Neural Networks
A Tuturial

Thomas Hellström
Department of Computing Science
Umeå University, Sweden
Department of Mathematics and Physics

Mälardalen University Västeras, Sweden

March 6, 1998

Contents

1 Introduction **3**

1.1 Two viewpoints on Neural networks 3

1.2 Computing elements 3

1.3 The Perceptron 6

2 Multilayer Networks **9**

2.1 Learning in Multilayer Networks 9

2.1.1 The Back-Propagation algorithm 10

2.1.2 Non statistical aspects on learning: 13

2.2 Neural networks as classifiers 13

3 Radial basis networks **15**

3.1 The activation function 15

3.2 Architecture 17

3.3 Learning in an RBF network 19

3.4 Applying an RBF network 19

3.5 Relations to a fuzzy rule bases 20

3.6 Remarks 20

4 Unsupervised learning **21**

4.1 Competitive Learning 22

4.1.1 Example 23

4.1.2 Other Similarity measures 24

4.1.3 Result of learning 25

4.1.4 The learning rate η 25

4.1.5 Deficiencies of Competitive Learning 25
4.2 Kohonen self organizing networks 26
4.2.1 Biological background 26
4.2.2 The architecture of the SOFM 27
4.2.3 The SOFM algorithm 28
4.2.4 SOFM after successful training 29

5 Learning Vector Quantization **31**
5.1 The Phonetic Typewriter 32
5.2 Other applications 33

6 Statistical Inference **35**
6.1 Bias and Variance 35
6.2 Weak and strong modelling 37
6.3 Overfitting and Underfitting 38
6.4 Overtraining 38
6.5 Measuring Generalization ability 38
6.5.1 Test-set validation 39
6.5.2 Cross-Validation 40
6.5.3 Algebraic estimates 40
6.6 Controlling Model complexity 41
6.6.1 Architecture selection in Neural Networks 41

[1]thomash@cs.umu.se

Chapter 1

Introduction

This tutorial is an introduction to Neural Networks The first three chapters covers superwised learning techniques; feedforward networks, perceptrons and multilayer architectures. Unsupervised networks are covered in chapter 4 with competetive learning algorithms and Kohonen nets. Learning Vector Quantization is described in chapter 5. Chapter 6 deals with statistical inference and issues such as bias/variance, overtraining and generalization capability.

1.1 Two viewpoints on Neural networks

A neural network is conceptually composed of a number of simple computing elements connected by links. Each link has a numerical weight associated with it. From a biological and AI point of view, the weights are the primary means of storing information in the neural network, and learning is a process where the weights are being updated to represent new knowledge. From a statistical point of view, the weights are free parameters in a complex non linear function. The correct values for these weights are found by regression.

1.2 Computing elements

The most common computing element have the following form and function:

The computing element, or **node** receives signals from other nodes or from network inputs, computes a new **activation level** that is sent along each of the output links. The computation is divided in two parts: a weighted sum in_i of the inputs:

$$in_i = \sum_j W_{j,i} a_j \tag{1.1}$$

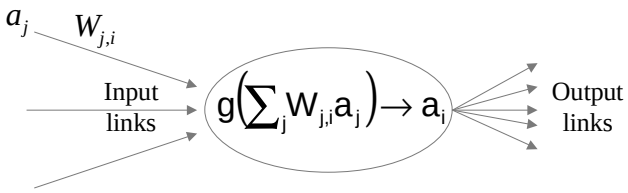
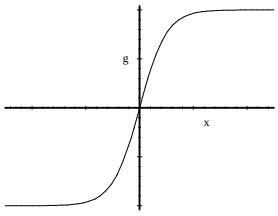


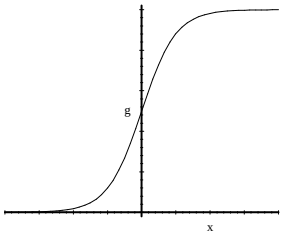
Figure 1.1:

and the **activation function** g. g can be either linear or non linear. The most common choises are:

- Step function $step_t(x) = \left\{ \begin{array}{l} 1, x \geq t \\ 0, x < t \end{array} \right\}$
- Sign function $sign_t(x) = \left\{ \begin{array}{l} +1, x \geq 0 \\ -1, x < 0 \end{array} \right\}$
- Sigmoid function $sigmoid(x) = \frac{1}{1+e^{-x}}$



- Tanh function $g(x) = \tanh(x)$



Of particular interest in early research on neural networks was the step function. .We will start by looking at what functions can be represented by a node with a step function as activiation function. It's convenient to replace the threshold t with an extra input weight $W_{0,i} = t$ connected to a fictive input $a_0 = -1$. In this way we get the following expression for the activation level a_i at unit i:

$$a_i = step_t\left(\sum_{j=1}^{N} W_j a_j\right) = step_0\left(\sum_{j=0}^{N} W_j a_j\right) \tag{1.2}$$

The step t can then be handled in the same way as the other weigths when adapting the behaviour of the node. Examples of boolean functions that can be represented by this simple computing element are shown in figure 1.2,

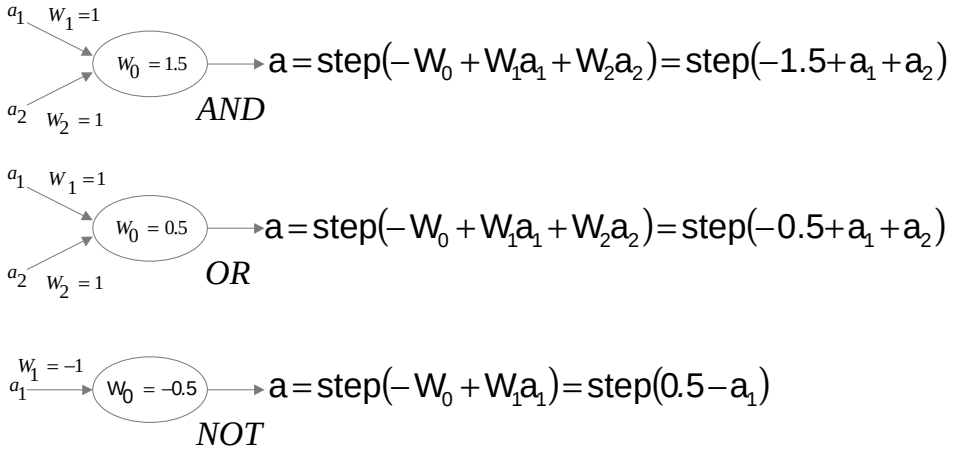


Figure 1.2:

It is however also easy to find boolean function that **can not** be represented by the same computing element. The exclusive or function XOR is such asn example. The XOR function has the following truth table:

a_1	a_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

The corresponding equations for the node activation function are

$$\left\{\begin{array}{l} -w_0 + 0\cdot w_1 + 0\cdot w_2 \leq 0 \\ -w_0 + 0\cdot w_1 + 1\cdot w_2 > 0 \\ -w_0 + 1\cdot w_1 + 0\cdot w_2 > 0 \\ -w_0 + 1\cdot w_1 + 1\cdot w_2 \leq 0 \end{array}\right\} \Leftrightarrow \left\{\begin{array}{c} -w_0 \leq 0 \\ -w_0 + w_2 > 0 \\ -w_0 + w_1 > 0 \\ -w_0 + w_1 + w_2 \leq 0 \end{array}\right\} \Leftrightarrow \left\{\begin{array}{c} w_0 \geq 0 \\ w_2 > w_0 \\ w_1 > w_0 \\ w_1 + w_2 \leq w_0 \end{array}\right\}$$

This is clearly a contradiction since we get $w_0 \geq 0$, $w_2 > w_0$, $w_1 > w_0$ and yet the sum $w_1 + w_2 \leq w_0$.Unfortunanately this example is not an exception. On the contrary it

can be shown that a majority of boolean functions can not be represented by a single perceptron. The situation can be formalized by the concept of *linear separability* as illustrated in the figure 1.3.

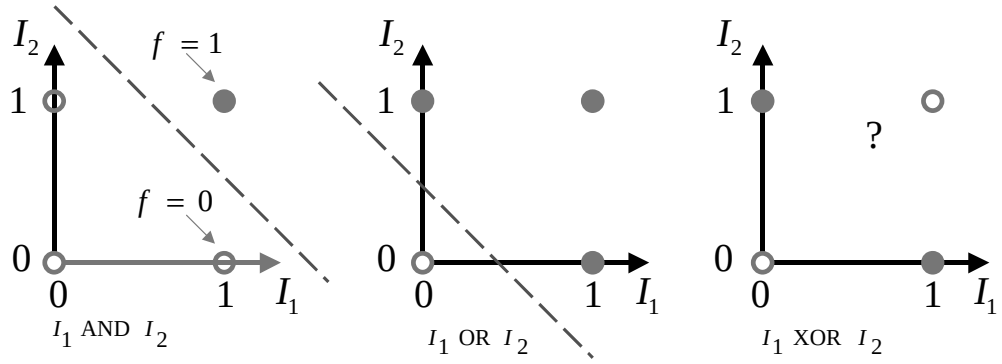


Figure 1.3:

In the general case, a function $f(\mathbf{X}) : R^N \rightarrow (0,1)$ is *linearelly separable* if there exists an $n-1-$dimensional hyper plane such that all points $\mathbf{X}$ where $f(\mathbf{X}) = 0$ are on the same side of the hyper plane.

