WARSAW UNIVERSITY OF TECHNOLOGY
FACULTY OF MATHEMATICS
AND INFORMATION SCIENCE

Neural Networks

Lecture 3

The convergence of the learning procedure

The convergence of the learning procedure

- The input patterns are assumed to come from a space which has two classes: **F^+** and **F^-**.
- We want the perceptron to respond with **1** if the input comes from **F^+**, and **-1** if it comes from **F^-**.
- The set of input values x_i as a vector in n-dimensional space **X**, and the set of weights w_i as the another vector in the same space **W**.
- Increasing the weights is performed by **W + X**, and decreasing by **W - X**.

The convergence of the learning procedure

```
start:     Choose any value for $\mathbf{W}$

test:      Choose an $\mathbf{X}$ from $\mathbf{F}^+$ or $\mathbf{F}^-$
             If $\mathbf{X} \in \mathbf{F}^+$ i $\mathbf{W} \cdot \mathbf{X} > 0 \Rightarrow$ test
             If $\mathbf{X} \in \mathbf{F}^+$ i $\mathbf{W} \cdot \mathbf{X} \leq 0 \Rightarrow$ add
             If $\mathbf{X} \in \mathbf{F}^-$ i $\mathbf{W} \cdot \mathbf{X} < 0 \Rightarrow$ test
             If $\mathbf{X} \in \mathbf{F}^-$ i $\mathbf{W} \cdot \mathbf{X} \geq 0 \Rightarrow$ subtract

add:         Replace $\mathbf{W}$ by $\mathbf{W} + \mathbf{X} \Rightarrow$ test

subtract: Replace $\mathbf{W}$ by $\mathbf{W} - \mathbf{X} \Rightarrow$ test
```

Notice that we go to ***subtract*** when $\mathbf{X} \in \mathbf{F}^-$, and if we consider that going to ***subtract*** is the same as going to ***add*** **X** replaced by **–X.**

The convergence of the learning procedure

```
start:    Choose any value for $\mathbf{W}$
test:     Choose a $\mathbf{X}$ from $\mathbf{F}^+$ or $\mathbf{F}^-$
              If $\mathbf{X} \in \mathbf{F}^-$ change the sign of $\mathbf{X}$
              If $\mathbf{W} \cdot \mathbf{X} > 0 \Rightarrow$ test
                  otherwise $\Rightarrow$ add
add:      Replace $\mathbf{W}$ by $\mathbf{W} + \mathbf{X} \Rightarrow$ test
```

We can simplify the algorithm still further, if we define **F** to be $\mathbf{F}^+ \cup -\mathbf{F}^-$ i.e. $\mathbf{F}^+$ and the negatives of $\mathbf{F}^-$.

The convergence of the learning procedure

```
start:   Choose any value for $\mathbf{W}$
test:    Choose any $\mathbf{X}$ from $\mathbf{F}$
             If $\mathbf{W} \cdot \mathbf{X} > 0 \Rightarrow$ test
                 otherwise $\Rightarrow$ add
add:     Replace $\mathbf{W}$ by $\mathbf{W} + \mathbf{X} \Rightarrow$ test
```

The Perceptron

Theorem and proof

Theorem and proof

Convergence Theorem:

Program will only go to *add* a finite number of times.

Proof:

Assume that there is a unit vector **W*,** which partitions up the space, and a small positive fixed number δ, such that

$$\mathbf{W}^*\cdot\mathbf{X} > \delta \text{ for every } \mathbf{X} \in \mathbf{F}$$

Define $G(\mathbf{W}) = \mathbf{W}^*\cdot\mathbf{W}/|\mathbf{W}|$

and note that $G(\mathbf{W})$ is the cosine of the angle between **W** and **W*.**

Theorem and proof

Since $|\mathbf{W}^*| = 1$, we can say that $G(\mathbf{W}) \leq 1$.

Consider the behavior of $G(\mathbf{W})$ through *add.*

The numerator: $G(\mathbf{W}) = \dfrac{\mathbf{W}^* \mathbf{W}}{|\mathbf{W}|}$

$$\mathbf{W}^* \cdot \mathbf{W}_{t+1} = \mathbf{W}^* \cdot (\mathbf{W}_t + \mathbf{X}) = \mathbf{W}^* \cdot \mathbf{W}_t + \mathbf{W}^* \cdot \mathbf{X} \geq \mathbf{W}^* \cdot \mathbf{W}_t + \delta$$

since $\mathbf{W}^* \cdot \mathbf{X} > \delta$.

Hence, after the m*th* application of *add* we have

$$\mathbf{W}^* \cdot \mathbf{W}_m \geq \delta \cdot m \qquad (1)$$

Theorem and proof

The denominator:

Since $\mathbf{W} \cdot \mathbf{X}$ must be negative (*add* operation is performed),

then $|\mathbf{W}_{t+1}|^2 = \mathbf{W}_{t+1} \cdot \mathbf{W}_{t+1} = (\mathbf{W}_t + \mathbf{X}) \cdot (\mathbf{W}_t + \mathbf{X}) =$

$$= |\mathbf{W}_t|^2 + 2\mathbf{W}_t \cdot \mathbf{X} + |\mathbf{X}|^2$$

Moreover $|\mathbf{X}| = 1$, so $|\mathbf{W}_{t+1}|^2 < |\mathbf{W}_t|^2 + 1$,

and after m*th* application of *add*

$$|\mathbf{W}_m|^2 < m. \qquad (2)$$

Theorem and proof

Combining (1) i (2) gives

$$\mathbf{G(W_m) = \frac{(W * W_m)}{|W_m|} > \frac{m\delta}{\sqrt{m}}}$$

Because G(**W**) $\leq$ 1, so we can write

$$\mathbf{\sqrt{m} \cdot \delta \leq 1 \qquad i.e. \qquad m \leq \frac{1}{\delta^2}}$$

Theorem and proof

What does it mean?? Inequality (3) is our proof.

In the perceptron algorithm, we only go to *test* if $\mathbf{W}\cdot\mathbf{X} > 0$. We have chosen a small fixed number δ, such that $\mathbf{W}\cdot\mathbf{X} > \delta$. Inequality (3) then says that we can make δ as small as we like, but the number of times, m, that we go to *add* will still be ***finite,*** and will be $\leq 1/\delta^2$.

In other words, perceptron will learn a weight vector **W**, that partitions the space successfully, so that patterns from F^+ are responded to with a positive output and patterns from F^- produce a negative output.

The perceptron learning algorithm

The perceptron learning algorithm

1 step – initialize weight and threshold:

Define $w_i(t)$, $(i=0,1,...,n)$ to be the weight from input *i* at time *t,* and Θ to be a threshold value ij the output node. Set $w_i(0)$ to small random numbers.

2 step – present input and desired output:

Present input $X = [x_1, x_2, ..., x_n]$, $x_i \in \{0,1\}$, and to the comparison block desired output d(t).

The perceptron learning algorithm

3 step:

Calculate actual output

$$y(t) = f\left[\sum_i w_i(t)x_i(t)\right]$$

4 step:

Adapt weights

The perceptron learning algorithm

<u>4 step (cont):</u>

if $y(t) = d(t)$ $\Rightarrow$ $w_i(t+1) = w_i(t)$

if $y(t) = 0$ and $d(t) = 1$ $\Rightarrow$ $w_i(t+1) = w_i(t) + x_i(t)$

if $y(t) = 1$ and $d(t) = 0$ $\Rightarrow$ $w_i(t+1) = w_i(t) - x_i(t)$

The perceptron learning algorithm

Algorithm modifications

4 step (cont.):

if $y(t) = d(t)$ $\Rightarrow$ $w_i(t+1) = w_i(t)$

if $y(t) = 0$ and $d(t) = 1$ $\Rightarrow$ $w_i(t+1) = w_i(t) + \eta \cdot x_i(t)$

if $y(t) = 1$ and $d(t) = 0$ $\Rightarrow$ $w_i(t+1) = w_i(t) - \eta \cdot x_i(t)$

$0 \leq \eta \leq 1$, a positive gain term that controls the adaptation rate.

The perceptron learning algorithm

Widrow and Hoff modification

4 step (cont.):

if $y(t) = d(t)$ $\Rightarrow$ $w_i(t+1) = w_i(t)$

if $y(t) \neq d(t)$ $\Rightarrow$ $w_i(t+1) = w_i(t) + \eta \cdot \Delta \cdot x_i(t)$

$0 \leq \eta \leq 1$ a positive gain term that controls the adaptation rate.

$\Delta = d(t) - y(t)$

The perceptron learning algorithm

The Widrow-Hoff delta rule calculates the difference between the weighted sum and the required output, and calls that the **error**.

This means that during the learning process, the output from the unit is not passed through the step function – however, actual classification is effected by using the step function to produce the +1 or 0.

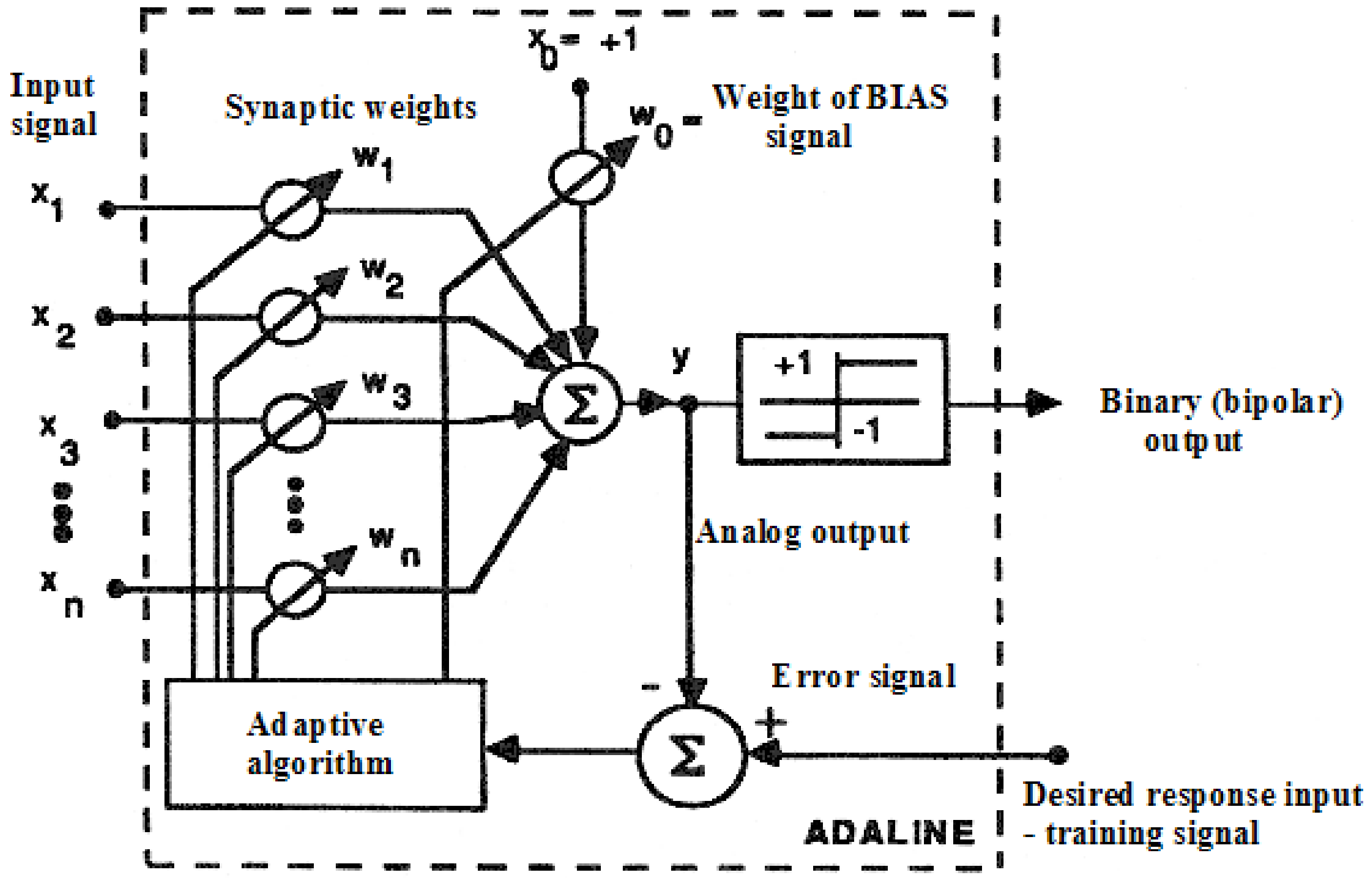
Neuron units using this learning algorithm were called ADALINE*s* (ADAptive LInear NEurons), who are also connected into a many ADALINE, or MADALINE structure.

