A Modular Neural Network Architecture with Additional Generalization Abilities for High Dimensional Input Vectors

A thesis submitted to the Manchester Metropolitan University in partial fulfillment of the requirements for the degree of Master of Science in Computing by:

Albrecht Schmidt

Manchester Metropolitan University,
Department of Computing,
September 1996.

Acknowledgement

I owe a debt of gratitude to my supervisor, Dr Zuhair A. Bandar from whom I have learned much about neural networks. I wish to acknowledge his support and guidance throughout the project.

I am greatly thankful to Dave Shield for his efforts in introducing me to the mysteries of the English language while proof-reading this thesis.

Special thanks to my dear friend Petra Dollinger for her encouragement over the last six months.

I would also like to express thanks to my parents for their support during my stay in Manchester.

Abstract

In this project a new modular neural network is proposed. The basic building blocks of the architecture are small multilayer feedforward networks, trained using the Backpropagation algorithm.

The structure of the modular system is similar to architectures known from logical neural networks. The new network is not fully connected and therefore the number of weight connections is much less than in a monolithic multilayer Perceptron.

The suggested training algorithm works in two stages and is easy to implement in parallel. Due to the used modular structure the training is very quick for large input vectors.

The modular architecture is designed to combine two different approaches of generalization known from connectionist and logical neural networks; this enhances the generalization ability, which is especially significant for a high dimensional input space.

An object-oriented implementation of the proposed model was written to simulate the behaviour.

The evaluation using different real world data sets showed that the new architecture is very useful for high dimensional input vectors. For certain domains the learning speed as well as the generalization performance in the modular system is significantly better than in a monolithic multilayer feedforward network.

Contents

List of Figures

List of Tables

Abbreviations

AI	Artificial Intelligence
AN	Adaptive Node
ANN	Artificial Neural Networks
ANS	Artificial Neural System
ART	Adaptive Resonance Theory
BP	Backpropagation
CALM	Categorization And Learning Module
CNS	Central Nervous System
ECG	ElectroCardioGram
HD	Hamming Distance
LNN	Logical Neural Networks
MLFFN	MultiLayer FeedForward Network
MLP	MultiLayer Perceptron
MMU	Manchester Metropolitan University
MNN	Modular Neural Network
NN	Neural Networks
NS	Nervous System
PET	Positron-Emission Tomography
PLN	Probabilistic Logical Node
PNS	Peripheral Nervous System
RAM	Random Access Memory
SOM	Self Organizing Map
WISARD	WIlki, Stonham, and Aleksander's Recognition Device
XOR	eXclusive OR

Chapter 1

Introduction

The simulation of human intelligence using machines was and is always a challenge to ingenuity. In the middle of this century a research discipline calling itself *Artificial Intelligence* (AI) emerged. The definition for the term AI is very indistinct; this has its major reason in the fact that there is no commonly accepted definition for 'intelligence'. The most comprehensive definition for AI includes all research aimed to simulate intelligent behaviour.

Despite the availability of massive computational power the results of current research are far from the aims proposed in the enthusiasm of the 1950's and 60's. Nevertheless systems built to simulate intelligence[1] in a limited domain, such as expert systems in medicine or forecasting applications, are already successfully used.

Artificial Neural Networks (ANNs) are of major research interest at present, involving researchers of many different disciplines. Subjects contributing to this research include biology, computing, electronics, mathematics, medicine, physics, and psychology. The approaches to this topic are very diverse, as are the aims. The basic idea is to use the knowledge[2] of the nervous system and the human brain to design intelligent artificial systems.

On one side biologists and psychologists are trying to model and understand the

[1] It will always depend on someone's view if the performed task is seen as *intelligent* or not.

[2] Which is surprisingly very limited.

brain and parts of the nervous system and searching for explanations for human behaviour and reasons for the limitations of the brain.

On the other, computer scientists and electronic engineers are searching for efficient ways to solve problems for which conventional computers are currently used. Biological and psychological models and ideas are often the resource of inspiration for these scientists.

In the computing environment the term *Neural Network* (NN) is usually used as synonym for artificial neural network.

1.1 The Project

This project is inspired by one main property of the nervous system: *Modularity*. The concept of modularity is investigated in the context of artificial neural networks.

First well known architectures and learning methods such as multilayer feedforward networks trained using the Backpropagation algorithm (see chapter 2) and logical networks (see chapter 3) were examined. The objective was to find networks which can be used as modules in a modular system as well as gaining knowledge of different architectures and their advantages.

In the next step, described in chapter 4, a further assessment of the nature of modularity was carried out. The aim was to find evidence for modular structures in the human nervous system and also to review recent developments in ANNs using the concept of modules.

Based on this knowledge a new modular artificial neural system was designed (see chapter 5), in chapter 6 the implementation is described and the evaluation of the model is given in chapter 7.

During the project a conference paper describing the most interesting parts of the architecture as well as some of results gained in the experiments was written, this is included in appendix A.

1.2 Biological Neural Networks

Artificial NN draw much of their inspiration from the biological nervous system. It is therefore very useful to have some knowledge of the way this system is organized.

Most living creatures, which have the ability to adapt to a changing environment, need a controlling unit which is able to learn. Higher developed animals and humans use very complex networks of highly specialized neurons to perform this task.

The control unit – or brain – can be divided in different anatomic and functional sub-units, each having certain tasks like vision, hearing, motor and sensor control. The brain is connected by nerves to the sensors and actors in the rest of the body.

The brain consists of a very large number of neurons, about 10^{11} in average. These can be seen as the basic building bricks for the central nervous system (CNS). The neurons are interconnected at points called synapses. The complexity of the brain is due to the massive number of highly interconnected simple units working in parallel, with an individual neuron receiving input from up to 10 000 others.

The neuron contains all structures of an animal cell. The complexity of the structure and of the processes in a simple cell is enormous. Even the most sophisticated neuron models in artificial neural networks seem comparatively toy-like.

Structurally the neuron can be divided in three major parts: the cell body (soma), the dentrites, and the axon, see Figure 1.1 for an illustration.

The cell body contains the organelles of the neuron and also the 'dentrites' are originating there. These are thin and widely branching fibers, reaching out in different directions to make connections to a larger number of cells within the cluster.

Input connection are made from the axons of other cells to the dentrites or directly to the body of the cell. These are known as axondentritic and axonsomatic synapses.

There is only one axon per neuron. It is a single and long fiber, which transports the output signal of the cell as electrical impulses (action potential) along its length.

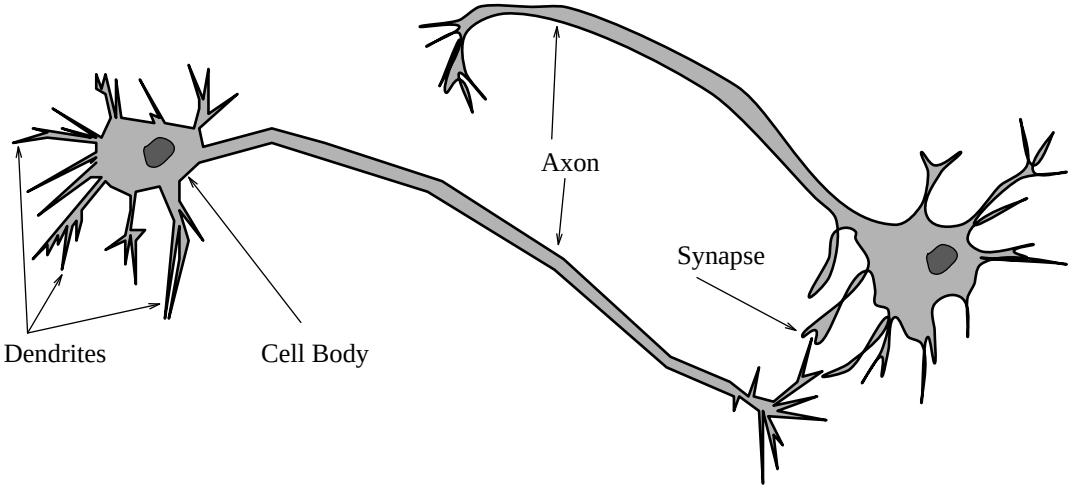


Figure 1.1: Simplified Biological Neurons.

The end of the axon may divide in many branches, which are then connected to other cells. The branches have the function to fan out the signal to many other inputs.

There are many different types of neuron cells found in the nervous system. The differences are due to their location and function.

The neurons perform basically the following function: all the inputs to the cell, which may vary by the strength of the connection or the frequency of the incoming signal, are summed up. The input sum is processed by a threshold function and produces an output signal. The processing time of about 1ms per cycle and transmission speed of the neurons of about 0.6 to 120 $\frac{m}{s}$ are comparingly slow to a modern computer [zell94, p24] , [barr88, p35].

The brain works in both a parallel and serial way. The parallel and serial nature of the brain is readily apparent from the physical anatomy of the nervous system. That there is serial and parallel processing involved can be easily seen from the time needed to perform tasks. For example a human can recognize the picture of another person in about 100 ms. Given the processing time of 1 ms for an individual neuron this implies that a certain number of neurons, but less than 100, are involved in serial; whereas the complexity of the task is evidence for a parallel processing, because a difficult recognition task can not be performed by such a small number of neurons, example taken from [zell94, p24]. This phenomenon is

known as the *100-step-rule*.

Biological neural systems usually have a very high fault tolerance. Experiments with people with brain injuries have shown that damage of neurons up to a certain level does not necessarily influence the performance of the system, though tasks such as writing or speaking may have to be learned again. This can be regarded as re-training the network.

In the following work no particular brain part or function will be modeled. Rather the fundamental brain characteristics of parallelism and fault tolerance will be applied.

1.3 Definitions

In the literature a wide variety of definitions and explanations for the terms *Artificial Neural Network* and *Neural Computing* can be found. The following definitions are balanced towards computing but are nevertheless very comprehensive in my opinion and they offer a wide range of views of what an ANN is.
The definition by Igor Aleksander is including a very wide range of methods and applications in the field of neural computing:

> "Neural computing is the study of networks of adaptable nodes which, through a process of learning from task examples, store experimental knowledge and make it available for use." [alek95, p1].

The following description by Laurene Fausett of artificial neural networks includes only the connectionist research approach [faus94, p3].

> "An *artificial neural network* is an information-processing system that has certain performance characteristics in common with biological neural networks. Artificial neural networks have been developed as generalizations of mathematical models of human cognition or neural biology, based on the assumption that:
>
> 1. Information processing occurs at many simple elements called neurons.
> 2. Signals are passed between neurons over connection links.
> 3. Each connection link has an associated weight, which, in a typical neural net, multiplies the signal transmitted.
> 4. Each neuron applies an activation function (usually nonlinear) to its net input (sum of weighted input signals) to determine its output signal."

Robbert L. Harvey focuses very much on the biological model. His definition excludes most parts of logical neural networks from the field of neural networks.

> "A neural network is a dynamical system with one-way interconnections. It carries out processing by its response to inputs. The processing elements are nodes; the interconnects are directed links. Each processing element has a single output signal from which copies fan out." [harv94, p2f]

The author of this report has the following definition which is more concerned with the fundamental ideas of neural systems and the basic properties of the brain rather than the aspect of modeling parts of the nervous system.

An Artificial Neural System (ANS)

- consists of simple interconnected modules.
- is based on communication between modules.
- performs its task by parallel processing.
- is fault tolerant.
- is learning from example.
- has the ability to generalize.
- performs complex tasks due to the whole architecture.

1.4 Memorization and Generalization

To simulate intelligent behavior the abilities of memorization and generalization are essential. These are basic properties of artificial neural networks. The following definitions are according to the Collins English Dictionary:

to memorize:	to commit to memory; learn so as to remember.
to generalize:	to form general principles or conclusions from detailed facts, experience, etc.

Memorizing, given facts, is an obvious task in learning. This can be done by storing the input samples explicitly, or by identifying the concept behind the input data, and memorizing their general rules.

The ability to identify the rules, to *generalize*, allows the system to make predictions on unknown data.

Despite the strictly logical invalidity of this approach, the process of reasoning from specific samples to the general case can be observed in human learning.

Generalization also removes the need to store a large number of input samples. Features common to the whole class need not to be repeated for each sample – instead the system needs only to remember which features are part of the sample. This can dramatically reduce the amount of memory needed, and produce a very efficient method of memorization.

1.5 Acquiring Knowledge by Learning

A vital property of neural networks is that they can learn the desired response from a set of examples in the domain. This contrasts with most other approaches in computing where an algorithm or rules are used to store the knowledge.

The advantage of learning from examples is that there is no need to explicitly form a rule system for the task. To extract rules from the knowledge in the domain implies that there is some expert interpretation. This process is often difficult, especially if the experts have different opinions on the problem. From an abstract point of view training a NN can be seen as an automatic process of extracting rules from a data set.

There are two basic paradigms of learning, supervised and unsupervised, both of which have their models in biology.

Supervised learning at its most general is a process where both information about the environment (e.g. the sensory stimuli) and the desired reaction of the system (e.g. the motor response) is given. It is analogous to human learning with a teacher who knows all the answers.

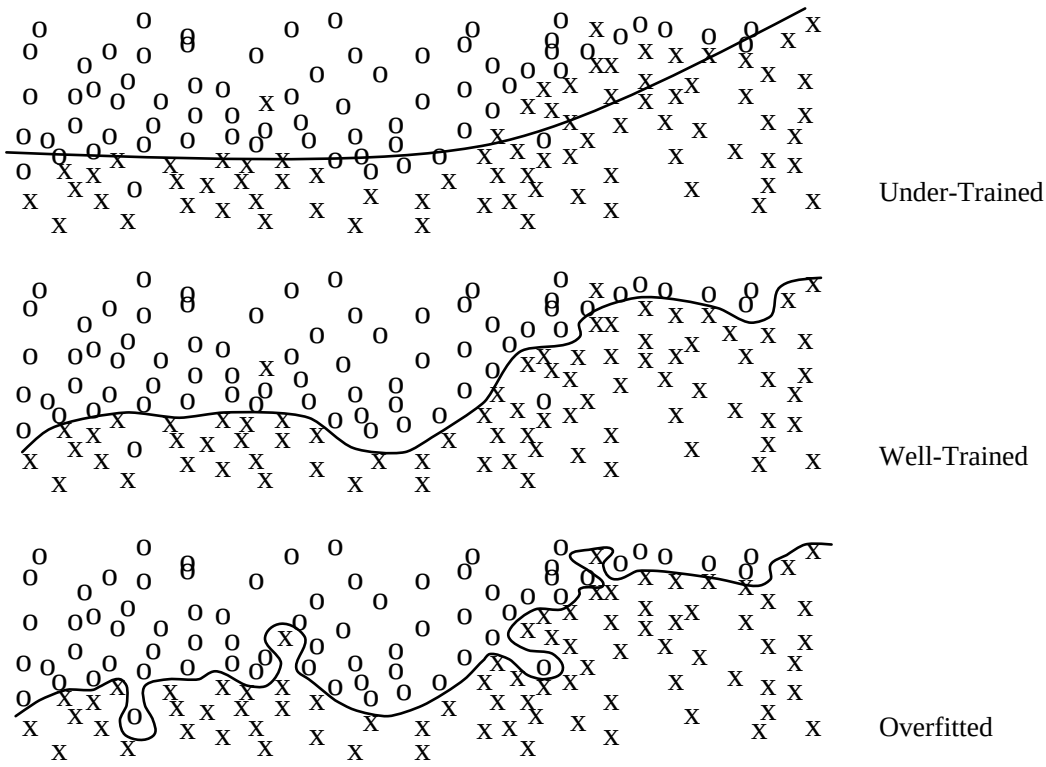


Figure 1.2: Learning and the Problem of Overfitting.

In an ANN context supervised learning is a process of memorizing vector pairs. The input vector together with the desired output vector is known. This method is often referred to as memorizing a mapping from the input space to the output space.

A special case is autoassociative mapping where the input pattern and the output pattern are equal (often written as a single vector). Autoassociative memories are often used to retrieve the original pattern for a distorted input.

Many algorithms for supervised learning work on a comparison between the calculated response of the network during training and the target values. There are also learning techniques where the input-output pair is directly used to calculate the weights in the network (e.g. the bidirectional associative memory [simp90, p61]).

A variant of supervised learning is called **reinforcement learning**. In this method the required output is not provided; the response by the teacher is only whether the calculated result is 'right' or 'wrong' [patt96, p28].

In supervised learning it is often difficult to determine when the learning process should be terminated. A network with a small error (the overall difference between the calculated and desired output) does not necessarily show a good performance on new data from the same domain. The problem is called **overfitting**. If the training process went on too long the network is biased towards the training set and the generalization ability of the network is decreased[3]. If the process is stopped too early the decision is very rough. In Figure 1.2 this is illustrated for the separation of two sets.

Unsupervised learning works only on the input vectors, and the desired output is not specified. This learning method can be compared to the process of categorization, discovering regularities, or adaptation to specific features.

The following sets explain the basic concept; S is the original set, the task is to split the set into two groups (here S_1 and S_2):

$$S = \{\sqcap, \vee, \sqcup, \cap, \wedge, \sqcup, \cup\}$$

$$S_1 = \{\sqcap, \wedge, \cap\}$$

$$S_2 = \{\vee, \sqcup, \cup\}$$

In this case the solution found, is a categorization according to the opening of the symbol. There are certainly other possible groupings, but the chosen one is very obvious.

Considering another set, difficulties with the unsupervised learning appear. If there are different ways to split the set, the process is not straight forward anymore. Assuming $S = \{a, A, b, B\}$, the following categories make equal sense: $S_1 = \{a, b\}$ and $S_2 = \{A, B\}$ or $S_1 = \{a, A\}$ and $S_2 = \{b, B\}$.

In many unsupervised models the categorization occurs according to the distance[4] between the input vectors. An example for a measure used is the Hamming distance

[3]This can also be observed in human behaviour; people who tend to learn everything literally often lose their ability to generalize.

[4]The word 'distance' is used as mathematical term.

for binary vectors. Generalization based on this approach groups input vectors in a way to minimize the distance between the members of a category for all categories.

If using an unsupervised model it is very important to analyze whether the clustering done by the network is the *right* way of grouping the data for the given problem.

1.6 Approaches to Neural computing

As already mentioned there are a huge range of approaches being used to tackle this topic. This section introduces two which are of particular importance to the project.

1.6.1 The Connectionist Approach

Connectionist artificial neural networks are an approach to neural computing which uses interconnected simple processors, called *neurons* to form a simplified model of the structures in the biological nervous system.

These neural networks have the ability to learn from examples and are trained to solve problems rather than programmed. The network has the ability to adapt to the environment.

The connectionist approach is very much inspired by biology and psychology. The very beginning of this field can be seen in the works of McCulloch and Pitts ([mccu43]), Hebb ([hebb49]), and Rosenblatt ([rose58]). This research does introduce the idea of parallel and interconnected neural systems based on simple units.

However there are various limitations to single layer networks which consist of simple neurons. These limitations, especially the classification problem of sets that are not linearly separable (in particular the XOR-Problem) were studied by Minsky and Papert ([mins69]). The publication of these results saw a significant reduction in research in neural networks.

In the 1980's advanced learning algorithms for multilayer networks (e.g. Back-propagation and derivatives) and ideas for different neural network architectures

such as Self Organizing Maps by Kohonen ([koho82]) and the ART network by Grossberg ([gros87]) re-vitalized research in this area. This increase in neural network research is still apparent, involving many different approaches. One is the idea of modular and multiple neural networks.

1.6.2 The Weightless Logical Approach

The logical approach to neural computing is different from the connectionist ideas for NN. The accumulated knowledge is stored as parts of patterns. This neural system has no weights.

The learning algorithm is very simple, the patterns are presented to the inputs of the network and then the patterns are stored in a certain way. This feature is very desirable from the application point of view, because it is fast to train and easy to use. However it lacks biological and psychological plausibility.

From the point of view of a biologist this kind of neural network has very few communalities with the biological description of the nervous system.

However, the architecture viewed from a more abstract point has some points in common with the biological model. The task is performed by many simple units; their only function is to store one piece of information for one input pattern. The ability to generalize is built in to the architecture of the network.

A major advantage of this approach is the ease of implementing the system in conventional hardware. One example of a hardware implementation is the WISARD, an adaptive pattern recognition device [alek95, p73ff].

1.7 Applications

Artificial neural networks are often used for applications where it is difficult to state explicit rules. Often it seems easier to describe a problem and its solution by giving examples; if sufficient data is available a neural network can be trained[5].

[5]Nevertheless one basic rule in information processing remains valid: *Garbage in* $\Rightarrow$ *Garbage out*. The results achieved by using neural network depend very much on the quality of data.

There is a wide range of application domains where ANNs are being used including classification, compression, noise reduction, optimization, prediction, and recognition. For a further description of applications see [simp90].

Chapter 2

Multilayer Networks and Backpropagation

Multilayer feedforward networks are one of the corner stones of research in ANN. In this project such networks are used as basic modules in a more complex structure.

This section discusses the usage of artificial neurons as building bricks for multilayer feedforward networks, together with learning algorithms needed to train such structures.

2.1 An Artificial Neuron

The artificial neuron shown in Figure 2.1 is a very simple processing unit. The neuron has a fixed number of inputs n; each input is connected to the neuron by a weighted link w_i. The neuron sums up the net input according to the equation: $net = \sum_{i=1}^{n} x_i \cdot w_i$ or expressed as vectors $net = \vec{x}^T \cdot \vec{w}$. To calculate the output a activation function f is applied to the net input of the neuron. This function is either a simple threshold function or a continuous non linear function. Two often used activation functions are:

$$f_C(net) = \frac{1}{1 - e^{-net}}$$

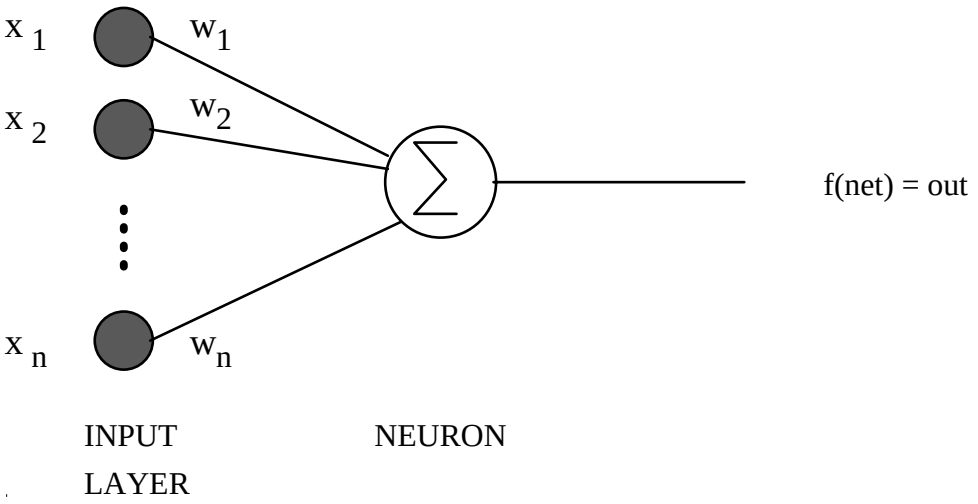


Figure 2.1: An Artificial Neuron.

$$f_T(net) = \begin{cases} if & a > \theta \quad then \quad 1 \\ else & 0 \end{cases}$$

The artificial neuron is an abstract model of the biological neuron. The strength of a connection is coded in the weight. The intensity of the input signal is modeled by using a real number instead of a temporal summation of spikes. The artificial neuron works in discrete time steps; the inputs are read and processed at one moment in time.

There are many different learning methods possible for a single neuron. Most of the supervised methods are based on the idea of changing the weight in a direction that the difference between the calculated output and the desired output is decreased. Examples of such rules are the *Perceptron Learning Rule*, the *Widrow-Hoff Learning Rule*, and the *Gradient descent Learning Rule*.

The Gradient descent Learning Rule operates on a differentiable activation function. The weight updates are a function of the input vector $\vec{x}$, the calculated output $f(net)$, the derivative of the calculated output $f'(net)$, the desired output d, and the learning constant η.

$$net = \vec{x}^T \cdot \vec{w}$$

$$\Delta w = \eta \cdot f'(net) \cdot (d - f(net)) \cdot \vec{x}$$

The delta rule changes the weights to minimize the error. The error is defined by the difference between the calculated output and the desired output. The weights are adjusted for one pattern in one learning step. This process is repeated with the aim to find a weight vector that minimizes the error for the entire training set.

A set of weights can only be found if the training set is linearly separable [mins69]. This limitation is independent of the learning algorithm used; it can be simply derived from the structure of the single neuron.

