$$H_1 = \frac{1}{2} \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix}$$

each row/column in the matrix are orthogonal to the other rows/columns.

We want to find the coefficients $a$ which solve the equation

$$X_1 \text{ OR } X_2 = \frac{1}{4}(a_1 + a_2 x + a_3 x_2 + a_4 x_1 x_2)$$

by using

$$\begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_{2^n} \end{bmatrix} = H_n \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_{2^n} \end{bmatrix}$$

$$\begin{bmatrix} a_1 \\ a_2 \\ a_3 \\ a_4 \end{bmatrix} = \frac{1}{4} \begin{bmatrix} 1 & 1 & 1 & 1 \\ -1 & 1 & -1 & 1 \\ -1 & -1 & 1 & 1 \\ 1 & -1 & -1 & 1 \end{bmatrix} \begin{bmatrix} -1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$$

which gives

$$X_1 \text{ OR } X_2 = \frac{1}{4}(2 + 2x_1 + 2x_2 - 2x_1x_2)$$

□
Chapter 4

The Perceptron

The perceptron is a weighted threshold logic unit.

\[
\begin{align*}
y &= \begin{cases} 
1 & \text{if } x^T w \geq \theta \\
0 & \text{if } x^T w < \theta
\end{cases} \\
x, w &\in \mathbb{R}^n
\end{align*}
\]

The perceptron can classify all input vectors belonging to linearly separable classes.

Example:

\[
\begin{align*}
\text{class } P & \quad \text{class } N \\
P \text{ and } N \text{ are linear separable!}
\end{align*}
\]
Example:

\begin{center}
\begin{tikzpicture}
\draw[->] (-2,0) -- (2,0) node[right] {$x_1$};
\draw[->] (0,-2) -- (0,2) node[above] {$x_2$};
\fill[gray,opacity=0.3] (-2,-2) -- (-1,1) -- (1,-1) -- (2,2) -- cycle;
\fill[gray,opacity=0.3] (2,-2) -- (1,1) -- (-1,-1) -- (-2,2) -- cycle;
\node at (0,0) {Input vector};
\node at (0,1) {$\text{class } P$};
\node at (0,-1) {$\text{class } N$};
\end{tikzpicture}
\end{center}

Not linear separable!

Example: Classification problem

Example:

\begin{center}
\begin{tikzpicture}
\draw[->] (-2,0) -- (2,0) node[right] {$\text{Pressure}$};
\draw[->] (0,-2) -- (0,2) node[above] {$\text{Heat}$};
\draw[dashed] (-2,-2) -- (2,2) node[above right] {$\text{Decision plane}$};
\fill[gray,opacity=0.3] (-2,-2) -- (-1,1) -- (1,-1) -- (2,2) -- cycle;
\fill[gray,opacity=0.3] (2,-2) -- (1,1) -- (-1,-1) -- (-2,2) -- cycle;
\node at (0,0) {$\text{Process OK (class } P\text{)}$};
\node at (0,-1) {$\text{Process error (class } N\text{)}$};
\node at (0,1) {$\text{Sensors}$};
\node at (1,0) {$\text{Heat}$};
\node at (0,1) {$\text{Pressure}$};
\node at (1,0) {$\text{Speed}$};
\end{tikzpicture}
\end{center}

\begin{center}
\begin{tikzpicture}
\draw[->] (-2,0) -- (2,0) node[right] {$x_1$};
\draw[->] (0,-2) -- (0,2) node[above] {$x_2$};
\fill[gray,opacity=0.3] (-2,-2) -- (-1,1) -- (1,-1) -- (2,2) -- cycle;
\fill[gray,opacity=0.3] (2,-2) -- (1,1) -- (-1,-1) -- (-2,2) -- cycle;
\node at (0,0) {Input vector};
\node at (0,1) {$\text{class } P$};
\node at (0,-1) {$\text{class } N$};
\end{tikzpicture}
\end{center}

\section{4.1 Perceptron training}

Assume we have sampled the sensors when the process is OK (P) as well as when its broken (N). We have a number of input vectors in each class.