Model ADALINE

Widrow and Hoff model

The structure ADALINE, and the way how it performs a weighted sum of inputs is similar to the single perceptron unit and has similar limitations. (for example the XOR problem).

Widrow and Hoff model

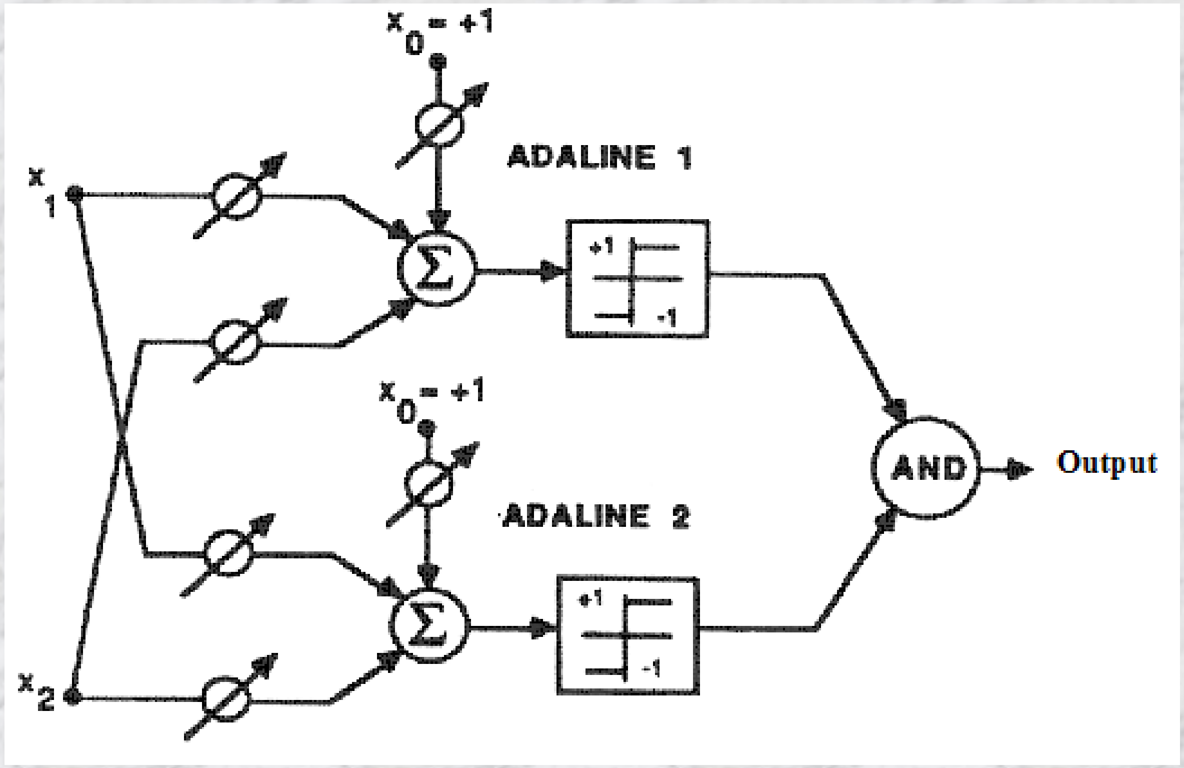


Widrow and Hoff model

When in a perceptron decision concerning change of weights is taken on the base of the output signal ADALINE uses the signal from the sum unit (marked Σ).

Widrow and Hoff model

The system of two ADALINE type elements can realize the logical AND function.



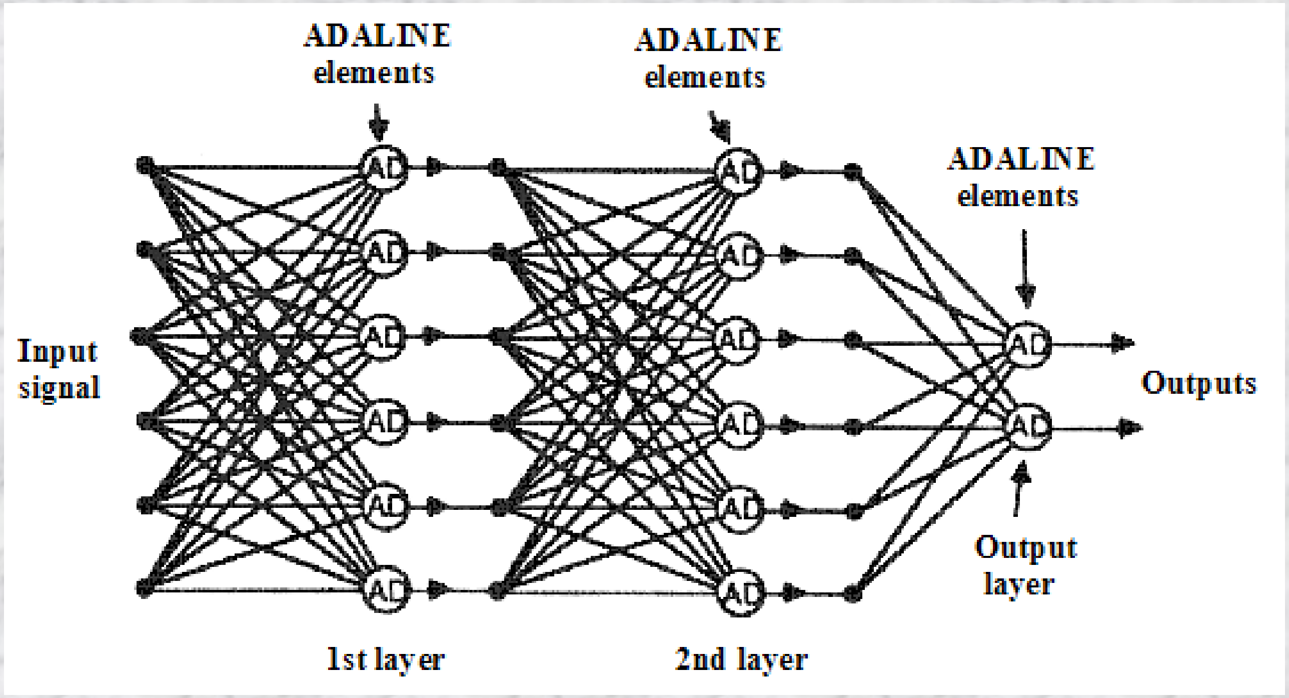
Widrow and Hoff model

Similarly to another multilayer nets (e.g. perceptron), from basic ADALINE elements one can create the whole network called ADALINE or MADALINE.

Complicated net’s structure makes difficult definition of an effective learning algorithm. The most in use is the LMS algorithm (Least-Mean-Square). But for the LMS method it is necessary to know the input and output values of every hidden layer. Unfortunately these information are not accessible.

Widrow and Hoff model

Three layer net composed of ADALINE elements create the MADALINE net.



Widrow and Hoff model

The neuron operation can be described by the formula (assuming the threshold = 0)

$$y = W^T \cdot X$$

where $W = [w_1, w_2, \ldots, w_n]$ is the weight vector

$X = [x_1, x_2, \ldots x_n]$ is the input signal (vector)

Widrow and Hoff model

From the inner product properties, we know that the out put signal will be bigger when the direction of the vector $\mathbf{x}_i$ in the *n*-dimensional space of input signals **X** will coincide with the direction of the vector $\mathbf{w}_i$ in the *n*-dimensional space of the weights **W**. The neuron will react stronger for the input signals more „similar” to the weight vector.
Assuming that vectors $\mathbf{x}_i$ i $\mathbf{w}_i$ are normalized (i.e. $|\mathbf{w}_i| = 1$ i $|\mathbf{x}_i| = 1$), one get

$$y = \cos\Phi$$

where Φ is the angle between the vectors $\mathbf{x}_i$ i $\mathbf{w}_i$.

Widrow and Hoff model

For the *m*-elements layer of the neurons (processing elements), we get

$$\mathbf{Y} = \mathbf{W} \cdot \mathbf{X}$$

where rows in the matrix **W** (1,2,...,m) correspond to the weights coming to particular processing elements from input nods, and

$$\mathbf{Y} = [y_1, y_2, \ldots, y_m]$$

Widrow and Hoff model

The net is mapping the input space X into R^m, $X \rightarrow R^m$. Of course this mapping is absolutely free. One can say that the net is performing the *filtering*.

A net operation is defined by the elements of a matrix W – i. e. the weights are an equivalent of the program in numerical calculation.

The a priori definition of weights is difficult, and in the multilayer nets – practically impossible.

Widrow and Hoff model

The one-step process of the weights determining can be replaced by the multi-step process – ***the learning process***.

It is necessary to expand a system adding the element able to define the output signal error and the element able to control the weights adaptation.

The method of operating the ADALINE is based on the algorithm called **DELTA** introduced by Widrow and Hoff. General idea: each input signal **X** is associated with the signal *d,* the correct output signal.

Widrow and Hoff model

The actual output signal *y* is compared with *d* and the error is calculated. On the base of this error signal and the input signal **X** the weight vector **W** is corrected.

The new weight vector **W'** is calculated by the formula

$$\mathbf{W'} = \mathbf{W} + \eta \cdot e \cdot \mathbf{X}$$

where η is the learning speed coefficient

Widrow and Hoff model

The idea is identical with the perceptron learning rule. When *d* > y it means, that the output signal was too small, the angle between vectors **X** and **W** – was too big. To the vector **W** it is necessary to add the vector **X** multiplied by the constant

$$(0 < \eta \cdot e < 1).$$

{This condition prevents too fast „rotations" of vector **W**}.

The vector **W** correction is bigger when the error is bigger – the correction should be stronger with the big error and m0ore precise with the small one.

Widrow and Hoff model

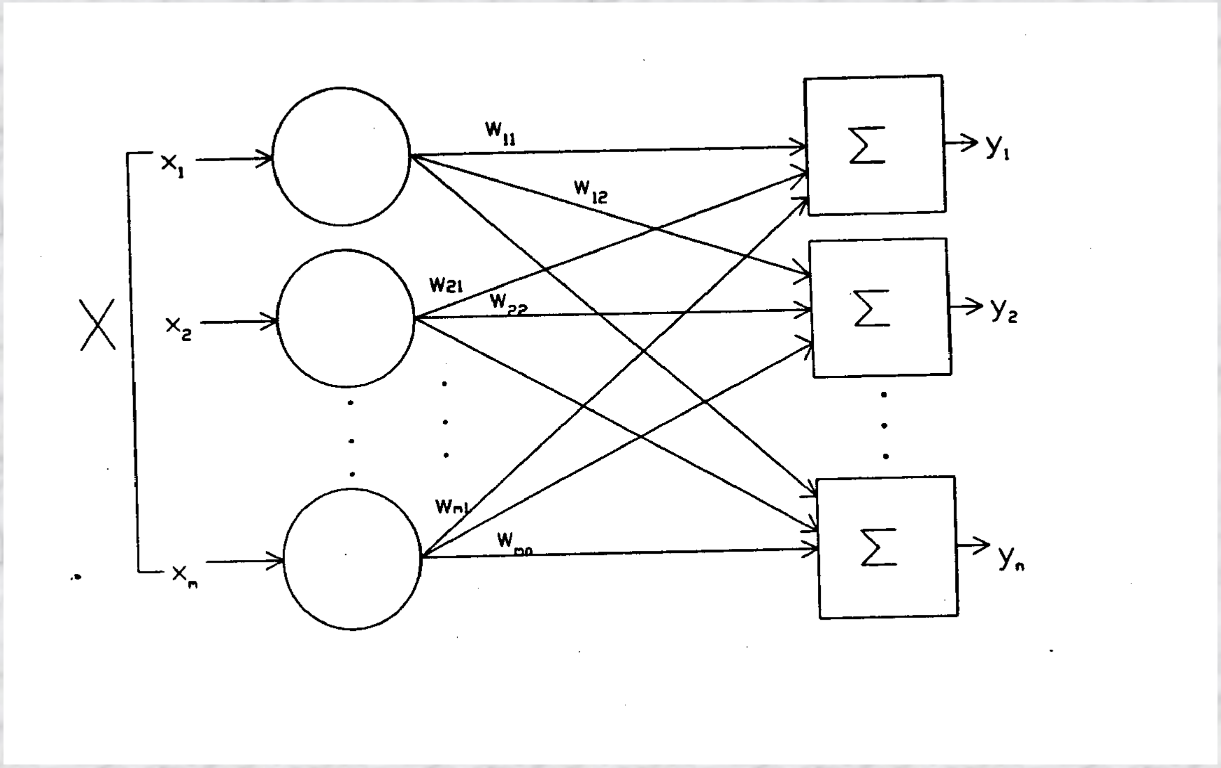
The rule assures, that *i-th* component of the vector **W** is changed more the bigger appropriate component of learned **X** was .