1.3 The Perceptron

Neural networks were studied in the late 1950s under the name **perceptrons.** A perceptrons is built from the the basic computational elements presented in the previous section. In figure 1.4 a perceptron with four inputs and three outputs is shown. The activation function g for the ouput nodes O_i is the step function previously described.

Even if perceptrons only can represent linearily separable functions, they achievde a great deal of interest when Rosenblatt proposed them in 1960. His **perceptron learing rule** was shown to converge to a set of weights that corectly represent any data set, as long as the data were linearily separable.

Given a set of K examples $\{(\mathbf{I}_i, \mathbf{T}_i), i = 1, K\}$, with $\mathbf{I}_p = (I_{p1}, ..., I_{pn})$ and $\mathbf{T}_p = (T_{p1}, ..., T_{pm})$, the learning rule iteratively updates the weights $w_{j,i}$ for each example p according to

$$W_{j,i} = W_{j,i} + \eta \; a_j \; (T_{p,i} - O_{p,i}) \tag{1.3}$$

where η is the *learning rate* that stabilizes the algorithm steps.

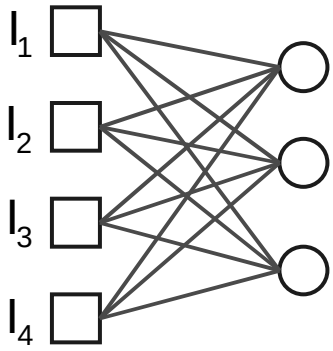


Figure 1.4:

Chapter 2

Multilayer Networks

The limitations with single layer networks are to some extent overcome by the introduction of multi-layer networks. In figure 2.1 a network with one additional *hidden* layer is shown. The activation functions for the hidden nodes are normally chosen as either the sigmoid or the tanh function. The output layer often has linear activation functions if the network is used for function approximation, and sigmoids if the network is used for classification purposes.

A two hidden layer feedforward network has been proved ([Cyb88]) to be a universal approximator, capable of approximating any continuous function. The learning algorithms will however be much more complex then in the case with single layer networks.

2.1 Learning in Multilayer Networks

From a statistical point of view a Feed Forward Neural network is a vector valued function $\mathbf{O}[\mathbf{W}](\mathbf{I})$ of an input vector $\mathbf{I}$ and a weight matrix $\mathbf{W}$.

The Learning task can be defined as:

Given a set of K examples $\{(\mathbf{I}_i, \mathbf{T}_i), i = 1, K\}$, find values for the weights $\mathbf{W}$ such that $\mathbf{O}[\mathbf{W}](\mathbf{I}_p) = \mathbf{T}_p$. Since exact match neither can nor should be achieved for all examples p, we define an error function that we want to minimize. For each example $p = (\mathbf{I}_p, \mathbf{T}_p)$ with $\mathbf{I}_p = (I_{p1}, ..., I_{pn})$ and $\mathbf{T}_p = (T_{p1}, ..., T_{pm})$, the error at output j is defined by:

$$\delta_{p,j} = T_{pj} - O_{pj} \tag{2.1}$$

The total error E_p for example p is normally computed as the individual $\delta_{p,j}$, squared and summed over all outputs j, as shown by

$$E_p = \frac{1}{2}\sum_{j=1}^{M}\delta_{p,j}^2 \tag{2.2}$$

We define the mean squared error MSE as then mean value of E_p computed over all examples , as shown by:

$$MSE = \frac{1}{K}\sum_{p=1}^{K}E_p \tag{2.3}$$

MSE is a function of the free parameters $\mathbf{W}$ that controls the network output values $\mathbf{O}$. For a given training set of examples, MSE represents a measure of training set learning performance. The objective of the learning process is to adjust the values of the free parameters $\mathbf{W}$ so as to minimize the MSE. The well known Back-Propagation algorithm is most often used even if numerical methods such as Gauss-Newton and Conjugate-Gradient methods often does the job much faster.

From a learning perspective, the Back-Propagation algorithm however is often more appealing since it reflects the incremental learning where knowledge gradually improves the system behavior. The Back-Propagation algorithm is also a major step towards parallelizable and biologically plausible learning mechanisms.

2.1.1 The Back-Propagation algorithm

We will derive the back propagation algorithm for the case with one hidden layer as shown in figure 2.1. The equations derived for the hidden layer turns out to look the same even if more hidden layers are added to the network.

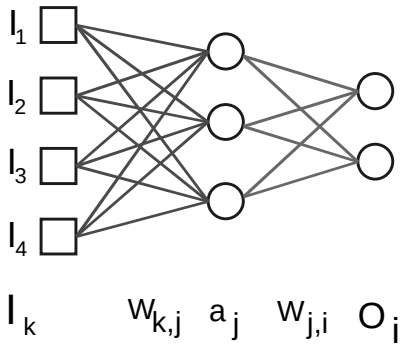


Figure 2.1: MLP with one hidden layer

The algorithm works in a pattern by pattern basis where each example is presented to the network in turn. The weights are updated to compensate for the error in

network output. The algorithm can be derived as a **gradient descent** in weight space. The gradient is computed on the function me want to minimize, i.e. E_p as defined in 2.2. Leaving out the pattern index p we get the following expression:

$$E(\mathbf{W}) = \frac{1}{2}\sum_i \left(T_i - g\left(\sum_j w_{j,i}a_j\right)\right)^2 \tag{2.4}$$

substituting $g\left(\sum_k w_{k,j}I_k\right)$ for the hidden unit activation a_j we get

$$E(\mathbf{W}) = \frac{1}{2}\sum_i \left(T_i - g\left(\sum_j w_{j,i}g\left(\sum_k w_{k,j}I_k\right)\right)\right)^2 \tag{2.5}$$

We compute the partial derivatives $\frac{\partial E}{\partial W_{j,i}}$ with respect to the weights $w_{j,i}$ connecting the hidden node j to the output node i. This entity tells how much the error would change if the weight $w_{j,i}$ was increased by a small amount. We get the following expression:

$$\frac{\partial E}{\partial W_{j,i}} = -a_j(T_i - O_i)g'\left(\sum_j w_{j,i}a_j\right) = -a_j(T_i - O_i)g'\left(in_i\right) = -a_j\Delta_i \tag{2.6}$$

where in_i is the input to the activation function in output node i. I.e:

$$in_i = \sum_j w_{j,i}a_j \tag{2.7}$$

and Δ_i is defined as the error in node i times the derivative computed in the same point. I.e.

$$\Delta_i = g'\left(in_i\right)\left(T_i - O_i\right). \tag{2.8}$$

The derivation of the partial derivatives with respect to the weights $w_{k,j}$ connecting the input k to the hidden layer node j yields a similar result even if the derivation of the equations are more complex. We obtain

$$\frac{\partial E}{\partial W_{k,j}} = -I_k\Delta_j \tag{2.9}$$

where

$$in_j = \sum_k w_{k,j}I_k \tag{2.10}$$

and

$$\Delta_j = g'(in_j) \sum_i W_{j,i} \Delta_i \tag{2.11}$$

The obtain the update rules for the minimization of E, we should take steps opposite to the gradient. Adding a learning rate η we get for the output layer weights

$$W_{j,i} = W_{j,i} + \eta \ a_j \ \Delta_i \tag{2.12}$$

and for the hidden layer weights

$$W_{k,j} = W_{k,j} + \eta I_k \Delta_j \tag{2.13}$$

The updates rules are implemented in a recursive algorithm as follows:

Given a set of examples $p = (\mathbf{I}_p, \mathbf{T}_p)$ with $\mathbf{I}_p = (I_{p1}, ..., I_{pn})$ and $\mathbf{T}_p = (T_{p1}, ..., T_{pm})$, the error at output j is defined by:

repeat for each example e

$(O_1, ..., O_m) = \mathbf{O}[\mathbf{W}]\ (\mathbf{I}_e)$ % compute the network output vector

for each output node i

$err_i = T_{e,i} - O_i$ % compute the error at each output node

$in_i = \sum_j w_{j,i} a_j$

$\Delta_i = g'(in_i)\ err_i$

$W_{j,i} = W_{j,i} + \eta \ a_j \ \Delta_i$ % update the weights leading to output layer

end

for each hidden node j:

$in_j = \sum_k w_{k,j} I_{e,k}$

% compute the error at each node by *propagating*

% the Δ_i from the output layer:

$\Delta_j = g'(in_j) \sum_i W_{j,i} \Delta_i$

$W_{k,j} = W_{k,j} + \eta \ I_{e,k} \ \Delta_j$ % weights leading to hidden layer

end

until converge

Each pass through all the examples in the training set is denoted an **epoch**.

As we can see in the update rules **??** and 2.12 a weight can be calculated by the unit to which it is attached, using only local information. This makes backpropagation

plausible as a biological learning mechanism. Furthermore it is appealing when implementing learning algorithms on parallel computer architectures.

Like all gradient descent based algorithms, back-propagation suffers from slow convergence and problems with local minima in the error surface. Numerous improvements to the basic algorithm have been proposed and made it by far the most commonly used learning method for neural networks.

