Online vs Offline learning with large datasets

Author: Yudong Wang
Studentnr: 9228284
Thesis number: 406
Mentor: Dr. Th. E. Schouten
Faculty of Informatics
K.U. Nijmegen

Preface

I study computer science because computers fascinate me, all the things that one can do with computers, besides playing those awesome games. During my study I became interested in artificial intelligence, the thought of building a machine that behaves like a human being, although this sounds like science fiction and daydreaming. To get a glimpse of what AI is real like in reality, I chose this project.

I would like to thank my supervisor Dr. Th. E. Schouten for his supervision. Also I want to thank A. Sardar and S. Hobo for their pleasant interruptions with ping-pong during this project.

Yudong Wang
March 1997

Contents

Chapter 1

Introduction

1.1 Performance of online and offline neural networks

Neural networks are important tool for pattern recognition and function approximation. There are many types of neural networks, a popular one is the backpropagation network. A neural network is hard to train. Considering the updating of the weights after a pattern is fed to a neural network, there are two extremes, online and offline backpropagation.

Offline learning, also known as batch-learning, updates the weights after a complete pass of the training data, using the error e over the whole training set. An epoch refers to a complete pass through the training data, thus, offline methods update the weights after each epoch.

Online learning updates the weights after each single training pattern (a pair of input and target vectors). The error e_p for a single pattern p is used. Online methods therefore update the weights several times (epoch size) during an epoch.

An offline optimization procedure can often be transformed into an online procedure by using the error for the current pattern, e_p, instead of the error over the whole training set, e. Weights may also be updated after any number of input/output presentations, so online and offline learning are two extremes.

Due to the arbitrariness in the presentation order of the training patterns, online learning is a stochastic process; whereas batch-mode learning is deterministic. It's assumed that online will outperform offline learning, since the random arrival of the patterns during an online learning process will introduce noise. This noise increases the possibilities to escape from a local minima and converge to a global minima of the weight-error space. In this paper a comparison between online and offline learning is made by experiments.

Chapter 2

Introduction to Neural Networks

2.1 Biological neurons

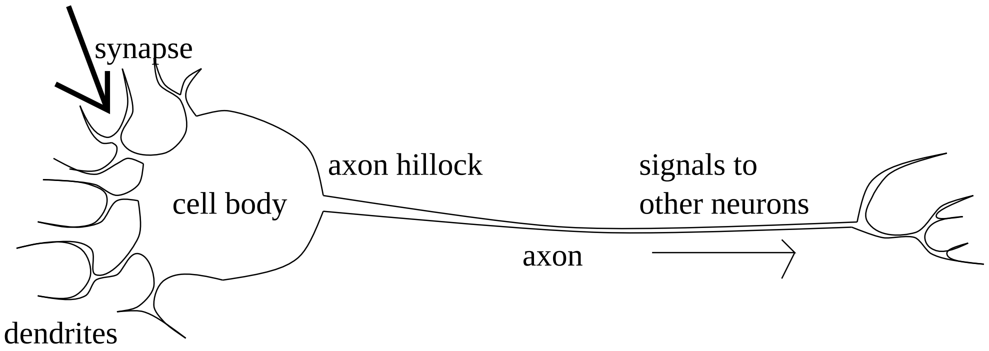


Figure 2.1: Biological neuron

Nervous signals are transmitted either electrically or chemically. Electrical transmission (action-potential or 'spike' train) prevails in the interior of a neuron, whereas chemical mechanisms operate between different neurons, i.e. at the synapses.

The electrical pulses impinge on the afferent neuron at terminals called synapses. These are found on a set of branching processes emerging from the cell body (soma) known as dendrites. Each pulse occurring at the synapse initiates the release of a small amount of chemical substance of neurotransmitters which travels across the synaptic cleft. These neurotransmitters are then received at the post-synaptic receptor sites on the dendritic side of the synapse.

The neurotransmitters becomes bound to molecular sites here which, in turn, causes a change in the dendritic membrane potential. This post-synaptic-potential (PSP) change may serve to increase (hyperpolarise) or decrease (depolarise) the polarization of the post-synaptic membrane. In the former case, the PSP tends to inhibit generation of pulses in the afferent neuron, while in the latter, it tends to excite the generation of pulses.

Each PSP will travel along its dendrites and spread over the soma, eventually reaching the base of the axon (axon-hillock). The afferent neuron sums or integrates the effects of thousands of such PSPs over its dendritic tree and over time. If the integrated potential at the axon-hillock exceeds a threshold, the cell 'fires' and generates an action potential or spike which starts to travel along its axon. This then initiates the whole sequence of events again in neurons contained in the efferent pathway[Gu96].