When the components of **X** can be both positive and negative – the sign of the error *e* defines the increase or decrease of **W.**

We'll take a 5-minute break now

The Delta Rule

The Delta learning rule



The one –layer network

The Delta learning rule

The perceptron learning rule is also the delta rule

if $y(t) = d(t) \Rightarrow \quad w_i(t+1) = w_i(t)$
if $y(t) \neq d(t) \Rightarrow \quad w_i(t+1) = w_i(t) + \eta \cdot \Delta \cdot x_i(t)$

where
$0 \leq \eta \leq 1$ is the learning coefficient
and $\Delta = d(t) - y(t)$

The Delta learning rule

The basic difference is in the error definition – discrete in the perceptron and continuous in the Adaline model.

The Delta learning rule

Let δ_k, the error term, defines the difference between the $\boldsymbol{d_k}$ desired response of the *k*-th element of the output layer, and is the actual response (real) $\boldsymbol{y_k}$.

Let us define the error function ***E*** to be equal to the square of the difference between the actual and desired output, for all elements in the output layer

$$E = \sum_{k=1}^{N} \delta_k^2 = \sum_{k=1}^{N} (d_k - y_k)^2$$

The Delta learning rule

Because

$$y_k = \sum_{i=1}^{n} w_{ik} x_i$$

thus

$$E = \sum_{k=1}^{N} (d_k - \sum_{i=1}^{n} w_{ik} x_i)^2$$

The Delta learning rule

The error function E is the function of all the weights. It is the square function with respect to each weight, so it has exactly one minimum with respect to each of the weights. To find this minimum we use the ***gradient descend method.***

Gradient of **E** is the vector consisting of the partial derivatives of **E** with respect to each variable. This vector gives the direction of most rapid increase in function; the opposite direction gives the direction of most rapid decrease in the function.

The Delta learning rule

So, the weight change is proportional to the partial derivative of a error function with respect to this weight with the minus sign.

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

where η is the learning rate

The Delta learning rule

Each weight can be fixed this way.

Lets calculate the partial derivative of ***E***

$$\frac{\partial E}{\partial w_{ik}} = \frac{\partial E}{\partial \delta_k}\frac{\partial \delta_k}{\partial w_{ik}} = 2\delta_k \frac{\partial \delta_k}{\partial y_k}\frac{\partial y_k}{\partial w_{ik}} = 2\delta_k(-1)x_i = -2\delta_k x_i$$

thus

$$\Delta w_{ik} = 2\eta\delta_k x_i$$

The Delta learning rule

The DELTA RULE changes weights in a net proportionally to the output error (the difference between the real and desired output signal), and the value of input signal

$$\boxed{\Delta w_{ik} = 2\eta\delta_k x_i}$$

The multilayer Perceptron

Many years the idea of multi layer perceptron was introduced . Multi – typically three

Layers: input, output and hidden.

In 1986 Rumelhart and McClelland described the new learning rule **the backpropagation learning rule**.

Neural network for Classification

Neural Network for Classification

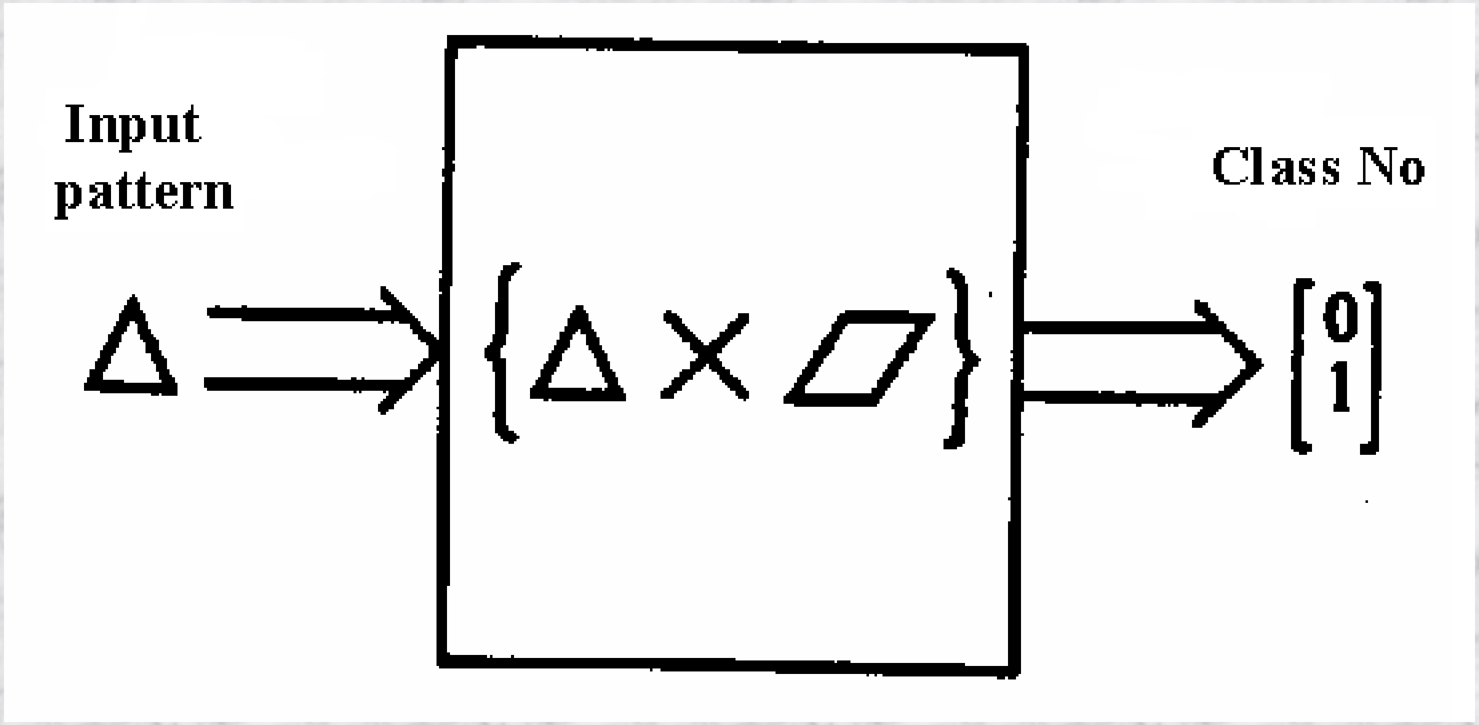
The fundamental objective for pattern recognition is ***classification*** and it is the most typical form of neural transformation. A pattern recognition system can be treated as a two stage device: **feature extraction** and **classification**. A feature is defined as a measurement taken on the input pattern that is to be classified but the classifier has to map the input features onto a classification state. The classifier must to decide which type of class category they match most closely.

Neural Network for Classification

Any given input pattern must belong to one of the classes that are in consideration. So, the mapping from the input to the required class must exists.

This mapping is a function that transforms the input pattern into the correct output class, and we will consider that our network has learnt to perform correctly, if it can carry out this mapping.

Neural Network for Classification

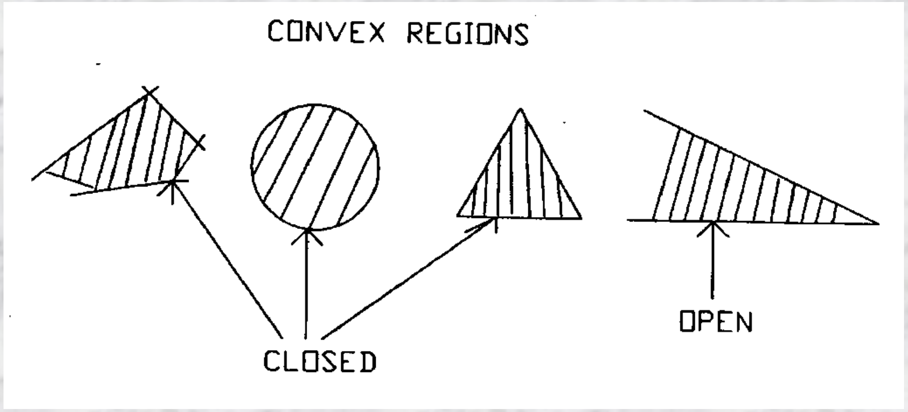


Neural Network for Classification

We spoke about the multilayer perceptron limitations. When the problem of linear separability was well understood it was also found that the problem of single-layer network can be overcome by adding more layers. For example, the two layer network may be formed by cascading two single-layer networks. These can perform more general classification, separating those points that are contained in convex open or closed regions.

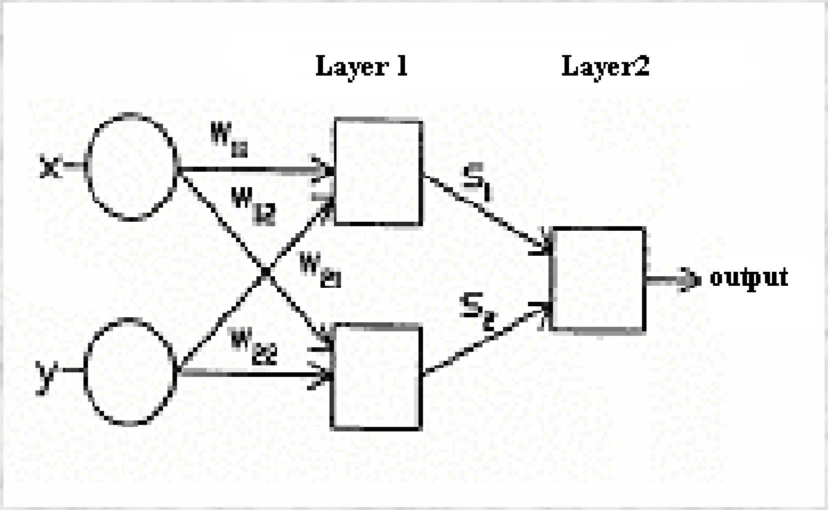
A convex regions

A convex region – is one in which any two points in the region can be joined by a straight line that does not leave the region.



Neural Network for Classification

To understand the convexity limitation, consider a simple two-layer network with two inputs going to two neurons in the first layer, both feeding a single neuron in the layer 2. The output neuron threshold is set to 0.75 and weights s_i are both set to 0.5



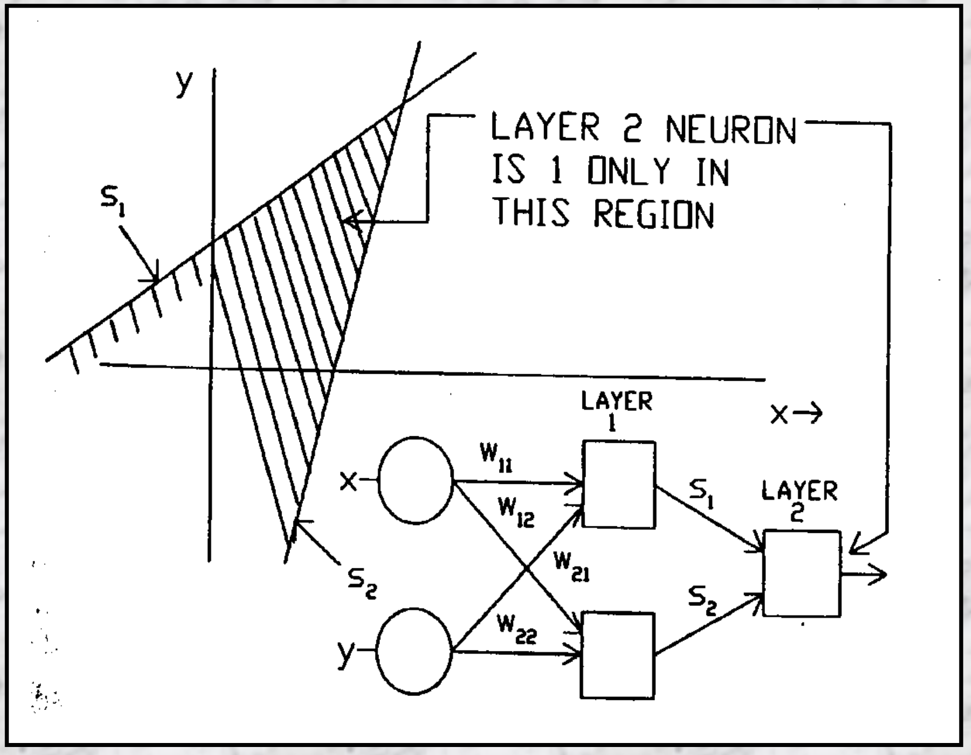
Neural Network for Classification

In this case the output of one (**1**) is required from both layer 1 neurons to exceed the threshold and to produce a one (**1**) on the output. Thus the output neuron performs a logical **AND** function.