2.1.2 Non statistical aspects on learning:

- We don't want performance on the training data !

  Minimization of the performance on the training data set often leads to "over training" where the computed model is to tightly connected to the training data. The generalization performance (how well it predicts unseen data) will therefor not be maximized by minimizing the mean squared error (2.3) on the training set data.

- Is the error function relevant (MSE)

  The mean squared error is not the only possible entity to minimize. Depending on the application, it may be more relevant to minimize the *maximum* error, the *median* error or some other measure of prediction quality. Also, in the case of multiple outputs, its not obvious that the error in all outputs are equally important to minimize.

- What about the learning process?

  Viewing the learning as a pure parameter estimation problem neglects the system behavior *during* the training process. It may in some cases be the whole purpose of a system to study the learning process as new data enters into the system. In other cases, such as certain real time systems, the behavior during training is crucial since useful predictions must be available even during the training process.

2.2 Neural networks as classifiers

Neural networks are also often used for classification problems where p-dimensional input vectors are to be mapped to an m-dimensional output vector with exactly *one* element set to *one* and the rest to *zero*. The position of the set element represents the class associated with the corresponding input vector. In this way, examples of input vectors and corresponding output vectors can be used to train the network to assign a class label to unknown input vectors. It is typically implemented with a multilayer perceptron with p inputs and m outputs. The output layer often uses the sigmoid

as activation function producing an m-dimensional output vector with values in the closed interval $[0, 1]$. It has been shown (see e.g. [**?**]) that the output from a successfully trained network is an asymptotic approximation of the *a posteriori* class probabilities.. This means that the optimum classification of an input vector is the position in the output vector that has the highest value. It is common to offset the target output vectors by a small amount ε such that the zero-elements are represented by ε and the one-element by $1 - \varepsilon$. The reason for this is to avoid saturation of the activation function in the training process. The output from the trained network can in this case not directly be interpreted as probabilities.. A linear transformation $[\varepsilon, 1 - \varepsilon] \rightarrow [0, 1]$ must first be applied. The position in the output vector that has the highest value is however still the optimum classification for a given input vector.

Chapter 3

Radial basis networks

A new and powerful type of feedforward artificial neural network is the radial basis function (RBF) network. It differs from the Multi Layer Perceptron (MLP) primarily in the activation function and how it is applied on the inputs to a node in the network.

3.1 The activation function

In a "normal" feed-forward neural network, the activation value (i.e. the "output") a_i from a node i is the activation function g applied on a weighted sum of the inputs $I_j, j = 1, n$ to the node (figure 3.1) . I.e.:

$$a_i = g\left(\sum_{j=1}^{n} W_{j,i} I_j\right).$$

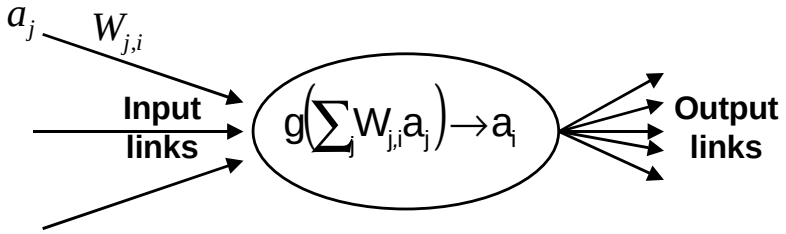


Figure 3.1: Node in a MLP

$W_{j,i}$ is the weight connecting input j to node i. g is typically chosen as the sigmoid defined as (figure 3.2)

$$g(x) = \frac{1}{1 + e^{-x}}. \tag{3.1}$$

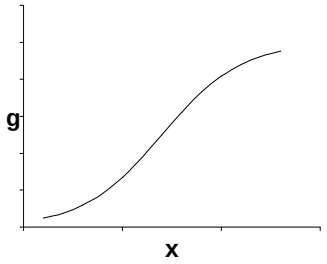


Figure 3.2: Sigmoid activation function

In the case of MLP, g contains no trainable parameters.

In a Radial basis network, each node has its own activation function denoted R (figure 3.3) The idea is that each node should react on different parts of the input space.

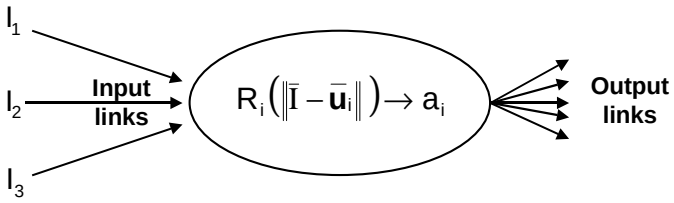


Figure 3.3: Radial basis node

The activation value a_i is therefor defined as a function of the distance between the input vector $\mathbf{I} = (I_i, ..., I_n)$ and a center point $\mathbf{u} = (u_{i1}, ..., u_{in})$:

$$a_i = R(\|\mathbf{I} - \mathbf{u}_i\|). \tag{3.2}$$

The activation function R should have its maximum at zero and decay symmetrically around zero. Typical choices are the gaussian (figure 3.4)

$$R(\|\mathbf{I} - \mathbf{u}_i\|) = \exp\left(-\|\mathbf{I} - \mathbf{u}_i\|^2 / 2\sigma_i^2\right). \tag{3.3}$$

and the logistic function

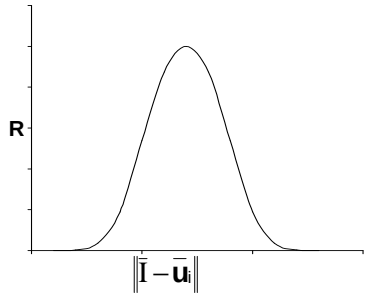


Figure 3.4: Gaussian activation function

$$R(\|\mathbf{I} - \mathbf{u}_i\|) = \frac{1}{1 + \exp\left(\|\mathbf{I} - \mathbf{u}_i\|^2 / 2\sigma_i^2\right)}. \tag{3.4}$$

Each node is thus specified by a center point $\mathbf{u}_i = (u_1, ..., u_n)$ and a standard deviation σ_i. Note that the inputs are connected directly to the node (in the case of MLP, the inputs are weighted before entering the node).

- The activation function R operates on the distance between the input vector $\mathbf{I}$ and the center point $\mathbf{u}$.
- R has its maxima for zero input and approaches zero at infinity. That is why the functions are called radial; their value is monotonically decreasing with the distance from a center point.
- The *receptive field* is the part of the input space that affects the output; a hyper sphere around $\mathbf{u}$ and a width determined by σ.
- The nodes are called localized receptive fields, locally tuned processing units or potential functions.

3.2 Architecture

Figure 3.5 shows a small RBF network. The hidden nodes have radial-basis activation functions as described above and the output layer nodes have weighted sums as activation functions:

$$a_i = \sum_{j=1}^{m} W_{j,i} a_j$$

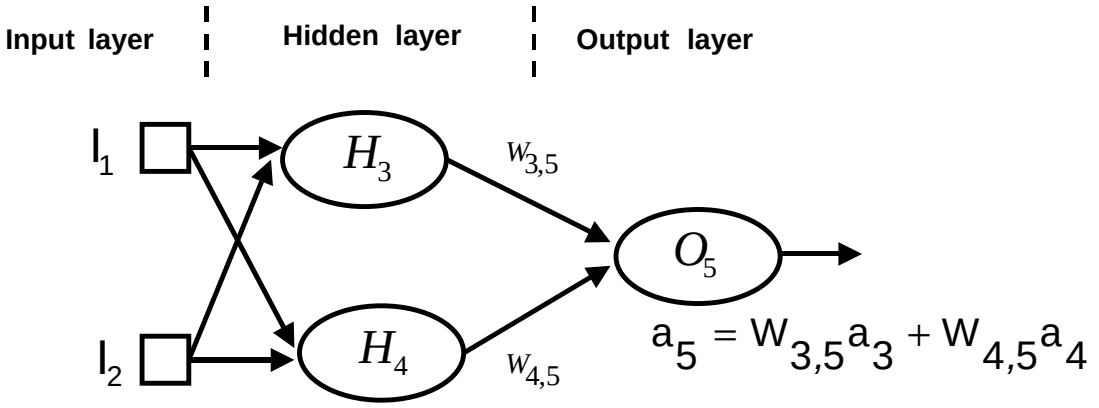


Figure 3.5: Radial basis network

where m is the number of hidden nodes connected to the output layer and $W_{j,i}$ is the weight connecting hidden unit j to output node i.

The number of hidden nodes can be determined in a number of ways:

- Each input vector can get its own node. The center is set equal to the input vector. The standard deviation is estimated statistically. The weights to the output layer are then estimated by ordinary linear regression. This method normally yields a vastly over parameterized model since the size of the model increases linearly with the number of input vectors.
- Another alternative is to cluster the input vectors and set the center points in the hidden layer nodes to the cluster centers. The standard deviation is also estimated from the clustering process.

Many other layouts have also been proposed for RBF networks. Depending on the chosen layout, some or all of the following parameters are subject to training:

- Center points $\mathbf{u}$ and widths σ for all the hidden nodes
- Weights between the hidden nodes and the output nodes

Weight initialization can also be done in several way:

- Random values
- Center points are set equal to the input vectors
- Center points are set equal to cluster means
- Standard deviations are estimated from the spread within clusters or from the original input data.