\textbf{Start:} Choose $w_0$ randomly, $t = 0$

\textbf{test:} Select a vector $x \in P \cup N$

\hspace{1cm} If all $x$ correct, stop!
\[
\left\{
\begin{array}{l}
\text{if } x \in P \text{ and } x^T w_t > 0 \text{ goto test} \\
\text{if } x \in P \text{ and } x^T w_t \leq 0 \text{ goto add} \\
\text{if } x \in N \text{ and } x^T w_t < 0 \text{ goto test} \\
\text{if } x \in N \text{ and } x^T w_t \geq 0 \text{ goto subtract}
\end{array}
\right.
\]

**add:**
1. \( w_{t+1} = w_t + x \) or
   \[ w_{t+1} = w_t - \frac{(x^T w_t - \varepsilon)x}{\|x\|^2} \], \( \varepsilon > 0 \)
2. \( t = t + 1 \)

**subtract:**
1. \( w_{t+1} = w_t - x \) or
   \[ w_{t+1} = w_t - \frac{(x^T w_t + \varepsilon)x}{\|x\|^2} \], \( \varepsilon > 0 \)
2. \( t = t + 1 \)

If the two sets \( N \) and \( P \) are not perfectly linear separable we want to make a good approximation of the classification.

The pocket algorithm:

**start:** Initialize \( w_0 \) randomly
   - define the best weight vector \( w_s = w \)
   - define \( h_s = 0 \)

**iterate:** Update \( w \) as before
   - Count the number \( h \) of consecutively correctly classifications.
   - If \( h > h_s \), set \( w_s = w \) and \( h_s = h \).
   - Continue iterating.
4.2 The Linear Neuron (LMS-algorithm)

Is a neuron with a linear function

Assume we have a signal \( x(n) \) which we want to transform linearly and match a signal \( d(n) \)

We form the error signal \( e(n) = d(n) - y(n) \). We define the instantaneous error function

\[
J = \frac{1}{2} e(n)^2
\]

We want to minimize \( J \). We use the steepest descent approach and calculate \( \nabla_w J \)

\[
\nabla_w J = \nabla_w \frac{1}{2} (d(n) - w^T x(n))^2
\]

Used the chain rule and the equation becomes

\[
\nabla_w J = (d(n) - w^T x(n)) \cdot (-x(n)) = -e(n) \cdot x(n)
\]

where the \(-x(n)\) is the inner derivative. The steepest descent states

\[
w^{k+1} = w^k - \alpha \nabla_w J
\]

\[
w^{k+1} = w^k + \alpha e(n) \cdot x(n) \quad \text{(LMS-algorithm)}
\]
Chapter 5

Layered Networks of perceptrons

We have seen that a single perceptron can separate 2 linearly separable classes. If we put several perceptrons in a multilayer network what’s the maximum number of classification regions, i.e. what’s the capacity of the network?

Proposition Let $R(m, n)$ denote the number of regions bounded by $m$ hyper-planes (of dim. $n - 1$) in a $n$-dimensional space(all hyperplane are going through origin), then

$$R(m, n) = R(m - 1, n) + R(m - 1, n - 1)$$

where

$$R(1, n) = 2, \quad n \geq 1$$

$$R(m, 0) = 0, \quad \forall m \geq 1$$

Example:

(fig 6.20) $R(m, n)$ is computed recursively.

<table>
<thead>
<tr>
<th>$M \backslash N$</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>0</td>
<td>2</td>
<td>6</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>0</td>
<td>2</td>
<td>8</td>
<td>14</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>0</td>
<td>2</td>
<td>10</td>
<td>22</td>
<td>30</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td>0</td>
<td>2</td>
<td>12</td>
<td>32</td>
<td>52</td>
<td>62</td>
</tr>
</tbody>
</table>
A formula for $R(m, n)$

$$R(m, n) = 2^{n-1} \sum_{i=0}^{n-1} \binom{m-1}{i}$$

For a network with 3 hidden units with structure $n-k-l-m$ that is, $n$ input nodes $k$ first hidden nodes, $l$ second hidden nodes and $m$ output nodes we get

$$\text{max # regions} = \min\{R(k, n), R(l, k), R(m, l)\}$$

**Example:** Structure 3-4-2-3

The maximum of regions in 3-dimensional input space classifiable by the network is

$$\min\{R(4, 3), R(2, 4), R(3, 2)\} = \min(14, 4, 6) = 4$$

That is the second hidden layer limits the performance $\Rightarrow$ often the hidden layers contain more units than the input and output layers.
Chapter 6

The backpropagation algorithm

Learning of a single neuron. Consider the following model of a neuron

\[ u = \mathbf{w}^T \mathbf{x} + \theta \]

\( \theta \) is a scalar parameter called the "bias". This bias gives the neuron the possibility of shifting the function \( f(\cdot) \) to the left or right (positive or negative bias).