Each neuron in layer 1 subdivides the **OXY** plane , producing tan output of one (**1**) on one side of the line.

Neural Network for Classification

The result of double subdivision when the output of one (**1**) of the layer 2 neuron is only over V-shaped region.



Neural Network for Classification

Similarly, three neurons used in the layer 1 further subdivide the plane, creating for example a triangle-shaped region. By including enough neurons in the layer 1, a convex region of any desired shape can be formed.

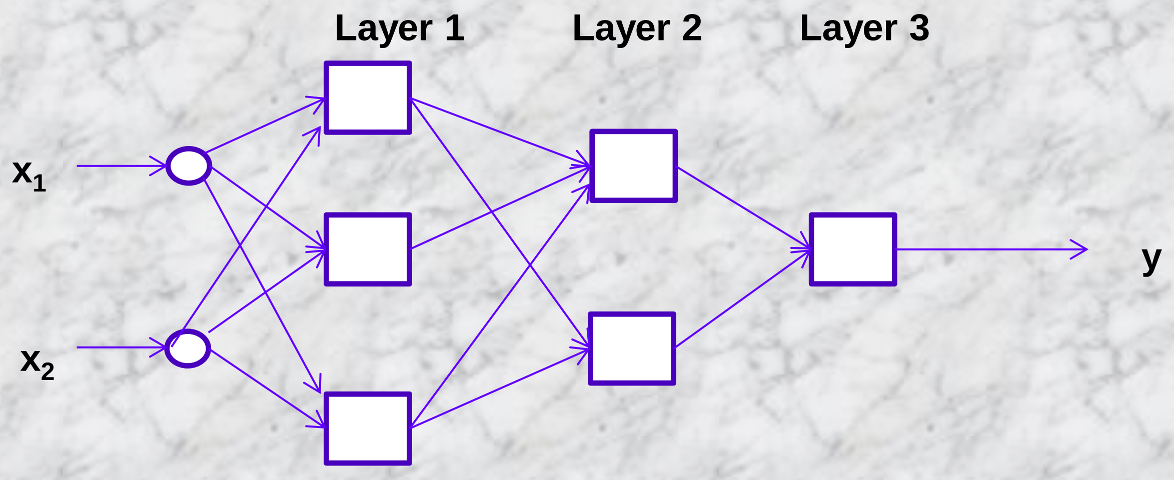
The layer 2 of course is not limited to the AND function, it can produce many other functions if the weights and threshold are suitably chosen.

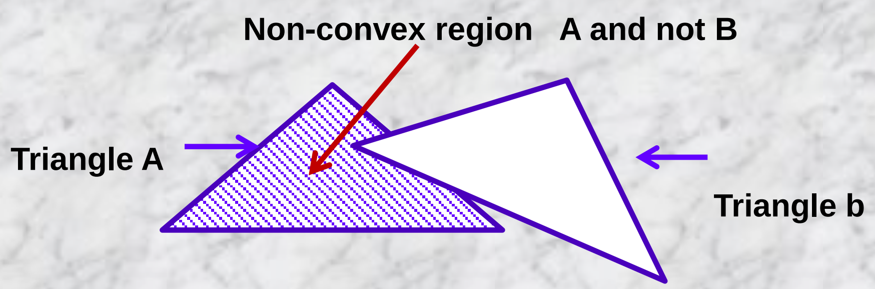
Neural Network for Classification

A three-layer network is still more general, its limitation capability is limited only by the number of neurons and weights. There are no convexity constrains; the layer 3 neuron receives as input a group of convex polygons, and the logical combination of that need not to be convex.

Neural Network for Classification

A concave decision region formed by intersection of two convex regions. Logical operation **A and not B**





Neural Network for Classification

Existence (Kolmogorov) Theorem

Any continuous function of n variables can be computed using only linear summations and nonlinear but continuously increasing functions of only one variable. It effectively states that a three layer perceptron with n(2n+I) nodes using continuously increasing non-linearities can compute any continuous function of n variables. A three layer perceptron could then be used to create any continuos likelihood function required in a classifier.

Neural Network for Classification

Conclusions

To create any arbitrary complex shape (decision region), we never need more that three layers in the network.

It gives the limitation on layers but does not define:

- how many elements is necessary to create a network (in general and in particular layers),
- how these elements should be connected,
- which weights value should be.

Network structure

Inconsistency in nomenclature

What is layer???

- some authors refer to the number of layers of variable weights
- some authors describe the number of layers of nodes

Usually, the nodes in the first layer, the input layer, merely distribute the inputs to subsequent layers, and do not perform and operations (summation or thresholding) n.b. some authors miss out these nodes.

Network structure

What is a network layer?

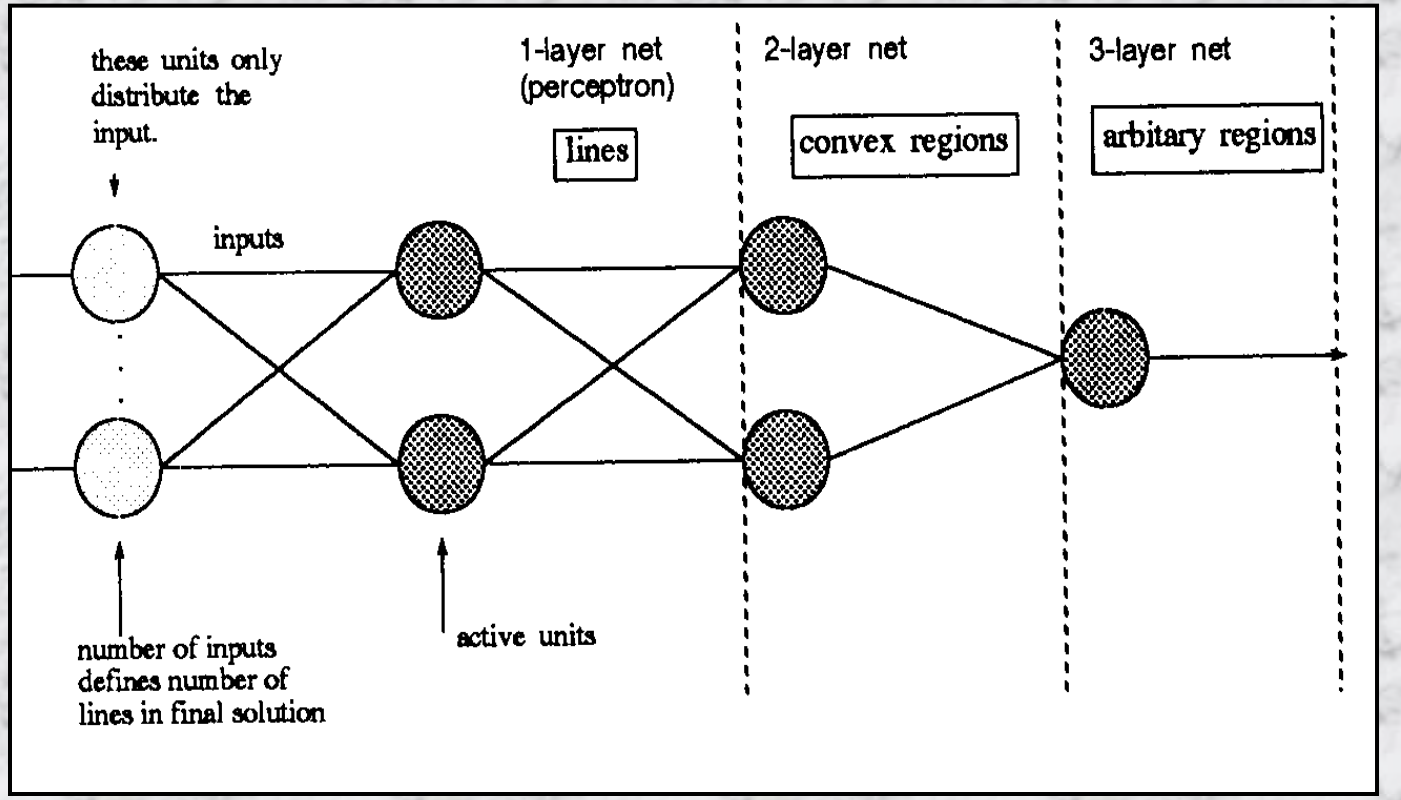
A layer - it is the part of network structure which contains active elements performing some operation. A multilayer network receives a number of inputs. These are distributed by a layer of input nodes that do not perform any operation – these inputs are then passed along the first layer of adaptive weights to a layer of perceptron-like units, which do sum and threshold their inputs. This layer is able to produce classification lines in pattern space.

Neural Network for Classification

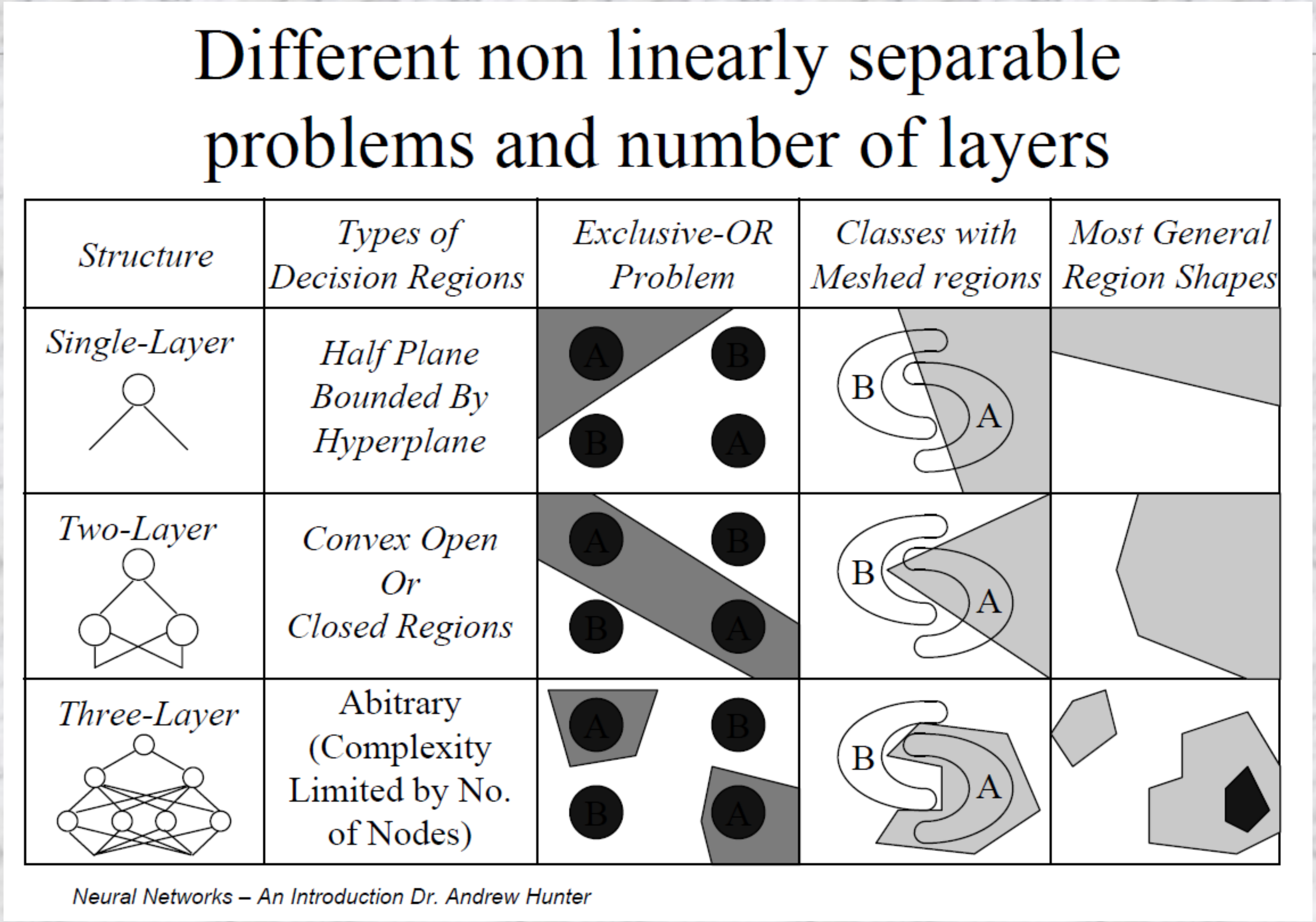
The output from this layer in then passed to another layer, and the output of this layer forms convex regions in pattern space. A further layer is able to define any arbitrary shape in pattern space.