3.3 Learning in an RBF network

The learning is done in the same manner as in the case of an MLP. An RBF neural network is a vector valued function $\mathbf{O}[\mathbf{W}](\mathbf{I})$ of an input vector $\mathbf{I}$ and a weight matrix $\mathbf{W}$. The number and types of weights depend on the chosen architecture as described in the previous section.

If the center points and the standard deviations are not subject to training, as described in the previous section, the learning task reduces to a simple linear regression problem This can be solved in a fraction of the time for the corresponding training of an MLP.

3.4 Applying an RBF network

After training the network is ready to be used on previously unseen data. A new input vector will probably not match any of the weight vectors (i.e. center points in the activation functions) exactly. It will more likely partially match several weight vectors and therefore activate the corresponding hidden nodes to varying degrees. The neuron is said to fire when an input vector falls within the receptive field. Normally many neurons are fired and their outputs are weighted together in the output nodes. In this way the network produces an interpolation between the centers of the fired neurons. Figure3.6 shows an example of a trained radial basis network. The individual activation functions $R_1, R_2, ...$ are displayed as contour curves as a function of the two inputs I_1 and I_2. The computed model will be local in the sense that it has little to say about what the underlying function looks like outside the areas where there are training data. Each activation function just represents a local model of the training data that falls inside the receptive field of the corresponding neuron. An ordinary MLP on the other hand produces a global approximation that is believed to be valid even outside the areas where it was actually trained.

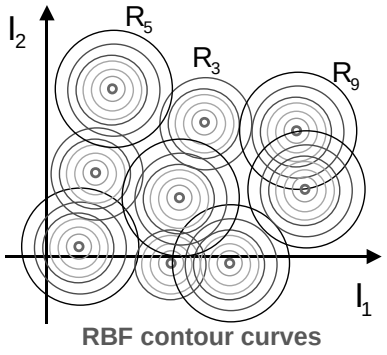


Figure 3.6: A trained Radial Basis Network

3.5 Relations to a fuzzy rule bases

A RBF network can be made equivalent to a first order Sugeno fuzzy inference system (FIS). The aggregation method where the consequent parts of the fuzzy rule base is combined corresponds to the processing performed by the output nodes in the RBF network. It is in both cases implemented as a weighted sum. The receptive fields in the hidden nodes correspond to Gaussian membership functions. If the fuzzy AND is implemented as multiplication, the one-dimensional membership functions will become multidimensional Gaussian functions and correspond to a Gaussian activation function in the RBF network.

3.6 Remarks

We can make the following conclusive remarks on RBF networks:

- In general an RBF network trains orders of magnitude faster then ordinary feed forward networks.
- It has been shown in Bianchini et al. (1995) that whenever the input space is separable by hyperspheres, the error function minimized in the training has no local minima. This is a great advantage compared to the ordinary MLP.
- An RBF network is normally slower to use due to the larger number of nodes.

Chapter 4

Unsupervised learning

When the data available for learning doesn't contain any correct answers or teacher's instructions, only the input vectors can be used for learning..Such an approach is commonly referred to as unsupervised learning. Whereas supervised learning usually involves some sort of function interpolation we are now instead trying to find features or regularities in the data. The goal is often to identify clusters that describe important aspects of the investigated data set. The unsupervised methodology is used in many applications:

- Pattern recognition: Automated cytology, Fingerprint recognition, Optical Character Recognition (OCR).
- Feature extraction: Finding out what parts of a large input vector is relevant for the problem.
- Pre processing of data: The clustering of data may serve as a pre processing stage for e.g. other types of neural networks.
- Data compression: Compression of images.

Many paradigms for unsupervised learning has been developed:

- Competitive learning
- Kohonen self-organizing feature map
- Boltzman machines
- Principal Component Analysis

We will describe the first two paradigms in the following sections.

4.1 Competitive Learning

The Competitive Learning network is a popular scheme to achieve unsupervised data clustering. Figure 4.1 shows a small example of a network for competitive learning. Each input is a 3-dimensional vector $\mathbf{x} = (x_1, x_2, x_3)$, and we expect the input vectors to be clustered in four clusters, hence the four output nodes $a_1, ...a_4$. Note that the data contain no output values. The analysis is based on the input vectors only.

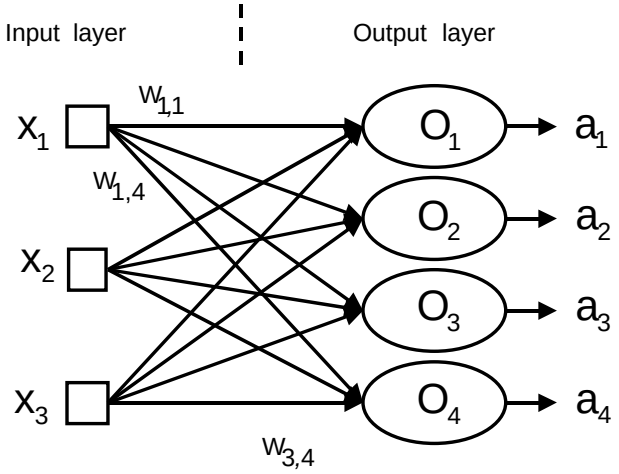


Figure 4.1: Network for Competitive learning

The activation level a_j (i.e. the output) from node j is normally defined as the Euclidean distance between the weight vector $\mathbf{w}_j = (w_{1,j}, w_{2,j}, w_{3,j})$ and the input vector $\mathbf{x}$

$$a_j = \sqrt{\sum_{i=1}^{n} (x_i w_{i,j})^2} = \|\mathbf{x} - \mathbf{w}_j\| \tag{4.1}$$

where $n = 3$ since the $\mathbf{x}$ vectors are 3-dimensional in the example.

In competitive learning each pattern is presented to the network one by one. The activation levels are computed and the most successful output is selected for updating. In this case the winner is the neuron that has the smallest activation level, i.e. the one with a weight vector $\mathbf{w}$ most similar to the input vector $\mathbf{x}$. This node is denoted k and the weight vector leading to it then adjusted further towards the input vector the following learning rule:

$$\mathbf{w}_k(t+1) = \mathbf{w}_k(t) + \eta\,(\mathbf{x}(t) - \mathbf{w}_k(t)) \tag{4.2}$$

where η is a learning rate parameter.

The weight vectors will eventually become cluster centers for all the inputs vectors where the associated output node was selected as winner.

The individual neurons learn to specialize on sets of similar patterns. They are therefor often denoted *feature detectors*. The cluster centers are often denoted *templates*, *reference vectors* or *codebook vectors*.

When the trained network is presented a new unclassified input, the winning output is taken as estimate or prediction of the classification of the input. If we want to decide whether two input vectors are equal we simply let the network classify them and see if they fall into the same cluster.

4.1.1 Example

Let us look at the example of a network for Competitive learning in figure 4.2.

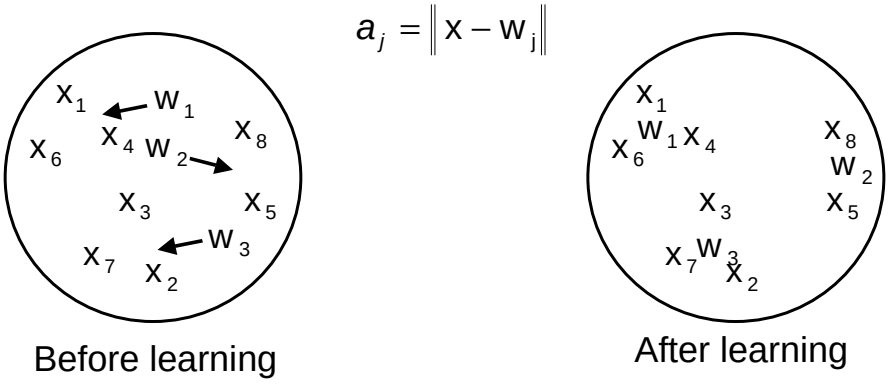


Figure 4.2: Illustration of Competitive learning

The network looks like the one in figure 4.1 but with 2-dimensional inputs I_1, I_2 and three output nodes $a_1, a_2 a_3$. The 8 inputs vectors are denoted $x_1, ..., x_8$ and are plotted as points in the 2-dimensional plane. We expect the input vectors to be clustered in 3 clusters, hence the 3 output nodes with associated weight vectors $\mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3$. The tree weight vectors are also plotted in the same plane. The arrows indicate how the eights are moved as result of the learning. The final result with the weight vectors as cluster centers are shown to the right in the figure.

1. One of the input vector $\mathbf{x}$ is applied as input to the network.
2. The activation levels are computed for all output nodes, i.e. for the three weight vectors. The activation level is defined as the Euclidean distance between the input vector $\mathbf{x}$ and the weight vector $\mathbf{w}$ for each node.

3. The weight for the output with the smallest activation level is selected for updating. I.e. the weight vector $\mathbf{w}$ that is closest to the input vector $\mathbf{x}$.

4. $\mathbf{w}$ is adjusted further towards the input vector by the learning rule 4.2.

5. The process is repeated from step 1 for all input vectors in one epoch.

6. The process is repeated with multiple epochs until all weights $\mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3$ have stabilized.