**Example:** \( f(x + \theta) = \text{sign}(x + \theta) \)

We can use the same terminology as before by defining the extended input vector \( \mathbf{x} = [x_1 \ x_2 \ \cdots \ x_n \ 1]^T \) and the extended weight vector \( \mathbf{w} = [w_1 \ w_2 \ \cdots \ w_n \ \theta]^T \Rightarrow y = f(\mathbf{w}^T \mathbf{x}) \)
Assume that we have a sequence of input vectors and a corresponding sequence of target (desired) scalars \((\bar{x}_1, t_1), (\bar{x}_2, t_2), \ldots, (\bar{x}_N, t_N)\)

We wish to find the weights of a neuron with a non-linear function \(f(\cdot)\) so that we can minimize the squared difference between the output \(y_n\) and the target \(t_n\), i.e.

\[
\min E_n = \min_1 \left( \frac{1}{2}(t_n - y_n)^2 \right) = \min_1 \left( \frac{1}{2}e_n^2 \right), n = 1, 2, \ldots, N
\]

We will use the steepest descent approach

\[
\bar{w}^{(n+1)} = \bar{w}^{(n)} - \alpha \nabla_w E
\]

We need to find \(\nabla_w E\)!

\[
\nabla_w E_n = \nabla_w \frac{1}{2} \left( t_n - f(\bar{w}^T \bar{x}_n) \right)^2
\]

The chain-rule gives \(\nabla_w g(h(f(u))) = \nabla_h g \nabla_f h \nabla_u f \nabla_w u\)

\[
\begin{align*}
\nabla_h g &= e_n \\
\nabla_f h &= t_n - f(u) = -1 \\
\nabla_u f &= \text{depends on the nonlinearity } f(\cdot) \text{ we choose} \\
\n\nabla_w u &= \bar{x}_n \\
\Rightarrow \bar{w}^{(n+1)} &= \bar{w}^{(n)} + \alpha e_n \nabla_u f \bar{x}_n
\end{align*}
\]

since \(f(\cdot)\) is a function \(f : \mathbb{R} \rightarrow \mathbb{R}\), i.e. one-dimensional. We can write \(\nabla_u f = \frac{df}{du}\)

\[
\Rightarrow \text{The neuron learning rule for general function } f(\cdot) \text{ is}
\]

\[
\bar{w}^{(n+1)} = \bar{w}^{(n)} + \alpha e_n \frac{df}{du} \bar{x}_n
\]

Where \(u = \bar{w}^T \bar{x}\) and \(\alpha\) is the stepsize.

OBS! If \(f(u) = u\), that is a linear function with slope 1, the above algorithm will become the LMS alg. for a linear neuron.

\[
\left( \frac{df}{du} = 1 \right)
\]
6.1 The non-linearity of the neuron

Any differentiable function $f(\cdot)$ can be used in the neuron. We will use 2 different functions, bi- and unipolar sigmoid function.

6.1.1 The bipolar sigmoid function

(the tangent hyperbolic)

$$f(u) = \tanh(\gamma u) = \frac{1 - e^{-2\gamma u}}{1 + e^{-2\gamma u}}$$

$\gamma$ is called the slope ($\gamma \in \mathbb{R}$)

Example:

![Graph of tanh(\gamma u) with \gamma = 1, 0.25, 2](image)

often $\gamma = 1$.

The derivative of $f(u)$ is

$$\frac{df}{du} = \gamma (1 - \tanh(\gamma u)^2) = \gamma (1 - f(u)^2)$$

43
6.1.2 The unipolar sigmoid function

\[ f(u) = \frac{1}{1 + e^{-\gamma u}} \]

Example:

![Graph of the unipolar sigmoid function with \( \gamma = 0.25, 1, 2 \)]

The derivative of \( f(u) \) is

\[
\frac{df}{du} = \gamma \frac{1}{1 + e^{-\gamma u}} \left(1 - \frac{1}{1 + e^{-\gamma u}}\right) = \gamma f(u)(1 - f(u))
\]

this gives the update formulas for the neuron

1. bipolar

\[
\tilde{W}^{(n+1)} = \tilde{W}^{(n)} - \alpha \gamma e_n (1 - y_n^2) \tilde{x}_n
\]

\[
y_n = f(u_n) = \tanh(\gamma u_n) = \tanh(\gamma \tilde{W}_n^T \tilde{x}_n)
\]

2. unipolar

\[
\tilde{W}^{(n+1)} = \tilde{W}^{(n)} - \alpha \gamma e_n y_n (1 - y_n) \tilde{x}_n
\]

\[
y_n = f(u_n) = \frac{1}{1 + e^{-\gamma u_n}} = \frac{1}{1 + e^{-\gamma \tilde{W}_n^T \tilde{x}_n}}
\]
6.2 The backpropagation algorithm for a 3-layer network

\[ \mathbf{\tilde{w}}^{[l]} \] - is the extended weight matrix. The last column consists of bias terms.