The incredible complexity of the human central nervous system, especially of the human brain, rests not so much in the complexity and diversity of the single nerve cell, which is quite limited, as in the vast number of its constituent units, i.e. of the neurons and their mutual connections[Re90].

The total numbers of neurons in the human central nervous system is in the vicinity of 10^{11}, and 10^{15} synaptic connections in the human brain. In the human cortex every neuron is estimated to receive converging inputs on the average from about 10000 synapses. Each cell feeds its output into many hundreds of other neurons. The above figures illustrate quantitative side of the human nervous system, even this can not be approximated by nowadays computer technology [1].

2.2 A Definition

The neural network model is derived from research into the nature of the human behavior, especially the human brain. This model is referred to by various names, neural networks, neuron-computers, parallel distributed processing (PDP) models, neuromorphic systems, layered self-adaptive networks, and connections models.

It is not merely researchers from computer science who are studying neural networks, also scientists from other fields like biology, physics and econ-

[1] The reader interested in the biology of the brain is encouraged to consult books dealing specially with this subject, e.g.[Bu77, Co72, Ec77, Ka85, Sh79]

omy are fascinated by the possibilities of a neural network. In mathematical terms a neural networks(neural nets for short) is defined as a directed graph with the following properties[Re90]:

1. A state variable n_i is associated with each node i.

2. A real-valued weight w_{ik} is associated with each link(ik) between two nodes i and k.

3. A real-valued bias ϑ_i is associated with each node i.

4. A transfer function $f_i[n_k, w_{ik}, \vartheta_i, (k \neq i)]$ is defined, for each node i, which determines the state of the node as a function of its bias, of the weights of its incoming links, and of the states of the nodes connected to it by these links.

2.3 A Neuron

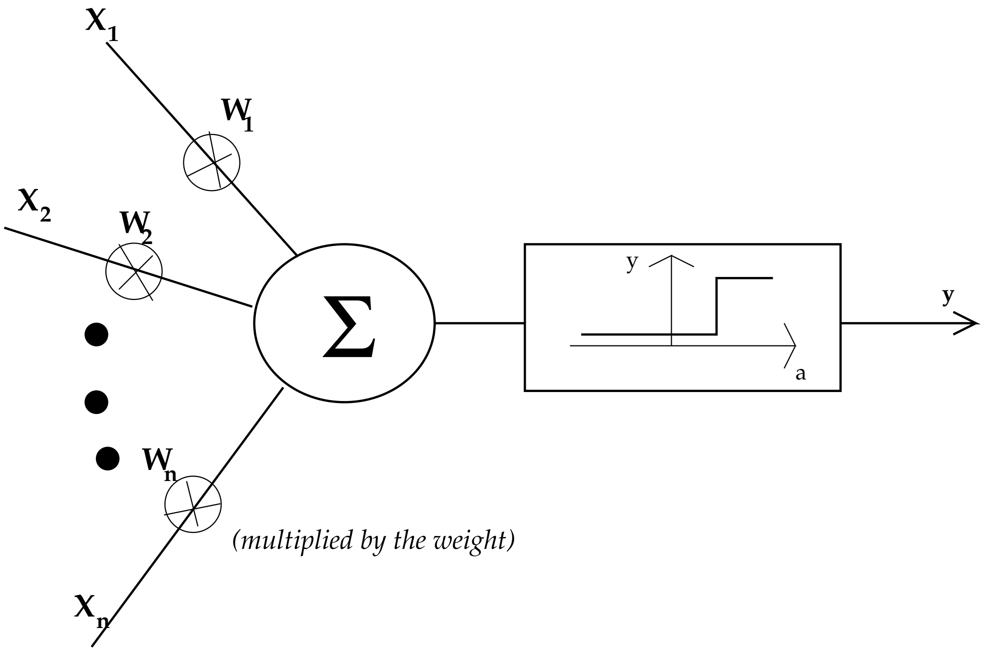


Figure 2.2: A simple neuron

The information processing performed by a simple neuron may be summarized as follows (figure 2.2):

- signals (action-potentials) appear at the unit's inputs (synapses)
- the effect each signal can be approximated by multiplying the signal by some number or weights indicating the strength of the synapse
- the weighted signals are now summed to produce an overall unit activation
- if this activation exceeds a certain threshold the unit procedures an output response (non-zero value).

This simple neuron is also known as the Threshold Logic Unit[Gu96], where the input and output signals are binary.