The weight vector will become cluster centers for all the inputs vector where the output node was selected as winner.

The input data is now divided into disjoint clusters such that the similarities between inputs in the same cluster are larger then those in different clusters. If we apply a totally new input vector $\mathbf{x}$ to the network, the node with a weight vector closest to $\mathbf{x}$ will have give the smallest activation level. $\mathbf{x}$ is then classified as belonging to the corresponding cluster.

4.1.2 Other Similarity measures

The concept of similarity is used to compute each neurons activation level when an vector is input. We have been using the Euclidean distance 4.1 so far, but other metrics can also be used. The scalar product between the input $\mathbf{x}$ and the weight vector $\mathbf{w}$ is often used as measure of dissimilarity. The activation level a_j for node j is normally defined as the Euclidean distance between the weight vector $\mathbf{w}_j = (w_{1,j}, w_{2,j}, w_{3,j})$ and the input vector $\mathbf{x} = (x_1, ..., x)$

$$a_j = \sum_{i=1}^{n} x_i w_{i,j} = \|\mathbf{x}\| * \|\mathbf{w}_j\| * \cos(\varphi_{w_j,x}) \tag{4.3}$$

As we can se from the definition, the scalar product for normalized weight and input vectors equals the angle between the two vectors. It can therefor be regarded as a measure of dissimilarity and the selection of winner in the competitive learning should be the neuron with largest activation value instead of the one with the smallest activation in the case of using the Euclidean distance. The normalization of the weight vector is taken care of by a modified learning rule

$$\mathbf{v}_k(t+1) = \mathbf{w}_k(t) + \eta\left(\mathbf{x}(t) - \mathbf{w}_k(t)\right) \tag{4.4}$$

$$\mathbf{w}_k(t+1) = \mathbf{v}_k(t+1) \;/\; \|\mathbf{v}_k(t+1)\| \tag{4.5}$$

and the normalization of the input vectors is taken care of before the learning phase.

4.1.3 Result of learning

We have claimed that the algorithms for the competitive learning divided the data into disjoint clusters such that the similarities between points in the same cluster are larger then those in different clusters. If we use the Euclidean distance as similarity measure we can prove that the competitive learning rule is a version of gradient descent applied on the squared error

$$E = \sum_p \left\| \mathbf{x}_p - \mathbf{w}_{i(x_p)} \right\|^2 \tag{4.6}$$

where $i(x_p)$ is the winning node for input vector $\mathbf{x}_p$. E measures the mean distance between each input vectors $\mathbf{x}$ and the cluster center $\mathbf{w}$ that $\mathbf{x}$ is classified as belonging to.

The algorithm for the competitive learning is an on-line method in the sense that the algorithm processes each data item in sequence as if it really was measured in real time, on-line. There are also off-line methods that operate on all data at once and perform the same minimization. An example is the K-means clustering algorithm.

4.1.4 The learning rate η

The learning rate is normally set dynamically during the learning process. A large η doesn't guarantee stability, the centers will move around. On the other hand a small η makes the algorithm less sensitive and the centers will not get updated when new data are presented. This trade-of between sensitivity and stability is called the *stability-plasticity dilemma* and is common for all learning intelligent systems. Adaptive resonance theory (ART) was introduced by Grossberg and proposes a solution to the dilemma in the network context.

4.1.5 Deficiencies of Competitive Learning

- Information is lost since an input vector is mapped onto one single winner node. It would be better if many nodes were involved.
- Waste of hidden nodes. Some of the nodes may never win if the initially set weights are to far away from all input vectors. In this way these nodes will never contribute to the learning process. Some techniques to deal with this problem are:
  - Initialize the weights to an existing inputs vector instead of to random values.
  - *Leaky learning*: Update all nodes, but with different learning rates η.

  - *Kohonen feature maps* is a well known network paradigm that updates nodes in a *neighborhood* around the winning node

- Hard to know how many output nodes to use!

- The underlying method of unsupervised learning uses a bottom-up representation and do not use any top-down information to form the clusters. It is not obvious that distinct clusters in the input space are relevant for the task at hand.

4.2 Kohonen self organizing networks

The self-organizing feature-mapping (SOFM) algorithm was developed by Teuvo Kohonen in 1982. It is based on competitive learning where incoming signal patterns of arbitrary dimension are mapped into a one- or two-dimensional discrete map. The updating or learning is done using not only the winning neuron but all neurons in a neighborhood around the winning node. The result of the training is the discrete map where input data has formed clusters. The SOFM algorithm is consequently a form of competitive learning, i.e. unsupervised learning where the data contain no target values. It is used for data clustering and also for pre processing in conjunction with other machine learning techniques.

4.2.1 Biological background

The self organizing artificial networks have been inspired by the studies of biological brains. The artificial networks are furthermore used to model and simulate biological brains in fields such as psychology and linguistics. We will therefor present some biological background for the self organizing artificial networks.

There are partial biological evidence that the brain organize information spatially. It is most easily studied by observing the neuron activity caused by a specific stimuli such as touching of the skin, sounds or light signals. The neuron activity may be studied by techniques such as radioactive tracers and gamma cameras and also magnetoencephalography (MEG) that analyses the small magnetic fields caused by neural responses. The reverse connection has been studied by stimulating a particular site in the brain with small electric currents and observing excitatory and inhibitory effects on certain cognitive abilities such as naming of objects.

In these ways a fairly detailed organizational view of the brain has evolved. The brain seem to contain many kinds of *maps* such that a neural response in a certain location in the map corresponds to a specific quality of stimuli. Examples are:

- The visual areas where some maps deal with line orientation or color perception. The spatial directions in the maps represent different aspects or features of the mapped entity.
- Some maps represent quite abstract qualities such as in the word-processing areas where the neural responses seem to be organized according to categories and semantic properties.
- Another example is the somotopic map, which contain a representation of the skin surface. An identical adjacent map takes care of the muscle control based on the same topology.

4.2.2 The architecture of the SOFM

The basic idea with the SOFM is that not only the winning neuron but all neurons in a neighborhood around the winning node should be affected by updating. The neighborhood concept is introduced by arranging the nodes in either a line or a square. The neighbors to a certain node are then defined to be the adjacent nodes in the square or on the line. The line or the square is denoted *map*.

In figure 4.3 we have 3-dimensional input vectors $\mathbf{x} = (x_1, x_2, x_3)$. All inputs are fully connected to each one of the 16 nodes through 16 3-dimensional weight vectors $\mathbf{w} = (w_1, w_2, w_3)$.

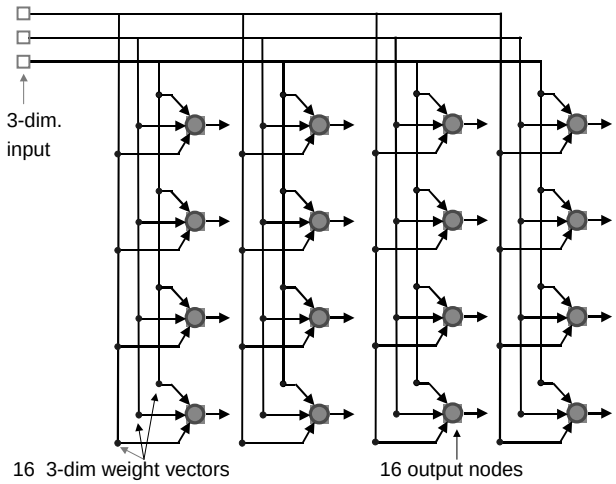


Figure 4.3: Kohonen Self Organizing Map

The activation level (i.e. the output) from each node is the same as in ordinary competitive learning. It may be the distance or the angle between the weight vector for the node and the input vector $\mathbf{x}$.

A neighborhood function Λ_i is also defined. It should map a neuron i onto the set of neurons that are regarded as the neighborhood to neuron i.

One way of doing this is shown in the figure 4.4. The size of the neighborhood is determined by the radius that can take values from 0 up to the full size of the map.

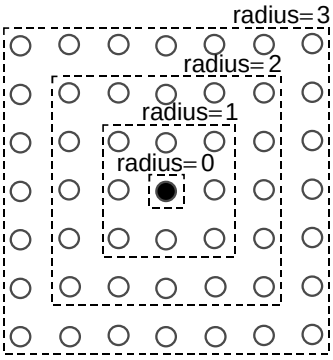


Figure 4.4: Neighbourhoods with varying radius

4.2.3 The SOFM algorithm

A high-level description of the SOFM algorithm is given below where N is the number of neurons in the map.

1. Initialization

   Choose random values for the initial weights vectors w_j. Different values for all $j = 1, 2, N$

2. Sampling

   Draw a sample vector $\mathbf{x}_p$ from the training set of inputs.