Assume we have a set of input vectors \( \mathbf{x}_n \) and a set of corresponding target vectors \( t_n, n = 1, 2, \ldots, N \) (a training set)

\[ (x_1, t_1), (x_2, t_2), \ldots, (x_N, t_N) \]

We want to minimize the squared difference between the output \( y_n \) of the network and the target vectors \( t_n \)

\[ \text{i.e. min} \sum_{i=1}^{P} e_i^2 \]

We first state the algorithm and then we prove it.

### 6.2.1 Backpropagation algorithm

Step 1. Initiate all the weight-matrices. This can be done by the following rule. Pick values randomly in the interval \((-0.5, 0.5)\) and divide with fan-in, which is the number of units feeding the layer.

**Example:**

\[
\mathbf{\tilde{W}}^{[1]} = \frac{\text{rand}(m, n + 1) - 0.5}{n + 1}
\]

\[
\mathbf{\tilde{W}}^{[2]} = \frac{\text{rand}(k, m + 1) - 0.5}{m + 1}
\]
\[
\tilde{\mathbf{W}}^{[3]} = \frac{(\text{rand}(p, k + 1) - 0.5)}{k + 1}
\]

where "rand" is uniformly distributed in [0, 1].

Step 2. Pick a input-target pair randomly from the training set. say \((\mathbf{x}_i, \mathbf{t}_i)\). Calculate the output when \(\mathbf{x}_i\) is the input according to

\[
\mathbf{y}_i = H \left[ \tilde{\mathbf{W}}^{[3]} \cdot G \left[ \tilde{\mathbf{W}}^{[2]} \cdot F \left[ \tilde{\mathbf{W}}^{[1]} \tilde{x}_i \right] \right] \right]
\]

where

\[
F = [f_1(u_1)^{[1]}, f_2(u_2)^{[1]}, \ldots, f_m(u_m)^{[1]}]^T
\]

\[
G = [g_1(u_1)^{[2]}, g_2(u_2)^{[2]}, \ldots, g_k(u_k)^{[2]}]^T
\]

\[
H = [h_1(u_1)^{[3]}, h_2(u_2)^{[3]}, \ldots, h_p(u_p)^{[3]}]^T
\]

All functions are chosen in advance.

Step 3.
Find the weight corrections for each layer.

First define the vector \(\mathbf{e}_i = \mathbf{t}_i - \mathbf{y}_i\).
Define the local error vector \(\delta^{[s]}\), \(s = 1, 2, 3\)

\[
\delta^{[3]} = \text{diag}(\mathbf{e}) \frac{\partial H}{\partial \mathbf{u}^{[3]}}
\]

\[
\Delta \tilde{\mathbf{W}}^{[3]} = \alpha \delta^{[3]} (\tilde{\mathbf{o}}^{[2]})^T, \alpha - \text{stepsize}
\]

OBS! \(\tilde{\mathbf{o}}^{[2]}\) is the extended vector!

\[
\delta^{[2]} = \text{diag} \left( (\mathbf{W}^{[3]})^T \delta^{[3]} \right) \frac{\partial G}{\partial \mathbf{u}^{[2]}}
\]

OBS! \(\mathbf{W}^{[3]}\) is without the biases!

\[
\Delta \tilde{\mathbf{W}}^{[2]} = \alpha \delta^{[2]} (\tilde{\mathbf{o}}^{[1]})^T
\]

\[
\delta^{[1]} = \text{diag} \left( (\mathbf{W}^{[2]})^T \delta^{[2]} \right) \frac{\partial F}{\partial \mathbf{u}^{[1]}}
\]

\[
\Delta \tilde{\mathbf{W}}^{[1]} = \alpha \delta^{[1]} \tilde{x}^T
\]

46
(The algorithm is called backpropagation since the local error is propagating backwards from output layer in to the hidden layers)

Step 4. Update the weight matrices.
\[
\begin{align*}
\bar{W}^{[3]} &= \bar{W}^{[3]} + \Delta \bar{W}^{[3]} \\
\bar{W}^{[2]} &= \bar{W}^{[2]} + \Delta \bar{W}^{[2]} \\
\bar{W}^{[1]} &= \bar{W}^{[1]} + \Delta \bar{W}^{[1]}
\end{align*}
\]