2.4 Threshold functions

Threshold functions determine the mapping of the input signals to a output domain. Here are some commonly used threshold functions:

- $f(x) = sign(x)$
  This so called step threshold function maps the inputs into two values. The step function is used for binary input/output values.
- Sigmoid functions:
  When the input and output domains are extended to real numbers, the step function is not sufficient anymore, and a sigmoid function is used. A sigmoid function is a continuous 'squashing' function, which is bounded, monotonic, non-decreasing. This function provides a graded and non linear response. Three examples of sigmoid functions:
  -
    $$f(x) = \frac{x}{1 + abs(x)}$$

    This is the Elliot function. This sigmoid approaches its extremes more slowly. Which means that more iterations is needed when trying to reach the target values. The advantage of Elliot is that it needs less CPU time to compute.

–

$$f(x) = \frac{1}{1 + e^{-x/p}}$$

This sigmoid has 0 and 1 as the bounded values. Here x is the weighted sum of the input signals, and p determines the shape of the sigmoid: a larger value making the curve flatter. Often this parameter is omitted so that is implicitly assigned the value 1.

–

$$f(x) = tanh(x)$$

The saturation levels of $tanh$ are -1 and $+1$.

2.5 Topology

The topology deals with the way in which the neurons are organized into layers and linked with weighted interconnections[Gu96].

2.5.1 Feed-forward nets

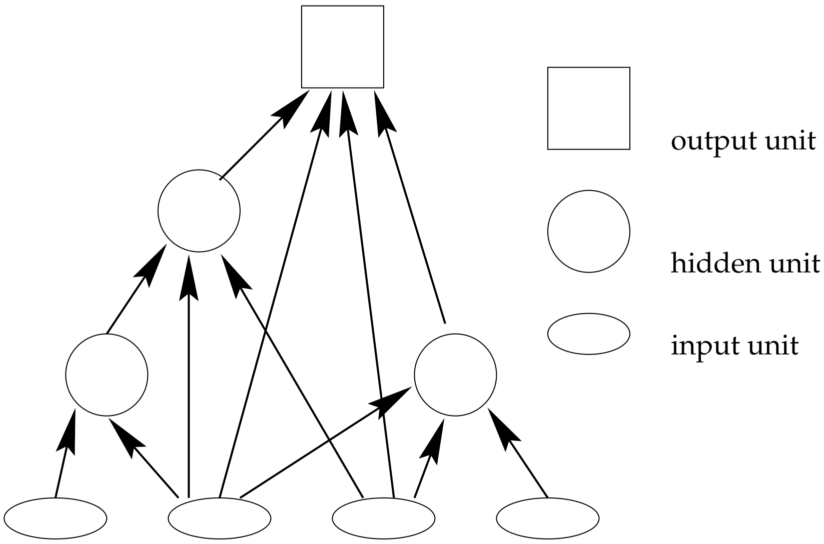


Figure 2.3: A feed-forward network

Feed-forward nets contain no feedback loops in their inter-node connection paths. Thus, starting from any node in the net and tracing its fan-out, after a finite number of steps, this process will terminate at an output node which does not have a fan-out (see figure 2.3).

Nodes

Concerning their place in the neural nets, there are three kinds of nodes:

- input nodes
  These nodes are accessible for the training environment and may therefore have signals directly imposed on them from outside. The data to be trained has to be in the format which is suited for the input nodes.

- output nodes
  These nodes are the complementary of the input nodes. The output nodes have no fan-out, but this is not always true as we will see later in case of recurrent nets.

- hidden nodes
  The hidden nodes resides between above two. They are not accessible by the environment. It is the task of these units to develop an internal representation of the training set.

Layers

With the general feed-forward net, it is possible to distinguish a series of distinct layers. Formally a layer may be described in the following way: there does not exist a set of three units A, B and C, such that C receives input from A and B, and B receives input from A. An example of the forbidden structure is shown in figure 2.4 together with a typical 3-layer feed-forward net. In a layered network, inputs are applied to the input nodes. The subsequent (hidden) layers evaluate their node outputs until the output layer is reached. So the computational process is done layer by layer. This process is referred to as a forward pass.