3. Similarity Matching:

   Find the winning neuron $i(\mathbf{x}_p) = \arg\min_j \|\mathbf{x}_p - \mathbf{w}_j\|$

4. Updating

   Update all weight vectors in the neighborhood $\Lambda_{i(\mathbf{x}_p)}$:

   $\mathbf{w}_j(t+1) = \mathbf{w}_j(t) + \eta\,(\mathbf{x}_p - \mathbf{w}_j(t))$for all $j \in \Lambda_{i(\mathbf{x}_p)}$

5. Continuation

6. Update the neighborhood function Λ (normally reduce the size) and the learning rate η (normally reduce the magnitude)

7. Continue with step 2 until all weights are stable.

The success of the generated feature map depends on how the learning rate η and neighborhood function Λ are initialized and updated in step 4. There is no theoretical basis for selection of these parameters. The following rules of thumb often provide a useful guide. (Kohonen, 1988b). The learning is assumed to contain two phases:

- The ordering phase. The topological ordering takes place. η should be around 1 during this phase for about 1000 iterations. Λ starts at full radius and decreases linearly with time down to a small value with only a couple of neighboring nodes.

- The convergence phase. Fine tuning the map. η should be on the order of 0.01. Many thousands of iterations. Λ is slowly reduced down to 1 or 0 neighboring neurons.

A lot of variants of the basic algorithm exist. Step 4 where the weights are being updated may for example use a negative feedback to update weight vectors that fall outside the neighborhood, i.e.: $\mathbf{w}_j(t+1) = \mathbf{w}_j(t) - \eta_2 (\mathbf{x}_p - \mathbf{w}_j(t))$for all $j \notin \Lambda_{i(\mathbf{x}_p)}$. This process is called *lateral inhibition*. Nodes close to the winning node are being *positively updated* (*reinforced* or *rewarded*) and nodes farther away from the winner are being *negatively updated* (*extinguished* or *punished*).

4.2.4 SOFM after successful training

What have we achieved when the SOFM algorithm has converged? Let $\mathbf{X}$ denote the continuous input space and let A denote the discrete output space, e.g. the map of neurons arranged in a line or in a square as in figure 4.5.

Let Φ denote a nonlinear transformation from $\mathbf{X}$ to A as shown by

$$\Phi : \mathbf{X} \rightarrow A \tag{4.7}$$

Φ may be viewed as an abstraction of step 3 in the SOFM algorithm: $i(\mathbf{x}_p) = \arg\min_j \|\mathbf{x}_p - \mathbf{w}_j\|$

which finds the best matching or winning neuron in the output space A.

This mapping, called the feature map, is totally defined by the weight vectors w. The weight vector for the winning neuron $i(x)$ may then be viewed as a pointer for the winning neuron back to $\mathbf{X}$. These two mappings are illustrated in figure 4.5.

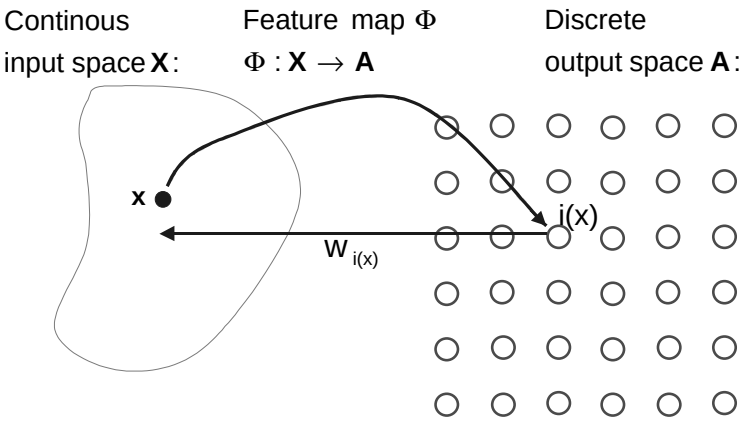


Figure 4.5: Self organizing feature map Φ

- If the input space has higher dimension than the map, we talk about dimensionality reduction and data compression. The described techniques has been applied successfully to image compression.
- The feature map Φ is topologically ordered in the sense that the spatial location of a neuron in the grid corresponds to a particular domain or feature of the input vectors. It can therefor be utilized for similarity detection and also as a preprocessing stage in other supervised algorithms.

Chapter 5

Learning Vector Quantization

This method was developed by Linde (1980) as a tool for image data compression. It was adapted by Kohonen (1986) for pattern classification, i.e.: assigning class labels to input data. The method combines unsupervised clustering and supervised learning and involves two steps:

- Locate cluster centers $\mathbf{w}$ by a clustering method such as SOFM.
- Fine tune the centers $\mathbf{w}$ to minimize the classification error.

Since this is a supervised learning technique, the target value for each input pattern $\mathbf{x}$ must be available to determine if the classification is correct. A high-level algorithm is given here:

1. Initialize the cluster centers $\mathbf{w}$ by a clustering method such as SOFM
2. Label each cluster $\mathbf{w}$ to the median values of each clusters's target values (voting).
3. Select a random input vector $\mathbf{x}$.
4. Find the closest cluster for $\mathbf{x}$: $i(\mathbf{x}) = \arg\min_j \|\mathbf{x} - \mathbf{w}_j\|$
5. If $\mathbf{x}$ is correctly classified (i.e. the target value for $\mathbf{x}$ equals the label of $i(\mathbf{x})$) then
   $$\mathbf{w}_{i(\mathbf{x})}(t+1) = \mathbf{w}_{i(\mathbf{x})}(t) + \eta\left(\mathbf{x} - \mathbf{w}_{i(\mathbf{x})}(t)\right)$$
   else
   $$\mathbf{w}_{i(\mathbf{x})}(t+1) = \mathbf{w}_{i(\mathbf{x})}(t) - \eta\left(\mathbf{x} - \mathbf{w}_{i(\mathbf{x})}(t)\right)$$
6. Decrease the learning rate η (typically from 0.01 down to 0 in 100000 steps) and repeat from step 3 maxiteration times.

The idea with the updating of weights in step 5 is to move the weight vectors away from the decision surface to demarcate the class borders more accurately (Kohonen 1990). In this way the classification performance is increased.

The described algorithm is LVQ1. Kohonen also suggested improved versions of the method, called LVQ2 and LVQ3, that update more then just the winning neuron.

5.1 The Phonetic Typewriter

A famous application of Learning Vector Quantization is Kohonens Phonetic Typewriter. It is an implementation of the Self-Organizing Map for recognition of phonemes in continuous speech (Finnish and Japanese). The input is a 15-components spectral vector from a 256-point FFT. It gives the frequency contents of the original speech signal. About 2000 samples is required to make the system learn a new speaker. The input vectors are clustered with the SOFM algorithm. The clusters are then labelled with phonemes and fine tuned according to the LVQ algorithm. The resulting map is shown in figure 5.1.

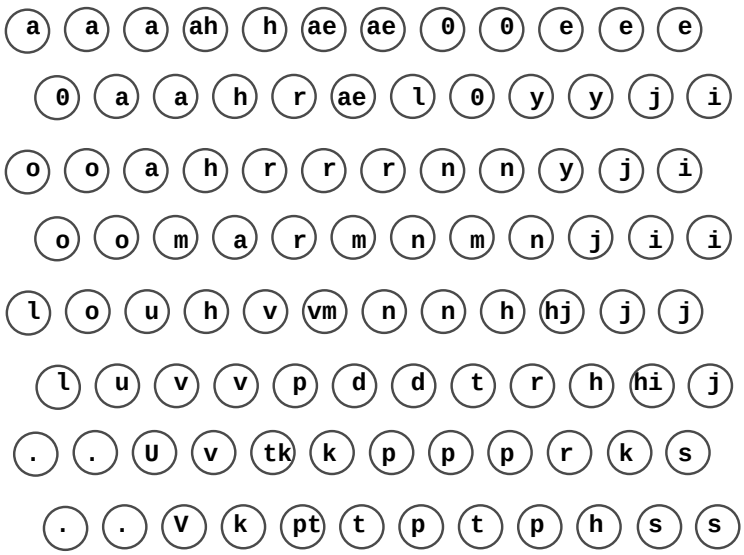


Figure 5.1: Map for the Phonetic Typewriter

Each node in the map is labelled with the classified phoneme. Most nodes give a unique answer, whereas the double labels indicate response to two phonemes. Classification errors less then 10% has been reported. This is claimed to be better then other methods. The method has been implemented in hardware for real time operation (Kohnonen1988).

5.2 Other applications

The self-organizing feature-mapping (SOFM) algorithm and then Learning Vector Quantization algorithm have been used in a variety of applications, including the following:

- Radar classification of sea ice
- Brain modelling
- Control of robot arms
- Speech recognition
- Sentence understanding
- Control of industrial processes
- Automatic synthesis of digital systems
- Optimization problems
- Image compression
- Classification of insect courtship songs

Chapter 6

Statistical Inference

Model-free estimation and non parametric statistical inference deal with the problem of finding a hypothesis function using a set of examples as primary information. The estimators can take many forms such as algebraic or trigonometric polynomials, neural networks, step wise regression etc. Many of these techniques have been shown to share the common property of being *universal approximators*, capable of approximating any continuous function. In the case of neural networks it was first shown by Cybenko [Cyb88]. Note that these are theoretical results which may require a model with arbitrary complexity for convergence. The convergence also assumes an infinite example set. The data must furthermore be free from all noise. Conditions which are seldom fulfilled in real applications. The general limitations of statistical inference are also clearly understood and is formulated as the "bias/variance dilemma"[GBD92].