To illustrate this consider an artificial neuron with two inputs and a threshold activation function f_T; this neuron is intended to learn the XOR-problem (see table). It can easily be shown that there are no real numbers w_1 and w_2 to solve the equations, and hence the neuron can not learn this problem.

Input Vector	Desired Output	Weight Equation
0 0	1	$0 \cdot w_1 + 0 \cdot w_2 > \theta \Rightarrow 0 > \theta$
1 0	0	$1 \cdot w_1 + 0 \cdot w_2 < \theta \Rightarrow w_1 < \theta$
0 1	0	$0 \cdot w_1 + 1 \cdot w_2 < \theta \Rightarrow w_2 < \theta$
1 1	1	$1 \cdot w_1 + 1 \cdot w_2 > \theta \Rightarrow w_1 + w_2 > \theta$

2.2 A Single Layer Network

A single layer network is a simple structure consisting of m neurons each having n inputs. The system performs a mapping from the n-dimensional input space to the m-dimensional output space. To train the network the same learning algorithms as for a single neuron can be used.

This type of network is widely used for linear separable problems, but like a neuron, single layer network are not capable of classifying non linear separable data sets. One way to tackle this problem is to use a multilayer network architecture.

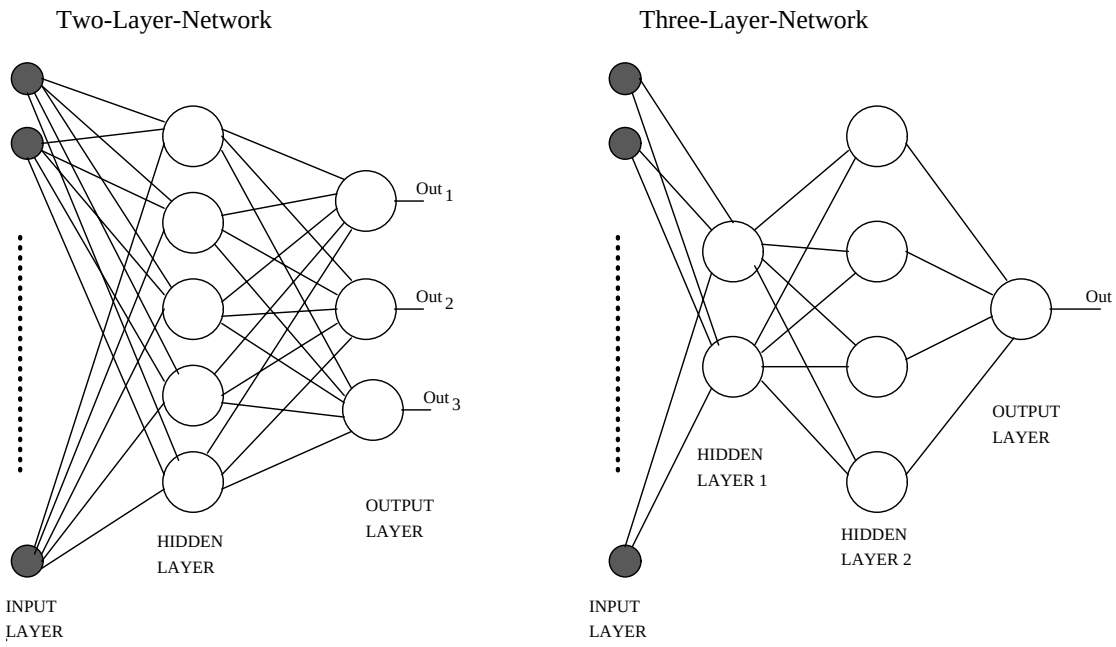


Figure 2.2: Examples of Multilayer Neural Network Architectures.

2.3 Multilayer Neural Network

Multilayer networks solve the classification problem for non linear sets by employing *hidden layers*, whose neurons are not directly connected to the output. The additional hidden layers can be interpreted geometrically as additional hyper-planes, which enhance the separation capacity of the network. Figure 2.2 shows typical multilayer network architectures.

This new architecture introduces a new question: how to train the hidden units for which the desired output is not known. The Backpropagation algorithm offers a solution to this problem.

The training occurs in a supervised style. The basic idea is to present the input vector to the network; calculate in the forward direction the output of each layer and the final output of the network. For the output layer the desired values are known and therefore the weights can be adjusted as for a single layer network; in the case of the BP algorithm according to the gradient decent rule.

To calculate the weight changes in the hidden layer the error in the output layer is back-propagated to these layers according to the connecting weights. This process is repeated for each sample in the training set. One cycle through the

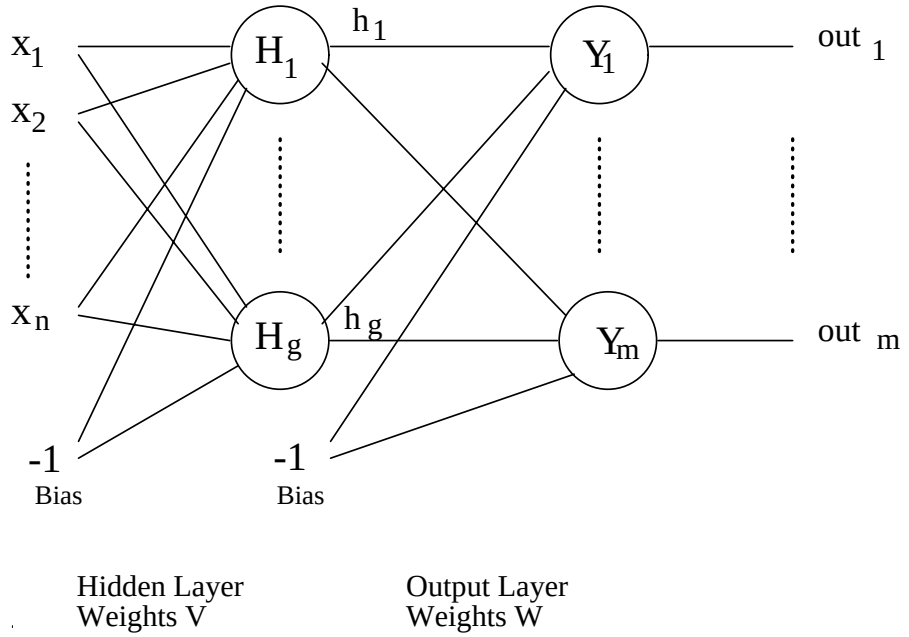


Figure 2.3: The Backpropagation Network.

training set is called an epoch. The number of epoches needed to train the network depends on various parameters, especially on the error calculated in the output layer.

The following description of the Backpropagation algorithm is based on the descriptions in [rume86],[faus94], and [patt96].

The assumed architecture is depicted in Figure 2.3. The input vector has n dimensions, the output vector has m dimensions, the bias (the used constant input) is -1, there is one hidden layer with g neurons. The matrix V holds the weights of the neurons in the hidden layer. The matrix W defines the weights of the neurons in the output layer. The learning parameter is η, and the momentum is α. For a discussion on the parameters see page 19.

The used unipolar activation function and its derivative are given by:

$$f(net) = \frac{1}{1 + e^{-\lambda \cdot net}}$$

$$f'(net) = \frac{e^{-\lambda \cdot net}}{(1 + e^{-\lambda \cdot net})^2}$$

The training set consists of pairs where $\vec{x}^p$ is the input vector and $\vec{t}^p$ is the desired output vector.

$$T = \{(\vec{x}^1, \vec{t}^1), \ldots, (\vec{x}^P, \vec{t}^P)\}$$

1. Initialize the weights V and W randomly with numbers from a suitable interval (e.g. -1 to 1). Select the parameters η and α.

2. Randomly take one unmarked pair $(\vec{x}, \vec{t})$ of the training set for the further steps and mark it as used.

3. Do the forward calculation. The notation $\vec{x}'$ is the input vector $\vec{x}$ enlarged by the bias, $\vec{h}'$ is the enlarged hidden layer output vector.

$$net_H = V^T \cdot \vec{x}'$$

$$h_i = f(net_{H_i})$$

$$net_y = W^T \cdot \vec{h}'$$

$$out_i = f(net_{y_i})$$

4. Do the backward calculation.

$$\delta_{out_i} = f'(net_{y_i}) \cdot (t_i - out_i)$$

$$\delta_{H_i} = f'(net_{H_i}) \cdot \sum_{j=1}^{m} w_{ij} \delta_{out_j}$$

5. Calculate the weight changes and update the weights.

$$\Delta W^T(t) = \eta \cdot \delta_{out} \cdot h'^T$$

$$\Delta V^T(t) = \eta \cdot \delta_H \cdot x'^T$$

$$W(t+1) = W(t) + \Delta W(t) + \alpha \cdot \Delta W(t-1)$$

$$V(t+1) = V(t) + \Delta V(t) + \alpha \cdot \Delta V(t-1)$$

6. `REPEAT` from step 2 `WHILE` there are unused pairs in the training.

7. Set all pairs in the training set to unused.

8. `REPEAT` from step 2 `UNTIL` the stop condition is `TRUE`.

Training continues until the overall error in one training cycle is sufficiently small; this **stop condition** is given by:

$$E_{max} > E$$

This acceptable error E_{max} has to be selected very carefully, if E_{max} is too large the network is under-trained and lacks in performance, if E_{max} is selected too small the network will be biased towards the training set (it will be overfitted).
One measure for the E is the root mean square error calculated by:

$$E = \frac{1}{P \cdot m} \sqrt{\sum_{p=1}^{P} \sum_{i=1}^{m} (t_i^p - out_i^p)^2}$$

The selection of the parameters for the Backpropagation algorithm and the initial settings of the weight influences the learning speed as well as the convergence of the algorithm.

The initial **Weights** chosen determine the starting point in the error landscape, which controls whether the learning process will end up in a local minimum or the global minimum. The easiest method is to select the weights randomly from a suitable range, such as between (-0.1,0.1) or (-2,2).

If the weight values are too large the *net* value will large as well; this causes the derivative of the activation function to work in the saturation region and the weight changes to be near zero. For small initial weights the changes will also be very small, which causes the learning process to be very slow and might even prevent convergence.

More sophisticated approaches to select the weights, such as the Nguyen-Widrow Initialization which calculates the interval from which the weights are taken in accordance with the number of input neurons and the number of hidden neurons, can improve the learning process. There are also statistical methods to estimate suitable initial values for the weights, [faus94, p297ff] and [patt96, p165f].

The **Learning Coefficient** η determines the size of the weight changes. A small value for η will result in a very slow learning process. If the learning coefficient is too large the large weight changes may cause the desired minimum to be missed.

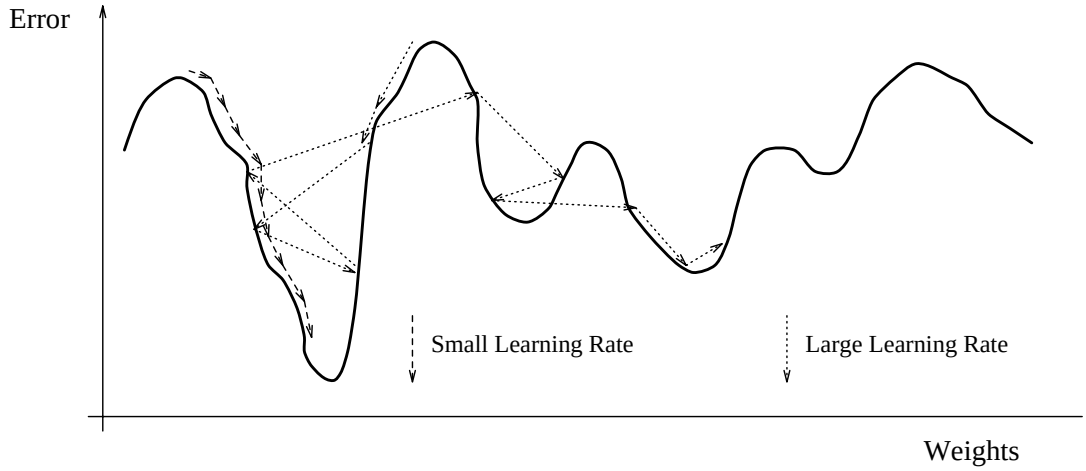


Figure 2.4: The Influence of the Learning Rate on the Weight Changes.

A useful range is between 0.05 and 2 dependent on the problem. The influence of the η on the weight changes is shown in Figure 2.4.

An improved technique is to use an adaptive learning rate. A large initial learning coefficient should help to escape from local minima, while reducing η later should prevent the learning process from overshooting the reached minimum.

The **Momentum** α causes the weight changes to be dependent on more than one input pattern. The change is a linear combination of the current gradient and the previous gradient. The useful range for this parameter is between 0 and 1. For some data sets the momentum makes the training faster, while for others there may be no improvement. The momentum usually makes it less likely that the training process will get stuck in a local minimum.

In recent years an enormous number of publications on refinements and improvements of the Backpropagation algorithms have been published. However most of the suggested improvements are only useful if the problem meets certain conditions. For examples see [patt96, p176f], [faus94, p305ff], and [zell94, p115ff].

Nevertheless the multilayer feedforward networks trained with the Backpropagation method are probably the most practically used networks for real world applications.

2.4 Other Methods to Train MLPs

There are many other ideas to improve the training of MLPs. The focus is on approaches using a layer by layer learning method.

The algorithm proposed in [leng96] works on optimizing an objective function for the internal representation in each layer separately. The weight updates in the hidden layers are independent from the final output of the network.

Another method based on information theory is described in [bich89]. The performance of a hidden unit is measured by its ability to transmit class information. The training process searches for a set of weights that minimize the conditional class entropy in each layer. In order to apply the concepts of information theory the neural network is viewed as a multistage encoder. To find the minimum the *simulated annealing* technique is used.

An accelerated learning algorithm for MLPs is given in [ergz95]. Only the neurons in the hidden layers use a sigmoidal transfer function; the neurons in the output layer use a linear function. The algorithm works iteratively and due to the linearization of the activation function the task of finding weights for the hidden layer is reduced to a linear problem. The weights in each layer are updated dependent on each other and on a particular cost function, but independent of the other layers.

All these methods showed superior performance on a number of applications. Most of the improved algorithms are useful for problems with certain properties.

2.5 Capabilities and Limitations of MLPs

In this section the question of capabilities and limitations of MLFFNs is addressed. In the second part networks trained by the Backpropagation algorithm are considered in more detail.

It is known that MLPs are universal approximators. The *Kolmogorov Theorem* states that any continuous function defined on a closed n-dimensional cube can be rewritten as a summation of applications of continuous function on one variable [kolm57].

For all functions $f(x_1, \ldots, x_n)$ with $x_i \in [0, 1]$ there are continuous functions ψ_i and ϕ_{ij} on one variable such that:

$$f(x_1, \ldots, x_n) = \sum_{j=1}^{2n+1} \psi_i(\sum_{i=1}^{n} \phi_{ij}(x_i))$$

However this theorem[1] is an existence theorem only! The functions ψ_i and ϕ_{ij} are dependent on the mapping function f and the theorem gives no help for finding these functions [patt96, p182] and [faus94, p328].

The *Hecht-Nielsen Theorem* which is based on the Kolmogorov theorem to shows that any continuous function $f : I^n \mapsto R^m$ can be approximated by a feedforward network with n inputs, $2n + 1$ hidden neurons, and m output nodes [hech87] and [hech89].

These theorems state that there is an appropriate network for such a problem, but give no assistance in how to find it. In particular they give no indication of a method to find the appropriate weights. The Backpropagation algorithm is *one* such method.

Usually the problem given to the neural network is described as a set of examples rather than a function. Therefore the number of points in the problem space is finite. Neural networks which are trained properly have a good interpolation performance, but a very poor extrapolation performance. Unfortunately there is no general rule to find out if a new pattern is within the interpolation space [hell95].

To define a measure for the generalization ability is very difficult. A good generalization always depends on the data set, what for one application might be desirable is useless for another. There are some theoretical approaches to this issue ([hold95] and the references of that paper).

[1] Found in 1957 as a solution to the 13th problem of Hilbert.

In [mraz95] a theoretical analysis of the robustness of BP networks is given. The paper proposes a definition for a separation characteristic which can be used to evaluate and compare BP networks. As a criteria for a robust network it requires that the network should respond with an output value 'no decision possible' for all new input patterns which are close to the separating hyperplane.

In [ston95] J. V. Stone and C. J. Thorton ask the question: "Can Artificial Neural Networks Discover Useful Regularities?"

In this paper they state the hypothesis that artificial neural networks trained with Backpropagation depend on correlations between input and output variables. They show that BP-MLPs have a very poor generalization ability for statistically neutral problems. Statistically neutral in this context means that no knowledge of the expected output value can be drawn from the knowledge of a single input variable. Only the relation between different input variables determines the output.

They suggest to using a sparse coding of the problem to supply the BP algorithm with correlated input variables and show that this approach can improve the generalization ability of BP trained ANN for statistically neutral problems.

However real world problems are rarely totally statistically neutral. A more appropriate question is therefore: "what kind of regularities can be discovered by an ANN?"

The idea to improve the generalization ability of a network by recoding the problem in a sparse code or binary (and therefore higher dimensional) seems very interesting.

Chapter 3

Logical Neural Networks

Logical neural networks (LNN) are based on simple processing units. These basic units have the ability to memorize input-output pairs. The logical model does not claim to implement the biological neural system. The basic unit in the logical approach is not a direct model of a neuron cell.

In the following section an adaptive node (AN) as a simple processing unit is introduced. The AN may be associated with a cluster of neurons or in special cases with a single neuron.

3.1 An Adaptive Node

An adaptive node has the ability to memorize the desired output for certain input patterns. All inputs and outputs are binary; the states are denoted with '0' and

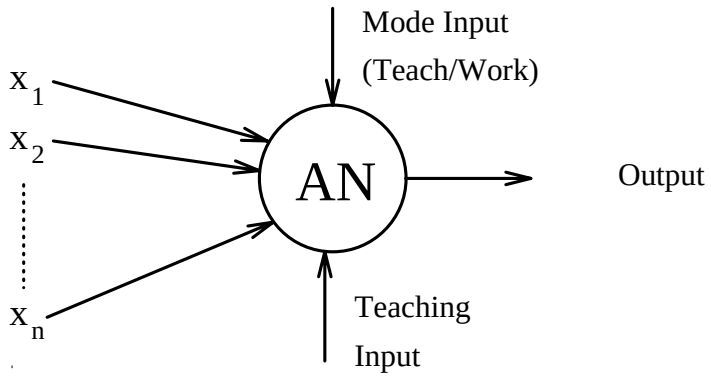


Figure 3.1: An Adaptive Node.

'1'. An AN has a number of data inputs, a mode input, a teaching input, and an output, see Figure 3.1 according to [alek95, p2].

The mode input determines whether the AN is learning or working. In the learning mode the node memorizes the input pattern (connected to the data inputs) together with the desired output (connected to the teaching input). In the working mode the response for a given input pattern (connected to the data input) is calculated and delivered to the output.

During the learning phase all the training pairs are presented to the AN; the node will memorize all pairs which are unambiguous. The response in the working phase will match the desired output for all correctly recognized training patterns.

For ambiguous and new patterns the AN will give the response '0' or '1' with equal probability (denoted here as '0/1'). Pairs are considered as ambiguous if the desired output can not be determined from the training set; e.g. one input pattern is given with different desired outputs. So far a single AN has only the function of a memory cell.

The search for a 'similar' pattern is implemented by a *firing rule*; this provides the ability to generalize on the input space. The response for an unseen input pattern is the same as the desired output for the memorized pattern with the minimal Hamming distance (HD) to the applied input pattern. If the minimal HD is equal to both a 0-taught pattern and a 1-taught pattern then the response is '0/1'.

To illustrate the firing rule of an AN consider a training set with two 0-taught vector (0 0 0) and (1 1 1); and a 1-taught vector (0 1 0). In the following table the responses to all possible 3-bit input patterns are shown.

pattern pattern	simple response	firing rule response
0 0 0	0	0
0 0 1	0/1	0
0 1 0	1	1
0 1 1	0/1	0/1
1 0 0	0/1	0
1 0 1	0/1	0
1 1 0	0/1	0/1
1 1 1	0	0

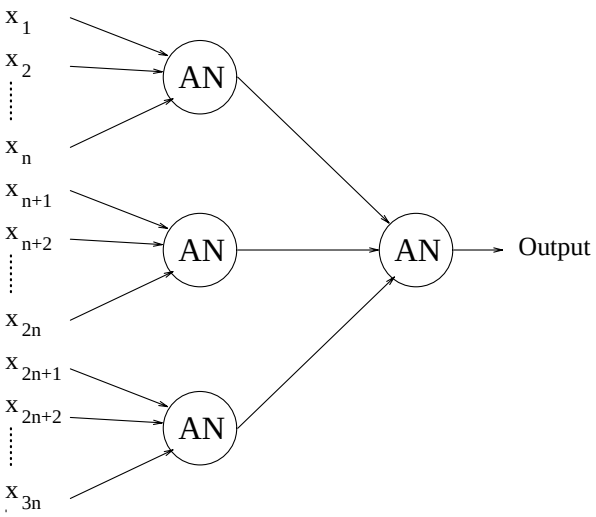


Figure 3.2: A Network of Adaptive Nodes.

The task performed by a single adaptive node with firing rule is simple template matching. The patterns are classified according to the Hamming distance.

A larger network consisting of ANs has additional generalization abilities. One possible network structure is shown in Figure 3.2. In the first layer the Hamming distance on small parts of the input pattern are calculated; in the second layer the results of the ANs of the first layer are combined to provide an overall result.

To calculate the HD of the whole input vector is often not appropriate as a large number of input samples would be necessary to enable generalization. If the Hamming distance is measured on smaller parts of the input vector it is much easier to obtain meaningful generalization from a small data set. It is more likely that subparts of a pattern are similar for the same class than that the whole vector is similar, see [alek95, p11].

3.2 A RAM Based Logical Neural Network

The most widely used form of logical neural networks is based on conventional random-access memory (RAM). This approach to neural computing works with a weightless network. On the first sight it looks very different from the biological idea of a neural network, but seen in connection with the adaptive node the differences are not so significant. The RAM unit is an adaptive node using the simple response

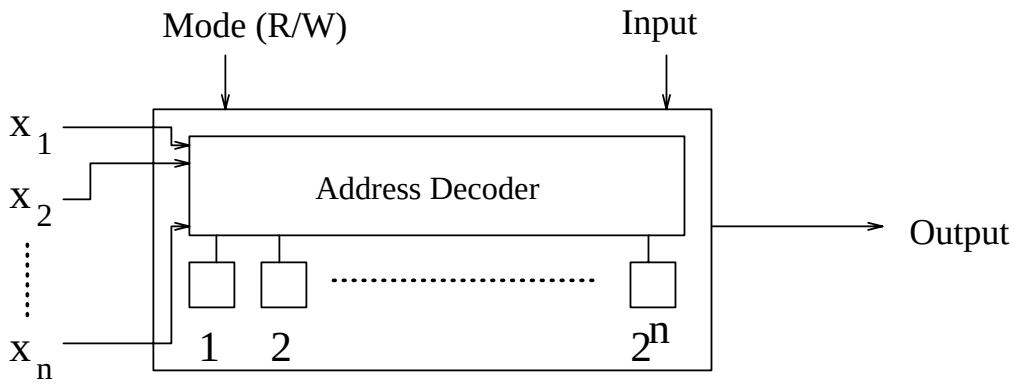


Figure 3.3: A Single RAM Unit.

schema.

The basic architecture is as follows: the input vector is divided into parts; each part is connected to the address inputs of a 1-Bit-RAM unit. The output of all the RAMs within one discriminator are summed up. The number of discriminators needed in a network is determined by the number of class which need to be distinguished by the network, see Figure 3.4.

The 1-Bit-RAM unit, depicted in Figure 3.3, is a device which can store one bit of information for each input address. A control input is available to switch the mode of the RAM between 'Write' and 'Read' for learning and recall.

Initially all memory units are set to '0'. During the learn ('Write') mode the memory is set to '1' for each supplied address; in the recall ('Read') mode the output is returned for each supplied address, either '1' (if the pattern was learned) or '0' (if the pattern was not learned).

The discriminator is the device which performs the generalization. It consists of several RAMs and one node which sums the outputs of the RAMs in recall mode. The discriminator is connected to the whole input vector; each RAM within the discriminator is connected to a part of this vector, so that each input bit is connected to exactly one RAM, see Figure 3.4(a). The connections are preferably chosen by random.