2.5.2 Recurrent nets

Contrary to feed-forward nets, recurrent nets have feedback loops in their inter-node connection paths. Thus there must exist at least one node such that, if the fan-out of that node is traced forward, then after a finite number of steps this node is revisited. In recurrent nets there is no distinction between input and output, rather there is a set of visible units which interface with the environment. Just like the feed-forward case, there is a number

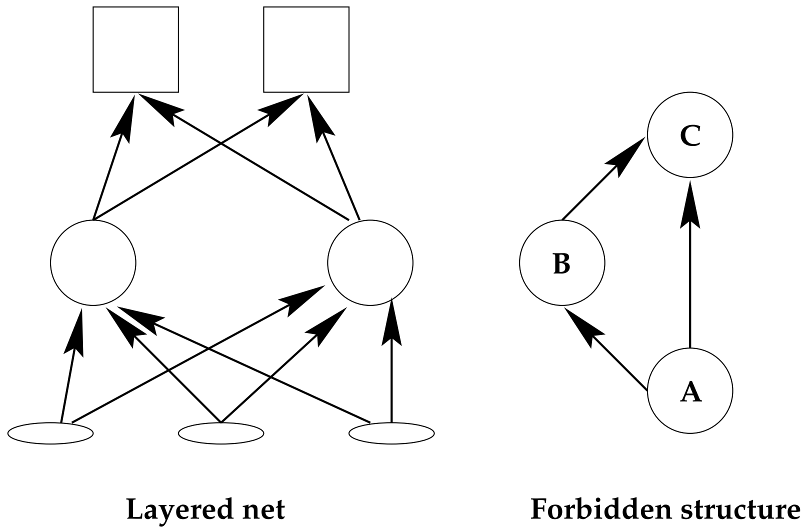


Figure 2.4: Forbidden layered structure

of hidden units in recurrent nets. With recurrent nets, the update can be done in two ways, synchronous or parallel update (asynchronous update). In the former case, all the nodes evaluate their new output together, and the net is 'clocked' globally. In the latter case, a node is selected at random to be updated. These are extreme cases. With the general case there is a probability p that any node gets updated.

In the case of synchronous update and with deterministic nodes, the net is actually a deterministic finite automation with a characteristic state space, where the state is the vector of unit outputs. The transition within the state space is only terminated when a cycle is reached. Therefore this type of dynamics implies that one has to wait for the net to reach some kind of equilibrium(cycle).

Asynchronous operation implies that the net is no longer deterministic, since nodes are selected for update at random. The state structure is probabilistic and more complex than the synchronous case. The concept of reaching equilibrium can be retained by the introduction of an 'energy function'[Gu96] associated with the net. The energy is chosen so that each neuron update implies as a decrease in its value, so in the energy 'landscape', the transition is guaranteed to be 'downhill' in the energy space.

Recurrent can be used for associative memory[2]. Compared to infor-

[2]Early studies of associative memory in the 70s[An72, Na72, Ko72, Am72], and later

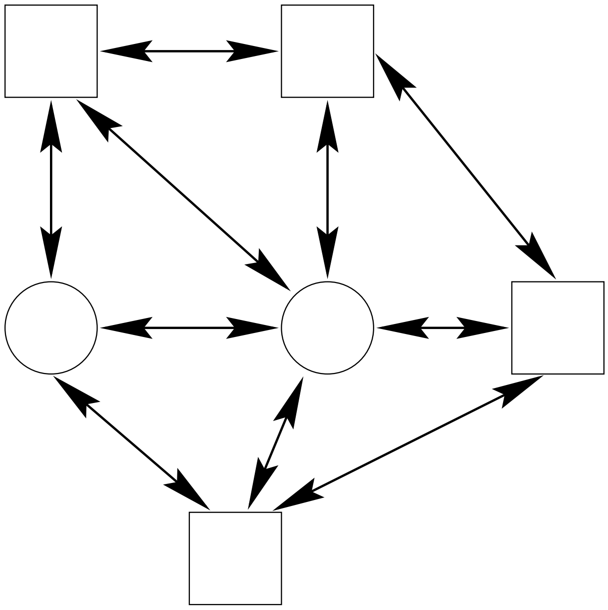


Figure 2.5: Recurrent network

mation retrieval with conventional computers, which is done by memory-address, information storage and recall using associative memory is done by association with other information. So this is done on the basis of a partial knowledge of its content, but without knowing its storage location. This is also called content-addressable memory. This way it is showing a similarity with the recall of stored information by the human brain, which also relies more on association with other stored data rather than on the order in which the memory was acquired.

2.6 Learning

2.6.1 Supervised learning

Supervised is learning is usually performed with feed-forward nets where training patterns are composed of two parts, an input vector and an output vector, associated with the input and output nodes respectively. A training cycle consists of the following steps:

- a pair of input and output vectors presented to the neural nets

by Hopfield[Ho82]

- forward pass with the input vector
- the errors between the desired (the presented one) and actual (computed by the forward pass) response for each node in the output layer are found
- these errors are then used to determine weight changes in the net according to the prevailing learning rule.