Return to step 2, stop when no changes in \( \Delta \bar{w}^{[s]} \)

The partial derivative are
\[
\frac{\partial H}{\partial u^{[3]}} = \begin{bmatrix}
dh_1 \\
dh_2 \\
\vdots \\
dh_m 
\end{bmatrix}^T
\]

If we use the bipolar sigmoid function
\[
\frac{\partial H}{\partial u^{[3]}} = \begin{bmatrix}(1 - \tanh(u_1^{[3]})^2) (1 - \tanh(u_2^{[3]})^2) \cdots (1 - \tanh(u_m^{[3]})^2)\end{bmatrix}^T
\]
and the same follows for \( \frac{\partial G}{\partial u^{[s]}} \) and \( \frac{\partial F}{\partial u^{[s]}} \)

### 6.3 The proof of the Backpropagation

We first define the error function for input-target pair \( i', t_i' \), i.e. \( (x_i', t_i') \)
\[
E_{i'} = \frac{1}{2} \sum_{j=1}^{p} (t_j - y_j)^2 = \frac{1}{2} \sum_{j=1}^{p} e_{ji}^2 = \frac{1}{2} \| e_{i'} \|_2^2
\]
that is we want to minimize the sum of the squared errors.

For every weight \( w_{ji}^{[s]} \) in layer \( s = 1, 2, 3 \) we will use the steepest descent rule to update the weight
\[
w_{ji}^{[s]} = w_{ji}^{[s]} - \alpha \frac{\partial E_{i'}}{\partial w_{ji}^{[s]}} = w_{ji}^{[s]} + \Delta w_{ji}^{[s]}
\]

We need to find the partial derivative \( \frac{\partial E_{i'}}{\partial w_{ji}^{[s]}} \) for every weight in every layer. First we start with the outputlayer \( (s = 3) \)
\[
\frac{\partial E_{i'}}{\partial w_{ji}^{[3]}} = -\frac{\partial E_{i'}}{\partial u_j^{[3]}} \frac{\partial u_j^{[3]}}{\partial w_{ji}^{[3]}}
\]
where

\[ u_j^{[3]} = \sum_{i=1}^{p} w_{ji}^{[3]} o_i^{[2]} \]

we define the local error \( \delta_j^{[3]} \) as

\[ \delta_j^{[3]} = -\frac{\partial E_i'}{\partial u_j^{[3]}} = -\frac{\partial E_i'}{\partial e_{ji}'} \frac{\partial e_{ji}'}{\partial u_j^{[3]}} = e_{ji}' \frac{\partial h_j}{\partial u_j^{[3]}} \]

And we get the update rule for all weights in the output layer \((s = 3)\) as

\[ w_{ji}^{[3]} = w_{ji}^{[3]} + \alpha e_{ji}^{(3)} \frac{\partial h_j}{\partial u_j^{[3]}} o_i^{[2]} = w_{ji}^{[3]} + \alpha \delta_j^{[3]} o_i^{[2]} \]

Which is the same rule as for the single neuron where the input \( o_i^{[2]} \) is the output from previous layer.

We continue to derive the update for the second hidden layer \((s = 2)\) As before

\[ -\frac{\partial E_i'}{\partial o_j^{[2]}} = -\frac{\partial E_i'}{\partial u_j^{[2]}} \frac{\partial u_j^{[2]}}{\partial o_j^{[2]}} = \delta_j^{[2]} o_i^{[1]} \]

The calculation of the local error \( \delta_j^{[2]} \) for the second hidden layer will be different

\[ \delta_j^{[2]} = -\frac{\partial E_i'}{\partial u_j^{[2]}} = -\frac{\partial E_i'}{\partial o_j^{[2]}} \frac{\partial o_j^{[2]}}{\partial u_j^{[2]}} \]

since \( o_j^{[2]} = g_j(u_j^{[2]}) \) where \( g_j(\cdot) \) - the function associated with neuron \( j \).