The network shown in Figure 3.4(b) can be used to distinguish k classes; it consists of k discriminators. For each class of input patterns one discriminator is

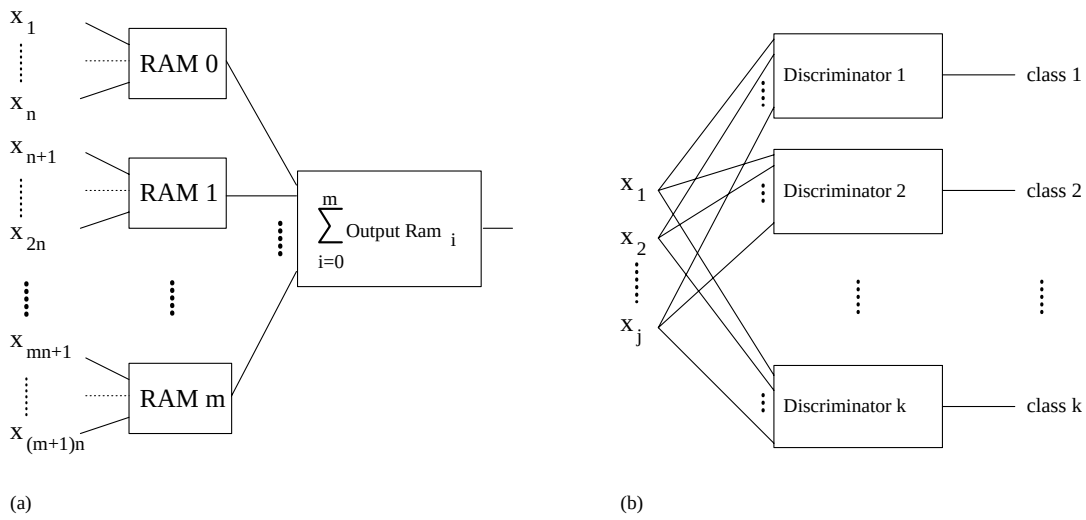


Figure 3.4: (a) A RAM-Discriminator
(b) A RAM-Based Logical Neural Network

needed, which is trained solely on the input data of its own class. In the recall mode each discriminator responds with the number of matching subpatterns. The pattern is classified according to the discriminator with the highest response.

The difference between the discriminator with the highest response and the runner-up is a measure of confidence ([alek89a, p174f]). It is also possible to set a threshold; if the output of a discriminator is higher than the threshold, then the pattern is accepted and recognized.

The major advantages of this kind of network are the ease of implementation and the ability to learn with only one presentation of the training patterns. These networks are mainly used for recognition purposes.

3.3 Selecting a Logical Network

To design a logical neural network appears much easier than choosing all the parameters for a Backpropagation MLFFN. The most important parameter is the size of the simple processing units; for a RAM based network this is the number of memory units per RAM. This size has a major influence on the generalization performance of the network. The **filling** of the network is the ratio between memory cells with content '1' to the size of the RAM. This is directly dependent on

the number of inputs to the RAM and therefore on the RAM-size and on the used training set.

If the number of inputs is small the filling will be very high. Assuming an extreme: a two input unit and therefore the RAM-size is four. If only four different subpatterns are in the training set the filling will be 100%.

A network with very high filling over-generalizes. The extreme case with a filling of 100% will return a discriminator response of '1' for all input patterns. This problem often appears when the training set is large or the data is very diverse.

Conversely a RAM unit with a large number of inputs is likely to have a low filling. For example, with 16 inputs there will be $2^{16} = 65536$ memory cells. To achieve an adequate filling in this case a large number of training examples is needed. Furthermore it is much more likely that different input vectors will share small subpatterns than large ones.

If the filling is low the generalization performance is poor. In theory, if the input size is equal to the dimension of input vector then there is no generalization at all; the vectors are simply stored.

The selected training vectors can be important, too. The simple method of using half the data set to train the network and the other half to test its performance may not always be appropriate, particularly for large data sets. It is proposed to use statistical methods or a genetic algorithm to select the most useful subset for training the network.

The best generalization performance for real world data is achieved with a filling in the range between 30% and 50%. For most applications this can be realized with a RAM-size of 256 (eight inputs) [band96].

3.4 An Example of a Logical Neural Network

This example illustrates the operation of the simplified recognition device shown in Figure 3.5, using a 3x3 input retina. The training set has four instances, rep-

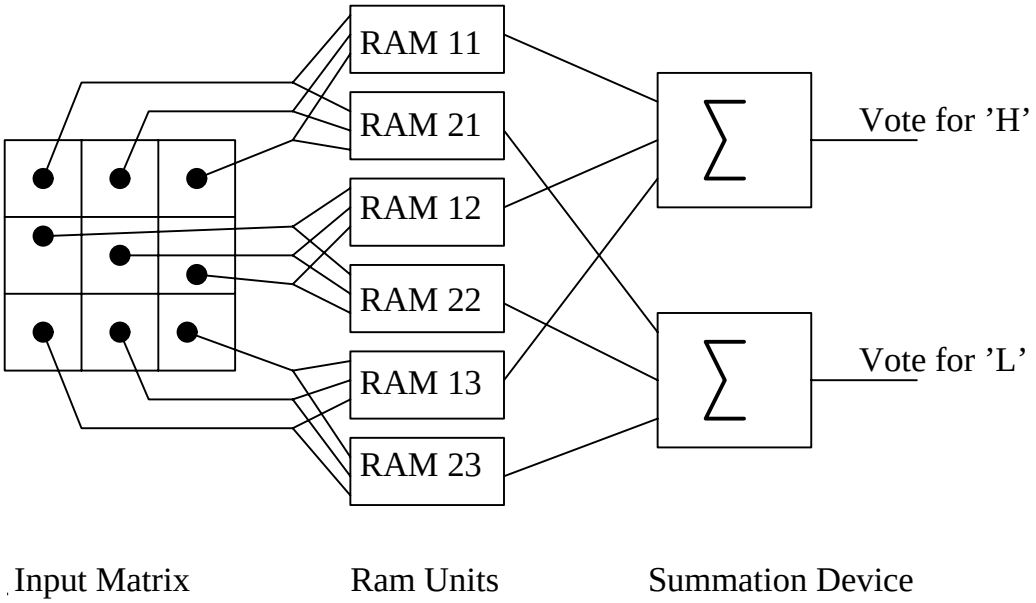


Figure 3.5: An Example Architecture Using RAM Elements.

resenting the simplified letters 'H' and 'L'. The test set consists of two distorted characters (see Figure 3.6).

After training the RAM units have the following filling:

	Address = Pattern							
	000	001	010	011	100	101	110	111
RAM11	0	0	0	0	1	1	0	0
RAM12	0	0	0	0	0	0	0	1
RAM13	0	0	0	0	0	1	0	0
RAM21	0	0	0	0	1	0	0	0
RAM22	0	0	0	0	1	0	0	0
RAM23	0	0	0	0	1	0	0	1

If the test patterns are supplied to the network the system gives the following response: For test pattern 1 the H-discriminator gives a response of '1' and the L-discriminator gives a '2'. For the second test pattern the H-discriminators give a '2' and the L-discriminator gives a '0'. If the threshold is set to two the distorted 'L' and 'H' are both correctly recognized.

3.5 Improvements on Logical Neural Networks

In the following section some new ideas in the field of logical neural networks are presented. The focus is particularly on concepts aimed on improving the basic

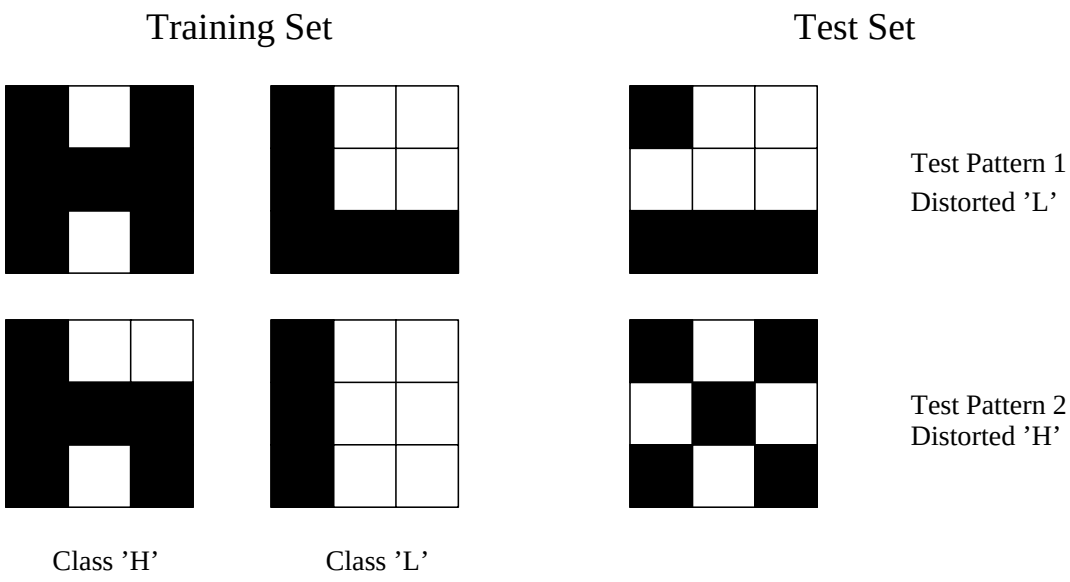


Figure 3.6: The Training and Test Sets.

processing unit.

In [canu95] a generalization process for weightless neurons is introduced, using a Radial-RAM. In this approach each of the adaptive nodes has the ability to generalize. The idea is directly based on simple RAM neurons and uses the same learning mechanism. The improvement is made in the recalling characteristics. Instead of just accessing the address that is assigned to an input pattern the region around the address is also read. The response is therefore dependent on all the learned patterns that are similar to the applied vector. Networks using this new type of neurons showed a better generalization performance than networks using simple RAM neurons.

Another attempt to improve logical NN uses probabilistic logic nodes (PLN) [alek95, 192f]. The PLN stores more than one bit of information at each address location. The content of an address determines the probability of firing.

A universal architecture for logical neural networks is discussed in [zhan91]. Most logical neural networks obtain their ability to generalize from the architecture of the system and from the training procedure used. The generalization desired may differ with the problem domain and therefore the optimal network structure

may differ as well[1].

The paper considers two extreme architectures used in LNNs: the single layer N-tuple RAM network and the multilayer pyramid architecture. The proposed universal architecture uses N-tuple networks as substructure within a pyramidal architecture. It also investigates how methods of spreading and training with noise can be used to improve the generalization performance. One observation was that N-tuple networks with a small number of inputs have difficulties in coping with noise.

A different approach to build better LNN is to improve the way the inputs are connected to the RAM units [alek89a, 202ff]. The most obvious method is to follow the biological model and use a genetic algorithm. This paper reports how the connections from an input retina to a logical neural network can be optimized with a genetic algorithm. The system with optimized connections proved to be superior to a randomly connected one.

[1]This also holds for connectionist Networks.

Chapter 4

Modularity

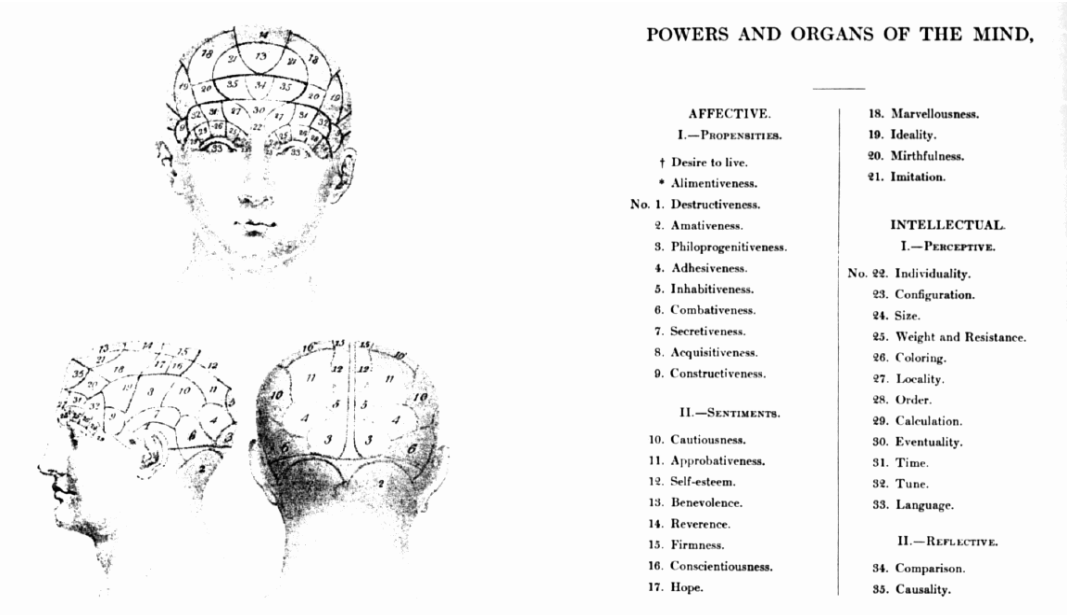
Modularity is a very important concept in nature. Modularity can be defined as subdivision of a complex object into simpler objects. The subdivision is determined either by the structure or function of the object and its subparts.

Modularity can be found everywhere; in living creatures as well as in inanimate objects. The subdivision in less complex objects is often not obvious.

At a very basic level electrons, positrons, and neutrons make the building blocks for any matter. At a higher level, atoms of elements are another form of simple modules of which everything is constructed. In living creatures proteins and on a higher level, cells could be seen as basic components. This idea of modules can be continued to more and more complex structures. Even looking at the universe planets can be seen as modules within a solar system.

Replication and decomposition are the two main concepts for modularity. These concepts are found in concrete objects as well as in thinking. It is often difficult to discriminate sharply between them; replication and decomposition often occur in combination.

Replication is a way of reusing knowledge. Once one module is developed and has proved to be useful it is replicated in a larger number. This principle is often found in living organisms. Observing a human this can be seen in a various scale: two similar legs, fingers, vertebra of similar structure, thousands of hair modules, and billions of cells. In electronics, the development of integrated circuits is based



POWERS AND ORGANS OF THE MIND,

AFFECTIVE.

I.—PROPENSITIES.

† Desire to live.
* Alimentiveness.
No. 1. Destructiveness.
2. Amativeness.
3. Philoprogenitiveness.
4. Adhesiveness.
5. Inhabitiveness.
6. Combativeness.
7. Secretiveness.
8. Acquisitiveness.
9. Constructiveness.

II.—SENTIMENTS.

10. Cautiousness.
11. Approbativeness.
12. Self-esteem.
13. Benevolence.
14. Reverence.
15. Firmness.
16. Conscientiousness.
17. Hope.
18. Marvellousness.
19. Ideality.
20. Mirthfulness.
21. Imitation.

INTELLECTUAL.

I.—PERCEPTIVE.

No. 22. Individuality.
23. Configuration.
24. Size.
25. Weight and Resistance.
26. Coloring.
27. Locality.
28. Order.
29. Calculation.
30. Eventuality.
31. Time.
32. Tune.
33. Language.

II.—REFLECTIVE.

34. Comparison.
35. Causality.

Figure 4.1: Spuzheim's Map of Brain Areas (1908).

on replication of simple units to build a complex structure.

Decomposition is often found when dealing with a complex task. It is a sign of intelligent behaviour to solve a complex problem by decomposing it into simpler tasks which are easier to manage and then reassemble the solution from the results of the subtasks. Constructing large software, building a car, or solving an equation are usually done by decomposing the problem.

4.1 Modularity in the Nervous System

The idea of a modular mind is very old. Researchers in brain theory have often tried to locate certain functions in the brain. An early attempt to develop a map of the mind was made by J.G. Spuzheim in 1908. In Figure 4.1 his idea is shown, taken from [kala92, p134]. The methods since then have changed dramatically; also the functions of modules which are proposed are different nowadays. In the following paragraphs a summary of the current view of the brain structure[1] is given.

[1] A lot of research in brain theory and anatomy of the nervous system is ambiguous. Perhaps people in one hundred years will look at this description in the same way we look at Spuzheims mind map.

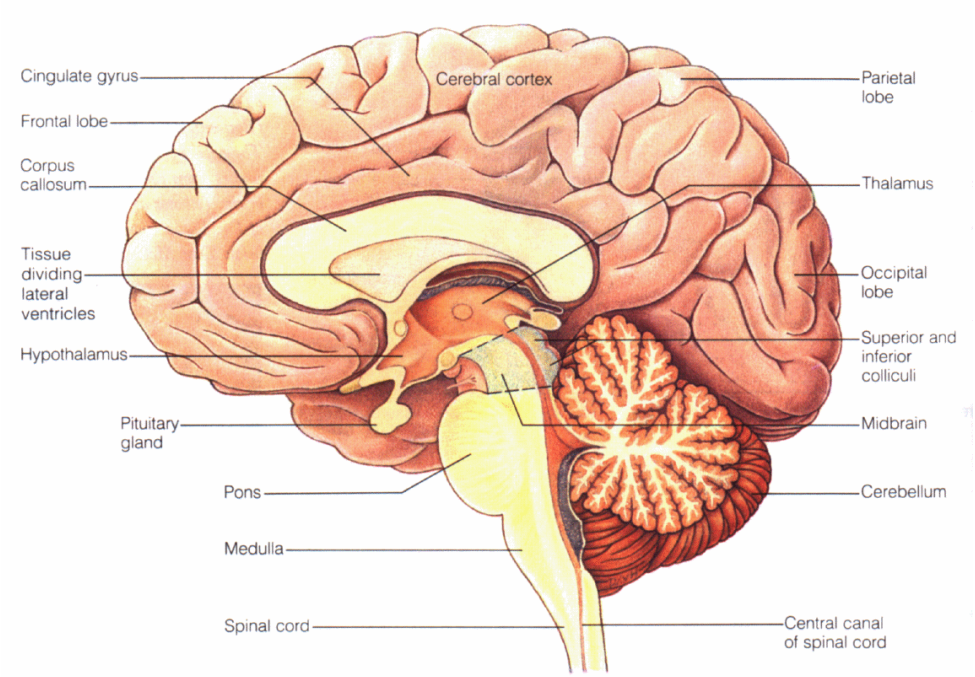


Figure 4.2: Anatomical Subdivision of the Human Brain.

The human central nervous system (CNS) can be subdivided into spinal cord, medulla oblongata, pons, midbrain, diencephalon, cerebellum, and the two cerebral hemispheres. All these parts have their own functions. Each region is interconnected with other parts of the brain. In Figure 4.2 a rough anatomic subdivision of the brain is given; the picture is taken from [barr88, p7].

The CNS is also connected by cranial nerves and by the spinal nerves with other regions of the human body. In the peripheral nervous system (PNS) sensory endings and effector endings are included. The interaction of the brain with the environment is realized by the PNS. The sensory nerves (afferent fibers) provide the CNS with information from the environment. The efferent fibers (motor neurons) control the muscles.

It is possible to divide the anatomic regions of the CNS further. The topology of the cerebral cortex is shown in Figure 4.3, from [kala92, p123]. In the cortex different regions have different higher brain functions. The modules should not be regarded as isolated parts; they are highly connected. The following description is according to [kala92] and [barr88].

The frontal lobe is the control unit for fine movement (the precentral gyrus).

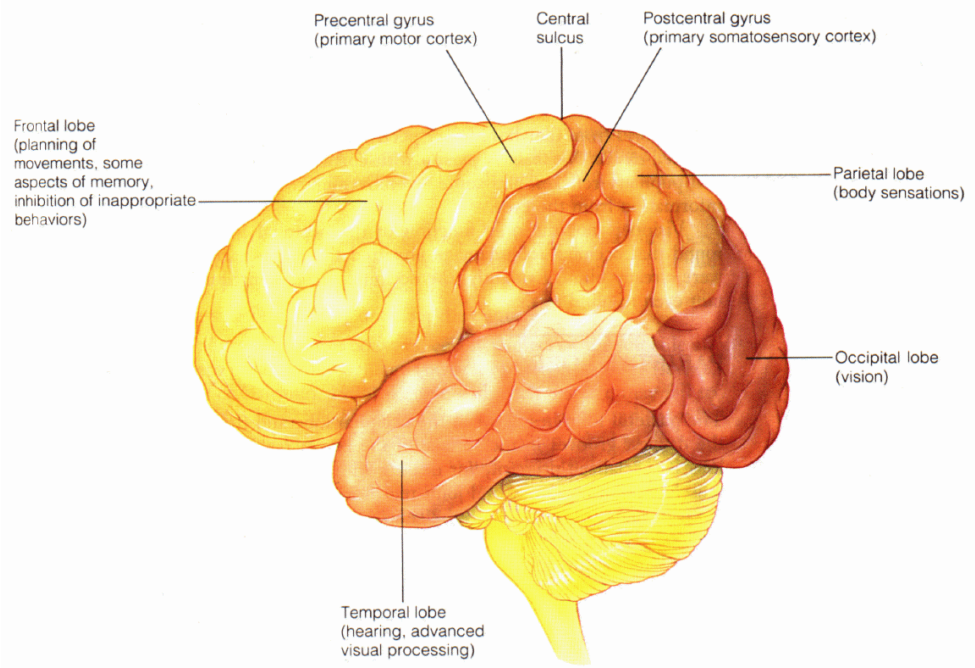


Figure 4.3: Functional Subdivision of the Cortex.

This part is also responsible for appropriate behaviour in accordance with the standards of the individual. Functions of memory and movement planning are located in this area, too.

The temporal lope is the primary processing unit for acoustic information. In this region complex recognition tasks, such as face recognition, are located. The comprehension of language is connected to this area (Wernicke's area).

The parietal lope deals with body sensation. Sensory information from all over the body is processed in the postcentral gyrus. The whole body is mapped onto this area.

The occipital lobe is the primary visual cortex. This area is mainly responsible for vision.

The CNS has also smaller functional units, called nuclei. These are clusters of neurons. Nuclei can be found in the most parts of the CNS. Each cranial nerve originates in a nucleus. This module is a unit to integrate the sensory input and to regulate the motor output.

Different areas in the brain have also a different cellular structure. Some parts consist mainly of cell bodies and dentrites (e.g. the surface of the cortex), this

is called gray matter. Other parts are containing mostly myelinated axons, these parts are called white matter.

A modern research method to locate functional units of the brain is to use Positron-emission tomography (PET). The test subject receives an injection of glucose with a radioactive label (with a very short half-live ranging). The regions which are more active need more energy; the concentration of glucose and therefore the concentration of the radioactive label will be higher in these parts than in other parts of the brain. Using PET these regions can be localized. It is assumed that the region which is most active, is the part of the CNS which deals with the task. For a more comprehensive description see [posn88].

The described anatomic structures underline the modular characteristics of the human brain on several levels. The connections and the structure of the modules and their interaction result in intelligent behaviour.

4.2 Modularity Derived from Psychology

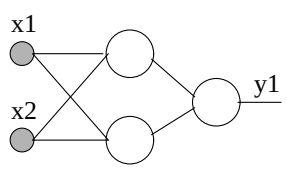
One cognitive task can involve different processes. Humans are able to do different processes in parallel. According to [eyse93, p4] this is the most likely explanation for the processing power of the brain. Most tasks involve a combination of serial and parallel processing.

It can be observed that while humans have the ability to do different things in parallel; some parallel tasks are easier than others. Most people have no problems with walking and talking in parallel; whereas they find listening to two different speakers at the same time very difficult. This implies that tasks which can be processed in different modules can be done easily in parallel, whereas tasks which need the same processing unit are difficult to manage concurrently.

Neuropsychologists study the information processing system of humans. They focus especially on patients with a partly damaged brain. The damage of a region of the brain has implications on the cognitive abilities which are situated in this area. The non-damaged parts are still working; in some cases their performance is

Training Set

x1	x2	y1
0	0	1
0	1	0
1	0	0
1	1	1



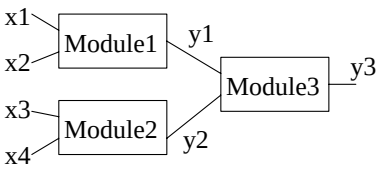


Figure 4.4: A Modular Solution for the 4-Bit-Parity Problem.

improved to compensate the loss in the other parts. These results suggest that the brain has a highly modular and parallel structure, [eyse93].

4.3 Modularity in ANN

The most used artificial neural networks have a monolithic structure. A lot of the models work on fully connected networks or layers (Hopfield or multilayer Perceptron). These networks perform well on a very small input space. However the complexity increases and the performance decreases rapidly with a growing input dimension [koho88]. The next paragraph gives a motivation for modular ANN.

4.3.1 Motivation: 4-Bit-Parity Problem

This is a small example to investigate the advantages of a modular structure over a monolithic network. A monolithic multilayer ANN can learn the 4-bit-Parity Problem. Due to the structure of this data set the learning will take rather long.