6.1 Bias and Variance

To estimate an unknown function $f(\mathbf{P})$that has produced a finite set of examples $(\mathbf{P}, \mathbf{T})$ the following general procedure for inductive learning can be applied:

1. Draw a random training set from the set of examples
2. Train one estimator $h_i(\mathbf{P})$ (e.g. a neural network)
3. Evaluate the estimator on a randomly picked test set of examples

Repeat 1-3 a large number of times, producing hypothesis functions $h_1, ..., h_N$.

Now lets concentrate on what we have produced for a specific point $\mathbf{p}$. We have a number of different estimators $h_1(\mathbf{p}), ..., h_N(\mathbf{p})$. Since they all depend on random

selections of training data, they can themselves be viewed as outcomes of a random variable h_P with mean m_P and variance V_P.

The mean squared error for the predictions in the point $\mathbf{p}$ can be written as:

$$E_p = E[(h_P - f(\mathbf{p}))^2] = E[h_P^2 + f(\mathbf{p})^2 - 2h_P f(\mathbf{p})] = E[h_P^2] + f(\mathbf{p})^2 - 2f(\mathbf{p})E[h_P] \tag{6.1}$$

By using the identity $V_p = E[h_p^2] - m_p^2$ we get:

$$E_p = V_p + m_p^2 + f(\mathbf{p})^2 - 2f(\mathbf{p})E[h_P] = V_p + (m_p - f(\mathbf{p}))^2 \tag{6.2}$$

The mean squared prediction error E_p thus is a random variable comprising two components termed *bias* and *variance*:

- Bias: $(m_p - f(\mathbf{p}))^2$. the amount by which the average estimator differs from the true value
- Variance: V_p. The variation among the estimators

The mean m_P can be estimated as $\sum_i h_i(\mathbf{p})/N$ and the variance V_P can be estimated as $\sum_i (h_i(\mathbf{p}) - m_P)^2/(N-1)$.Note that explicit estimation of E_P requires knowledge of the underlying function f. This is seldom if ever the case, otherwise there would be no need to estimate it. Statistical methods to estimate E_P without this knowledge of f exist. For example, the Jackknife, and the Bootstrap techniques[Efr82]. It is however possible to proceed without explicit calculation of E_P by looking at how the estimators complexity affect the distribution for E_P. Depending on the estimators complexity, we get the following extreme cases:

- To low complexity (e.g. a straight line or a neural network with few weights) The computed estimators $h_1(\mathbf{p}), ..., h_N(\mathbf{p})$ will produce roughly the same values since a low complexity model don't have the power to express minor differences between different samples of training data. Hence the variance V_P will be low. The bias will however by high since all of the estimators differ in the same way from the true value $f(\mathbf{P})$.
- To high complexity (e.g. a neural network with several layers and many weights) The computed estimators $h_1(\mathbf{p}), ..., h_N(\mathbf{p})$ will train precisely to each training set, thereby producing high variance V_P for the estimators since each estimator operate on different random samples of training data. The bias will however be low since the mean value m_P computed as $\sum_i h_i(\mathbf{p})/N$ will be centered around the true value $f(\mathbf{P})$.

Now turn to the "real" situation where only one single estimate $h_k(\mathbf{P})$ has been produced. It is still an outcome from the random variable $h(\mathbf{P})$ with its associated mean and variance. So, what distribution on the estimates do we prefer to pick the single estimate from, one with low-variance/high-bias or one with low-bias/high-variance? The choice is partly controlled by the complexity of the model. Looking at equation 6.2, the correct answer would be that neither of the alternatives are optimal in terms of giving a minimal prediction error E_p. The best choice is a trade-off between low bias and low variance. Since the complexity affects the entities in different directions we are facing what is called the "bias/variance" dilemma. A large variance has to be accepted in order to keep the bias low and vice versa.

According to Casdagli and Weigend [CW93], the position taken by most statisticians is, "for reasons of conservatism", to favor low-variance/high-bias over low-bias/high-variance. This results in nonlinear models with relatively low complexity and few parameters. These models work fine if the underlying function is equally simple and the interesting behavior is mainly due to outside perturbations. For more complicated functions, models with higher complexity must be used to get acceptably low prediction error. The price to be payed for this additional expressiveness is higher variance.

In some cases, a low complexity model can be designed using prior knowledge about the specific application of interest. In such cases the bias is "harmless" since it is directed towards the real function $f(\mathbf{P})$. The bias can then be brought down without increasing the variance. This approach can applied even to black box models such as neural networks. In *weight sharing* several synapses (connections between nodes) are using the same weight. In *radial basis networks* the receptive field of the neurons in the hidden layer can be preset to reflect known properties in the function to be modelled. The general situations are described with the terms "Weak" and "Strong" modelling.

6.2 Weak and strong modelling

The terms weak and strong modelling refers to the degree to which a model is pre conditioned to reflect the underlying process to by modelled. A weak model makes few assumptions on what the real process looks like whereas a strong model makes many assumptions. The traditional choice in the natural sciences has been to prefer strong models which are tightly connected to the actual process. The parameters in such models can often be given a "meaning" in terms of slopes, constants, thresholds etc. There are however also examples of weak modelling. Physicists sometimes model dynamical systems as fairly general nonlinear functions [CW93]. The classical ARMA-models are also examples of weak models with few domain-specific assumptions.

6.3 Overfitting and Underfitting

The term Overfitting is used to denote the situation where the selected model type has to high complexity with respect to the "bias/variance" trade-off.. The term underfitting denotes situations where the model has to low complexity.

6.4 Overtraining

The term overtraining refers to a situation where a model is fit to closely to the training data, thereby decreasing the computed models generalization performance. The problem is caused by allowing the learning algorithm to keep iterating in an attempt to bring the total prediction error on the training data to an absolute minimum. When the prediction error has reached below a certain, problem specific limit, the algorithm will however start fitting properties in the training data which are not general but rather to be considered as noise. The generalization performance will therefor decrease from that point on in the learning process. The phenomena is normally connected to "weak modelling" where the model is totally unbiased and capable of fitting data from any data source. It is also connected to the issue of model complexity since the point where the overtraining starts depends on the models expressiveness.. A model with low complexity (e.g. a straight line) will be less sensitive to the problems with overtraining. A high noise level in data does also increases the risk of overtraining since noisy data more easily gets interpreted as real data by the training process. The risk for overtraining is also affected by the amount of data used for the training of the model. A lot of data prevents a model of a certain complexity to interpret noisy data as real.

The problem with overfitting can either be solved indirectly by reducing the model complexity or directly by interrupting the learning algorithm. Refer to section 6.6 for further discussions.

6.5 Measuring Generalization ability

The crucial measure for all learning algorithms is the ability to perform good when presented unseen data (also called out-of-sample performance). The difference between the performance on the training data and on unseen data is dramatically increased when low-bias models such as artificial neural networks are considered.. The importance of good estimations of the performance on unseen data can in such situations hardly be overestimated. Apart from being the principal performance measure for a modelling problem, the generalization ability is also an important tool for model selection.

The theoretical aspects of why and when learning works is covered by *computational learning theory*, a field in the intersection of Artificial Intelligence and computer science. A result from the sub field *PAC-learning (probably approximately correct)* gives the following relation between the number of necessary examples m and the set of possible hypotheses H.

$$m \geq \frac{1}{\varepsilon}\left(\ln\frac{1}{\delta} + \ln|H|\right) \tag{6.3}$$

where ε is the error in a specific hypothesis and δ is the probability for a non correct hypothesis being consistent with all examples. Thus, by using at least m examples in the training then with probability at least $1-\delta$, the produced hypothesis has an error at most ε. Even if the practical results from PAC-learning still are limited, it puts focus on two important things:

- The aim of learning is to find a hypothesis which is approximately correct. Traditional learning theory focused on the problem of *identification in the limit* where the hypothesis should match the true function exactly.
- The size $|H|$ of the hypothesis space, i.e. the model complexity, is a key issue for both estimation and control of generalization ability.

A number of methods for estimation of the generalization ability exist.

6.5.1 Test-set validation

The standard procedure for validation in most machine learning techniques is to split the data into a training set and a test set. Some times the training set is further divided to extract a cross validation set (used to determine the stopping point to avoid overfitting). The test set is only used for the final estimation of the generalization performance. This approach is wasteful in terms of data since not all of it can be used for training. The advantage is that no assumptions regarding error distributions or model linearity have to be made. In the case of neural networks, Weigend and LeBaron [WL94] have shown that the variation in results, due to how the split in the three sets is done, is much larger then the variation due to different network conditions such as architecture and initial weights. The method used to show this is a variation of the procedure described in the section "Bias and Variance" above. By really generating a huge number of independent hypothesis functions the variance can be explicitly estimated. Weigend and LeBaron uses predictions of traded volume on the New York Stock Exchange as application in their report. Volume data is known to contain more forecastable structures then typical price series. It is however dominated by a noise component which makes the sensitivity to how the data is split very prominent.

This result should not be seen as a disqualification of the method of test-set-validation, but rather as an illustration of the general problem with all inductive inference methods.