We have

\[ \delta_j^{[2]} = -\frac{\partial E_i'}{\partial o_j^{[2]}} \frac{\partial g_j}{\partial u_j^{[2]}} \]

To calculate \( -\frac{\partial E_i'}{\partial o_j^{[2]}} \) we use the chain rule once again to express the derivative in terms of the local error in layer 3.

\[ -\frac{\partial E_i'}{\partial o_j^{[2]}} = -\sum_{i=1}^{p} \frac{\partial E_i'}{\partial u_i^{[3]}} \frac{\partial u_i^{[3]}}{\partial o_j^{[2]}} = \sum_{i=1}^{p} \left( -\frac{\partial E_i'}{\partial u_i^{[3]}} \right) \frac{\partial}{\partial o_j^{[2]}} \sum_{k=1}^{p} \left( w_{ik}^{[3]} o_k^{[3]} \right) = \delta_j^{[3]} \]
The sum comes from the fact that one output $o_j$ in layer 2 is connected to all $u_i$ in layer 3.

$$\frac{\partial E'}{\partial o_j^2} = \sum_{i=1}^{p} \delta_i^3 w_{ij}^3$$

which gives the update rule for the second hidden layer

$$w_{ji}^2 = w_{ji}^2 + \alpha \delta_j^2 o_i$$

where

$$\delta_j^2 = \frac{\partial g_j}{\partial u_j^2} \sum_{i=1}^{p} \delta_i^3 w_{ij}^3$$

In the same way the update for the first hidden layer can be derived using the local error of the following layer

$$w_{ji}^1 = w_{ji}^1 + \alpha \delta_j^1 x_i$$

where

$$\delta_j^1 = \frac{\partial f_j}{\partial u_j^1} \sum_{i=1}^{k} \delta_i^2 w_{ij}^2$$

Using matrix notation the algorithm is derived.$\square$
6.4 The rule of the hidden layer

Consider a two layer feedforward neural network, one hidden and one output layer, without non-linearities. The output vector will then be

\[ y = W^2(W^1x) \iff y = Wx \]

which means that the network can be reduced to a single layer network with weight matrix \( W = W^2W^1 \).

\( \Rightarrow \) Every multilayer feedforward network with linear neurons can be reduced to an equivalent single layer network.

**Proposition** A three layer feedforward neural network with sigmoidal functions can approximate any multivariate function, with finite number of discontinuities, at any desired degree of accuracy.

\( \square \)

Provided we have enough number of neurons.

6.5 Applications of back-propagation alg.

the solution to a set of linear equations

\[ Ax = b \quad (A \text{ can be any size } (m \times k)) \]

\[ \Rightarrow x^{opt} = A^+b \]

(in the LS-sense if no solution exists)

How can we use Neural networks to find the pseudo-inverse of \( A \)?

We will use a single layer NN with linear neurons (no biases). Write the matrix \( A \) as

\[ A^T = [a_1 \ a_2 \ \cdots \ a_m] \]
and define the columns of a identity matrix

\[ I_{m \times m} = [e_1 \ e_2 \ \cdots \ e_m] \]

i.e.

\[ e_1 = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \ e_2 = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}, \ \ldots, \ e_m = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix} \]

then the training set is

\[ (a_1, e_1), (a_2, e_2), \ldots, (a_m, e_m) \]

we use the LMS algorithm to adapt the weights \( W \) to the training set with one modification, that is

\[ W^{(k+1)} = (1 - \gamma) W^k + \alpha e x^T \]

\( \gamma \ll 1 \)- leaky factor, \( \alpha \)- stepsize, \( x_k=\text{input}=a_k \) and \( e=\text{error vector} \)

(remember we have linear functions, the algorithm is the MIMO-LMS, Multiple-Input Multiple-Output)

After convergence \( W^T = A^+ \) (Pseudoinverse of \( A \)) □

6.5.1 One more example of Backpropagation

An alternative way to find the pseudo inverse of a matrix \( A \).

We will use a linear two layer network. We create a random sequence of input vectors and use the same vectors as targets.

\[ \Rightarrow \text{Training sequence}(x_1, x_1), (x_2, x_2), \ldots, (x_N, x_N) \]

We fix the first layer weights to \( A \) and we only update the output layer weights \( W \).
Since we know that for any input vector $x_i$ we have
\[ \hat{x}_i = WAx_i \]
and that backpropagation minimizes the norm of
\[ \min \| x_i - WAx_i \| \Rightarrow W = A^+ \]

### 6.6 Application speech synthesis-NETtalk

People in the field of linguistics have been working with rule based speech synthesis systems for many years. Speech synthesis is a way of converting text to speech and a feedforward neural network, NETtalk, can be trained to solve this. It uses a 7 character sliding window to read the text and it is trained with backpropagation to classify the phonemes of the text’s pronunciation (See figure 9.14).

Another application is the reverse problem of listening to speech and then convert to text. They use Hidden Markov Models trained with Backpropagation.