In Figure 4.4 a modular design and the adjusted training set is depicted. In this example it is obvious that it is much easier to train a single module (with 6 weights and 4 data tuples) and replicate this three times than to train the bigger monolithic MLP (with 26 weights and 16 data tuples). The task performed by the trained networks will be the same. The truth-table for the modular solution it is shown in Table 4.1.

In this example it can be seen that the modular approach is superior to a monolithic network.

x_1	x_2	x_3	x_4	y_1	y_2	$y_3 = y_{desired}$
0	0	0	0	1	1	1
0	0	0	1	1	0	0
0	0	1	0	1	0	0
0	0	1	1	1	1	1
0	1	0	0	0	1	0
0	1	0	1	0	0	1
0	1	1	0	0	0	1
0	1	1	1	0	1	0
1	0	0	0	0	1	0
1	0	0	1	0	0	1
1	0	1	0	0	0	1
1	0	1	1	0	1	0
1	1	0	0	1	1	1
1	1	0	1	1	0	0
1	1	1	0	1	0	0
1	1	1	1	1	1	1

Table 4.1: The 4-Bit-Parity Problem: The Truth-Table.

The main problem remains: how to chose modules, how to structure the problem? In the example above human design knowledge was used to restructure the problem and the data set for the modular solution. This approach is only applicable for a very limited domain of problems.

In [boer92] a genetic algorithm was used to find an optimal network architecture. The problem is that the number of possible network structures is huge even for very simple problems. With no structural limitations to the network architecture the complexity is too great. Therefore the problem can not be solved with reasonable computational power.

The use of a genetic algorithm offers enormous potential if the search space could be limited. This approach is biologically very plausible. It is very likely that connections in the brain have developed as a result of the evolutionary process.

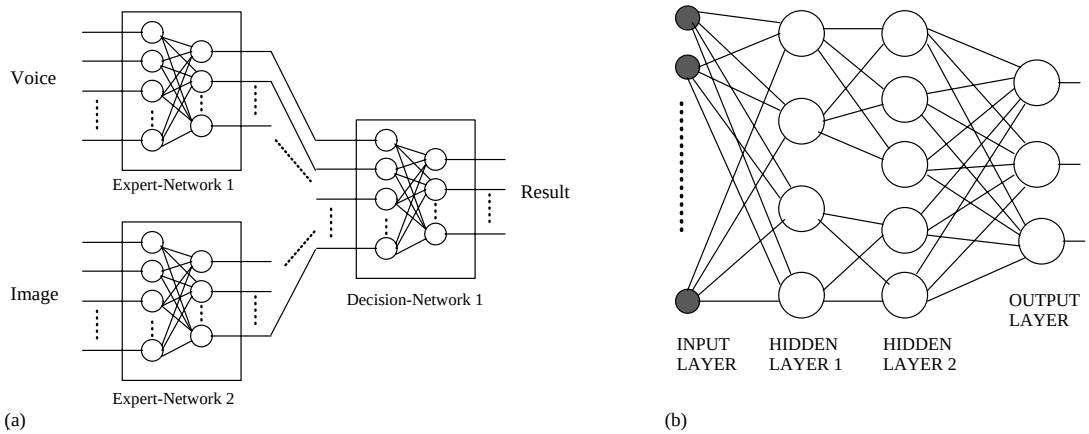


Figure 4.5: (a) A Multiple Neural Network
(b) A Modular Architecture

4.4 Multiple and Modular Artificial Neural Networks

The term *Multiple Neural Networks* is used for strongly separated architectures. Each of the networks works independently on its own domain. The single networks are built and trained for their specific task. The final decision is made on the results of the individual networks, often called expert networks or agents. The decision system can by implemented in many different ways; depending on the problem a simple logical majority vote function, another neural network, or a rule based expert system may be employed.

Multiple NNs are used if different information source (different sensors) are available to give information on one object. Another method to use multiple neural networks is to preprocess the input data in different ways (e.g. different filters) and train a network on each preprocessed input. The networks within the multiple network can be of any architecture.

Each network can be seen as an expert on its domain. The individual network is trained on its domain only. The output by a single network is according to its specific input. In the example given in Figure 4.5(a) one of the networks is trained to identify a person by voice while the other network is trained to identify the person by vision.

The outputs of the expert networks are the input data of the decision network which is trained after the expert networks have been trained. The decision is made according to the outputs of the experts, not directly from the input data. For further examples on such network architectures see [patt96, p17, p235].

The term *Modular Neural Networks* (MNN) is very fuzzy. It is used for many different structures. Everything that is not monolithic seems to be modular. In the next paragraph two types of modular NN are briefly introduced.

One idea of a modular neural network architecture is to build a bigger network by using modules as building blocks.

All modules are neural networks. The architecture of a single module is simpler and the sub-networks are smaller than a monolithic network. Due to the structural modifications the task the module has to learn is in general easier than the whole task of the network. This makes it easier to train a single module.

In a further step the modules are connected to a network of modules rather than to a network of neurons. The modules are independent to a certain level which allows the system to work in parallel.

For this modular approach it is always necessary to have a control system to enable the modules to work together in a useful way.

Another idea of modularity is a not-fully connected network. In this model the structure is more difficult to analyze, for an example see Figure 4.5(b). A clear division between modules can not be made. A module is seen as a part of the network that is locally fully connected. This modular approach is biologically very plausible. Experiments with this structure are described in [boer92].

4.5 Publications on Aspects of Modular Neural Network

There are many articles and papers published in the field of neural computing. In the following section some articles which are of major interest for the project are reviewed, in particular those dealing with modular and multiple neural networks.

4.5.1 Modularity as a Powerful Concept

An interesting investigation of the relation between structure and function of modular neural networks is given in [happ94].

The article "Design and Evolution of Modular Neural Network Architectures" examines the structural evidence for a modular architecture in the human brain which is given by different psychologists, biologists, and neurologists. Several levels of modularity in the brain are described. Human multitasking abilities and disabilities are explained with the modular and parallel structure of the brain.

Happle and Murre conclude "[...]that the nature of information processing in the brain is modular. Individual functions are broken up into subprocesses that can be executed in separate modules without mutual interference." [happ94, p984]. They further speculate that there is a process of subdividing modules into submodules and tasks into subtasks up to a very basic level, and that the modular architecture of the brain, which has developed in a long evolutionary process, may be the key issue for this division of complex tasks into subtasks.

They suggest building more modular artificial neural networks which are similar to the modular structure of the brain. These new architectures may then increase the ability of the network to solve more complex real world problems.

The authors point out that most well established ANN do not have any preimposed modular structure. These networks suffer from problems in convergence and highly increased training expenditure for high dimensional input spaces. It is also addressed that the generalization ability decreases in huge ANNs.

Following this motivation for a modular architecture, a new network structure is introduced. The basic building block in this network is the CALM (Categorization And Learning Module), which works on a competitive and unsupervised basis and has the ability to differentiate input patterns in different categories. For a very detailed description of the CALM see [murr92, p15ff].

The article also discusses the process of adaption, and describes the following basic biological principles:

- Evolution as slow process of high level adaption over generations.
- Ontogenesis and Learning as intermediate levels of adaption.
- Neural Activation as the lowest and fastest level of reaction to changes in the environment.

The evolutionary idea concludingly suggests the use of genetic algorithms to optimize the structure of a modular neural network. For a further investigation of this issue see [boer92].

The evidence for a modular design in natural neural systems pointed out in this article and in several reference given in the paper is sufficiently encouraging to develop and test ideas of modular structures within the field of artificial neural computing.

The problems occurring in large conventional ANNs such as multilayer feedforward networks or Hoppfield networks mentioned by the authors are very significant, especially for the simulation of huge brain-like structures. The modular approach seems very promising for tackling those problems.

Learning and forgetting as a concept to develop modules is described in [ishi95]. A technique called "Learning of modular structured networks" is proposed to overcome the problem of training large-scale networks. The learning occurs in two stages, first the connections in the small modules are learned; then the inter-modular connections are established. Modules that have already learned a function can also be employed elsewhere within the network.

Modularity beyond neurons is shown in [svan92]. The basic units in this approach are blocks of synapses, blocks of neurons, and a control block. The system is realized as a hardware implementation. It is more flexible than a model which uses only neurons.

4.5.2 Reducing the Complexity of the Problem

A very common method in MNNs is to construct an architecture that supports a division of the complex task into simpler tasks.

Basit Hussain and M. R. Kabuka present a new architecture in "A Novel Feature Recognition Neural Network and its Application to Character Recognition" [huss94]. They implemented the idea of decomposing the task of recognition. A complex object (a character) can be detected by recognizing its subpatterns.

The network is based on a two level detection scheme. In the first level subpatterns are recognized. Multiple detectors are used for each segment to recognize shifted sub-patterns. The second level detects the class according to the information provided by the first level.

The training algorithm which is calculating the weights directly from the input patterns is very simple. The training is therefore very fast.

The tests showed that the network is capable of recognizing distorted, noisy, scaled, and shifted patterns. The complexity of the proposed architecture is much less than in a comparable Neocognitron.

A Divide-and-Conquer Methodology is proposed by Chiang and Fu [chia94]. This works on an architecture consisting of two general components; a Divide-and-Conquer Engine built out of subnetworks capable of managing a subset of the data, and an Integration Engine, that determines which of the subnetwork outputs will be used as the final output of the system.

The training set is divided according to the training error value. Each subnetwork learns its partition of the training set. This process is repeated until the modules in the Divide-and-Conquer Engine can learn the subset with a sufficiently small overall error. The Integration Engine is trained to determine to which partition an input vector belongs.

The article "A Parallel and Modular Multi-Sieving Neural Network Architecture for Constructive Learning" by Lu et al. introduces an approach to modular neu-

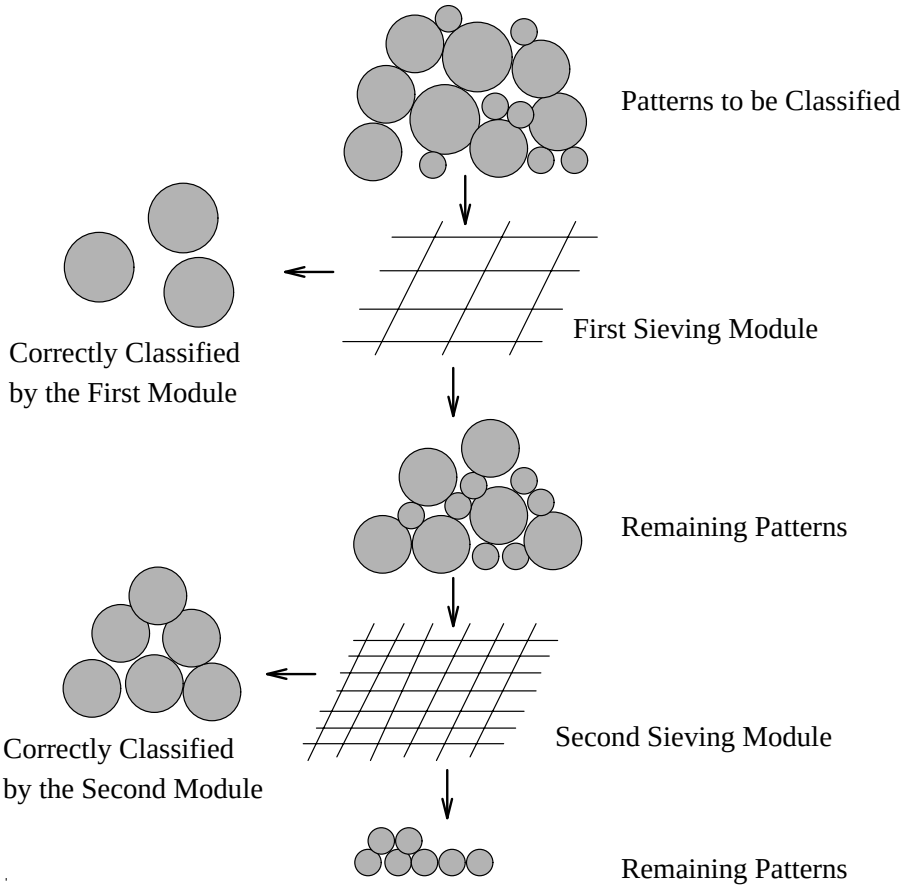


Figure 4.6: The Basic Idea of a Sieving Algorithm.

ral networks, which segments the training data automatically and then employes different modules to handle different subsets [lu95].

The basic idea of this network is based on a multi-sieving learning algorithm, depicted in Figure 4.6. The patterns are classified by this algorithm on different levels. In the first level - *a very rough sieve* - some patterns may be recognized correctly while others will not. The correctly classified samples are taken out of the training set. The next level - *a less rough sieve* - is only trained on the remaining ones. After the training of this level the correctly recognized patterns are removed from the training set. The remaining patterns form the training set for the next level. This process is repeated untill all patterns are classified correctly.

Each level of learning - *each sieve* - generates a neural network with the ability to recognize a subset of the original training set. These networks, called *sieving modules*, face a simpler recognition task than the whole problem.

The modules are considered in a hierarchical order. The output of the first module classifying the pattern will be taken as as the response of the system; only if the current module can not classify the pattern the network on the next lower level will be asked. To determine whether the output for an unknown input is accepted as classification or not a 1-out-of-N coding is used.

All modules work in parallel; therefore the speed of the system is nearly independent of the number of modules involved in the recognition process.

The authors state that the retraining of the system with a slightly modified training set is very easy.

One of the main advantages of this system is that the decomposition of the problem and the generation of an appropriate network is made by the algorithm and not by the user.

Furthermore the idea of sieving seems very intuitive to human understanding and thinking. The concept of a self growing and massive parallel system is very promising.

The problem of deciding whether the classification of an unknown input pattern on a certain level is correct is not easy. The 1-out-of-N coding is a way to decrease the probability that outputs are classified as valid by chance; the algorithm will have to show if this is sufficient for real world data sets.

The idea of reducing the input dimension on single modules within a network is introduced in [kim94]. The system for a N-dimensional input vector consists of N competitive modules. Each of the modules is connected only to $(N-1)$ of the inputs; and each input is connected to $(N-1)$ modules. The competitive learning in each module is now based on slightly different $(N-1)$-dimensional input vectors. In a second stage a recognition layer decides according to the outputs of all submodules.

It is shown that this parallel architecture is superior to a single N-dimensional competitive network.

4.6 Modular Neural Network Applications

In recent years modular neural networks have became more popular for applications in various areas, some examples are presented here.

In the article [kwon94] the usage of a modular neural network for function approximation is described.

A stock market prediction system using MNN is presented in [kimo90]. The system is based on expert modules, realized as MLFFNs. Each expert has its own input domain and preprocessing unit. A final postprocessing unit combines the results from all the modules to an overall output.

An adaptive MNN for character recognition is introduced in [mui94], and it is demonstrated that the modular topology has a high fault tolerance.

Bellotti et. al. describe a network consisting of a self organizing map and a MLP [bell95]. It is used to separate the signal from the background noise in a cosmic ray space experiment.

A similar modular structure is suggested in [blon93]. The task is the classification of 'remote senced data'. The architecture uses an unsupervised module to compress the high dimensional input data and a MLP as a classifier.

A description of a patient independent ECG recognition system based on a modular connectionist NN is found in [farr93]. A combination of associative and competitive learning is used.

Chapter 5

A New Modular Neural Network

The aim of the project was to develop a parallel and fault tolerant modular network architecture. The focus on parallelism and fault tolerance was motivated by the structure of the human nervous system.

In this chapter the evolution of the model is shown; then the proposed network together with a training algorithm is described.

In the last part of this chapter some aspects of the proposed model are analysed and discussed.

5.1 The Evolution of the Model

In the early stage of the project a large number of possible network structures were considered. The following list shows the main design questions that had to be decided before setting up the model.

- Is the network homogenous or heterogeneous?

  Are all modules of one type or are there different types of modules used?

- What type of architecture is used for the modules?

  MLP, LNN, SOM, ART1, ART2

- How are the modules interconnected?

  Are only feedforward connections used, are recurrent connections allowed, are

connections only made from one layer to next?

- What is the general network structure?
  Has the network a single layer, n-layers, or pyramidal architecture?
- How are the inputs connected to the network?
  Are the inputs connected to all modules (overlapping) or only to one module (non-overlapping)? Are these connections made randomly or is information about the input space used (e.g. locally connected)?
- What training algorithm is used for the modules?
  Is a supervised or unsupervised learning method used?
- How is learning organized for the whole network?
  Does the network learn in stages; is noise used during the training?

From the list above it can be seen that the number of possible network structures is huge. In Figure 5.1 four example architectures are given.

The network shown in Figure 5.1(a) is a heterogeneous architecture consisting of two modules. One is a self-organizing map (SOM) and is used to reduce the input dimension; the other one is a multilayer Perceptron that works on the reduced input space. This architecture is already used in different applications; one example is shown in [bell95].

To train a MLP on the output of logical adaptive nodes is one idea for another heterogeneous structure, see Figure 5.1(b).

The distribution of output values may have important information about the input pattern, which is not used in a standard LNN. The suggested architecture uses all the outputs as input vector for a MLP; this might increase the performance of the system. To the best of my knowledge there is currently no published work on this type of network.

A homogeneous pyramidal structure is shown in Figure 5.1(c). This is similar to an architecture proposed by Aleksander [alek89b], but uses small MLPs instead of

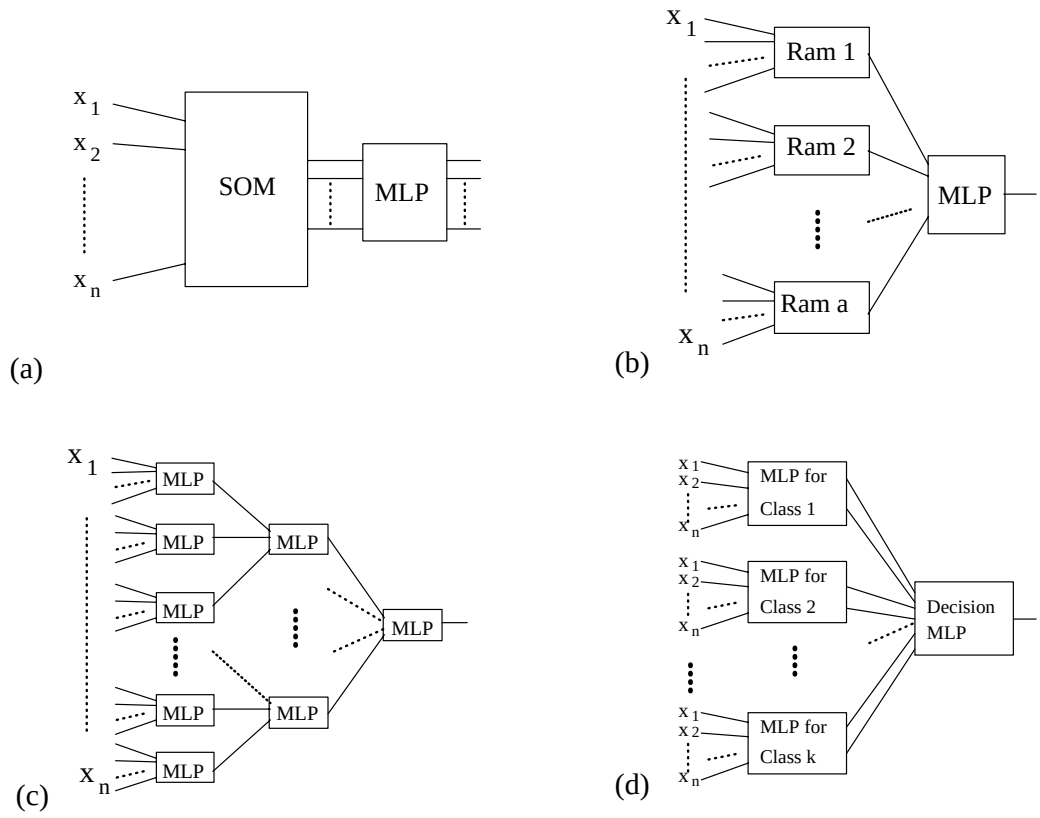


Figure 5.1: Some Examples of Modular Architectures.

logical nodes. This should provide fast training of the network as well as improved generalization ability. To the best of my knowledge there is no research published on this topic.

The final network shown in Figure 5.1(d), uses a different approach. The input and output dimensions of the problem remain the same.

Each subnetwork is only concerned with a single class. For any input vector, the output of a particular module indicates whether that input is recognized as belonging to the relevant class or not. This structure is investigated in [anan95] and [zhao95].

In the process of developing a modular neural network a number of other ideas were considered. These included using partly overlapping modules or different training data sets. Other ideas focused on the connections between the inputs and the modules, including the use of statistical methods or the entropy of the input attributes to structure these connections.

Due to the time constraints involved (less than six month) the project concentrated on one structure. The architecture used is a less general version of the network depicted in Figure 5.1(c).

5.2 The Network Architecture

The proposed network system consists of a layer of input modules and an additional decision module. All sub-networks are MLPs. Only the number of inputs and the number of outputs of the module is determined by the system. The internal structure, such as the number of hidden layers and the number of neurons in each hidden layer can be chosen independent of the overall architecture.

Each input variable is connected to only one of the input modules. These connections are chosen at random. The outputs of all input modules are connected to the decision network. In the discussion that follows the dimension of the input vector is denoted by l and the number of classes by k.

To determine a modular network it is necessary to specify either the number of inputs per input module or the number of input modules. These parameters are dependent on each other. It is assumed here that the number of inputs per module in the first layer is chosen as n; the number of input modules in the input layer can therefore be calculated as $m = \lceil \frac{l}{n} \rceil$.

It is further assumed that $l = m \cdot n$. If this is not the case the spare inputs can be connected to constant inputs; in the implementation of the model all 'free' inputs were connected to the constant value '0'. Alternatively, it would be possible to alter the size of one module or of a group of modules.

Each module in the input layer can have either k or $\lceil \log_2 k \rceil$ outputs. The network with k intermediate outputs is refered to as *large* intermediate representation. It only is useful if the number of classes is very small. For problems with a larger number of classes the *small* intermediate representation ($\lceil \log_2 k \rceil$) is more appropriate.

From an information theory point of view the small intermediate representation

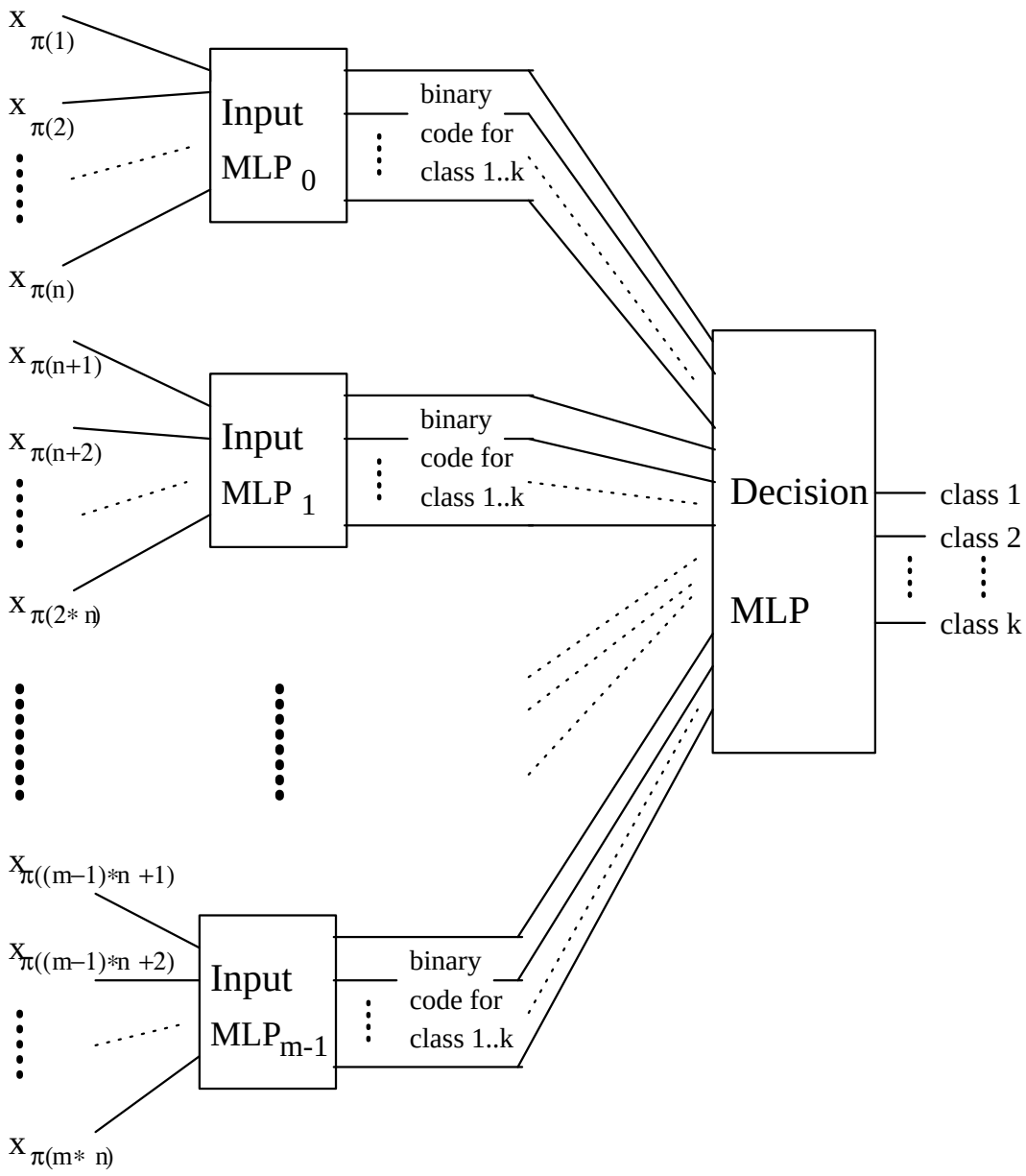


Figure 5.2: The Proposed Modular Neural Network Architecture.

should be sufficient because only this number of output neurons is required to represent all the classes in a binary code.