Neural Network for Classification

Neural networks and their corresponding decision regions



Neural Network for Classification



Different non linearly separable problems and number of layers

Structure	*Types of Decision Regions*	*Exclusive-OR Problem*	*Classes with Meshed regions*	*Most General Region Shapes*
Single-Layer	*Half Plane Bounded By Hyperplane*	A B B A	B A	
Two-Layer	*Convex Open Or Closed Regions*	A B B A	B A	
Three-Layer	Abitrary (Complexity Limited by No. of Nodes)	A B B A	B A	

Neural Networks – An Introduction Dr. Andrew Hunter

Neural Network for Classification

A multilayer perceptron is **fault-tolerant**, since it is distributed parallel processing element, with each node contributing to the final output.

If a node or its weights are lost or damaged. recall is impaired in quality, but the distributed nature of the information means that the damage has to be extensive before the network's response degrades badly. The network demonstrate graceful degradation rather that catastrophic failure.

We'll take a 5-minute break now

Backpropagation Algorithm

Backpropagation

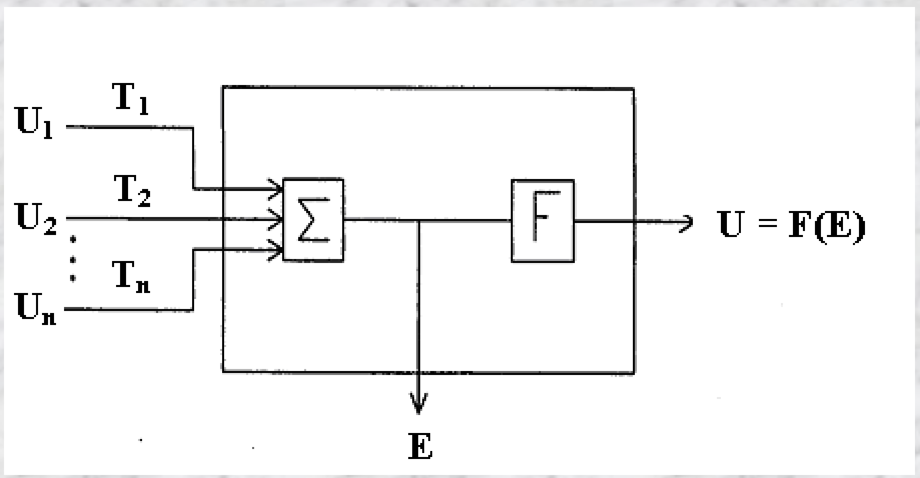
For many years there was no theoretical sound algorithm for training multilayer artificial neural networks. Since single-layer networks proved severely limited in what they could represent, the entire field went into virtual eclipse.

In 1986 Rumelhart and McClelland suggested a new learning rule known as ***a backpropagation rule*** which is today used in many practical applications like for example in solving the optimization problems.

Backpropagation

The invention of the backpropagation algorithm has played a large part in the resurgence of interest in artificial neural networks.

Backpropagation is a systematic method for training multilayer artificial neural networks.



Backpropagation

It is the rule how to change the weights $\mathbf{T_{ij}}$ between network elements.

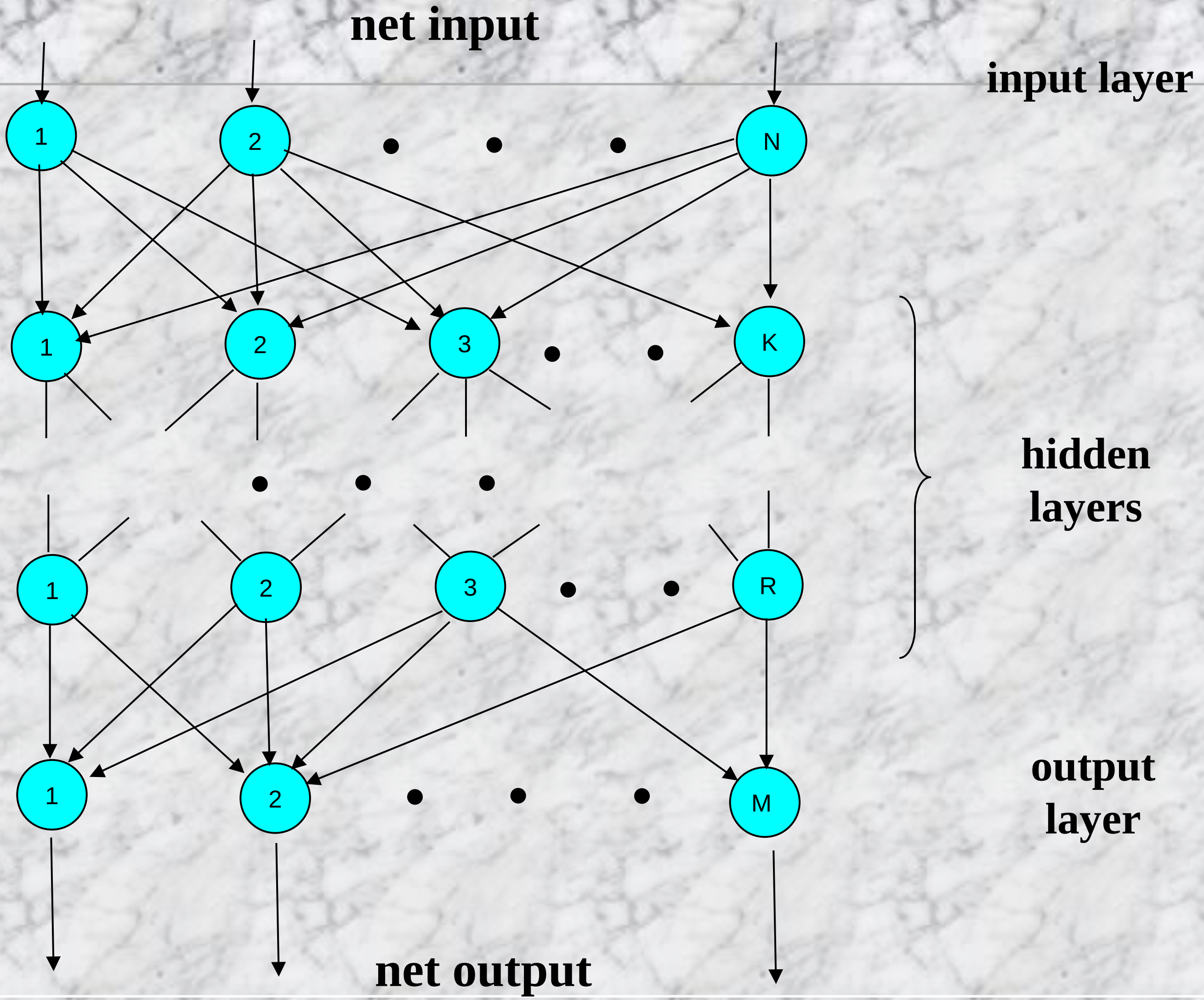
The algorithm is based on the idea to minimize the square root of errors by use of the gradient descent method.

Backpropagation

Assumptions:

- the net is the regular, multilayer structure
- the first layer – input layer
- the last layer – output layer
- layers between – hidden layers
- feed forward propagation only

Backpropagation



Backpropagation

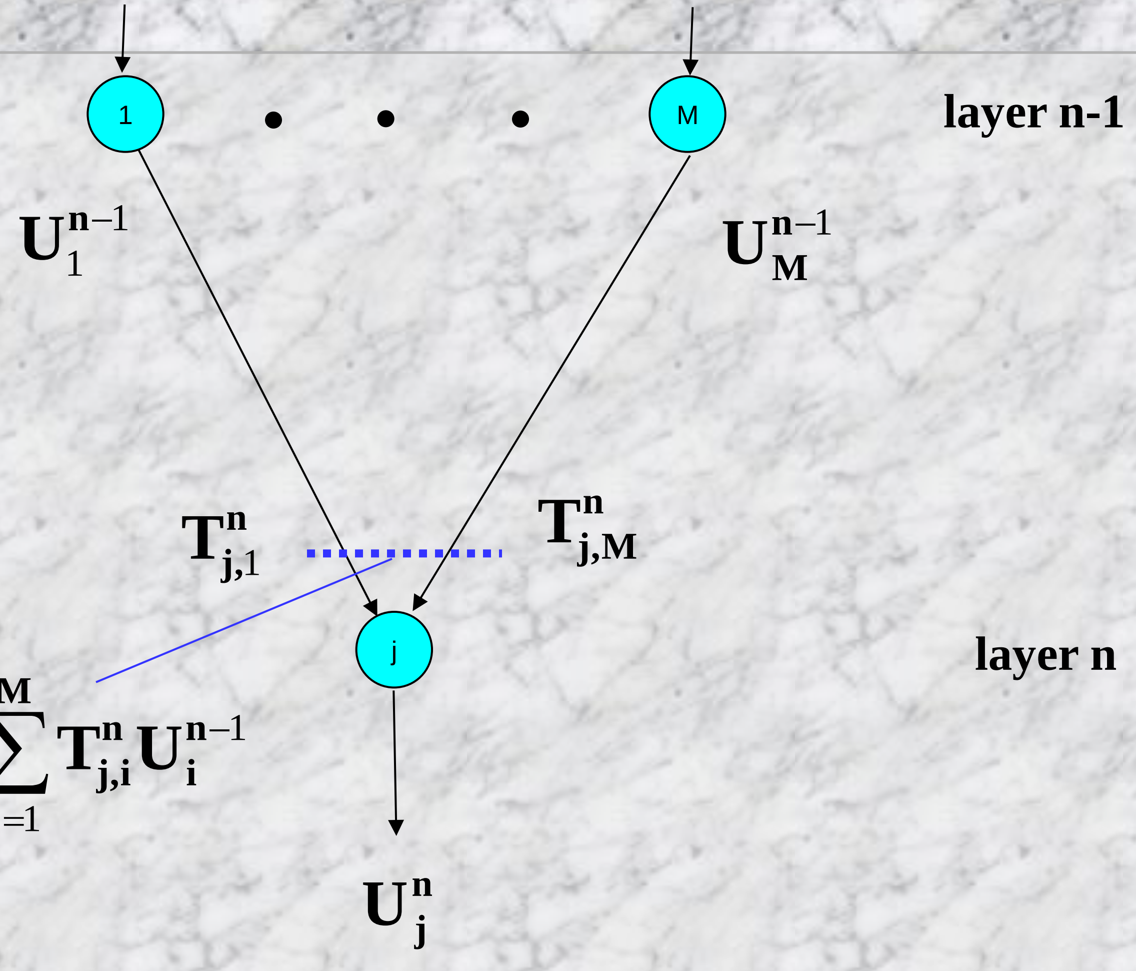
To define a state of *j*-th neuron in layer *n* we calculate the weighted sum of its **M** inputs

$$\mathbf{E}_j^n = \sum_{i=1}^{M} \mathbf{T}_{j,i}^n \mathbf{U}_i^{n-1} \qquad \textbf{(1)}$$

where

$\mathbf{E}_j^n$	weighted input sum of *j*-th neuron in layer *n*
$\mathbf{T}_{j,i}^n$	connection weight between *i*-th neuron inlayer *n*-1 and *j*-th neuron in layer *n*
$\mathbf{U}_i^{n-1}$	out put signal of *i*-th neuron in layer *n*-1

Backpropagation



$$E_j^n = \sum_{i=1}^{M} T_{j,i}^n U_i^{n-1}$$

Backpropagation

output signal of *j*-th neuron in layer *n* is defined by

$$\mathbf{U}_{\mathbf{j}}^{\mathbf{n}} = f(\mathbf{E}_{\mathbf{j}}^{\mathbf{n}}) \qquad \textbf{(2)}$$

where *f* is the neuron transfer function

$$\mathbf{U}_{\mathbf{j}}^{\mathbf{n}} = f(\mathbf{E}_{\mathbf{j}}^{\mathbf{n}}) = \frac{1}{1 + exp[-(\mathbf{E}_{\mathbf{j}}^{\mathbf{n}} - \Theta_{\mathbf{j}}^{\mathbf{n}})/\Theta_0]} \qquad \textbf{(3)}$$

f – sigmoid function, the output values $\in$ (0;1),

$\Theta_{\mathbf{j}}^{\mathbf{n}}$ - the threshold, Θ_0 - slope

Backpropagation

The global error **D** is a differentiable function of weights

$$D = \frac{1}{2} \cdot \sum_{j=1}^{M} (U_j^* - U_j^{out})^2 \qquad \textbf{(4)}$$

where

$\mathbf{U_j^*}$	desired output (a target) of *j*-th neuron in the output layer
U_j^{out}	actual output of *j*-th neuron in the output layer

Backpropagation

Aim: minimization of a global error **D**, modifying the network's weights.