The term 'supervised' originates from the fact that the desired signals on individual output nodes are provided by an external 'teacher'. This technique occurs in the backpropagation algorithm, the delta rule and the perceptron rule.

2.6.2 Unsupervised learning

Unsupervised learning is usually found in recurrent and competitive nets. There is no separation of the training set into input and output pairs. A training cycle will consist of the following steps:

- a vector is applied to the visible (input) nodes
- the net is allowed to reach equilibrium
- weight changes are made according to some prescription

The disappearance of the external supervisor providing target outputs gives this scheme its name. This process relies only upon local information and internal control. Unsupervised learning is sometimes referred to as self-organization.

2.7 Linear separation of classes

Consider a simple neural nets (figure 2.6). The table of response is shown in the following table:

x_1	x_2	activation	output
0	0	0	0
0	1	1	0
1	0	1	0
1	1	2	1

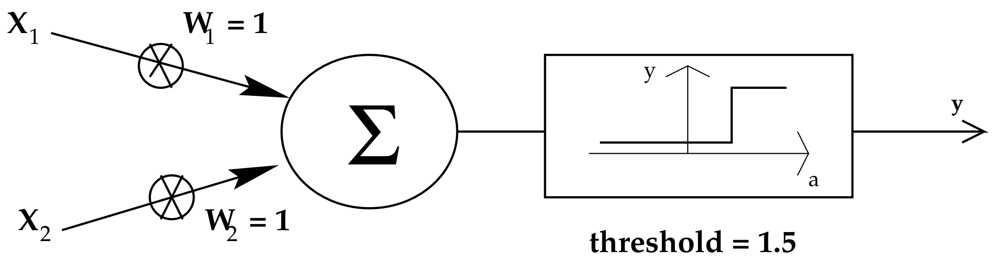


Figure 2.6: A simple neural net

where :

$$\begin{aligned} a(ctivation) &= x_1 \times w_1 + x_2 \times w_2 \\ o(utput) &= 0 : a(ctivation) < t(hreshold) \\ &= 1 : else \end{aligned}$$

The input patterns are classified into two classes: those that give output '1' and those that give output '0'. Each input pattern has two components, x_1, x_2. Therefore these patterns can be represented in two-dimensional space (figure 2.7). The space in which the inputs reside is referred to as the pattern space. Each pattern is a point in the space by using its component values as space-coordinates. In general, for n-inputs, the pattern space will be n-dimensional.

In 2-D case:

$$w_1 x_1 + w_2 x_2 = t$$

can be written as:

$$x_2 = -\frac{w_1}{w_2} x_1 + \frac{t}{w_2}$$

This is a equation of a line:

$$x_2 = a x_1 + b$$

In this case, $a = -1$ and $b = 1.5$ The two classes are therefore separated by the 'decision' line (see figure 2.8). This is called linear separable. It is possible to generalize this result with n inputs. In 3-D the two classes are separated by a decision-plane. In n-D this becomes a decision-hyperplane.