### 6.7 Improvements to backpropagation

There are many ways to improve the convergence of backpropagation.
Main problem is the steepest descent algorithm. We know that steepest descent is perpendicular to the contours of the error function.

We also know that the more elliptic the contour is the slower the convergence will be. The ratio of the ellipses diameters are controlled by the eigenvalues of the correlation matrix $\mathcal{R}_{xx}$ ($x$-input). If we decorrelate the input before the network we will get better performance.

$\lambda_1$, $\lambda_2$ – eigenvalues of $\mathcal{R}_{xx}$

Decorrelation can be done by an unsupervised algorithm (The Generalized Hebbian algorithm) by finding the Principal Components (PCA). We will see this later in this course.

### 6.8 Momentum update

In order to drop faster in the error surface we use the average of previous gradients when we update the weights

$$\Delta w_{ij}^{(k)} = -\frac{\partial E}{\partial w_{ij}} + \beta \Delta w_{ij}^{(k-1)}$$

momentum term, $\beta < 1$
The backpropagation algorithm can cause the weight update to follow a fractal pattern (fractal means it follow some rules which are applied statistical), (see figure 8.11)
⇒ The stepsize and momentum term should decrease during training to avoid this.

### 6.9 Adaptive step algorithms

These algorithms modify the stepsize during training in order to speed convergence. The idea is to increase the stepsize, $\alpha$, if the sign of the gradient is the same for two following updates and to decrease if the sign change.

*Example: Error function*

![Error function diagram](image)

### 6.10 Silva and Almeidal’s algorithms

\[
\alpha_i^{(k+1)} = \begin{cases} 
\alpha_i^{(k)} u & \text{if } \nabla_iE^{(k)}\nabla_iE^{(k-1)} > 0 \\
\alpha_i^{(k)} d & \text{if } \nabla_iE^{(k)}\nabla_iE^{(k-1)} < 0 
\end{cases}
\]

$u, d$ parameters (e.g. $u = 1.1, d = 0.9$) and index $i$ stands for weight no. $i$

Other similar methods are *Delta-bar-delta* and *Rprop*.

### 6.11 Second order algorithms

These methods use the Newton’s algorithm instead of the steepest descent.

\[
w^{(k+1)} = w^{(k)} - \left[ \frac{\partial^2 E}{\partial w^2} \right]^{-1} \nabla_w E
\]

54
One can derive the updates for the network in each layer as before but the computational complexity will increase drastically in the training of the network.

Every neuron needs a matrix inversion!

There are good approximations to these algorithms, the so called Pseudo-Newton’s methods, e.g. Quickprop, QRprop etc.

6.12 Relaxation methods

These methods are used to avoid calculating the gradient. They use a trial and error approach, change the weights in any way - if the error is dropping use the new weights otherwise try again.

6.13 Generalization

The learning process may be viewed as a ”curve-fitting” problem. The network itself may be considered simply as a non linear input-output mapping.

When training Neural Networks it is important to get a network that has good generalization. A network is said to ”generalize” well when the input-output mapping computed by the network is correct (or nearly so) for test data never used in creating or training the network.

Over-fitting is one of the biggest threats to a good generalization. Over-fitting occurs during training when the number of free parameters, or weights, is too large. What happens is that the error during training is driven to a very small value, but when new data is presented to the network the error is large. This depend on that the networks complexity is too big which results in that the entire training patterns is memorized and that it has big enough degree of freedom to fit the noise perfectly. Of course this means bad generalization to new test data. The figure tries to visualize this problem. One solution to this problem is to reduce the number of free parameters. This is easiest done by reducing the number of hidden neurons. Generalization is influenced by three factors:

- The size of the training set, and how representing it is for the environment of interest.
• The architecture of the neural network

• The physical complexity of the problem of hand.

We may view the issue of generalization from two different perspectives.

• The architecture of the network is fixed. We have to determine the size of the training set needed for a good generalization to occur.

• The size of the training set is fixed. We have to determine the best architecture of the network for achieving good generalization.

6.13.1 The size of the training set

We may in generally say that for good generalization, the number of training examples $P$ should be larger than the ratio of the total number of free parameters (the number of weights) in the network to the mean square value of the estimation error.