Goal: define the rules of the weights adaptation to minimize the global error

Solution: the gradient descent method

$$\Delta \mathbf{T}^{\mathbf{n}}_{\mathbf{j,i}} = -\eta \frac{\partial \mathbf{D}}{\partial \mathbf{T}^{\mathbf{n}}_{\mathbf{j,i}}} \qquad \textbf{(5)}$$

where η is the learning coefficient

Backpropagation

Each weight is changed according to the value and direction of negative gradient on the hyperplane **D(T)**.

The partial derivative in (5) can be calculated by use of the chain rule.

Backpropagation

$$\frac{\partial D}{\partial T_{j,i}^{n}} = \frac{\partial D}{\partial E_{j}^{n}} \cdot \frac{\partial E_{j}^{n}}{\partial T_{j,i}^{n}} \qquad \mathbf{(6)}$$

We define the change in error as a function of the change in the net inputs to a *j*-th neuron as

$$d_{j}^{n} = -\frac{\partial D}{\partial E_{j}^{n}}$$

Backpropagation

$$\frac{\partial E_j^n}{\partial T_{j,i}^n} = -\frac{\partial}{\partial T_{j,i}^n}\sum_{k=1}^{M} T_{j,k}^n U_k^{n-1} = \sum_{k=1}^{M}\frac{\partial T_{j,k}^n}{\partial T_{j,i}^n} U_k^{n-1} = U_i^{n-1} \qquad (7)$$

because

$$\frac{\partial T_{j,k}^n}{\partial T_{j,i}^n} = \begin{cases} 0 & \text{for } k \neq i \\ 1 & \text{for } k = i \end{cases}$$

Backpropagation

Finally, we get

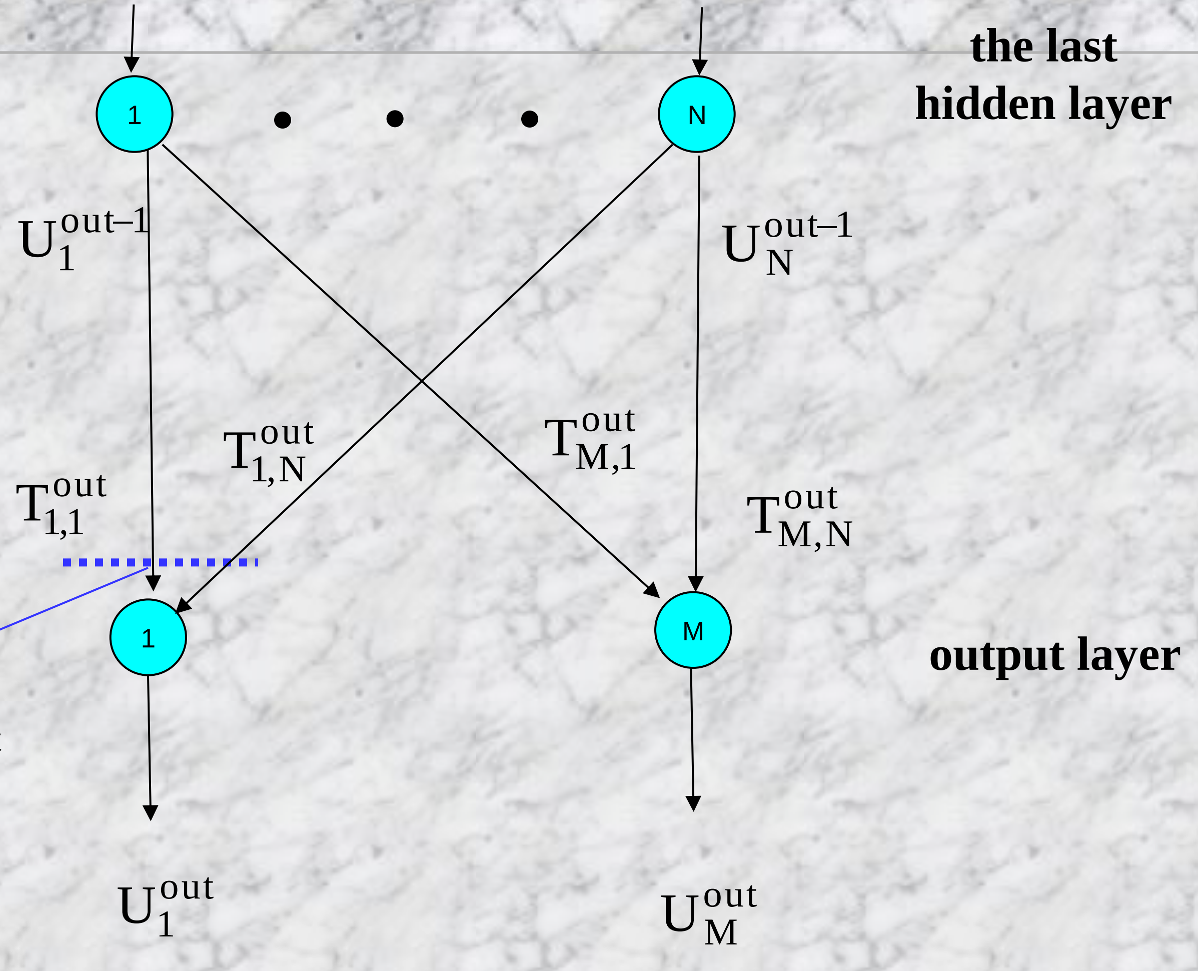
$$\boxed{\Delta \mathbf{T}^{\mathbf{n}}_{\mathbf{j,i}} = \eta \mathbf{d}^{\mathbf{n}}_{\mathbf{j}} \mathbf{U}^{\mathbf{n\text{-}1}}_{\mathbf{i}}} \qquad \textbf{(8)}$$

For the elements in the output layer

$$\Delta \mathbf{T}^{\mathbf{out}}_{\mathbf{j,i}} = \eta \mathbf{d}^{\mathbf{out}}_{\mathbf{j}} \mathbf{U}^{\mathbf{out\text{-}1}}_{\mathbf{i}} \qquad \textbf{(9)}$$

Decrease of **D** means the changes proportional to $\mathbf{d}^{\mathbf{out}}_{\mathbf{j}} \mathbf{U}^{\mathbf{out\text{-}1}}_{\mathbf{i}}$

Backpropagation



$$E_1^{out} = \sum_{i=1}^{N} T_{1,i}^{out} U_1^{out}$$

Backpropagation

Now, we need to know what $\mathbf{d_j^{out}}$ is for each of the units

$$\mathbf{d_j^{out}} = -\frac{\partial \mathbf{D}}{\partial \mathbf{E_j^{out}}} = \frac{\partial \mathbf{D}}{\partial \mathbf{U_j^{out}}} \frac{\partial \mathbf{U_j^{out}}}{\partial \mathbf{E_j^{out}}} \qquad \mathbf{(10)}$$

where

$$\frac{\partial \mathbf{U_j^{out}}}{\partial \mathbf{E_j^{out}}} = f'(\mathbf{E_j^{out}})$$

Backpropagation

differentiate $\mathbf{D}$ with respect to $\mathbf{U}_j^{out}$ giving

$$\frac{\partial \mathbf{D}}{\partial \mathbf{U}_j^{out}} = -(\mathbf{U}_j^{*} - \mathbf{U}_j^{out}) \qquad \textbf{(11)}$$

thus

$$\mathbf{d}_j^{out} = (\mathbf{U}_j^{*} - \mathbf{U}_j^{out}) \cdot f'(\mathbf{E}_j^{out}) \qquad \textbf{(12)}$$

Backpropagation

This is useful for the output units to modify weights $T^{out}_{j,i}$ between the neurons of the output layer and the neurons of the last hidden layer (since the target and output both are available)

$$T^{out}_{j,i}(t) = T^{out}_{j,i}(t-1) + \tau\, \Delta T^{out}_{j,i}$$

where τ is a learning coefficient **(13)**

Backpropagation

The formula (9) can be rewritten to avoid oscillations

$$\Delta T_{j,i}^{out}(t) = \alpha \Delta T_{j,i}^{out}(t-1) + (1-\alpha) d_j^{out} \boldsymbol{f}(E_i^{out-1}) \quad \textbf{(14)}$$

where $\mathbf{E}_{\mathbf{j}}^{\mathbf{wy-1}}$ is the weighted input sum of *i*-th element from the last hidden layer, *α* (so called smoothing parameter), is a constant value defining the effect of the previous weights modification on the actual modification .

Backpropagation

This method is very useful for the output layer elements because we have access both to the output signal, target signal.

However, for the elements located in the hidden layers unfortunately does not work.

Backpropagation

If *j*-th neuron **does not belong** to the output layer but is an element of a hidden layer *s*, then its local weight modification is calculated from

$$\mathbf{d}_j^s = -\frac{\partial \mathbf{D}}{\partial \mathbf{E}_j^s} \qquad \textbf{(15)}$$

Because

$$\frac{\partial \mathbf{D}}{\partial \mathbf{E}_j^n} = \frac{\partial \mathbf{D}}{\partial \mathbf{U}_j^n} \cdot \frac{\partial \mathbf{U}_j^n}{\partial \mathbf{E}_j^n} = f'(\mathbf{E}_j^n)\frac{\partial \mathbf{D}}{\partial \mathbf{U}_j^n} \qquad \textbf{(16)}$$

Backpropagation

also

$$\frac{\partial \mathbf{D}}{\partial \mathbf{U}_j^n} = \sum_{k=1}^{N} \frac{\partial \mathbf{D}}{\partial \mathbf{E}_k^{n+1}} \frac{\partial \mathbf{E}_k^{n+1}}{\partial \mathbf{U}_j^n} = \sum_{k=1}^{N} \left[\frac{\partial \mathbf{D}}{\partial \mathbf{E}_k^{n+1}} \frac{\partial}{\partial \mathbf{U}_j^n} \left(\sum_{i=1}^{M} \mathbf{T}_{k,i}^{n+1} \mathbf{U}_i^n \right) \right] =$$

$$= \sum_{k=1}^{N} \frac{\partial \mathbf{D}}{\partial \mathbf{E}_k^{n+1}} \mathbf{T}_{k,j}^{n+1} = \sum_{k=1}^{N} \mathbf{d}_k^{n+1} \mathbf{T}_{k,j}^{n+1} \qquad \mathbf{(17)}$$

then

$$\mathbf{d}_j^n = -\frac{\partial \mathbf{D}}{\partial \mathbf{E}_j^n} = f'(\mathbf{E}_j^n) \sum_{k=1}^{N} \mathbf{d}_j^{n+1} \mathbf{T}_{k,j}^{n+1} \qquad \mathbf{(18)}$$

Backpropagation

it modify the weight $\mathbf{T}^{n}_{j,i}$ between *i*-th element of the layer *n*-1 and *j*-th element of the layer *n*

$$\mathbf{T}^{n}_{j,i}(\mathbf{t}) = \mathbf{T}^{n}_{j,i}(\mathbf{t-1}) + \tau\, \Delta\mathbf{T}^{n}_{j,i} \qquad \mathbf{(19)}$$

a

$$\Delta\mathbf{T}^{n}_{j,i}(\mathbf{t}) = \alpha\Delta\mathbf{T}^{n}_{j,i}(\mathbf{t-1}) + (1-\alpha)\,\mathbf{d}^{n}_{j}\, f(\mathbf{E}^{n-1}_{i}) \qquad \mathbf{(20)}$$

Iterative calculations correct weight „backwards”, to the input layer.

Backpropagation

This learning procedure is repeated for each learning pattern, until the network will generate the correct answer for every input signal (pattern) – of course with the defined accuracy.