The decision network has $(m \cdot k)$ or $(m \cdot \lceil \log_2 k \rceil)$ inputs, dependent on the intermediate representation used. The number of outputs is k, one output neuron for each class. The structure of the modular network is depicted in Figure 5.2 using a small intermediate representation. The function $\pi : X \mapsto X$ gives a permutation on $X = \{1 \ldots l\}$. This permutation is randomly chosen and constant for a network.

Definition: A Module

A module is a multilayer feedforward neural network defined by a 3-tuple:

$$\mathcal{M} = (a, b, \mathcal{H})$$

Where a is the number of inputs of the module, b is the number of output nodes, and $\mathcal{H}$ is a list containing the numbers of neurons in each of the hidden layers.

Example:

A multilayer Perceptron module $\mathcal{M}$ with eight inputs, twelve neurons in the first hidden layers, ten neurons in the second hidden layer, and four outputs is described as: $\mathcal{M} = (8, 4, [12, 10])$.

Definition: A Modular Neural Network

A modular neural network is a set of interconnected modules defined by a 7-tuple.

$$N = (l, k, m, r, \pi, \mathcal{I}, \mathcal{D})$$

Where l is the number of inputs, k the number of classes, m the number of modules in the input layer, r is the type of the intermediate representation ($r \in \{small, large\}$), π is the permutation function, $\mathcal{I}$ is the input layer module, and $\mathcal{D}$ is the decision module.

Example:

A modular network with 210 inputs, four classes, 15 input modules (each having 14 inputs, one hidden layer with six neurons and two outputs), a small intermediate

representation, a permutation function p, and a decision network (with 30 inputs, one hidden layer with ten neurons and 4 output neurons) is described as: $N = (210, 4, 15, small, p, (14, 2, [6]), (30, 4, [10]))$.

5.3 Training the System

Training occurs in two stages, using the Backpropagation algorithm described in section 2.3.

In the first phase all sub-networks in the input layer are trained. The individual training set for each sub-network is selected from the original training set and consists of the components of the original vector which are connected to this particular network (as an input vector) together with the desired output class represented in binary or 1-out-of-k coding.

In the second stage the decision network is trained. To calculate the training set each original input pattern is applied to the input layer; the resulting vector together with the desired output class (represented in a 1-out-of-k coding) form the training pair for the decision module.

To simplify the description of the training a *small* intermediate representation is used, further it is assumed that the permutation function is the identity $\pi(x) = x$.

The original training set TS is: $(x_1^j, x_2^j, \ldots, x_l^j; d^j)$, and where $x_i^j \in I\!R$ is the ith component of the jth input vector, d^j is the class number, $j = 1, \ldots, t$, where t is the number of training instances.

The module MLP_i is connected to:

$x_{i \cdot n+1}, x_{i \cdot n+2}, \ldots, x_{(i+1) \cdot n}$

The training set TS_i for the module MLP_i:

$(x^j_{i \cdot n+1}, x^j_{i \cdot n+2}, \ldots, x^j_{(i+1) \cdot n}; d^j_{BIN})$

for all $j = 1, \ldots, t$, where d^j_{BIN} is the output class d^j represented in a binary code.

The mapping performed by the input layer is denoted by:

$\Phi : R^{n*m} \mapsto R^{m*\lceil \log_2 k \rceil}$

The Training Algorithm:

- Stage 1 – Training the Input Layer:
  1. Select the training sets TS_i from the original training set TS, for all $i = 0 \ldots m-1$.
  2. Train all modules MLP_i on TS_i using the BP algorithm.
- Stage 2 – Training the Decision Network
  1. Calculate the response r of the first layer for each input vector j.
     $r^j = \Phi((x_1^j, x_2^j, \ldots, x_l^j))$.
  2. Build the training set for the decision network
     $TS_d = \{(r^j; d_{BIT}^j) | j = 1, \ldots, t\}$
  3. Train the decision network on the set TS_d using the BP algorithm

Figure 5.3: The Training Algorithm.

The training set for the decision network:
$(\Phi((x_1^j, x_2^j, \ldots, x_l^j)); d_{BIT}^j)$ and $j = 1, \ldots, t$. Where d_{BIT}^j is the output class d^j represented in a 1-out-of-k code.

The mapping of the decision network is denoted by:
$\Psi : R^{m*\lceil \log_2 k \rceil} \mapsto R^k$

The training algorithm is summarized in Figure 5.3.

The training of each module in the input layer is independent of all other modules so this can be done in parallel. The training is stopped either when each module has reached a sufficient small error or a defined maximum number of steps has been performed. This keeps the modules independent.

Alternatively training can be stopped if the overall error of all modules is suffi-

ciently small or the number of maximum steps has been performed. This assumes that the training occurs step by step simultaneously in all modules.

5.4 Calculation of the Output

The calculation of the output also occurs in two stages. First the input subvectors for each module are selected from the applied input vector according to the permutation function and the intermediate output is calculated by all modules. In a second stage all the outputs from the input layer are used as input to the decision network; then the final result is computed.
The mapping of the whole network is denoted by:

$$\Phi \circ \Psi : R^l \mapsto R^k$$

The response r for a given test input $(a_1, a_2, \ldots, a_l)$ is determined by the following function:

$$r = \Psi(\Phi(a_1, a_2, \ldots, a_l))$$

The k-dimensional output of the decision module is used to determine the class number for the given input. In the experiments the output neuron with the highest response was chosen as the calculated class. The differences between this neuron and the runner-up may be taken as a measure of accuracy.

5.5 Analysis of the New Model

In this section some aspects of the model are considered. This analysis presents some background to the network behaviour, and also addresses the question: *why does the network work?* – focusing is on aspects of speed, learning and generalization. Theoretical limitations of the model are also considered.

The structure of the proposed network is based on modules with each input being connected to a single module. This feature results in a faster training procedure and advances the generalization performance, but also introduces certain

limitations. The number of connections within the modular network is significant less than in a comparable monolithic architecture. A concrete comparison is given in chapter 7.

5.5.1 A Fast Network

For the analysis the following assumptions are made:

- The modules used in the modular neural network have only one hidden layer with four neurons.
- The long intermediate representation is used for the modular neural system (for the short representation the number of weights is even smaller).
- The function ν takes a module and returns the number of weight connections. (e.g. $\nu(M) = i$, where $M = (a, b, [h])$ and $i = a \cdot h + h \cdot b$ for a one hidden layer module and where $M = (a, b, [h_1, h_2])$ and $i = a \cdot h_1 + h_1 \cdot h_2 + h_2 \cdot b$ for a two hidden layer network.)
- The function τ represents the time needed for training a module. The analysis assumes that τ is dependent only on the number of weights and is monotonously increasing $i > j \Rightarrow \tau(i) > \tau(j)$. This means the training time is longer if there are more weights in the network. In a real system it is likely to be dependent on other parameters as well.

Training the new network architecture is faster than training a monolithic modular network on the same problem for three reasons:

1. The number of connections in the modular network and hence the number of weights, is much less than in a monolithic MLP. Fewer weights lead to fewer operations during the BP-training. This results directly in a speed-up of the learning procedure.

   Consider a modular network with ten input modules, each with n inputs, $h_m = 4$ hidden layer neurons, and k outputs. The decision module has

$(10 \cdot k)$ inputs, $h_m = 4$ hidden layer neurons, and k outputs. This is denoted by: $M_{mod} = (l, k, 10, (n, k, [h_m]), (10, k, [h_m]))$, where $l = 10 \cdot n$.

A monolithic network with the same number of inputs and outputs, and with two hidden layers, each with h_s neurons can be denoted by: $BP = (l, k, [h_s, h_s])$

For the number of neurons to be equal in the two networks:

$$10 \cdot (h_m + 1) + h_m + k = 2 * h_s + k$$

$$\Rightarrow h_s = \frac{11h_m + 10}{2} = 27$$

The number of weights in each network is:

$$\nu(BP) = 27 \cdot l + 27 \cdot k + 729$$

$$\nu(M_{mod}) = 4 \cdot l + 84 \cdot k$$

If the input is sufficiently large so that:

$$l > 32 + 2.5 \cdot k$$

then

$$\nu(BP) > \nu(M_{mod})$$

and hence

$$\tau(\nu(BP)) > \tau(\nu(M_{mod}))$$

since τ monotonously increasing.

2. The modules in the input-layer are mutually independent, so the training can be performed in parallel. The training time for a full parallel implementation is the maximum time needed for training one of the input modules plus the time to train the decision module. Therefore the number of weights that have to be regarded as time factor in a parallel training is only the number of weights in an input module plus the number of weights in the decision

module. Assuming M_i is a module in the input layer and M_d is the decision module the training time T can be calculated as follows:

$$T = \tau(\nu(M_i)) + \tau(\nu(M_d))$$

Assuming the example from above (M_{mod} and BP) the speed-up is significant. The number of inputs per module is assumed to be larger than eight ($n > 8$).

The number of weights to consider for training in each network is:

$$\nu(BP) = 27 \cdot l + 27 \cdot k + 729$$

$$\nu(M_{mod}) = 2 \cdot (4 \cdot l + 4 \cdot k)$$

The ratio between the numbers of weights to train ($k = 2, n > 8, l = 10 \cdot n$):

$$\frac{\nu(BP)}{\nu(M_{mod})} = \frac{270 \cdot n + 783}{8 \cdot n + 16} > \frac{270 \cdot n + 783}{10 \cdot n} > \frac{270 \cdot n}{10 \cdot n} = 27$$

The number of weights to consider for the time need to train the network is at least 27 times less than in a monolithic MLP.

3. Splitting the training vector into parts often helps to focus on common attributes. Consider the following (admittedly contrived) example:

Original Set		Set MLP_1		Set MLP_2	
$x_1x_2x_3x_4x_5x_6$	y	$x_1x_2x_3$	y	$x_4x_5x_6$	y
0 0 0 0 0 1	0	0 0 0	0	0 0 1	0
0 0 0 0 1 0	0	0 0 0	0	0 1 0	0
0 0 0 1 0 0	0	0 0 0	0	1 0 0	0
1 1 0 1 0 1	1	1 1 0	1	1 0 1	1
1 0 1 1 0 1	1	1 0 1	1	1 0 1	1
0 1 1 1 0 1	1	0 1 1	1	1 0 1	1

Class '0' is determined by x_1, x_2, and x_3; which will be learned very quickly by MLP_1, which sees this tuple '0 0 0 : 0' three times during one training cycle.

Similarly class '1' is determined by the x_4, x_5, and x_6, which will be quickly learned by MLP_2.

It is unlikely the that a real world data set has the same structure as the example. However, it can be conceivably produce significant improvements, particularly with large input dimensions.

5.5.2 Learning Problems

Splitting-up the training set into subsets can also bring problems. The number of equal input vectors with different possible output values may increase, especially for modules with a small number of input variables. Consider the worst case: a 4-Bit-Parity problem:

Original Set		Set MLP_1		Set MLP_2	
$x_1x_2x_3x_4$	y	x_1x_2	y	x_3x_4	y
0 0 0 0	0	0 0	0	0 0	0
0 0 0 1	1	0 0	1	0 1	1
0 0 1 0	1	0 0	1	1 0	1
0 0 1 1	0	0 0	0	1 1	0
0 1 0 0	1	0 1	1	0 0	1
0 1 0 1	0	0 1	0	0 1	0
0 1 1 0	0	0 1	0	1 0	0
0 1 1 1	1	0 1	1	1 1	1
1 0 0 0	1	1 0	1	0 0	1
1 0 0 1	0	1 0	0	0 1	0
1 0 1 0	0	1 0	0	1 0	0
1 0 1 1	1	1 0	1	1 1	1
1 1 0 0	0	1 1	0	0 0	0
1 1 0 1	1	1 1	1	0 1	1
1 1 1 0	1	1 1	1	1 0	1
1 1 1 1	0	1 1	0	1 1	0

Assuming two input modules; MLP_1 connected to x_1 and x_2, MLP_2 connected to x_3 and x_4. The resulting training set for each of the two input modules has 16 2-Bit vectors, each of the four different vectors appearing twice with the desired output '1' and twice with the desired output '0'. It is impossible for the individual modules to distinguish these cases, so after training the module response will be '0.5' for **any** input pattern. All the information has been lost in the input layer, so no decision is possible.

To discuss this problem more general the following definition is necessary. $P(y = a)$ is the probability of the output variable y having the value a. And

$P(y = a|x = b)$ is the conditional probability of $y = a$ if $x = b$.

Definition: A Set of Statistically Neutral Variables

Consider the function $f(x_1, \ldots, x_n, \ldots, x_m) = y$. A subset of the input $\{x_1, \ldots, x_n\}$ with $n < m$ is called *statistically neutral* if the knowledge of the values of these variables does not increase the knowledge of the result y.

Formally: The set of input variables $\{x_1, \ldots, x_n\}$ is statistically neutral if:

$$P(y = a) = P(y = a|x_1 = b_1 \wedge \ldots \wedge x_n = b_n)$$

If the whole set of input variables and all possible subsets for all modules are statistically neutral, the network can not learn the task. If only some of the modules are supplied with a statistically neutral set of input variables the network may perform satisfactory.

In [ston95] it is shown that monolithic multilayer feedforward networks can learn statistically neutral problems but they can not generalize on them.

This situation sounds very unlikely in real world data, and the expectation was observed during experimentation.

Especially in tasks with a large input space, such as picture recognition, this problem can be ignored. In tasks with a small number of input attributes, one way to reduce the possibility of having statistically neutral input sets is to recode the data in a sparse code.

5.5.3 On Generalization

The ability to generalize is the main property of neural networks. This is how neural networks can handle inputs which have not been learned but which are *similar* to inputs seen during the training phase. The proposed architecture combines two methods of generalization.

One method is built-in to the MLP. Each of the networks has the ability to generalize on its input space. This type of generalization is common to connectionist systems.

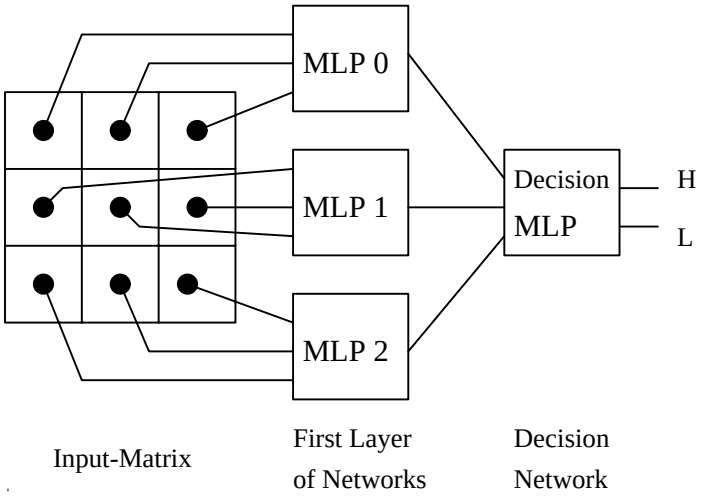


Figure 5.4: An Example Architecture.

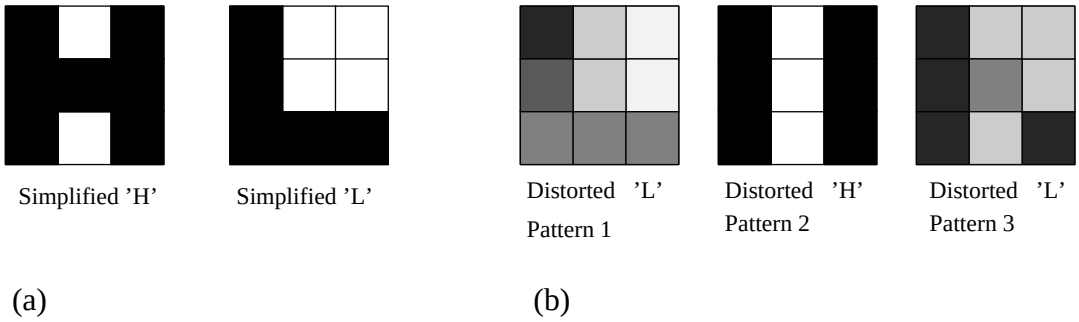


Figure 5.5: (a) The Training Set.
(b) The Test Set.

The second method of generalization is due to the architecture of the proposed network. It is a way of generalizing according to the similarity of input patterns. This method is found in logical neural networks [patt96, p172ff].

To explain the behaviour more concretely consider the following simplified example of a recognition system.

A 3x3 input retina with the architecture shown in Figure 5.4 is assumed. Each of the nine inputs reads a continuous value between zero and one, according to the recorded gray level (black=1; white=0).

The network needs to be trained to recognize the simplified letters 'H' and 'L', using the training set is shown in Figure 5.5(a). The desired output of the input networks is '0' for the letter 'H' and '1' for the letter 'L'.

The training subsets for the networks MLP_0, MLP_1, and MLP_2 are:

MLP_0	MLP_1	MLP_2
(1,0,1;0)	(1,1,1;0)	(1,0,1;0)
(1,0,0;1)	(1,0,0;1)	(1,1,1;1)

After training of the first layer of networks it is assumed that the calculated output is equivalent to the desired output. The resulting training set for the decision network is:

$$(\Phi(1,0,1,1,1,1,1,0,1);1,0) = (0,0,0;1,0)$$
$$(\Phi(1,0,0,1,0,0,1,1,1);0,1) = (1,1,1;0,1)$$

After the training of the decision network the assumed response of the system to the training set is:

$$r_H = \Psi(\Phi(1,0,1,1,1,1,1,0,1)) = \Psi(0,0,0) = (1,0)$$
$$r_L = \Psi(\Phi(1,0,0,1,0,0,1,1,1)) = \Psi(1,1,1) = (0,1)$$

To show different effects of generalization three distorted characters, shown in Figure 5.5(b) are used as the test set:

The first character tests generalization within the input modules, the second tests the generalization on the number of correct sub-patterns, and the third character is a combination of both. (The figures in the input vectors are according to the gray-level in the pattern; the outputs are taken from a typical neural network).

$$\begin{aligned} r_1 &= \Psi(\Phi(0.9,0.2,0.1,0.7,0.2,0.1,0.5,0.5,0.5)) \\ &= \Psi(0.95,0.86,0.70) = (0.04,0.96) \quad \Rightarrow \quad 'L' \end{aligned}$$

$$\begin{aligned} r_2 &= \Psi(\Phi(1.0,0.0,1.0,1.0,0.0,1.0,1.0,0.0,1.0)) \\ &= \Psi(0,0.49,0) = (0.91,0.09) \quad \Rightarrow \quad 'H' \end{aligned}$$

$$\begin{aligned} r_3 &= \Psi(\Phi(0.9,0.2,0.2,0.9,0.5,0.2,0.9,0.2,0.9)) \\ &= \Psi(0.92,0.65,0.09) = (0.15,0.89) \quad \Rightarrow \quad 'L' \end{aligned}$$

Chapter 6

The Implementation

The main motivation for developing the software was to evaluate practically the suggested model. For bigger neural networks it is very difficult to analyze the behaviour of the system theoretically, in particular the generalization performance. One way to demonstrate that the model is working is to run a simulation. This does not formally prove that the proposed architecture has certain properties because only a finite number of examples can be simulated. Nevertheless this methodology is widely used in the field of neural networks. From a more practical point of view a successful simulation does *'prove'* that the architecture is capable of solving a certain type of problem.

In this section aspects of the implementation are discussed, including the structure of the software for the modular neural network, the usage of the programs and the interfaces to the developed library. The implementation also included mechanisms for preprocessing of the data.

6.1 The Concept

The following issues had to be considered for the implementation:

- From the outset, the implementation was seen as a prototype for the suggested model. It was inevitable that alterations would need to be made during the project, and this required the development of a flexible implementation.

- One area of interest was to investigate the generalization performance on large input spaces. This requires a fast and efficient implementation.
- In order to compare the monolithic network with the modular system it was desirable to have the same implementation for both.
- It would also be advantageous to be able to use the implementation to model different modular architectures. This implies the development of a 'construction kit' form of the implementation.

These constraints as well as the hardware available led to the decision to use the C++ programming language. This provided an object-oriented environment, that was both flexible and fast. Another advantage of C++ that was used, is the portability of the language.

6.2 The Software Platform

The use of C++ made it possible to restrict the system dependencies to a few functions. By re-implementing these functions, the software could be used on other systems with a C++ compiler. The implementation used pre-processor directives to control these system specific functions; by globally defining the used system (either `#define PC` or `#define UNIX`) the correct code is used.

The implementation was mainly developed on a SUN-workstation running under SUN-OS, using the `gcc` compiler and the standard C/C++ libraries. To simplify the process of compiling and linking a `Makefile` was used. First a library was built and then the programs were developed using the library functions.

The program can also be complied using TURBO C++ 3.0 on a PC. To simplify the programming a `project file` is used. To write new programs on a PC using the developed modules all the supplied classes have to be included in the new `project file`.

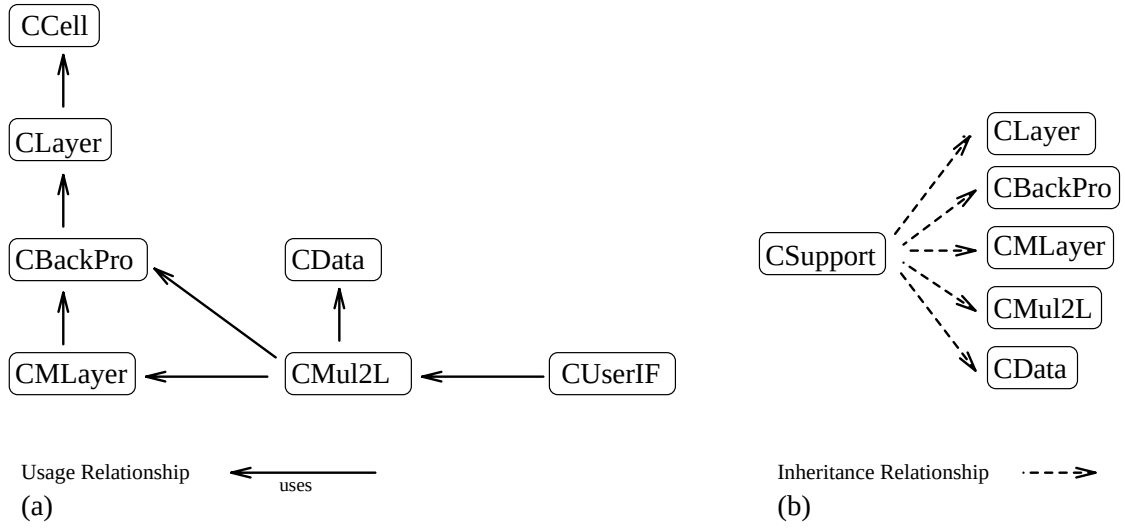


Figure 6.1: The Structure of the Neural Network Library.

6.3 Preprocessing of the Data

The preprocessing of the raw input data may appear to be a trivial task; but can have a major impact on the performance [ston95]. In the project C/C++ routines and UNIX tools like `sed` and `awk` were used to preprocess the data. The pictures were converted into an `ASCII` format using the UNIX program `xv`. To use the UNIX text processing tools proved to be quite efficient and easy, see [chri88] for further details.