6.13.2 Number of hidden neurons

To choose the correct number of hidden neurons is an important aspect when designing a Neural Network. Too few hidden neurons gives us a net that has difficulty to adapt to the training data and too many neurons may result in a network with bad regularization. A theoretical and a more straightforward approach on finding the number of neurons will be discussed in the following subsections.
A theoretical approach

How many neurons should be used in relation to the number of training examples? Information theory suggests that the number of input vectors should be equal to the number of weights in the system. The number of weights, \( W \), for a multi-layer network can be calculated by the expression:

\[
W = (I + 1)J + (J + 1)K
\]  

(6.1)

where \( I \) is the number of inputs, \( J \) the number of hidden neurons and \( K \) is the number of outputs. Another point of view is the one from Widrow and Lehr (1990). They argue that the number of training examples \( P \) should be much greater than the network capacity, which is defined as the number of weights divided by the number of outputs \((W/K)\). In their paper they also refer to Baum and Hesslers (1989) statement that the number of training examples into a multi-layer network could be calculated as follows:

\[
P = \frac{W}{\varepsilon}
\]  

(6.2)

where \( \varepsilon \) is an "accuracy parameter". An "accuracy parameter" means the fraction of tests from the test set that are incorrectly classified. A good value for \( \varepsilon \) is 0, 1, which corresponds to an accuracy level of 90\%. This gives us the rule of thumb below.

Lower bound for number of input examples, \( P \), is \( P = W \). And a realistic upper bound for \( P \) is \( P = 10W \).

This is just a rule of thumb and there is actually no upper bound for \( P \).

Example

Suppose we know the number of training examples \( P \) and we would like to calculate the number of hidden neurons. The input-vector is of length 24, \( I = 24 \), and the number of training examples is 4760, \( P = 4760 \). The number of outputs is one, \( K = 1 \). This gives us the following equations:

\[
W = (24 + 1)J + (J + 1)1 \iff W = 26J + 1
\]  

(6.3)

Since the rule of thumb says that,

\[
W \leq P \leq 10W \iff (26J + 1) \leq 4760 \leq (260J + 10)
\]  

(6.4)

we get that the limitations for the hidden neurons, \( J \).

\[
18 \leq J \leq 183
\]  

(6.5)
6.13.3 Cross validation

A standard tool in statistics known as cross-validation provides an appealing guiding principle. First the available data set is randomly partitioned into a training set and a test set. The training set is further partitioned into two disjoint subsets:

- **Estimation subset**, used to select the model.
- **Validation subset**, used to test or validate the model.

The motivation here is to validate the model on a data set different from the one used for parameter estimation. In this way we may use the training set to assess the performance of various candidate models, and thereby choose the "best" one. There is, however, a distinct possibility that the model with the best-performing parameter values so selected may end up overfitting the validation subset. To guard against this possibility, the generalization performance of the selected model is measured on the test set, which is different from the validation subset.

Extremely low training error is not hard to achieve when training Neural Networks. But a low training error does not automatically mean that the training result is good. Two methods that can improve generalization, regularization and early stopping, are shortly discussed in the following subsections.

**Early Stopping**

The training set is used to calculate the gradients and to update the network weights and biases. And as the number of epochs increase the mean square error usually decrees. It is possible for the network to end up over-fitting the training data if the training session is not stopped at the right point. The validation set is used to improve generalization. During training the validation error is monitored. As long as the network shows no signs of overfitting, the validation error is normally decreasing. The sign that the method is waiting for is when the error goes from decreasing to increasing, which means that the network starts to over-fit. The training is stopped when the error has been increased for a specific number of steps and the weights and biases at the minimum validation error are returned.
Regularization

There are a couple of regularization techniques to be used which all are based on the same idea, to generate a smooth mapping that does not fit the noise in the training data. This means that over-fitting should not be a problem, even when the network has lots of hidden neurons.

The basic idea of regularization is that the squared error function, or performance function, $E_{mse}$ is added with a penalty function $\Psi$, to get a new error function $E_{msereg}$. Here we define the penalty function as the square norm of the weight vector in the network, but there are other definitions as well. The error function is usually given by:

$$E_{mse} = \frac{1}{N} \sum_{i=1}^{N} (e_i)^2$$  \hspace{1cm} (6.6)

where $e_i$ is network error and $N$ the number of errors. The penalty function:

$$\Psi = \frac{1}{n} \sum_{j=1}^{n} (w_j)^2$$  \hspace{1cm} (6.7)

where $w$ are the weights and $j$ the number of weights. These two functions, together with a multiplicative constant $\mu$ can be used to get a regularized error function:

$$E_{msereg} = \mu E_{mse} + (1 - \mu)\Psi$$  \hspace{1cm} (6.8)

Using this error function, weights and biases will be forced to small values which smoothes the network response and makes it less likely to over-fit. To use a regularization method is especially important when the amount of training data is limited.