Backpropagation

Example:

If $$f(\mathbf{E}) = \frac{1}{1+exp(-k\mathbf{E})}$$

f**(E)** $\in$ **(0;1), k>0**

when **k** $\Rightarrow \infty$

then f**(E)** $\Rightarrow$ step function

$$\mathbf{U_j^{out}} = \mathrm{f}(\mathbf{E_j^{out}}) = \frac{1}{1+\exp(-\mathrm{k}\mathbf{E_j^{out}})}$$

Backpropagation

calculating the derivative *f'*(**E**) for *j*-th element:

$$f'(\mathbf{E}) = \frac{k\mathbf{e}^{-\mathbf{kE}}}{(\mathbf{1}+\mathbf{e}^{-\mathbf{kE}})^{2}} = kf(\mathbf{E})\cdot[\mathbf{1}-f(\mathbf{E})] =$$

$$= k\mathbf{U}_{\mathbf{j}}^{\mathbf{out}}(\mathbf{1}-\mathbf{U}_{\mathbf{j}}^{\mathbf{out}})$$

(this derivative simplify he calculations)

Backpropagation

for the elements in the output layer

$$\mathbf{d}_j^{\mathbf{out}} = (\mathbf{U}_j^{*} - \mathbf{U}_j^{\mathbf{out}}) \cdot f'(\mathbf{E}_j^{\mathbf{out}}) =$$

$$= k \cdot \mathbf{U}_j^{\mathbf{out}} \cdot (\mathbf{1} - \mathbf{U}_j^{\mathbf{out}}) \cdot (\mathbf{U}_j^{*} - \mathbf{U}_j^{\mathbf{out}})$$

for the elements in the hidden layers

$$\mathbf{d}_j^{\mathbf{n}} = k \cdot \mathbf{U}_j^{\mathbf{n}} \cdot (\mathbf{1} - \mathbf{U}_j^{\mathbf{n}}) \cdot \sum_{\mathbf{k}=\mathbf{1}}^{\mathbf{N}} \mathbf{d}_{\mathbf{k}}^{\mathbf{n}+\mathbf{1}} \mathbf{T}_{\mathbf{j,k}}^{\mathbf{n}+\mathbf{1}}$$

Recurrent MultiLayer Perceptron (RMLP)

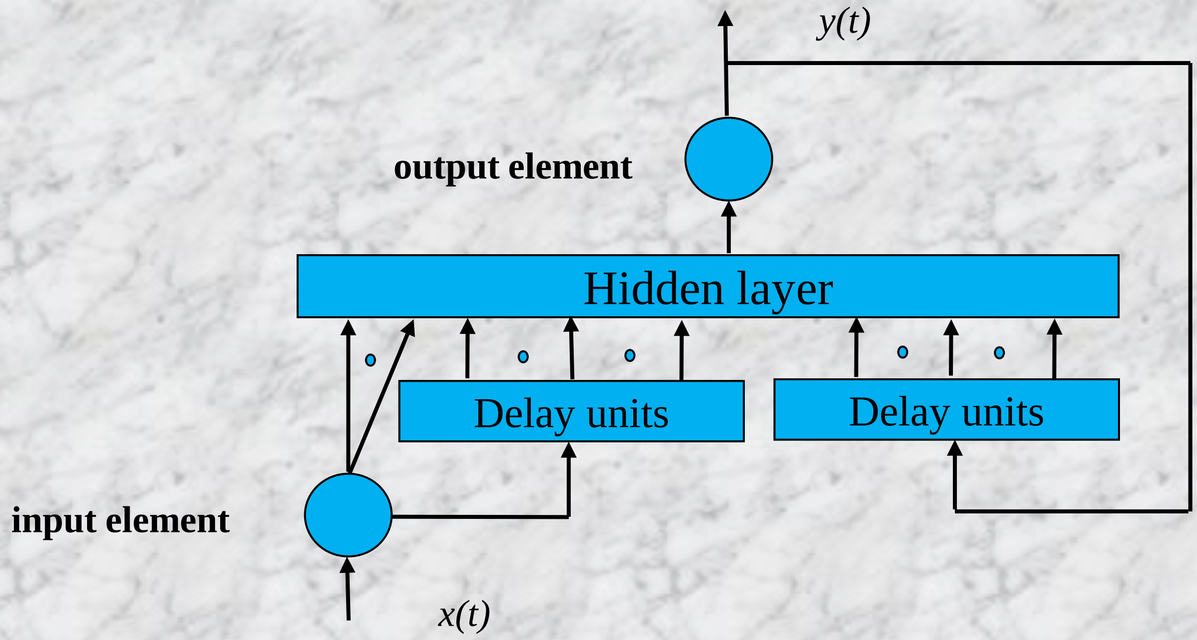
RMLP

Recurrent neural network architectures consists of a standard Multi-Layer Perceptron (MLP) plus added loops.

RMLP

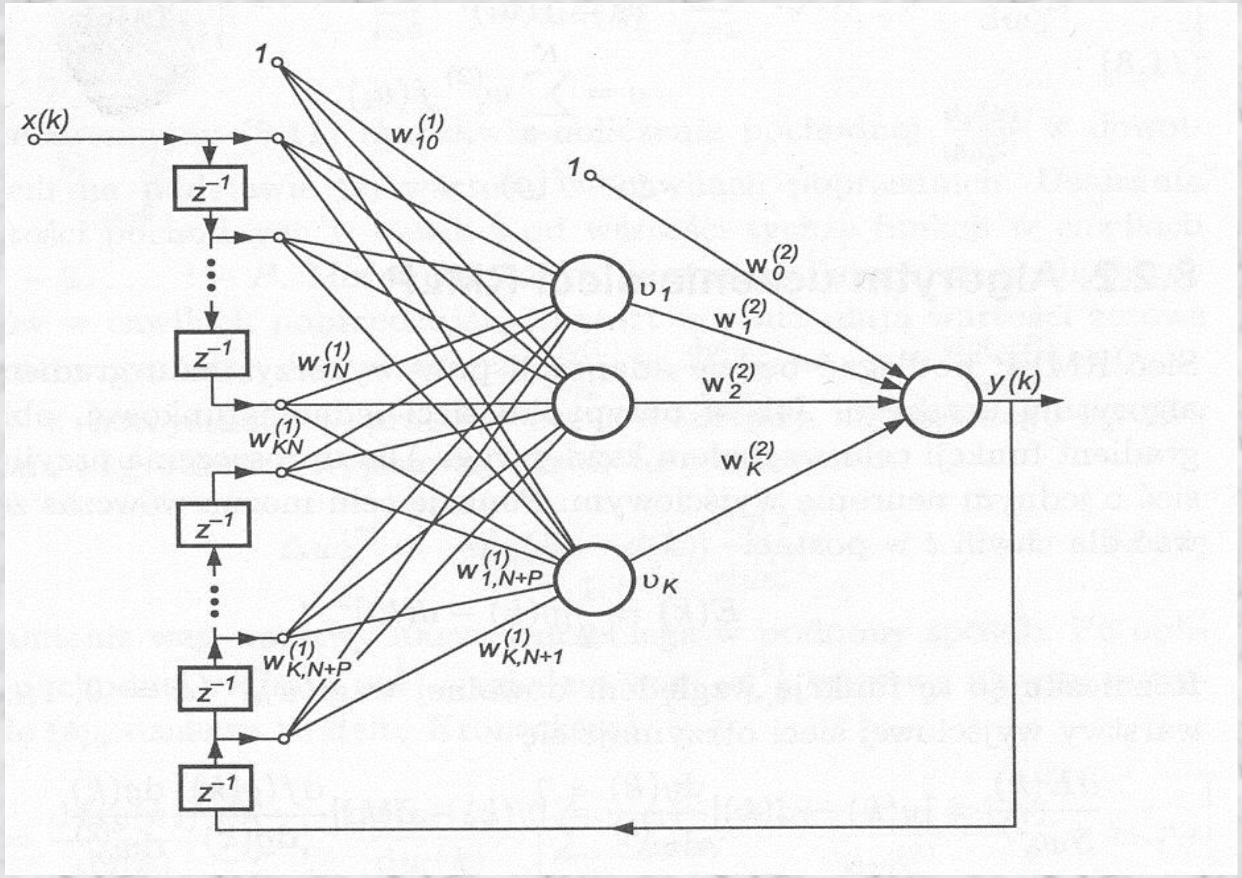
Assumption: only one input nod (input signal *x(t)*) and one output nod (output *signal y(t)*), and one hidden layer.

The network input signal is composed from input *x*, delayed inputs and external recurrence output signal *y(t)* with some unit time delays.



RMLP

Structure of the RMLP network



RMLP

RMLP is a dynamic network with delayed input and output signals.

The farther analysis is performed for only one input node (signal *x(t)*), one output node (signal *y(t)*) and one hidden layer.

The function

$$y(t+1)=f(x(t),x(t-1), \ldots, x(t-(N-1)),y(t-1),y(t-2),\ldots,y(t-P))$$

where

$N-1$	⇨ number of delayed input signals,
P	⇨ number of delayed output signals,
K	⇨ number of neurons in the hidden layer,

RMLP

Vector input signal ***x*** applied to the network input in the time *t*

x*(t)=[1,x(t),x(t-1), ..., x(t-(N-1)),y(t-P),y(t-P+1),...,y(t-1)]*

Denote

$u_i = \sum_{j=0}^{N+P} w_{ij}^{(1)} x_j$	weighted input signal of the *i*-th neuron of the hidden layer
$v_i = f(u_i)$	output signal of the *i*-th neuron of the hidden layer
$g = \sum_{i=0}^{K} w_i^{(2)} f(u_j)$	weighted input signal of the output neuron
$y = f(g)$	output signal of the output neuron

(1-4)

RMLP

RMLP Learning Algorythm

The gradient learning algorithm is used. The gradient of ab error function with respect to each networks’ weight is calculated. For RMLP network the error function can be define by:

$$E(t)=\frac{1}{2}[y(t)-d(t)]^2$$

Differentiating with respect to each weight $w_{\alpha}^{(2)}$ in the second layer

RMLP

$$\frac{\partial E(t)}{\partial w_\alpha^{(2)}} = [y(t) - d(t)]\frac{dy(t)}{dw_\alpha^{(2)}} = [y(t) - d(t)]\frac{df(g(t))}{dg(t)}\frac{dg(t)}{dw_\alpha^{(2)}} =$$

$$= [y(t) - d(t)]\frac{df(g(t))}{dg(t)}\sum_{i=0}^{K}\frac{d(w_\alpha^{(2)} v_i(t))}{dw_\alpha^{(2)}} =$$

$$= [y(t) - d(t)]\frac{df(g(t))}{dg(t)}\left[v_\alpha(t) + \sum_{i=0}^{K} w_\alpha^{(2)}\frac{dv_i(t)}{dw_\alpha^{(2)}}\right]$$

because

$$\frac{d(w_\alpha^{(2)})}{dw_\alpha^{(2)}} = \begin{Bmatrix} 1 & \text{for i} = \alpha \\ 0 & \text{for i} \neq \alpha \end{Bmatrix}$$

RMLP

$$\frac{dv_i(t)}{dw_\alpha^{(2)}} = \frac{df(u_i(t))}{du_i(t)} \sum_{j=0}^{N+P} w_{ij}^{(1)} \frac{dx_j}{dw_\alpha^{(2)}} =$$

$$= \frac{df(u_i(t))}{du_i(t)} \sum_{j=N+1}^{N+P} w_{ij}^{(1)} \frac{dy(t-P-1+(j-N))}{dw_\alpha^{(2)}} =$$

$$= \frac{df(u_i(t))}{du_i(t)} \sum_{j=0}^{P} w_{ij}^{(1)} \frac{dy(t-P-1+j)}{dw_\alpha^{(2)}}$$

RMLP

next

$$\frac{dy(t)}{dw_{\alpha}^{(2)}} =$$

$$= \frac{df(g(t))}{dg(t)} \left[v_{\alpha}(t) + \sum_{i=0}^{K} w_i^{(2)} \frac{df(u_i(t))}{du_i(t)} \sum_{j=1}^{P} w_{i,j+N}^{(1)} \frac{dy(t-P-1+j)}{dw_{\alpha}^{(2)}} \right]$$

this formula enable calculater the $\frac{dy(t)}{dw_{\alpha}^{(2)}}$ in the epoch *t* on the base of its previus values

RMLP

Using the gradient descent method the weight change of the output layer is defined by

$$\Delta w_{\alpha}^{(2)} = -\eta[y(t) - d(t)]\frac{dy(t)}{dw_{\alpha}^{(2)}} \qquad \textbf{(5)}$$

Similarly can be calculated the weight change in the hidden layer. $w_{\alpha,\beta}^{(1)}$

After calualtion of the derivative of a signal *y(t)* with respect of the weight ijn the hidden layer

RMLP

$$\frac{dy(t)}{dw^{(1)}_{\alpha,b}} = \frac{df(g(t))}{dg(t)} \sum_{i=0}^{K} w_i^{(2)} \frac{df(u_i(t))}{du_i(t)} \left[\sum_{j=1}^{P} w^{(1)}_{i,j+N} \frac{dy(t-P-1+j)}{dw^{(2)}_{\alpha}} \right]$$

and the formula for the weight $w^{(1)}_{\alpha,\beta}$ change in the hidden layer

$$\Delta w^{(1)}_{\alpha,\beta} = -\eta[y(t)-d(t)]\frac{dy(t)}{dw^{(1)}_{\alpha,\beta}} \qquad (6)$$

RMLP

Summary of a learning algorithm

1. Initialize the weight vectors in the hidden layer and output layer
2. Evaluate the neuron state in epoch (t) with input x (1-4)
3. Calculates the values of $\frac{dy(t)}{dw_{\alpha}^{(2)}}$ and $\frac{dy(t)}{dw_{\alpha\beta}^{(1)}}$ for every *αβ*
4. Updates the weights according to 5 i 6
5. Go to step 2 algorithm.