6.4 The Neural Network Library

To develop the modular neural network first a library was written. All the important functions and objects are included in this library. The structure of the classes used in the modular network is shown in Figure 6.1. As it can be seen from the names in the diagram the classes represent concrete object like cells, layers, etc. This way of structuring was preferred over the usage of more abstract objects like vector or matrix; in my opinion the usage of concrete objects makes it easier to use the library as a kind of construction kit.

The concept of containment of simple object within a complex object was used as the main design method, Figure 6.1(a). Inheritance was used to provide all

objects with some general functions, Figure 6.1(b).

One problem with using C++ was the garbage collection; there is no built-in mechanism to support this. In the high-level interface this problem does not appear to the user of the library, but in the low-level interface this is an important point. In the whole implementation the following convention is made to solve this problem: the user of a low-level methods has to allocate the memory for the return values; and he is also responsible for freeing this memory again. This convention is very common in C programming.

To give an overview the used data structures and classes are described below. For the more important object some parts of the interface are explained.

6.4.1 New Data Types

The following additional data types are used. The structure `TNetDef` is used to describe a single module. It is similar to the theoretical definition made in section 5.2 with some additional information. More complex data types are realized as classes.

```
enum    TFunction {bi_pol, uni_pol}; // for the activation function

typedef double*     TVector;     // for weights, input & output vectors

typedef struct TNetDef           // to describe a BP module
{
  int       no_inputs;           // # of inputs
  int       no_layers;           // # of layers (without the input layer)
  double    lambda;              // lambda value for activation function
  TFunction fct;                 // transfer function (bi_pol or uni_pol)
  int       neuronL[MAX_LAYER];  // # of neurons in each layer
};

enum TAction {new_net, load, store, train, train_inp,    // for the input
              train_dec, test, reset, work, help, quit}; // choice
```

6.4.2 The Class `CData`

Objects of this class can be used as a fast file interface. The basic task performed by this object is to load the data file into the memory. It makes it possible that

each input tuple can be accessed in any order without reading the file again. The important methods of this object are shown below.

```
CData::CData();  // constructor

// method to read the file into memory
// returns the number of records in the file
int    CData::Read(char * filename,    // the data-file name
                   int input_length,   // number of input attributes
                   int output_length); // number of output attributes
                                       // if a class number is used 1

// methods to access the data
TVector CData::GetInV(TVector vect, int no); // read the input vector
TVector CData::GetOutV(TVector vect, int no);// read the output vector
```

To use the facilities a `CData`-object is created in the application code, the function to read the file into the object memory is called; after this the data is available for use. The following example will explain the use of this class and also shows the necessary memory allocation.

```
...
int no_inputs  = 10;  // number of input attributes in the file
int no_outputs = 2;   // number of output attributes in the file
int no_records;       // variable to store the number of records
CData aDataObj;       // the data object
// the vectors to store the returned tuples
TVector inputVec  = new double[no_inputs];
TVector outputVec = new double[no_outputs];
...
// read the file "file.dat"
no_records = aDataObj.Read("file.dat", no_inputs, no_outputs);
...
for(i=0;i<no_records;i++)
{
    inputVec  = aDataObj.GetInV(inputVec, i);  // get input tuple  i
    outputVec = aDataObj.GetOutV(outputVec, i);// get output tuple i
    ...
}
...
// free the memory
delete[] inputVec;
delete[] outputVec;
```

6.4.3 The Class `CSupport`

This class is used as a base class to provide general functions, like reading or writing files, generating random numbers, methods to read the system time, and print functions for the used vectors. All advanced class inherit from `CSupport`.

This class contains most of the system specific code. To compile the software on another system only this class would need to be changed.

6.4.4 The Class `CEncode`

This class provides methods for recoding data files, including converting continuous attributes into normalized values or binary codes. Using an object of this type it should be very straight forward to recode a new data file into the used file format.

6.4.5 The Class `CCell`

A cell-object is used as the basic building block of the network, analogous to the nervous system. Each neuron has a number of inputs and a number of outputs. Each input has a assigned weight, representing the connection strength of the synapse.

The object of type `CCell` can calculate the `net`-value for an input vector as well as the output response.

6.4.6 The Class `CLayer`

In this class one layer of neurons is implemented, consisting of a number of neurons and a constant input called *bias*. The neurons within the layer are of type `CCell`.

A object of class `CLayer` can calculate the output of the layer for a given input; this method is used in the working mode of the network as well as during learning. Most other methods in this class implement the BP-learning for one layer. The `CLayer`-object has methods to calculate the error signal (δ) for a hidden layer and for an output layer. This makes it possible to use the class to implement both types of layers in a MLP.

6.4.7 The Class `CBackPro`

This class implements a Backpropagation module. Objects of this type can be used as building bricks in a modular neural network. The interfaces to the most important methods are shown below.

Two constructors are provided. The first creates a network from a set of parameters; in this form the number of neurons in the last used layer must be equal to the number of outputs; the weights are initialized randomly. The other constructor load a module from file. The variables and weights are set according to the values in the file. There is also a method to store the module in a file.

```
/* Constructor using parameters:
CBackPro::CBackPro(int no_inputs,  // no of inputs
                   int no_layers,  // no of layers
                   double lambda,  // lambda value
                   TFunction func, // transfer function
                   char *logfile,  // name of the log file
                                   // use NO_LOG for skipping the log
                   int neuronL0,   // no of neurons in layer 0
                   int neuronL1,   // no of neurons in layer 1
                   int neuronL2=0,
                   int neuronL3=0, // no of neurons in the following
                   int neuronL4=0, // layers (default = 0)
                   int neuronL5=0);

/* Constructor to load a stored network file
   The first parameter is the name of file which contains a description
   of a network (with or without weights). The net description files
   might be typed in by using any editor or generated by the method
   CBackPro::StoreNet(...) . The second parameter is the
   name of the logfile (NO_LOG makes no logfile).             */
CBackPro::CBackPro(char *filename,      // net description file
                   char *logfile);      // name of the logfile

// Store the network in a file
void CBackPro::StoreNet(char *filename);
```

The module offers methods to learn vector pairs using the BP algorithm and to calculate the response for an input vector. A method to reset the weights in a given range is provided.

```
// Learn one vector for which the result is known
// Returns the propagated error
double CBackPro::Learn(TVector inpV,     // the vector to learn
                       TVector tarV,     // the desired result
                       double eta,       // the learning constant
                       double alpha);    // the momentum

// calculate the output for a given input
// Returns the calculated response
TVector CBackPro::Apply(TVector outV,   // the allocated result vector
                        TVector inpV);  // the input vector

// Reset the weights to a random value in a range between min and max
void CBackPro::ResetWeights(double min=MIN_RND, double max=MAX_RND);
```

6.4.8 The Class `CMLayer`

In this class a number of modules are used to built a layer of networks. The modules within the layer are implemented as objects of type `CBackPro`. Each input of a module is connected to a single input variable, and each input variable is connected to only one module. The main task of the `CMLayer`-objects is to perform the mapping from the input space onto the modules; the object provides functions to do either a linear or random mapping. The class also deals with the case when $l \neq m \cdot n$, see section 5.2.

The functions for learning and calculating the output is mainly performed by mapping the input data onto the modules and reassembling the results given by the modules.

6.4.9 The Class `CMul2L`

This class is a representation of the proposed architecture. It consists of an object of class `CMlayer` used to implement the input layer and an object of class `CBackPro` used as the decision network. An object of class `CData` is used to handle the file access.

To create the modular neural network two constructors are offered. One is using a set of parameters to initialize the network, the other one loads the description from file. There is also a method to store the modular neural network on disk.

```
// the constructor with parameters
CMul2L::CMul2L( int no_u_inps,     // number of user inputs
                int no_cl,         // number of classes
                TNetDef *netL1,    // modules used in the input layer
                TNetDef *net_m,    // the decision module
                char *logfile = "multi.log",// the log file
                int mapping = RND_MAPPING,  // mapping: inputs-modules
                TOutRep out_rep = small);   // the representation of the
                                            // classes in the first layer

// the constructor from file
// method to set up the modular network according to the description file
CMul2L::CMul2L( char *filename,               // the description file
                char *dirname = "net_dir",    // the sub directory
                char *logfile = "multi.log"); // the log file

// function to store the network
int CMul2L::StoreNet( char *filename ,      // the main description file
                      char *dirname = "net_dir");// the directory
```

Two types of interfaces are provide by the class: a text-file based interface and a low-level interface. The file based interface offers methods to deal with a whole data file, such as train the network on a file, test the network on a file, or calculate the output for a given file.

```
// train the whole network on a data file
void CMul2L::Train(char *datafile,    // the data file
                   int maxStep_1,     // max of steps in input layer
                   double maxError_I, // max accep. error in input layer
                   int maxStep_m,     // max of steps in decision net
                   double maxError_D, // max accep. error in decision net
                   char *err_file_I = "input.err" ,  // error file name
                   char *err_file_D = "decision.err",// error file name
                   double eta=1,      // learning constant
                   double alpha = 0); // momentum

// test the performance of the network on a data file
//Returns the performance in percent (0..100%)
double CMul2L::Test(char *filename, // the data file
                    double diff =0.01, // the required difference between
                                       // winner and runner-up
                    int displ = 0);    // if displ >= 1 each result is
                                       // printed on the screen
```

```
// calculate the output for given input data in a file
void CMul2L::Work(char *infile,  // the input data
                  char *outfile, // the output file
                  int datefiletype =1, // the type of the data file
                                       // 0 = long    ( 1 0 1 0 : 1 ;)
                                       // 1 = short   ( 1 0 1 0 ;)
                  int displ = 0, // if displ > 0 each result is
                                 // printed on the screen
                  double diff = 0.01,  // the required difference between
                                       // winner and runner-up
                  int o_mode = 1);     // output mode 0= class only
                                       //             1= vector and class
```

The low-level interface allows working with single vectors. Explicit functions to calculate the results for a given input vector are provided. To learn single vectors methods in the input layer or in the decision network must be called.

```
// calculate the response for an input vector (the vector)
// returns the calculated output vector
TVector CMul2L::Apply( TVector resV,  // allocated result vector
                       TVector inV ); // input vector

// calculate the class number for an input vector
// Returns the class number or -1 if the difference
// between the winner and the runner-up is smaller than diff
int CMul2L::Detect(TVector inV,          // input vector
                   float diff = 0.01);  // the wanted difference
```

6.4.10 The Class `CUserIF`

A text-menu based interface is implemented in this class. A text-menu was preferred over a graphical interface because it makes the user interface platform independent. Given a specific operation system it would be straight forward to implement a graphical user interface.

All the functions can be called from a menu; the parameters are asked from the user, where possible a default value is given. The main menu is shown in Figure 6.2.

The class `CUserIF` can also be seen as an example of how to use the underlying objects.

```
Welcome to the
         M O D U L A R  N E U R A L  N E T W O R K
                                          User Interface!

------------------------------------------------------------
   <N>ew network. User will be asked for parameters.
   <L>oad a network form a description-file.
   <S>tore the actual network in a description-file.

   <T>rain the whole network on a data-file.
   <I>nput network training on a data-file.
   <D>ecision network training on a data-file.

   <P>erformance test of the network (on a data-file)
   <C>alculate outputs for given input file.
   <R>eset the weights of the actual network.

   <H>elp gives help on data-file structure


   <Q>uit!
------------------------------------------------------------

>>> _
```

Figure 6.2: The Menu of the Modular Neural Network Program.

6.5 The File Formats

The high-level interface to the library is using ASCII-text files. The advantages of this approach is that text editors, filters, or tools like `vi`, `sed`, `grep` or `awk` can be used to maintain and manipulate the files. There are also UNIX-programs working on text files for visualization of the data; in the project the program `gnuplot` was used.

This makes it easier for an user to view data input files as well as results written by the system. The exchange of files between different platforms is straight forward when using ASCII-text files. It is possible to train the system on a fast UNIX-workstation and then copy the network description file to a PC and use the trained network without conversion.

The main disadvantage of using text files is that they are larger than the equivalent binary files would be. The longer time needed to read the file from disk is not significant in comparison to the training time of the network; particularly by using the class `CData` where the file is only accessed once.

To archive larger data files or network description files, all platforms offer tools

to compress the text files; on the UNIX system `compress` and on the PC `pkzip` were used.

6.5.1 The Data File

The format for the data file was designed to be readable to humans. For the following description it is assumed that the network has l inputs.

Each tuple consists of:

- an input part comprising l real numbers separated by spaces (`'\32'`)
- a colon separator (`':'`) between input and output
- the desired output class, represented by an integer

The tuples are separated by a semicolon (`';'`), the last line may be concluded by a semicolon or by a dot (`'.'`).

Here are some lines from an example data file with ten input attributes; the numbers used show the various possible formats recognized.

```
0.233 0.12 -1 0.99   -0.121   .5 1 0.001 0 -1 : 3 ;
1 -0.5 -0.1 -.99 0.21 0.45 1 -0.911 0 -0.8181 : 0 ;
...
-1 0.5 -0.01 0.99 -1 -1 1 -0.11 0.1 -0.1199   : 4 .
```

This file format is referred to as 'long file format' because it stores the information of the input vector and the output class. This file type can be used for training the network and for testing the performance of the network. A 'short file format' containing only information about the input vectors, is used in the working mode. New input patterns are presented and the response of the system is calculated. The following lines are taken from a typical 'short' file:

```
0.913 -0.2 -1 0.119   0.11   .95 0.1 0.001 1 -1 ;
0 -0.5 -0.1 -.9 0.21 0.5 -0.91 -0.11 0.10 -0.11 ;
...
-0.2 0.2 0.01 0.99 -0.75 -1 0 -0.11 -0.341 0.19 .
```

6.5.2 The Network Description File

The network description is stored in an ASCII file. Each module is stored in a separate file, with all modules for a particular network being grouped in a separate directory. This makes it possible to view, modify, or test individual modules using the program `bpnet`.

The main network description file stores the information about the overall network structure and the names of the description files for the input layer and for the decision module. The conventions for the names of these files are `<filename>.net` for the main network decision file, `<filename>.inp` for the input layer file, and `<filename>.dec` for the description of the decision network. The input layer file stores information about the layer as a whole and also the names of the description files for the sub-modules. The description files for the modules have the names `<filename>.<no>` with `<no>` the number of the module. The `<filename>` needs to be selected such that the created filenames are valid on the operation system which is used.

The description file for a module stores the information about the number of layers, number of inputs, the lambda value used, the type of the activation function, the number of neurons in each layer, and the connecting weights. For an example see appendix B.2.

6.5.3 The Error Output File

For each training cycle the calculated and propagated error is recorded in the error file, by convention `<filename>.err`.

This information is very useful to find out how fast a network is converging and in determining an optimal maximum training error. The format of the file is:

```
0 0.455532
1 0.426511
2 0.391984
...
999 0.001322
```

Files in this format can be read and visualized by many programs, including `gnuplot` on the UNIX system and MS-Excel on PCs.

6.5.4 The Logfile

All important information about the creation, training, and testing of a network is recorded in the logfile. After running the program, information about the training parameters, the time needed to train, or the performance on the data set can be recalled. The file is called `<filename>.log` .

The function `fflush(FILE*)` is used after logfile entries have been made to ensure that the work done up to this point is not lost in case of failure.

For an example of a logfile see appendix B.1. The information stored in the logfile made the evaluation of tests much easier.

6.6 The Developed Programs

Here is a short survey on the developed programs. For a more comprehensive description see the `README` files for the programs.

6.6.1 The Modular Neural Network: `modnet`

The program `modnet` was the main tool used for the experiments in the project. The implementation is based on the library described above. The program employes an object of type `CUserIF`; an event-handle loop is created using the methods `CUserIF::Menu()` and `CUserIF::Handle(TAction)`.

The menu created by the `CUserIF` object allows the user to chose what operation should be performed, prompting for the necessary parameters for that function. In the cases where this is possible the user is provided with a default value or information about a suitable range for the parameter.

6.6.2 The Monolithic Neural Network: `bpnet`

The program `bpnet` provides a similar interface as seen above but dealing with one module as a single network. It is a standard Backpropagation implementation working on the same file formats as the program `modnet`.

This program can be used to create a monolithic network for a given data set as well as to evaluate single modules out of the modular neural network.

6.6.3 The Logical Neural Network: `logicnet`

The program `logicnet` was developed to make a comparison between a logical NN and the connectionist networks shown above. It is also a prototype implementation using the same concepts as for the modular neural network.

The basic building brick is the class `CRam`, implementing a simple RAM-node. The class `CDiscrim` implements a discriminator based on a number of `CRam`-objects. A complete logical neural network is implemented in the class `CRamNet`.

The user interface is similar to the ones given above. It is implemented in the class `CRamUI`.

Chapter 7

Experimental Evaluation

To evaluate the proposed modular neural network a number of tests were carried out, examining the ability to memorize different data sets and to generalize on them. A variety of different data sets were used to find a framework for the possible application domain for the proposed architecture.

A minimum criteria for experimental evaluation of neural network learning algorithms is: *"An Algorithm evaluation is called acceptable if it uses a minimum of two real or realistic problems and compares the results to those of at least one alternative algorithm."* [prec95, p227]. The experimental studies during the project were made to be well ahead of this guideline.

The new architecture was compared to two other well known and well researched network types: logical neural networks and BP-trained multilayer feedforward networks. For data sets that were represented in a binary form the comparison was made with both reference networks; for continuous data the comparison was made with the MLFFN only.

A number of different real world data sets were used. Some were based on binary input variables, other were coded with continuous values. The input dimension varied from eight to 6750, and the number of instances in the different training sets were between four and 384.

A simulation is always based on a finite number of test cases. It is therefore difficult to draw a general conclusion from the tests. This problem particularly

appears in comparing different networks. A reference network with a different learning parameter or number of hidden layers might have given a better result; or a different number of inputs to each module in the modular network might have increased the performance.

Both the BP-network and the modular neural network have a number of variables and parameters that may be changed for the test. To estimate the number of test networks for a comprehensive study consider the following variables:

- Five different random mappings from the inputs to the modules.
- Five different combinations of learning constant and momentum.
- Five different numbers of modules in the input layer.
- Two intermediate representations (small, large).
- Two different types of input modules (number of neurons in the hidden layer).
- Four different types of decision modules (number of layers, number of neurons).
- Ten different random initializations of the weights.

This would result in:

$$5 \cdot 5 \cdot 5 \cdot 2 \cdot 2 \cdot 4 \cdot 10 = 20000$$

different test networks that would have to be trained and evaluated. This huge number is reached despite the fact that the figures for each individual variable are too low to be statistically reliable. This only considers a single data representation. The problem of the huge number of possible networks is addressed in section 8.1.

Because of this a different test strategy was used. In each experiment the focus was on a particular issue and all other variables were kept constant.

In some of the experiments the optimum solution for the MLFFN was found using the program `opti`. This was developed during an in-course project at the MMU [schm96], and is based on the BP-algorithm and searches for the optimal set of weights for a given training and test set. The basic idea of the program is

to check the performance on the test set after each training cycle to prevent the network from overfitting.

7.1 Test Methodology

The test sets used are either obtained from neural network benchmark data bases [prob96], [ucim96], or [nnbc96], or were developed from raw data.

The following methodology was used for the tests:

Two data sets were prepared: a training set and a test set. Some data sets were split randomly in two sets of equal size (plus/minus one if the number of instances was odd). For other tests selected instances were used for training and the others for measuring the performance.

The networks first learnt the training set and then the recognition or prediction performance was calculated on both sets. The result of the training set is denoted by P_M and it is a measure for the memorization capabilities. The performance on the test set is given as P_G and indicates the generalization abilities of the trained network.

Depending on the test, different network structures, various numbers of hidden units and network layers, or alternative network parameters were investigated. Furthermore the influence of the number of training steps and of the error value as stop condition were examined.

The time specified in the tests is measured on the same system for one experiment. Due to the hardware availability different experiments were carried out on different systems, therefore a direct comparison between different experiments is not valid. The notation x'y" stays for x minutes and y seconds. No parallel training was used.

In the experiment to investigate the fault tolerance of the modular neural network architecture a different methodology was used, this is described in section 7.5.

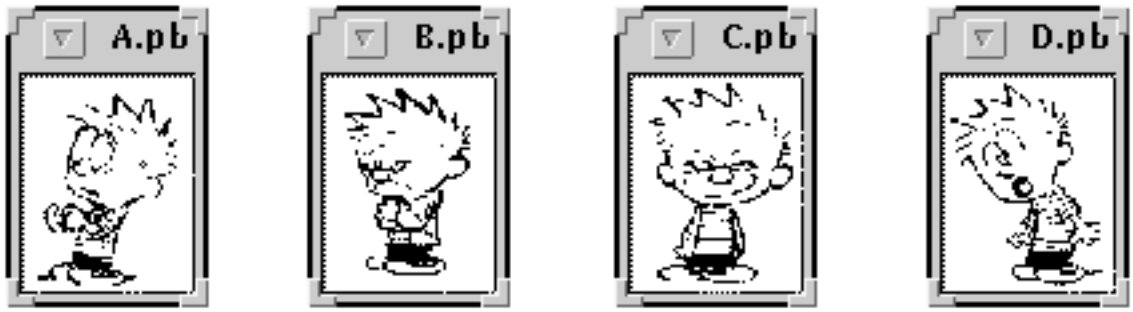


Figure 7.1: The Four Original Pictures.

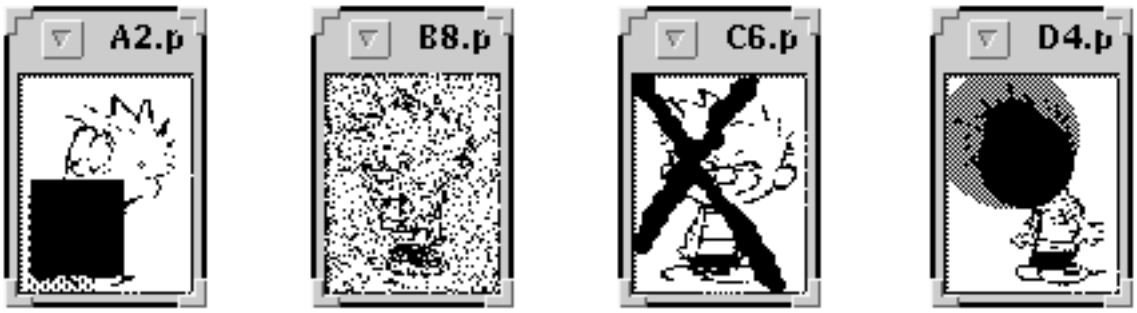


Figure 7.2: Examples of Distorted Pictures.

7.2 Binary Pattern Recognition

For problems in pattern recognition, particularly for binary patterns, logical neural networks have been proved to be very useful. LNN can perform a classification according to the distance between the test and training patterns. The hypothesis is that a modular neural network should outperform a monolithic network on this domain; because the internal structure of a MNN is similar to a LNN.

In this experiment different networks were trained to memorize four different but similar binary pictures, shown in Figure 7.1. The pictures had a size of 66 by 83 pixels (5478 binary input variables).

After the networks had successfully learnt the training set, a number of distorted pictures were presented. Examples of the distorted pictures are given in Figure 7.2, and the full picture set is given in appendix C. The task was to assign the class number to each picture.

As expected the RAM based logical network performed very well on this prob-

lem. For a RAM size of 256 (eight inputs) a recognition performance of 100% was achieved. This good result was achieved because the number of instances in the training set was very small.

Twelve modular networks using between 548 and three input modules were tested. The mapping function π was a random permutation. The number of training steps in the input layer and in the decision network was varied. Different learning parameters and momentum values were used. Throughout the experiment the modular network converged for nearly any setting of the parameters.

Four different monolithic networks were trained; they had different numbers of hidden layers and different numbers of neurons. It was difficult to find a parameter set that ensured convergence for the training. In Table 7.1 some of the best results gained with these networks are presented.

Network	P_G	Time
$M_1 = (5478, 4, [32, 16, 8])$	72.0%	12'
$M_2 = (5478, 4, [8, 6])$	77.7%	3'26"
$N_1 = (5478, 4, 548, small, \pi, (10, 2, [2]), (1096, 4, [8]))$	88.8%	2'28"
$N_2 = (5478, 4, 110, small, \pi, (50, 2, [2]), (220, 4, [8]))$	91.6%	1'36"
$N_3 = (5478, 4, 28, small, \pi, (200, 2, [2]), (56, 4, [8]))$	86.1%	4'32"

Table 7.1: The Binary Picture Recognition Example.