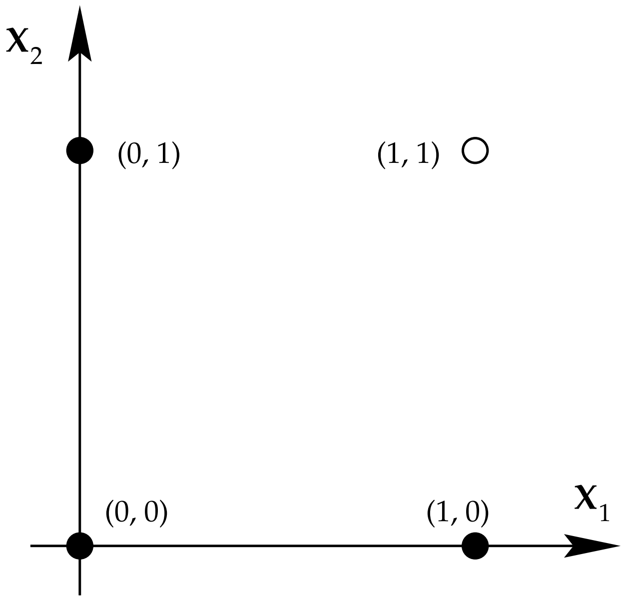


Figure 2.7: Patterns in 2D space

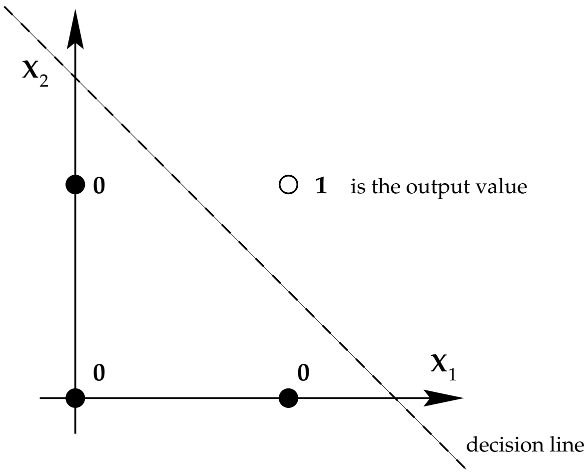


Figure 2.8: Patterns separated by a decision line

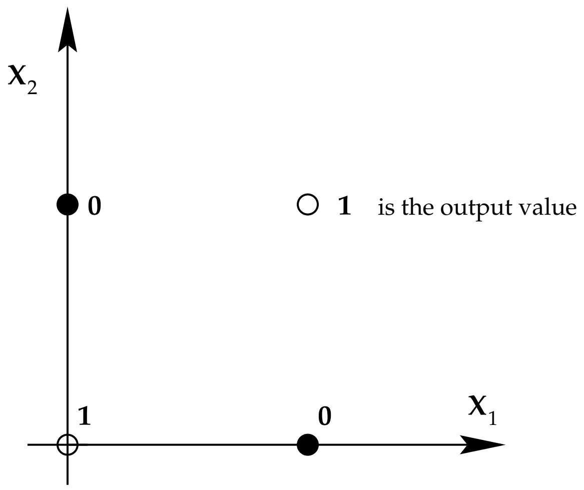


Figure 2.9: Patterns of the XOR-problem in 2D space

The famous XOR problem in figure 2.9 can not be solved by the simple neural nets from figure 2.6. Since one can not draw a line to separate these two classes. The problem is solved by using two lines. These two lines can be realized by a neural nets of two layers. Further expansion of the neural nets is illustrated in figure 2.10 [3].

A three-layer network implements decision surfaces of arbitrary complexity. The number of nodes used in each layer determines the complexity. A multi-layer neural nets can be trained by using the backpropagation algorithm.

[3] adapted from [Go92]