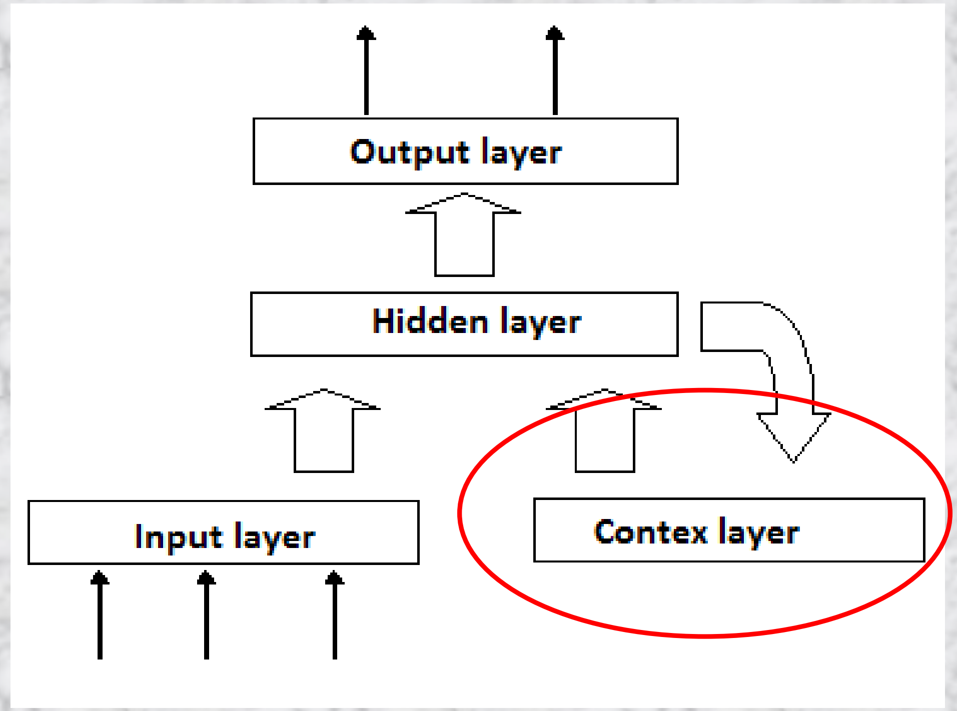
Elman network

Elman Network

An Elman network is an MLP with a single hidden layer and in addition it contains connections from the hidden layer’s neurons to the ***context units***. The context units store the output values from the hidden neurons in a time unit and these values are fed as additional inputs to the hidden neurons in the next time unit.

Elman Network

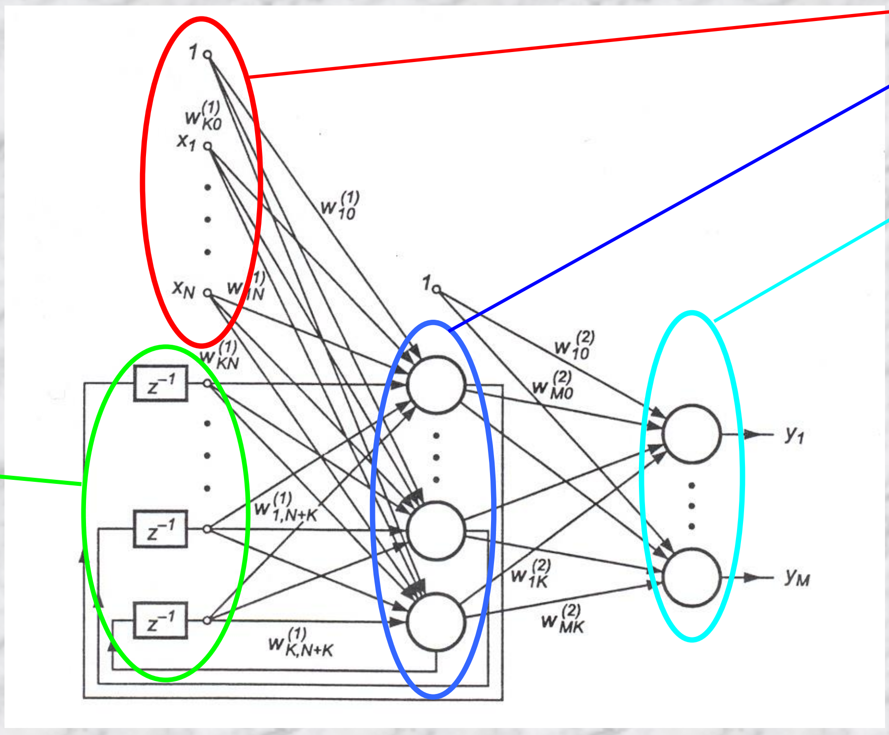
Elman Network – SRN (Simple Recurrent Network) is a simplification of Multi Layer Perceptron



Elman Network

A structure of an Elman Network

Contex layer



N- number of external inputs

K – number of neurons in a hidden layer

M – number of neurons in an output layer

Elman Network

Vector input signal:

$$[1, x_1^{(1)}(t), \dots, x_N^{(1)}(t), x_{N+1}^{(1)}(t), \dots, x_{N+K}^{(1)}(t)]$$

the elements denoted *j*=*N*+1,..,*N*+*K* are from the hidden layer from the previous epoch

$$[1, x_1^{(1)}(t), \dots, x_N^{(1)}(t), y_1^{(1)}(t-1), \dots, y_K^{(1)}(t-1)]$$

Elman Network

Notation

$u_i(t) = \sum_{j=0}^{N+K} w_{ij}^{(1)} x_j(t)$	weighted input signal of the *i*-th neuron in a hidden layer
$v_i = f_1(u_i(t))$	output signal of the *i*-th neuron of a hidden layer
$g_i(t) = \sum_{j=0}^{K} w_{ij}^{(2)} v_j(t)$	weighted input signal of the *i*-th neuron of the output layer
$y_i = f_2(g_i(t))$	output signal of the *i*-th neuron of the output layer

Elman Network

Elman learning algorithm

The network will be learn according to steepest descent algorithm. Similarly to feedforward network a gradient of the cost function will be calculated with respect to every network`s weight. For the Elman network a cost function can be defined by

$$E(t)=\frac{1}{2}\sum_{i=1}^{M}[y_i(t)-d_i(t)]^2=\frac{1}{2}\sum_{i=1}^{M}[e_i(t)]^2$$

Differentiating this function with respect to any output layer weight $w_{\alpha\beta}^{(2)}$ we obtain

Elman Network

$$\nabla^{(2)}_{\alpha\beta}E(t)=\frac{\partial E(t)}{\partial w^{(2)}_{\alpha\beta}}=\sum_{i=1}^{M}[y_i(t)-d_i(t)]\frac{df_2(g_i(t))}{dg_i(t)}\frac{dg_i(t)}{dw^{(2)}_{\alpha\beta}}=$$

$$=\sum_{i=1}^{M}[y_i(t)-d_i(t)]\frac{df_2(g_i(t))}{dg_i(t)}\sum_{j=0}^{K}\frac{d(w^{(2)}_{ij}v_j(t))}{dw^{(2)}_{\alpha\beta}}=$$

$$=\sum_{i=1}^{M}[y_i(t)-d_i(t)]\frac{df_2(g_i(t))}{dg_i(t)}\sum_{j=0}^{K}\left[\frac{d(v_j(t)}{dw^{(2)}_{\alpha\beta}}w^{(2)}_{ij}+\frac{d(w^{(2)}_{ij}(t)}{dw^{(2)}_{\alpha\beta}}v_j(t)\right]$$

Because connections between the hidden layer and output layer are unidirectional

$$\frac{d(v_j(t))}{dw^{(2)}_{\alpha\beta}}=0$$

Elman Network

yields to

$$\nabla^{(2)}_{\alpha\beta}E(t)=\frac{\partial E(t)}{\partial w^{(2)}_{\alpha\beta}}=\sum_{i=1}^{M}[y_i(t)-d_i(t)]\frac{df_2(g_i(t))}{dg_i(t)}\sum_{j=0}^{K}\frac{d(w^{(2)}_{ij}(t)}{dw^{(2)}_{\alpha\beta}}v_j(t)$$

(7)

According to the method of steepest descent the weights' change in output layer are defined by

$$w^{(2)}_{\alpha\beta}(t+1)=w^{(2)}_{\alpha\beta}(t)-\eta\nabla^{(2)}_{\alpha\beta}E(t)$$

(8)

Elman Network

Weights change in the hidden layer is more complicated because of the feedbacks.

$$\nabla^{(1)}_{\alpha\beta}E(t) = \frac{\partial E(t)}{\partial w^{(1)}_{\alpha\beta}} = \sum_{i=1}^{M}[y_i(t) - d_i(t)]\frac{df_2(g_i(t))}{dg_i(t)}\sum_{j=0}^{K}\frac{d(w^{(2)}_{ij}v_j(t))}{dw^{(1)}_{\alpha\beta}} =$$

$$= \sum_{i=1}^{M}[y_i(t) - d_i(t)]\frac{df_2(g_i(t))}{dg_i(t)}\sum_{j=0}^{K}\left[\frac{d(v_j(t)}{dw^{(1)}_{\alpha\beta}}w^{(2)}_{ij}\right] \qquad \textbf{(9)}$$

$$\frac{dv_j(t)}{dw^{(1)}_{\alpha\beta}} = \frac{df_1(u_j)}{du_j}\sum_{m=0}^{N+K}\frac{d(x_m(t)w^{(1)}_{jm})}{dw^{(1)}_{\alpha\beta}} = \frac{df_1(u_j)}{du_j}\left[\delta_{j\alpha}x_\beta + \sum_{m=0}^{N+K}\frac{dx_m(t)}{dw^{(1)}_{\alpha\beta}}w^{(1)}_{jm}\right]$$

Elman Network

and next

$$\frac{dv_j(t)}{dw^{(1)}_{\alpha\beta}} = \frac{df_1(u_j)}{du_j}\left[\delta_{j\alpha}x_\beta + \sum_{m=N+1}^{N+K}\frac{dv_{(m-N)}(t-1)}{dw^{(1)}_{\alpha\beta}}w^{(1)}_{jm}\right] = \tag{10}$$

$$= \frac{df_1(u_j)}{du_j}\left[\delta_{j\alpha}x_\beta + \sum_{m=1}^{K}\frac{dv_m(t-1)}{dw^{(1)}_{\alpha\beta}}w^{(1)}_{j,m+N}\right]$$

The last formula (10) allows to calculate the derivatives of cost function with respect to weights of hidden layer in moment *t*. It is the recurrent formula defining derivative in a moment *t* dependence of a derivative in a moment *t*-1.

Elman Network

Assuming the initial values in a moment *t* = 0

$$\frac{dv_1(0)}{dw_{\alpha\beta}^{(2)}} = \frac{dv_2(0)}{dw_{\alpha\beta}^{(2)}} = \ldots = \frac{dv_K(0)}{dw_{\alpha\beta}^{(2)}} = 0$$

and basing on the steepest descent method the weights' change in the hidden layer is defined by

$$w_{\alpha\beta}^{(1)}(t+1) = w_{\alpha\beta}^{(1)}(t) - \eta \nabla_{\alpha\beta}^{(1)} E(t) \qquad \textbf{(11)}$$

Elman Network

Elman training algorithm

```
1. Initialize the weight vector with random values in , the learning rate ,
   the repetitions counter () and the epochs counter (). Initialize the
   context nodes at 0.5.
2.Let the network’s weight vector in the beginning of epoch
  2.1 Start of epoch. Store the current values of the weight vector
  2.2 For n=1,2,...,N
      2.2.1 Select the training example ($x^n$, $t^n$)and apply the error
        backpropagation in order to compute the partial derivatives $\eta \frac{\partial E^n}{\partial w_i}$
      2.2.2 Update the weights $w_i(k+1) = w_i(k) - \eta \frac{\partial E^n}{\partial w_i}$
      2.2.3 Copy the hidden nodes’ values to the context units.
  2.3 End of epoch k. Termination check. If true, terminate.
3. k=k+1.  Go to step 2.
```