The maximum performance of the modular network was significantly better than the best result achieved with a monolithic network, but did not achieve the perfect recognition of the logical network.

In Figure 7.3 a comparison of the best networks of each type is given. It can be seen that the LNN is the most suitable for this application domain (small number of binary patterns, high dimensional input space). It delivered the best generalization performance with the shortest training time.

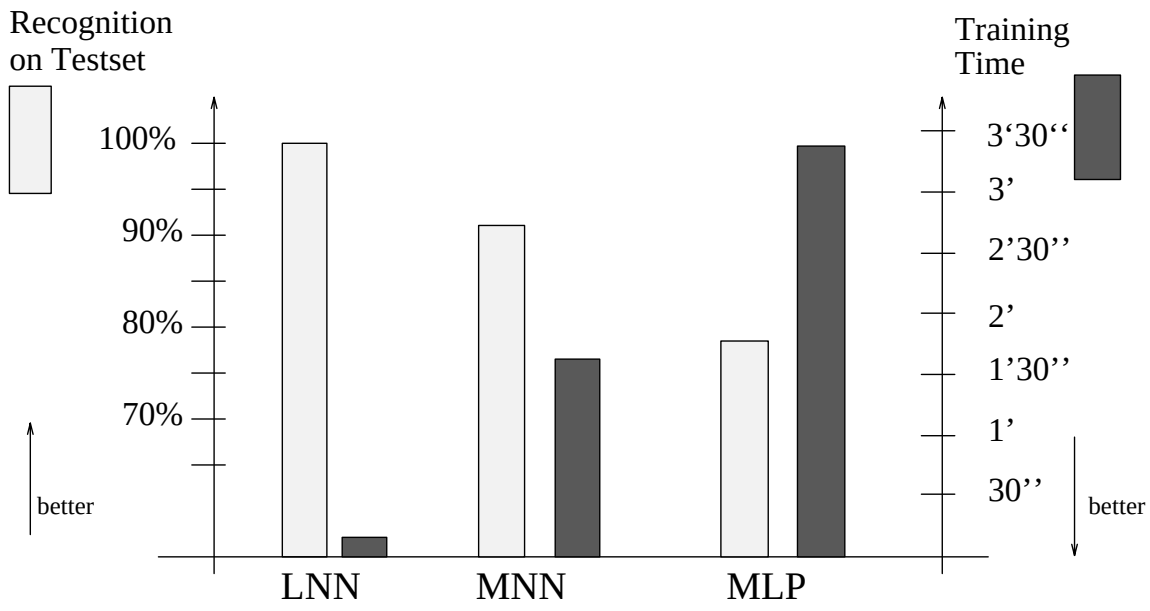


Figure 7.3: Comparison of the Best Networks for the Binary Recognition Task.

7.3 Problems with a Small Input Dimension

Typical application domains for MLPs are data sets with a small input dimension. All the data sets described here represent real world problems, and are obtained from different neural network benchmark databases. Most sets have already been used to investigate the behaviour of BP-trained neural networks by other researchers; some of the results were compared to the findings in these experiments.

7.3.1 Prediction on Diabetes Data

This data set is called the ‘Pima Indians Diabetes Database’ [sigi96]. The original set has 768 instances, each tuple having eight input attributes normalized in the interval $[-1, 1]$. Instances are assigned to one of two classes, 500 of the records belonging to class ‘1’ and 268 to class ‘2’.

Testing with a single MLP and the eight continuous input variables gave a generalization performance of about 76%, similar to results stated in the informations obtained with the data set.

To allow a comparison between all three types of networks the eight continuous valued attributes were transformed into 80 binary input attributes.

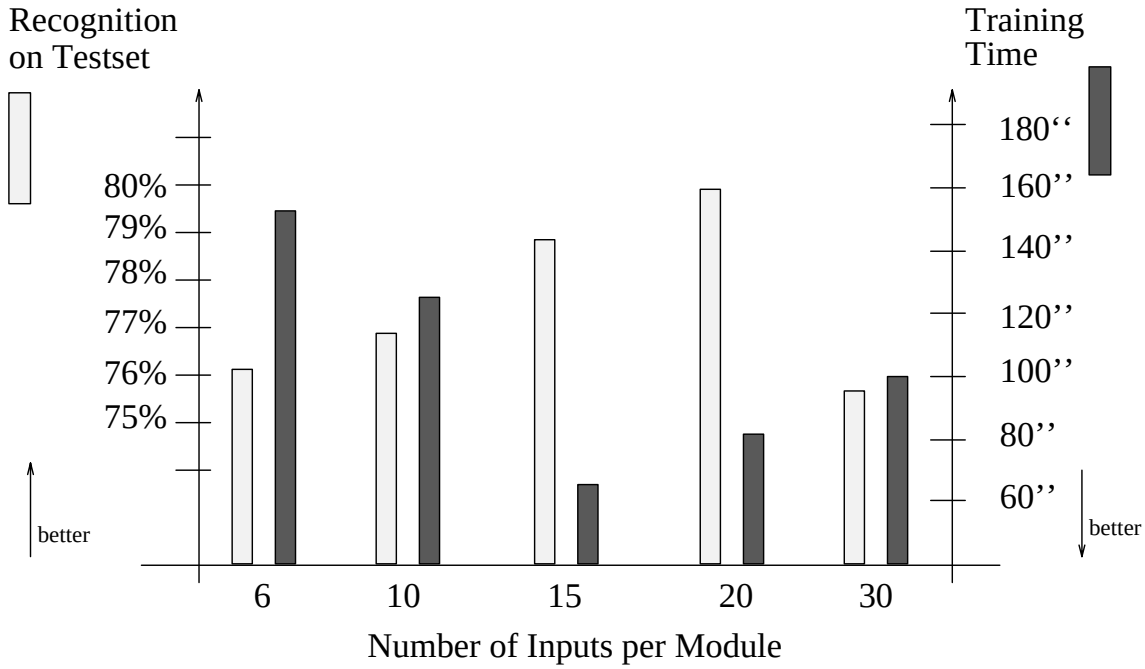


Figure 7.4: Generalization and Training Time for the Diabetes Data.

The logical network was not at all useful for this data set. For any configuration (tested RAM-Sizes: 64, 256, 1024, 4096, 16384, 65536, 262144, 1048576) the generalization performance was about 50%. For a two class problem, that is exactly the probability expected by classifying input vectors at random. This can be explained by the large number of data tuples in the training set; the RAM-network was overfilled.

The results achieved with a single multilayer feedforward network were similar to results achieved using the proposed architecture; the best modular neural network $M = (80, 2, 4, small, \pi, (20, 2, [2]), (4, 2, [2])$ achieved a generalization performance of 79.7%; the best MLP found by manually selecting the parameters had $P_G = 78.3\%$; the optimal solution found with the program `opti` was 80.1%.

In Figure 7.4 the generalization performance depending on the number of inputs to the modules is shown, together with the time needed to train the networks.

7.3.2 Prediction on Heart Disease Data

This task is to predict on a two class problem. The data set contains 304 records, with 13 input variables. The variables were recoded into 28 binary inputs. A

comparison was only made between a MLFFN and a modular network because the number of inputs was too small to use a LNN.

The data set was investigated in the project [schm96]. The best result achieved in a long number of tests with the program `opti` was 88.15%.

Different MNNs using standard parameters (learning parameter $\eta = 1$, momentum $\alpha = 0$, and the steepness of the activation function $\lambda = 1$) for learning resulted in a performance of 86%. The training was about four times quicker.

7.3.3 Prediction on Credit Card Data

The Credit Approval data set contains 690 instances concerning credit card applications [quin96]. There are 15 attributes describing the applicant; some of the input values are continuous, others are symbolic. The two classes '+' or '-' indicate whether the application was successful. The class distribution set is fairly evenly balanced (44.5% '+' and 55.5% '-').

For the experiment all input variables were converted into numeric values and normalized in the interval $[0, 1]$.

In the test the performance of a single BP trained MLP was compared to two different modular networks, one with three sub-nets in the input layer and one with two. The best results achieved with both modular configurations was between 84% and 85%. The best result achieved by a single network was 86%.

7.3.4 Classification of Radar Signals

The Ionosphere database contains data of radar signals, used to find structures in the ionosphere. It is a two-class prediction problem. The signal is given as 350 instances of 34 continuous values normalized between -1 and 1. The first 200 records were used for training the remaining 150 for testing (this is according to the previous usage of the data).

Training a single network $M = (34, 1, [8, 4])$ over 49 cycles (time needed 18") led to a performance of 97.3%. The very low number of training cycles shows that

the data is very easy to memorize and generalization is simple on the domain.

To compare all three network types the data was converted into 340 binary inputs, using the coding:

```
[-1,-0.8)   => 0 0 0 0 0 0 0 0 0 1
[-0.8,-0.6) => 0 0 0 0 0 0 0 0 1 1
[-0.6,-0.4) => 0 0 0 0 0 0 0 1 1 1
[-0.4,-0.2) => 0 0 0 0 0 0 1 1 1 1
[-0.2,0)    => 0 0 0 0 0 1 1 1 1 1
[0,0.2)     => 0 0 0 0 1 1 1 1 1 1
[0.2,0.4)   => 0 0 0 1 1 1 1 1 1 1
[0.4,0.6)   => 0 0 1 1 1 1 1 1 1 1
[0.6,0.8)   => 0 1 1 1 1 1 1 1 1 1
[0.8,1)     => 1 1 1 1 1 1 1 1 1 1
```

Table 7.2 shows the best results achieved with each type of network on the recoded data set:

Network	P_G	Training Time
RAM based Logical network	92.5%	5”
$M = (340, 1, [10, 6])$	97.3%	9’01”
$N = (340, 2, 7, small, \pi, (50, 1, [4]), (7, 2, [2]))$	97.3%	2’21”

Table 7.2: The Radar Signal Example.

On the recoded data set the time difference is significant, but it must be recalled that the single network using the original data set was much faster. For this data set the recoding brought no advantage, in fact a significant disadvantage.

7.3.5 Classification of Sonar Data

The sonar data set ([sejn96]) is a collection of sonar signals, coded as 60 continuous attributes. The task is to discriminate between signals bounced off a rock or a metal cylinder (2 classes). The test was done with a number of networks and very similar results obtained independent of the architecture. Table 7.3 shows the number of neurons in the network, the number of weights to train, and the performance achieved on the test set.

Number of Network	Number of Neurons	Number of weights	Achieved Result
Modular: 15 + 1	74	340	87.5%
Modular: 10 + 1	74	440	88.5%
Modular: 4 + 1	62	664	88.5%
Modular: 3 + 1	48	620	90.4%
Single: 1	23	950	90.4%

Table 7.3: The Sonar Classification Example.

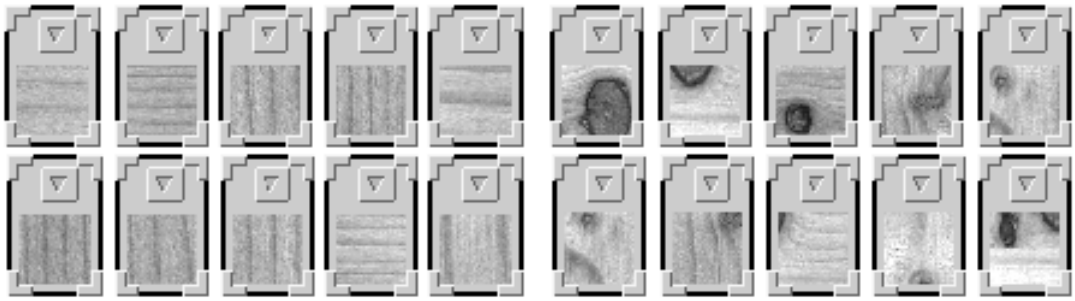


Figure 7.5: Examples of *good* and *faulty* Pictures of Wood.

7.4 Problems with a Large Continuous Input Space

These experiments investigated the behaviour of the modular neural network on high dimensional input vectors. The comparison was made between MLFFNs and the modular network.

7.4.1 Discrimination Between Good and Faulty Wood

The wood data set was generated to test the ability of a network to generalize on a huge number of continuous inputs.

To generate the data set a board of wood was scanned in as a gray-scale picture. Then a program `SelectPic` was written to select sub-pictures of a particular size out of the original picture. These sub-pictures were then classified manually into the two categories.

The basic idea is to discriminate pictures of wood, to determine if they are

faulty or not. The data set consists of 100 gray scale pictures of size 30 by 30 pixels; the gray level of each pixel is coded as value between 0 and 1. The output is either 'good' or 'faulty'. The result is a data set with 100 instances each having 900 continuous inputs and two output classes. In Figures 7.5 examples of training and test pictures are given.

The modular structure learnt the examples in very short time and gave a performance of 100% on the training set and 92% on the test set. This results were achieved with different system structures.

It was very difficult to find a parameter set for the monolithic MLFFN that converged on the data set. After a lot of tests (about 40 different configurations) a network was found which performed very well. A small learning constant ($\eta \leq 0.15$) was necessary to ensure a successful learning process. The best network gave also the results of 100% on the training set and 92% on the test set.

The time difference between training the MLP and the modular NN was significant. The fastest monolithic network trained successfully in about ten minutes whereas the average training time for the modular neural networks with $P_G \geq 90\%$ was two and a half minutes; this is a speed-up of 400%.

The significant gain in training speed can be explained by the number of weights in the different networks, see Table 7.4.

Number of Network	Number of Neurons	Number of weights	Achieved Result
Modular: 90 + 1	276	2348	92%
Modular: 23 + 1	121	3872	92%
Modular: 18 + 1	112	3710	90%
Single: 1	61	36820	92%

Table 7.4: The Wood Classification Example.

7.4.2 Gray Scale Picture Recognition

In this experiment the task was to memorize five pictures of different faces. Each gray-level picture had a size of 75 by 90 pixels (6750 continuous input variables). The gray-value (0-255) of a pixel was converted into a normalized value in the

Figure 7.6: The Original Pictures used as Training Set.

interval $[0, 1]$. The original pictures are from [pict96]. After training the MLP and the modular network the recognition performance was tested with distorted pictures.

No comparison with the logical neural network was made because continuous input values were used.

In Figure 7.6 the five original pictures are shown. This set was used as training set for the following two tests. The training was stopped if the propagated error was sufficiently small. Both network types memorized the training set very well. For all pattern the response was below 0.1 for a desired '0' and over 0.9 for a desired '1'.

The training time for the modular network was much shorter than for the monolithic MLP. This is easy to explain by the number of weights used in each network. In Table 7.5 an overview of the training times is given.

Network	Number of Weights	Training Time
$M_1 = (6750, 5, [10])$	67550	20'08"
$M_2 = (6750, 5, [20, 20])$	135500	16'40"
$N_1 = (6750, 5, 135, small, \pi, (50, 3, [4]), (405, 5, [5]))$	30670	2'27"
$N_1 = (6750, 5, 225, small, \pi, (30, 3, [2]), (675, 5, [8, 6]))$	20328	2'07"
$N_1 = (6750, 5, 23, small, \pi, (300, 3, [8, 6]), (69, 5, [5]))$	57088	4'03"

Table 7.5: The Continuous Picture Recognition Example.

Figure 7.7: Examples of Manually Distorted Pictures.

It can be seen that the training time to achieve a fixed error value is not directly dependent on the number of weights, if the number of training steps is fixed then the time is proportional to the number of weights.

In this experiment the training was stopped when the root mean square error in the network was below 0.01. A larger number of nodes might learn the data set in less training cycles so that the overall time is shorter; this effect can be seen comparing M_1 and M_2 in Table 7.5.

Manually Distorted Pictures

In this test the recognition performance was measured on manually distorted pictures. The pictures in the test set were alter by using a graphic editor. Some of the changed pictures are shown in Figure 7.7.

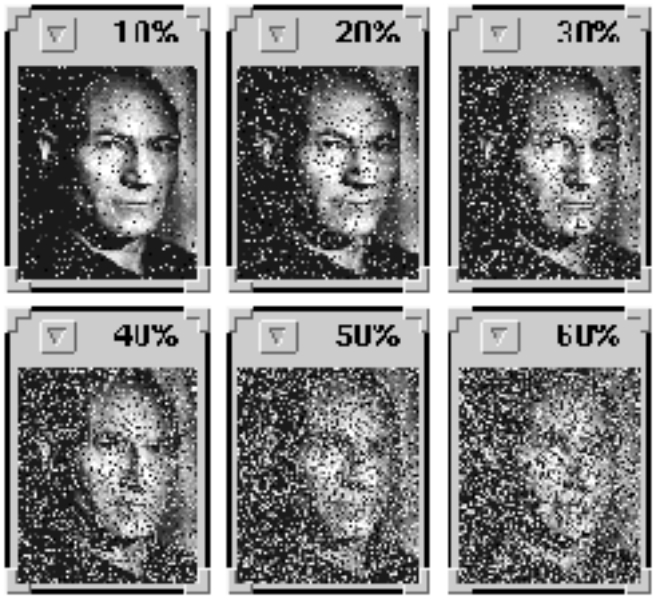


Figure 7.8: Examples of Noisy Test Pictures.

The recognition performance of the modular neural network was about 87.8%, the monolithic network could recognize 78.1%. Furthermore the discrimination difference of the modular network was much higher than for the single network. By increasing the demanded confidence (difference between winner and runner-up) the performance difference became even larger.

Noisy Pictures

Another comparison was made on the ability to recognize noisy inputs. The noise on the pictures was generated randomly. The noise-level is the probability for each pixel to be altered to a random value in the interval [0,1] (an uniform distribution was used). In Figure 7.8 pictures with different noise-levels are shown.

The modular network could recognize pictures with a significant higher noise-level than the single MLP; the results are shown in Figure 7.9.

From the above experiments it can be seen that the modular network has superior generalization abilities on this type of high dimensional input vectors.

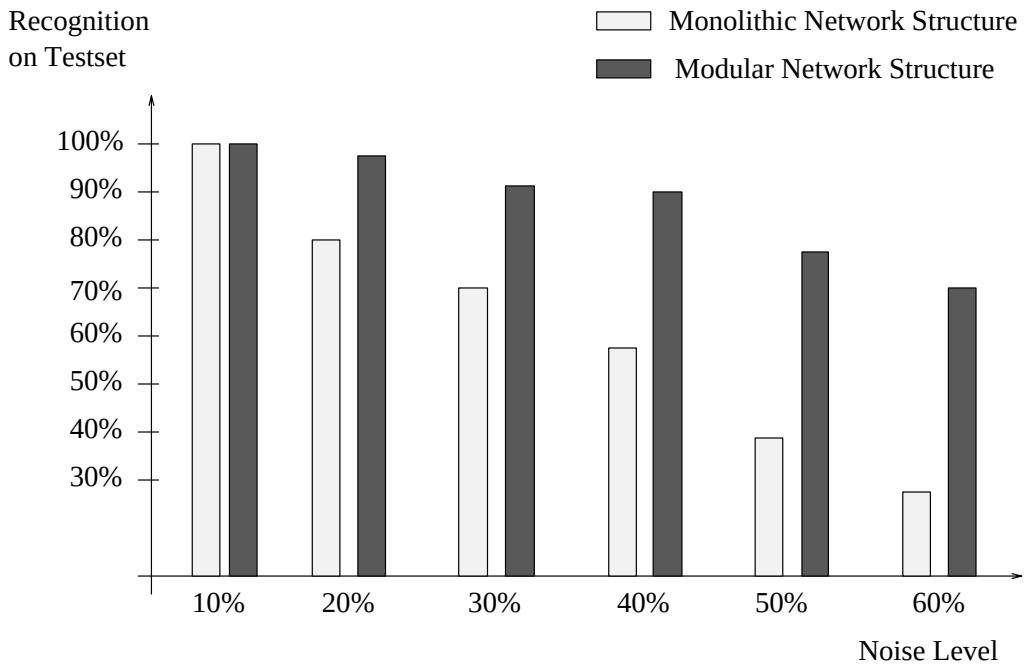


Figure 7.9: The Performance on Noisy Inputs.

7.5 Experiments on the Fault Tolerance of the System

The experiment of the fault tolerance of the system is another investigation in possible advantages of a modular architecture. For this test the Diabetes data set described in section 7.3.1 was used.

The network $M = (80, 2, 5, small, \pi, (16, 1, [4]), (5, 2, [4]))$ was used. 250 training steps were performed on the input layer of the modular neural network; this took 33 minutes and resulted in a root mean square error of the input layer of 0.184. Then 200 training steps in the decision network were made (10 minutes, root mean square error for the decision network of 0.238). The achieved performance on the training set was $P_M = 85.4\%$ and the generalization performance on the test set was $P_G = 75.0\%$.

One of the five modules in the input layer was reset to random weights while the other four modules remained in the trained state. The performance of the partly damaged network was tested ($P_{M_{damage}}$ for the performance on the training set and $P_{G_{damage}}$ for the performance on the test set).

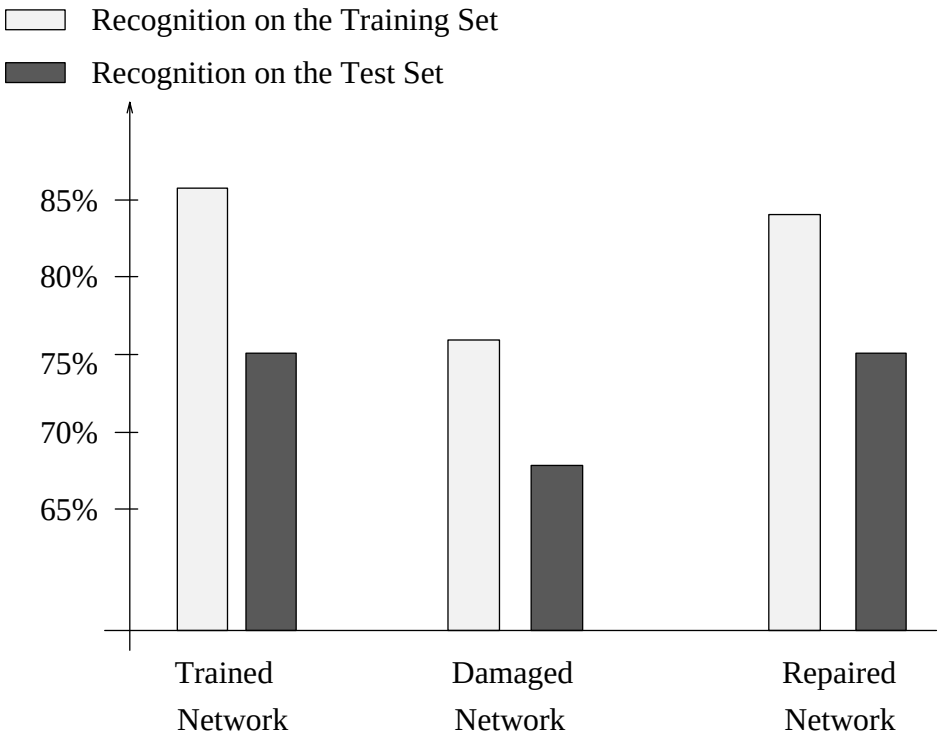


Figure 7.10: Tests on the Fault Tolerance of the System.

Than the decision network was trained another 50 steps; this took in average about three minutes. Than the performance of the repaired network ($P_{M_{repair}}$ and $P_{G_{repair}}$) was calculated.

This was repeated for all modules in the first layer, in each test only one module was reseted randomly, this is equal to about 18.5% of the weights.

The results are visualized in Figure 7.10. This result is very astonishing. The loss of about 10% performance can be retrained to nearly the original values with very little effort.

It is interesting that a similar effect can be observed in human neuro psychology. Patients with a partly damaged brain lose some of their abilities but most of them are able to relearn the things in a very short time [kala92].

Chapter 8

Conclusion

The proposed architecture proved to be useful for many different real world data sets. The experiments showed that the model performed satisfactory for all of the tested application domains. In some of the experiments the MLP or LNN was superior to the suggested model; but in these cases the performance of the modular network was close to that of the better performing network.

In most data sets with a medium input dimension (12-320 variables) the new model showed similar memorization and generalization performance as a Backpropagation trained monolithic multilayer feedforward network.

For larger input dimensions the modular architecture proved to be superior over the monolithic MLFFN, both for memorization as well as generalization. In particular the generalization ability on large continuous input vectors was significantly better. This can be explained by the combined generalization mechanism in the new model.

The theoretical limitation for learning statistically neutral data sets did not appear in the test cases used. This problem seems to be unlikely in practice, particularly when dealing with a large number of inputs.

An important improvement is the speed of training in the proposed network. The training process was much faster due to the smaller number of weight connections and the splitting of the input vector in smaller parts. This effect was particularly significant in problems with high dimensional input spaces.