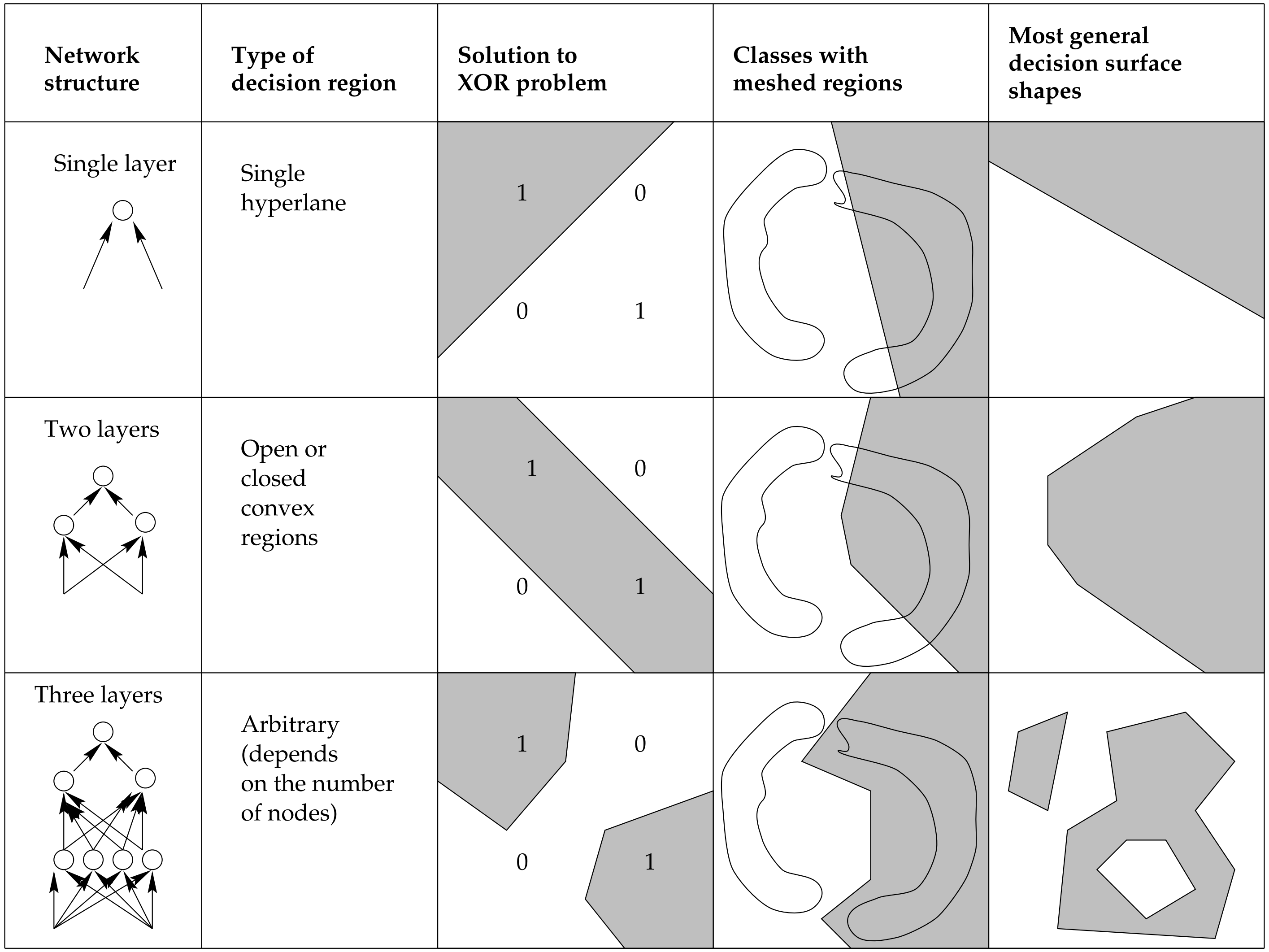


Figure 2.10: Classification by multi-layered network

Chapter 3

Backpropagation

Backpropagation is a common used algorithm for training a feed-forward, multi-layered neural nets. It was developed independently around 1985 by several research groups [Pa85, Ru86a]. The algorithm is called backpropagation because it first computes the changes needed to the weights in the last layer, then uses this to compute the changes to the weights in the preceding layer, and finally the initial layer. This procedure is explained heuristically in the next section, then described mathematically in its following section[Wi92].

3.1 Heuristical approach

In a feed-forward neural nets, the input is fed to the input nodes, after going through the layers of the net, an actual output o is computed at the output nodes. If this output is not equal to the desired output d, then we have an error, $d - o$, which depends on the current weight settings w. A big change to the weights is useful, if this change can reduce a large output error and if the size of that reduction is substantial. On the other hand, if a change to the weights does not result in a reduction of any output error substantially, little should be done to that weight.

Most of the computations needed to compute the change to any particular weight is also needed to compute the changes to weights that are closer to the output nodes. In figure 3.1 , once you see how to compute a change for a particular weight, $w_{i\to j}$, between a node in layer i and a node in layer j, you see that the required computations involve computations needed for the weights, $w_{j\to k}$, between a node in layer j and a node in layer k.

Let's give some thoughts to the way how the changes to the weights

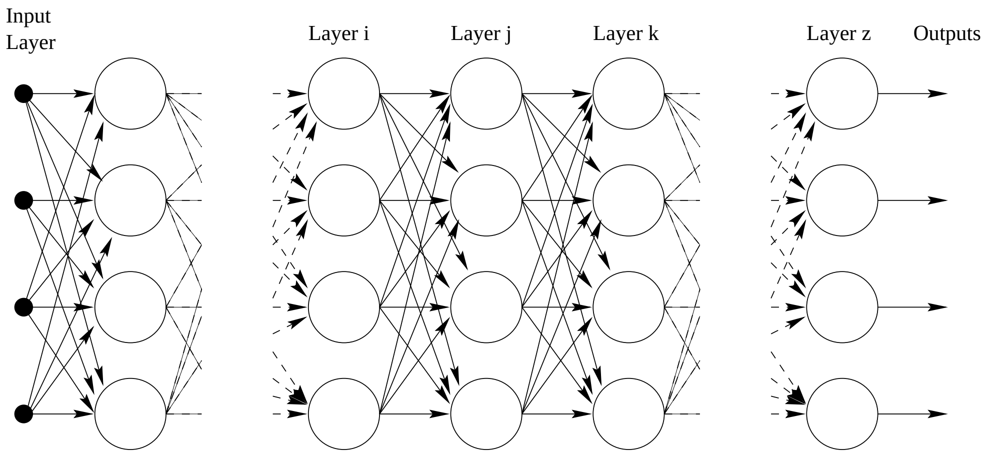


Figure 3.1: A multi-layered network

should be done. First let's consider the fact that every node has a threshold function. So a change in the input at node j results in a change in the output at node j that depends on the slope of this threshold function, o'_j. Where the slope is steepest, a change in the input has the maximum effect on the output. So the change in $w_{i \to j}$ is proportional to the slope of the threshold function of layer j.