6.5.2 Cross-Validation

Cross-validation takes the idea of Test-set-validation to its extreme. Assume that the entire data set contains N data samples. One sample is left out and the remaining samples are used to train a model. The performance of this model is estimated by the squared error in the left out sample. The procedure is repeated for all N samples in the data set, thus producing N models with associated error estimates for the single left out point. The mean of these estimates are used as a total estimate of the prediction error for the model. If N is large, the method easily gets too expensive in terms of computations. Variations where more the one sample is removed from the data set was introduced in [Gei75]. A method specifically developed for artificial neural networks was proposed in [MU95]. Instead of starting the training from scratch with each new training set, the weights from previous training is kept and used as starting values for the next model.

6.5.3 Algebraic estimates

The various methods with cross validation require the data to be split in separate sets for training and estimation of generalization error. This is often not possible to do when the total number of data points are very limited or the data is very noisy (a large training set is then necessary). A number of algebraic estimates of the generalization error however exist and work without any split of the data. They all impose prior assumptions on the statistical error distribution. Some well known formulas are Akaike's final prediction error (FPE) and Generalized cross validation (GCV):

$$FPE = MSE\left(\frac{1+\frac{Q}{N}}{1-\frac{Q}{N}}\right) \tag{6.4}$$

$$GCV = MSE\frac{1}{\left(1-\frac{Q}{N}\right)^2} \tag{6.5}$$

MSE is the average squared error and is computed over the whole data set. The methods work by penalizing the MSE with a term to compensate for the complexity of the model. A sufficiently powerful model with many weights can, as is well known, fit any data set. A low value of MSE is therefor not sufficient to guarantee a low generalization error. The complexity is estimated by the number of *free parameters* Q.

In the case of neural network models, the value on Q is far from trivial! If the training is done with regularization such as early stopping, Tikhonov regularization or weight elimination, the number of free parameters is not the same as the number of weights!

6.6 Controlling Model complexity

As been described above, the model complexity is intimately connected the "bias /variance dilemma". In powerful models such as neural networks with many weights, it's often necessary to impose some restrictions on the model in order to avoid too high model variance. The methods used are often termed "regularization" and include techniques such as Architecture selection, Adding noise to data, Early stopping and Tikhonov regularization. Tikhonov's *regularization theory* applied to neural network training algorithms can be found in [EGLW97].

6.6.1 Architecture selection in Neural Networks

The architecture of an ordinary feed-forward neural network is characterized by

- The number of hidden layers
- The number of hidden nodes
- The set of non zero weights

In the following we assume a network with one hidden layer. The problem of finding the best number of hidden nodes and the best set of nonzero weights can be approached in one of two ways:

- Network growing methods.

The network is gradually increased in size by adding hidden nodes until the best performance is achieved. Methods for doing this includes *structure-level adaption* and *cascade correlation learning architecture [FL90].*

- Network pruning methods

The network is gradually decreased in size by either removing nodes (such as in *optimal brain surgeon OBS [?] and optimal brain damage OBD [?]),* by removing weights (such as in *weight-elimination [?]*) or by reducing weights (such as in *weight-decay[?].* Haykin gives a survey of these common methods in [?].

Moody [Moo94] presents some other common algorithms for control of the network complexity: the *minimum description length (MDL)* [Ris78] and *an information theoretic criterion* [**?**]. Both methods work by adding a complexity term to the ordinary mean squared error that is minimized in the training process.

Structure-level adaption

In this method the networks performance is monitored after the training phase has completed. If the estimation error is larger then a desired value, a new node is added to the network which is then trained again. If the weights connected to the inputs of a node fluctuate a lot between successive trainings it may be inferred that the node does not contribute to the function approximation and should therefor be removed.

Cascade correlation learning architecture

The procedure starts with an empty hidden layer and an input and output layer determined by the specific problem at hand. New hidden nodes are then added one by one. Each new node gets weights connecting it to the input layer and to the existing hidden nodes. The new weights and some of the old weights are trained repeatedly when each new node is added. new nodes are added until satisfactory performance is attained.

Optimal brain surgeon OBS and Optimal brain damage OBD

The basic idea of these methods is to identify the weights whose deletion from the network will cause the least increase in the value of the error function (normally the mean square error). This is achieved by a quadratic approximation of the error surface. The methods are very computationally intensive and require inversion of the Hessian matrix for the network.

Weight-decay

The idea behind this method is to force weights to take values either close to zero or relatively large. Add to the usual cost function a term which is a scaled sum of all squared weights, and minimize this new sum in the training phase. I.e. for a neural network with K weights and N examples in the training set we want to minimize the following cost function:

$$\sum_{i=1}^{N} (y'(t) - y(t))^2 + \lambda \sum_{i=1}^{K} w_i^2 \tag{6.6}$$

The last term works by grouping the weights of the network in two categories: those that have a large influence on the mean squared error and those that have little influence on it. Weights that have little effect on reducing the mean squared error will be penalized because of the last term. Their absolute values will therefor take values close to zero. Weights that have large effect on reducing the mean squared error will on the other hand not be penalized since the last term is balanced by a hopefully greater reduction in the first term. The value on λ is crucial for successful function of the method.

Weight-elimination

The idea behind this method is simple: Add to the usual cost function a term which estimates the number of significant parameters, and minimize this new sum in the training phase. I.e. for a neural network with K weights and N examples in the training set we want to minimize

$$\sum_{i=1}^{N} (y'(t) - y(t))^2 + \lambda \sum_{i=1}^{K} \frac{w_i^2/w_0^2}{1 + w_i^2/w_0^2} \tag{6.7}$$

The last term is an estimate of the number of significantly sized weights. "Significant" is defined by the choice of w_0 .For $|w_i| \ll w_0$ the cost is close to zero. All significantly sized weights will result in an extra cost and will therefor be penalized when the training algorithm attempts to minimize the cost function. It should be noted that the values on λ and w_0 are crucial for the function of the method. Guidelines for selection of λ can be found in *[?]*.

Early Stopping

Early stopping is a method that addresses the complexity issue by combining training and estimation of the generalization performance. It is normally used for neural networks but could also be applied to other types of inductive learning techniques. Weigend et al. [WRH90] suggests setting part of the training data apart, introducing a cross-validation set. The learning is done using the training set data but is stopped at the point where the performance (e.g. mean square error) on the cross-validation set has its minimum. This performance measure for the cross validation set is therefor computed in parallel with the ordinary training algorithm that attempts to minimize the mean squared error on the training set data. The weights in the epoch where the mean square error on the cross validation set has its minimum are then regarded as optimal and are selected for the model. The produced model is then applied to the test set for a final estimation of the generalization performance.

In [WR92] and [**?**] Weigend and Rumelhart performs an interesting analysis on how the effective dimension of the hidden units in a neural network is changed during

back propagation training. The effective dimension is a measure of the complexity in the model that is represented by the network at each instance during training. It turns out that the effective dimension starts at almost zero and gradually increases during training. This provides a justification for the use of oversized neural networks and early stopping. The early stopping should therefor by viewed as a architecture selection tools rather then an obscure method that stops the regression before the actual minima is reached.

Bibliography

[CW93] Martin C. Casdagli and Andreas S. Weigend. *Exploring the continuum Between Deterministic and Stochastic Modeling*, pages 347–369. Addison-Wesley, 1993.

[Cyb88] G. Cybenko. Approximation by superpositions of a sigmoidal function. Technical report, University of Illinois, 1988.

[Efr82] B. Efron. The jackknife, the bootstrap and other resampling plans. *Society for Industrial and Applied Mathematics (SIAM)*, 38, 1982.

[EGLW97] Jerry Eriksson, Mårten Gulliksson, Per Lindström, and Per-Åke Wedin. Regularization tools for training large Feed-Forward neural networks using automatic differentiation. *Optimization Methods and Software*, page ?, 1997.

[FL90] S.E. Fahlman and C. Lebiere. *The cascade-correlation learning architecture*. Morgan Kaufmann, 1990.

[GBD92] Geman.S, E. Bienstock, and R. Doursat. Neural networks and the bias/variance dilemma. *Neural Computation*, 5:1–58, 1992.

[Gei75] S. Geisser. The predictive sampling reuse method with applications. *Journal of The American Statistical Association*, 1975.

[Moo94] John Moody. Prediction risk and architecture selection for neural networks. In V. Cherkassky, J. H. Friedman, and H. Wechsler, editors, *From Statistics to Neural Networks: Theory and Pattern Recognition Applications*, NATO ASI Series F. Springer-Verlag, 1994.

[MU95] John Moody and Joachim Utans. Architecture selection strategies for neural networks: Application to corporate bond rating prediction. In Apostolos-Paul Refenes, editor, *Neural Networks in the Capital Markets*, pages 277–300. John Wiley & Sons, 1995.

[Ris78] J Rissanen. Modelling by shortest data description. *Automatica*, 14:465–471, 1978.

[WL94] Andreas S. Weigend and Blake LeBaron. Evaluating neural network network predictors by bootstrapping. Technical Report CU-CS-725-94, Department of Computer Science and Institute of Cognitive Science, University of Colorado, Boulder, CO, May 1994.

[WR92] Andreas S. Weigend and David E. Rumelhart. *Weight Elimination and Effective Network Size*, chapter 16, pages 457–476. ?, 1992?

[WRH90] Andreas S. Weigend, David E. Rumelhart, and Bernardo A. Huberman. Predicting the future: a connectionist approach. *Internatioanl Journal of Neural Systems*, pages 193–209, 1990.