The training of the decision network proved to be very easy; often very few

cycles were needed to reach a sufficiently small error value. This can be explained by the resulting training set for the decision network. The output of each module in the input layer is the class number of the input vector and the task of the decision network is reduced to a very simple mapping. For some experiments the decision module converged in less than 20 cycles.

Throughout the experiments it appeared that nearly any configuration of the modular neural network could be trained to convergence. The individual input modules proved very robust, due to their small size. The modular approach delivered a significant advantage for problems with large input vectors, where it was very difficult to find a parameter set for a monolithic network.

All modules in the input layer are mutually independent, which makes parallel training of the input layer straight forward. Since there is no communication during the learning process, this also allows a distributed implementation without additional problems.

Drawing on the modular nature of the human nervous system, an attempt to explore modularity was made in this project. The results which are far from comprehensive are very promising. The findings during the experiments are encouraging to search for new artificial neural networks based on parallelism and modularity.

8.1 Further Work

The project showed a lot of ways into the world of artificial modular neural networks. The time for experiments was very limited and many possible approaches were left unexplored. In this section some ideas for further work are given.

The aim of this project was to answer the general question: *Is the architecture useful?*. The next step should be to investigate further issues:

- How do the following parameters influence the performance of the network:
  - the number of inputs per module
  - the number of training steps in each module

  - the learning parameters
  - the number of neurons in the modules
  - the intermediate representation

  This test should be made using a single data set. Because of the huge number of possibilities using a genetic algorithm might be appropriate.

- The mapping between the input variables and the modules can be explored; The experiments assigned this mapping randomly. Making connections based additional information on the domain could improve the performance. Statistical methods could be used to calculate a mapping from the inputs to the modules that the class information is maximal for all modules. This could also be used to avoid training modules with statistically neutral data sets.

- A more general architecture, such as a pyramidal structure, can be investigated; it is also to consider if a different intermediate representation is more appropriate.

- Different training algorithms can be explored. The effect of training the decision network with noise in the input vectors can be investigated.

- The behaviour of the modular architecture for data sets with a larger number of classes may be analyzed.

- In the experiment it was shown that the decision network can be retrained very quickly. This feature could be used to adapt the network to a changing environment. To investigate this the whole network should be trained using the original data; if the data set changes only the decision network is retrained. The performance on the new data set is then measured.

- A theoretical investigation in the generalization behaviour of the network can give additional evidence for a suitable application domain.

- Using a more flexible structure could improve the performance. To explore this it is suggested to use modules of different size in the input layer and

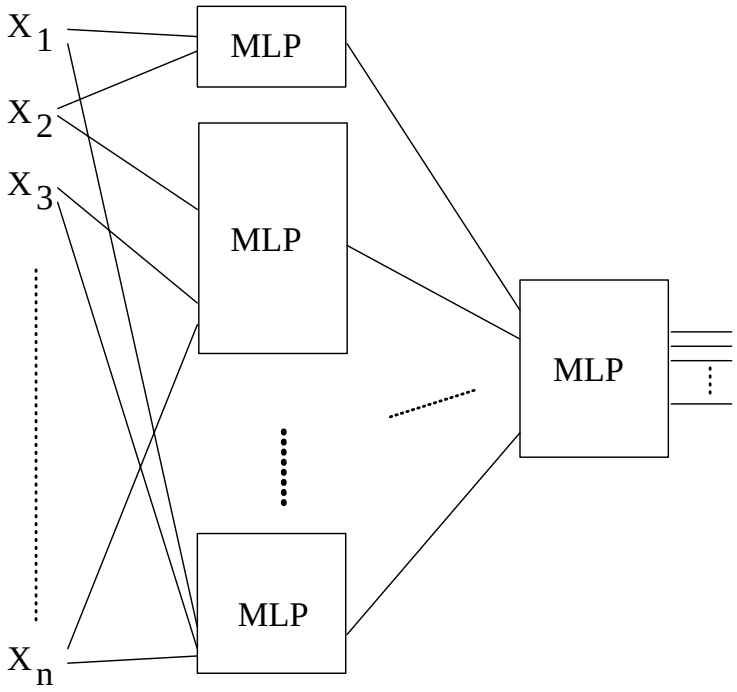


Figure 8.1: A Suggested Architecture for Further Experiments.

connect each input to two modules, see Figure 8.1. The resulting number of possible network architectures is huge therefore a genetic algorithm seems to be a reasonable method to find a good structure for a particular training set.

To avoid training of all networks in a population to calculate the fitness the statistical knowledge of the input variables could be used instead. To define a measure for the class information carried by a subset of input variables is probably a way to create a fitness function that can be evaluated quickly.

- If the system will be used for real applications a parallel implementation would certainly be desirable.

- To make the usage of the modular neural network easier, especially for users with a non-computing background, a graphical user interface should be provided.

Appendix A

Conference Paper

During the project selected topics of the work were used to write a conference paper. In particular the new architecture, the learning and generalization in the system, and the experiment on high dimensional continuous input vectors were described.

The paper was published at the *Third International Conference on Artificial Neural Networks and Genetic Algorithms (ICANNGA) '97*. A further paper on the topic was published at *IASTED International Conference on Computer Systems and Applications (CSA'98)*. See below for the full references.

Albrecht Schmidt and Zuhair Bandar. **A modular neural network architecture with additional generalisation abilities for large input vectors.** *Third International Conference on Artificial Neural Networks and Genetic Algorithms (ICANNGA)* 1997 in Norwich/England.

Albrecht Schmidt and Zuhair Bandar. **Modularity - a Concept for new Neural Network Architectures.** *IASTED International Conference on Computer Systems and Applications (CSA'98)* 1998 in Irbid/Jordan.

Please see my web-page or contact me, if you like a copy.

Appendix B

Example Files

The file formats have already been described in section 6.5; these examples are to illustrate the description.

B.1 A Typical Logfile

The class `CMul2L` writes a very detailed logfile. This is very useful for evaluation of experiments. All relevant information can be found in this file. The used text format is easy to read for the human user. It can also be used with text filters like `grep` or `awk` to extract information from a number of files.

```
 This is the logfile of the

 M O D U L A R  N E U R A L  N E T W O R K

                                  program


 Running Version: 1.0

Initialized at Thu Aug 22 13:06:04 1996
with the following pararmeters:
# of user inputs       : 80
# of different classes : 2

# of nets in the inp layer     : 5
# inputs in inp layer          : 16
# layers in inp layer          : 2
# lambda in inp layer          : 1.000000
# function in inp layer        : 1
```

```
# neurons in 0. layer          : 4
# outputs per net in inp layer : 1

# inputs in the decision net   : 5
# inputs in the decision net   : 0
# layers in the decision net   : 2
# lambda in the decision net   : 1.000000
# function in the decision net : 1
# neurons in 0. layer          : 4
# outputs in the decision net  : 2

this logfile name: multi.log
the mapping code 17
small intermediate representation

weights are initilalized randomly!


Training the whole Network !

Training the input layer !

Parameters: max. # of steps: 250
            max. Error     : 0.010000
            data file name : diab.t
            # train. recs. : 384
            error file name: input.err
            learning const.: 1.000000
            momentum const.: 0.000000

Start training at Thu Aug 22 13:06:39 1996

Stop training (input layer) at Thu Aug 22 13:39:57 1996

 Results : # of steps: 250
           Error     : 0.184489

Training the Decision Network !

Parameters: max. # of steps: 200
            max. Error     : 0.010000
            data file name : diab.t
            # train. recs. : 384
            error file name: decision.err
            learning const.: 1.000000
            momentum const.: 0.000000

Start training at Thu Aug 22 13:40:02 1996

Stop training (decision net) at Thu Aug 22 13:50:01 1996
```

```
Results : # of steps: 200
          Error     : 0.238476

Write the Network to file: diab80 in dir: nets
                  at Thu Aug 22 14:06:59 1996

Test the performance !

Parameters: data file name : diab.t
            # records      : 384
            difference     : 0.010000
            display mode   : 0

Start testing at Thu Aug 22 14:07:27 1996

 Test Statistics:

 Results: # records   : 384
          # goods     : 328
          # bads      : 56

          recognition : 85.416667 %

Test the performance !!!

Parameters: data file name : diab.p
            # records      : 384
            difference     : 0.010000
            display mode   : 0

Start testing at Thu Aug 22 14:07:48 1996

 Test Statistics:

 Results: # records   : 384
          # goods     : 288
          # bads      : 96

          recognition : 75.000000 %


The work is done ?! at Thu Aug 22 14:07:57 1996

 B Y E !!!
```

B.2 The Network Description File

When the network is stored on disk, a new directory is created, containing a number of files representing different parts of the modular network. Considering a MNN called `multi` with five modules in the input layer and a decision network, the following files would be created:

```
% ls
multi.0
multi.1
multi.2
multi.3
multi.4
multi.dec
multi.inp
multi.net
```

The file `multi.net` is the main description file:

```
% cat multi.net
80 :no_user_inps
2 :no_classes
5 :no_nets_L1
1 :no_out_net_L1
5 :no_main_inps
2 :no_main_outs
multi.inp :filenames_L1
multi.dec :filename_m
```

The file `multi.inp` is the input layer description file:

```
% cat multi.inp
5 :no_of_nets
16 :no_of_inputs_p_net
80 :no_of_inputs_user
80 :no_inputs_p_layer
1 :no_outputs_p_net
17 :rnd_seed
-----the_net_description_files----
multi.0
multi.1
multi.2
multi.3
multi.4
```

The file `multi.dec` is the decision network description file:

```
% cat multi.dec
2 :no_layers
5 :no_of_inputs
1.000000 :lambda
1 :function
4 2 :no_neurons_per_layer
-----------------

4.858376 -8.901467 3.486753 -7.278128 -8.241478 -7.557687 ;
7.999052 0.022757 8.855008 5.218701 1.913574 11.766893 ;
-1.990103 -3.054739 0.905476 -1.190806 -1.460390 0.501350 ;
-6.788280 -4.840351 3.461928 -3.261231 -2.135085 -1.465473 ;

-----------------

2.141190 -3.346811 1.701500 3.693212 0.955209 ;
-2.151395 3.352007 -1.418384 -3.804537 -0.951071 ;

END.
```

The files `multi.?` are the description files for the modules in the input layer. They all have the same format; one is shown as an example:

```
% cat multi.0
2 :no_layers
16 :no_of_inputs
1.000000 :lambda
1 :function
4 1 :no_neurons_per_layer
-----------------

-0.443 -3.3273 -0.0486 -0.6680 -4.7130 3.5100 0.0980 3.0047
-3.984 -0.924 -0.82 -0.386 4.196 -0.152 -1.095 0.2077 1.8940 ;
-1.110 -2.3721 -0.640 -0.4547 -2.667 -3.932 0.201 1.311 4.40
0.5744 -0.7802 -0.3600 -7.4475 -5.5865 -7.9615 0.4967 -5.0221 ;
3.784 1.732 0.2465 -0.76 4.2979 4.784 0.4800 -3.2210 1.7857
-4.1440 2.8404 0.3333 8.7203 -3.1325 -1.4015 -0.2533 6.6640 ;
-1.354 -0.9127 0.766 0.673 1.322 -1.1493 0.078 -4.991 -6.721
1.8525 -1.924 0.966 -3.2772 -1.8000 -1.5567 0.1700 -1.7079 ;

-----------------

-5.660216 -1.537823 6.177195 -1.776341 -0.448600 ;

END.
```

Appendix C

Binary Pictures

Please contact me if you like to see the full set of pictures used in the binary picture recognition experiment described in section 7.2.

References

[alek89a] Igor Aleksander (ed.). Neural Computing Architectures. The Design of Brain-Like Machines. North Oxford Academic Publishers Ltd. 1989.

[alek89b] Igor Aleksander. Canonical Neural Nets Based on Logical Nodes. In: IEE Proceedings. International Conference on Artificial Neural Networks. Pages 110-114. London 1989.

[alek95] Igor Aleksander and Hellen Morton. An Introduction to Neural Computing. Second Edition. Chapman & Hall 1995.

[anan95] Rangachari Anand, Kishan Mehrotra, Chilukuri K. Mohan, Sanjay Ranka. Efficient classification for multiclass problems using modular neural networks. In: IEEE Transactions on Neural Networks, Vol. 6, No. 1. Pages 117-124. January 1995.

[band96] Zuhair Bandar. Lecture on Neural Networks. MSc Course. Department of Computing. Manchester Metropolitan University. Not Published. 1996.

[barr88] Murray L. Barr & John A. Kiernan. The Human Nervous System. An Anatomical Viewpoint. Fifth Edition. Harper International 1988.

[beal90] R. Beale. Neural Computing. An Introduction. IOP Publishing. London 1990.

[bell95] R. Bellotti, M. Castellano, C. De Marzo, G. Satalino. Signal/Background classification in a cosmic ray space experiment by a modular neural system. In: Proc. of the SPIE - The International Society for Optical Engineering, Vol. 2492, No. pt.2. Pages 1153-1161. 1995.

[bich89] M. Bichsel. Image processing with optimum neural networks. In: First IEE International Conference on Artificial Neural Networks, Conference Publication No. 313 IEE. Pages 374-377. London, 16-18 October 1989.

[blon93] P. Blonda, V. LaForgia, G. Pasquariello, G. Satalino. Multispectral classification by modular neural network architecture. In: IGARSS '94. International Geoscience and Remote Sensing Technologies, Data Analysis and Interpretation, Vol. 4. Pages 1873-1876. New York, USA 1993.

[boer92] Egbert J.W. Boers and Herman Kuiper. Biological metaphors and the design of modular artificial neural networks. Master's thesis. Departments of Computer Science and Experimental and Theoretical Psychology at Leiden University, the Netherlands. 1992.

[canu95] A. M. P. Canuto, E. C. B. C. Filho. A Generalization Process for Weightless Neurons. In: Artificial Neural Networks, Conference Publication No. 409 IEE. Pages 183-188. 26-28 June 1995.

[chia94] Cheng-Chin Chiang and Hsin-Chia Fu.Divide-and-conquer methodology for modular supervised neural network design. In: Proceedings of the 1994 IEEE International Conference on Neural Networks. Part 1 (of 7). Pages 119-124. Orlando, FL, USA 1994.

[chri88] Kaare Christian. The UNIX Operating System. Second Edition. 1988.

[creu77] O.D. Creutzfeld. Generality of the functional structure of the neocortex. Naturwissenschaften, Vol. 64. Pages 507-517. 1977.

[dayh90] Judith Dayhoff. Neural Network Architectures. An Introduction. New York 1990.

[ergz95] S.Ergezinger and E. Thomsen. An Accelerated Learning algorithm for Multilayer Perceptrons Optimization Layer by Layer. In: IEEE Transactions on Neural Networks, Vol.6, No. 1. January 1995.

[eyse93] Michael W. Eysenck. Principles of Cognitive Psychology. LEA. Hove 1993.

[farr93] S. Farrugia, H. Yee, P. Nickolls. Modular connectionist architectures for multi-patient ECG recognition. In: 3rd International Conference on Artificial Neural Networks, IEE Conference Publication No. 372. Pages 272-276. Brighton, England 1993.

[faus94] Laurene Fausett. Fundamentals of Neural Networks. Arichtectures, Algorithms, and Applications. Prentice-Hall. New Jersey 1994.

[gros87] S. Grossberg Competitive Learning: From Interactive Activation to Adaptive Resonance. In: Cognitive Science, Vol. 11. Pages 23-63. 1987.

[happ94] Bart L.M. Happel & Jacob M.J. Murre. Design and Evolution of Modular Neural Architectures. Neural Networks, Vol. 7, No. 6/7, Pages 985-1004. 1994.

[harv94] Robbert L. Harvey. Neural Network Principles. Prentice Hall. New Jersey 1994.

[hass95] Mohamad H. Hassoun. Fundamentals of Artificial Neural Networks. MIT Press 1995.

[hayk94] Simon Haykin. Neural Networks. A Comprehensive Foundation. New York 1994.

[hebb49] D. O. Hebb The Organization of Behavior. John Wiley & Sons. New York 1949.

[hech87] R. Hecht-Nielsen. Kolmogorovo's Mapping Neural Networks Existence Theorem. Proceedings of the First IEEE International Conference on Neural Networks, Vol. 3. Pages 112-114. San Diego 1987.

[hech89] R. Hecht-Nielsen. Theory of Backpropagation Neural Networks. Proceedings of the International Joint Conference on Neural Networks, Vol. 1. Pages 593-605. Washington 1989.

[hell95] I. S.Helliwell, M. A. Turega, and R. A. Cottis. Accountability of neural networks trained with 'real world' data. In: Artificial Neural Networks, Conference Publication No. 409 IEE. Pages 218-222. 26-28 June 1995.

[hold95] Sean B. Holden and Peter j. W. Rayner. Generalization and PAC Learning: Some New Results for the Class of Generalized Single-Layer Networks. In: IEEE Transactions on Neural Networks, Vol.6, No. 2. March 1995.

[huss94] Basit Hussain and M. R. Kabuka. A Novel Feature Recognition Neural Network and its Application to Character Recognition. In: IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 16, No. 1. Pages 98-106. January 1994.

[ishi95] Masumi Ishikawa. Learning of modular structured networks. In: Artificial Intelligence, Vol. 75, No. 1. Pages 51-62. 1995.

[kala92] James W. Kalat. Biological Psychology. Fourth Edition. Brooks/Cole Publishing Company. 1992.

[kim94] Jongwan Kim, Jesung Ahn, Chong Sang Kim, Heeyeung Hwang, Seongwon Cho. Multiple neural networks using the reduced input dimension. In: Proceedings – ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing, Vol 2. Pages 601-604. Piscataway, NJ, USA, 1994.

[kimo90] Takashi Kimoto, Kazuo Asakawa, Morio Yoda, Masakazu Takeoka.Stock market prediction system with modular neural networks. In: 1990 International Joint Conference on Neural Networks – IJCNN 90. Pages 1-6. Piscataway, NJ, USA 1990.

[koho82] T. KOHONEN Self-organized Formation of Topologically Correct Feature Maps. In: Biological Cybernetics, Vol. 43. Pages 59-69. 1982.

[koho88] T. KOHONEN, G. BARNA, AND R. CHRISLEY. Statistical pattern recognition with neural networks: benchmarking studies. In: Proc. IEEE International Conference on Neural Networks. Pages 61-67. San Diego. 1988.

[kolm57] A. N. KOLMOGOROV. On the Representation of Continuous Functions of Many Variables by Superposition of Continuous Functions of One Variable and Addition. Doklady Akademii Nauk SSR, Vol. 114. Pages 953-956. 1957. English translation. Mathematical Society Transaction, Vol 28. Pages 55-59. 1963.

[kung93] S. Y. KUNG. Digital Neural Networks. Prentice Hall. New Jersey 1993.

[kwon94] TAEK KWON, HUI CHENG, MICHAEL ZERVAKIS. Modular neural networks for function approximation. In: Proceedings of the Artificial Neural Networks in Engineering Conference (ANNIE'94). Pages 11-16. St. Louis, MO, USA 1994.

[leng96] RÉGIS LENGELLÉ AND THIERRY DENOEUX. Training MLPs Layer by Layer Using an Objective Function for Internal Representation. In: Neural Networks, Vol.9 , No. 1. Pages 83-97. 1996.

[lu95] B.-L. LU, K. ITO, H. KITA, Y. NISHIKAWA. Parallel and modular multisieving neural network architecture for constructive learning. In: Proceedings of the 4th International Conference on Artificial Neural Networks. Conference Publication No. 409. Pages 92-97. Cambridge, UK 1995.

[marr82] DAVID MARR. Vision: a computational investigation into human representation and processing of visual information. San Francisco, 1982.

[mast93] TIMOTHY MASTERS. Practical Neural Network Recipes in C++. Academic Press, INC 1993.

[mccu43] W. S. McCulloch and W. Pitts A Logical Calculus of the Ideas Immanent in Nervous Activity. In: Bulletin of Mathematical Biophysics, Vol. 5. Pages 115-133. 1943.

[mins69] M. Minsky and S. Papert. Perceptrons. MIT Press. Cambridge, MA. 1969.

[mraz95] I. Mrázová. The Robustness of BP-Networks. In: Artificial Neural Networks, Conference Publication No. 409 IEE. Pages 234-239. 26-28 June 1995.

[mui94] L. Mui, A. Agarwal, A. Gupta, P. Shen-Pei Wang. An Adaptive Modular Neural Network with Application to Unconstrained Character Recognition. In: International Journal of Pattern Recognition and Artificial Intelligence, Vol. 8, No. 5, Pages 1189-1204. October 1994.

[murr92] Jacob M.J. Murre. Learning and categorization in modular neural networks. 1992.

[nnbc96] The Neural-Network Benchmark Collection. Via FTP from ftp.cs.cmu.edu [128.2.206.173] in directory /afs/cs/project/connect/bench. 1996.

[patt96] Dan W. Patterson. Artificial Neural Networks. Theory and Applications. Prentice Hall. Singapure 1996.

[pict96] Picture-Directory. University Stuttgart. Via FTP from ftp.uni-stuttgart.de in directory /pub/graphics/pictures. 1996.

[posn88] M.I. Posner, S.E. Peterson, P.T. Fox, M.E. Reichle. Localization of cognitive operations in the human brain. In: Science. Vol. 240. Pages 1627-1631. 1988.

[prec95] L. Prechelt. A Quantitaive Study of Experimental Neural Network Learning Algorithm Evaluation Practicies. In: Artificial Neural Networks, Conference Publication No. 409 IEE. Pages 223-227. 26-28 June 1995.

[prob96] PROBEN1. Neural Network Benchmark Database.
Via FTP from ftp.cs.cmu.edu [128.2.206.173] as
/afs/cs/project/connect/bench/contrib/prechelt/proben1.tar.gz
or ftp.ira.uka.de [129.13.10.90] as /pub/neuron/proben.tar.gz. 1996.

[quin96] QUINLAN@CS.SU.OZ.AU. Credit Approval. Via FTP from [128.195.1.1] in directory /pub/machine-learning-database/. 1996.

[rose58] F. ROSENBLATT The Perceptron: a Probabilistic Model for Information Storage and Organization in the Brain. In: Psychological Review, Vol. 65. Pages 386-408. 1958.

[rume86] D.E. RUMELHART, G.E. HINTON, AND R.J. WILLIAMS. Learning internal representations by error propagation. In: D.E. Rumelhart, and J.L. McClelland (Eds.), Parallel distributed processing. Volume I: Foundations. Cambridge, MA: MIT Press 1986.

[sarl96] WARREN S. SARLE. FAQs on Neural Networks. Via FTP from ftp.sas.com as /pub/neural/FAQ.html. 1996.

[schm96] ALBRECHT SCHMIDT. Implementation of a Multilayer Backpropagation Network. In-Course-Project. Department of Computing. Not Published. MMU 1996.

[sejn96] TERRY SEJNOWSKI AND R. PAUL GORMAN. Sonar, Mines vs. Rocks.
Via FTP from ftp.cs.cmu.edu in directory
/asf/cs.cmu.edu/project/connect/bench. 1996.

[sigi96] VINCENT SIGILLITO. Pima Indians Diabetes Database. National Institute of Diabetes and Digestive and Kidney Diseases. Via FTP from [128.195.1.1] in directory /pub/machine-learning-database/. 1996.

[simp90] PATRICK K. SIMPSON. Artificial Neural Systems. Foundations, Paradigms, Applications, and Implementations. Pergamon Press 1990.

[ston95] J. V. Stone and C. J. Thorton. Can Artificial Neural Networks Discover Useful Regularities?. In: Artificial Neural Networks, Conference Publication No. 409 IEE. Pages 201-205. 26-28 June 1995.

[svan92] Josef Švanda. Modular set of analog neural blocks. In: Neurocomputing, Vol. 4, No. 1-2. Pages 103-107. Febuary 1992.

[ucim96] UCI machine learning database. Via FTP from ics.uci.edu [128.195.1.1] in directory /pub/machine-learning-databases. 1996.

[zell94] Andreas Zell. Simulation Neuronaler Netze. Addison-Wesley 1994.

[zhan91] Sheng-Wei Zhang and T.J. Stonham. Universal Architectures for Logical Neural Nets. In: Second International Conference on Artificial Neural Networks, Conference Publication No. 349 IEE. Pages 262-266. 18-20 November 1991.

[zhao95] Mingsheng Zhao, Youshou Wu, Xianquig Ding. Classification for multiclass problems based on modular neural networks of two class problems. In: Proceedings of International Conference on Neural Information Processing (ICONIP '95), Vol. 2. Pages 845-848. 30.10.-2.11 1995.