Next, the change in the input to the nodes of layer j, depends on the output of layer i. So the change in $w_{i \to j}$ is proportional to o_i, the output at layer i.

At last, the weight change $w_{i \to j}$, should also capture how beneficial it is to change the output nodes at layer j. The Greece letter β is commonly used for this purpose, it stands for the benefit obtained by changing the output value of a node. So concluded: The change to $w_{i \to j}$ should be proportional to $o_i \times o'_j \times \beta_j$.

Now consider the fact that a node in layer i is connected to many nodes in the next layer. So the overall benefit is the sum of the individual benefits: $\beta_i = \sum_j w_{i \to j} o'_j \beta_j$

To finish the analysis, we have to deal with the nodes in the output layer. This value depends on how wrong the output node's value happens to be. If the difference between the d_z, the desired output at node z, and the o_z, the actual output at node z, is small, then the change in the output at node z should be small too. On the other hand, if the difference is large, then the change in the output of node z should be large too. So the change to

o_z should be in proportion to the difference, $d_z - o_z$. In the framework of benefit, we have:
$\beta_z = d_z - o_z$
Finally, weight changes shouldn't be too large, it depends on a rate parameter r, that should be as large as possible to increase the performance, but not too large to cause overshooting the desired solution. This parameter r can only be determined experimentally.

Now the formulas for backpropagation:
$\Delta w_{i \to j} = r o_i o'_j \beta_j$,
$\beta_j = \sum_k w_{j \to k} o'_k \beta_k$ for nodes in the hidden layers,
$\beta_z = d_z - o_z$ for nodes in the output layer.

3.2 Mathematical approach

Two ideas from calculus:

- y is a smooth function of several variables, x_i. If you want to make incremental changes to the initial values of each x_i so as to increase the value of y as fast as possible. Then, the change to each initial x_i value should be in proportion to the partial derivative of y with respect to that particular x_i:

$$\Delta x_i \propto \frac{\partial y}{\partial x_i}$$

  Making such a change is called gradient ascent.

- y is a function of several intermediate variables x_i, and that each x_i, on its turn, is a function of one variable, z. For computing the derivative of y with respect to z, you can obtain that by summarize the results that you obtained by multiplying each partial derivative of y with respect to the x_i by the derivative of x_i with respect to z :

$$\frac{dy}{dz} = \sum_i \frac{\partial y}{\partial x_i} \frac{dx_i}{dz} = \sum_i \frac{dx_i}{dz} \frac{\partial y}{\partial x_i}$$

  This is called the chain rule.

Now let's return to our problem with the neural nets, where we have a set of nodes with their corresponding weights, which we want to improve with respect to the inputs along with each input's desired output. To do this, we need a way to measure the performance of the weights, and a way to improve that performance.

A commonly used way of measuring performance is using the squared error of the outputs. You can do this for each sample input by sum the errors over each them, then add a minus before the formula, this will cause the formula to peak at 0 in the optimal case:

$$P = -\sum_s (\sum_z (d_{sz} - o_{sz})^2)$$

where:

P	is the measured performance
s	is an index over the sample inputs
z	is an index over the output nodes
d_{sz}	is the desired output for sample input s at output node z
o_{sz}	is the actual output for sample input s at output node z

The performance measure P is a function of the weights. So we can use the idea of gradient ascent if we can calculate the partial derivative of P with respect to each weight. With these partial derivatives, the overall performance P can be improved rapidly by altering the weights in proportion to the corresponding partial derivative. Note that above the performance is given as a sum over all sample inputs. So reduce notational complexity, we dropped the s subscript, thus focusing on one sample input at a time. Since this will have the same result if we adding up all these partial derivatives later. The partial derivative of the performance:

$$\frac{\partial P}{\partial w_{i \to j}}$$

where $w_{i \to j}$ is a weight connecting a node from the ith layer to a node in the jth layer.

The goal is to find an efficient way to compute the partial derivative of P with respect to $w_{i \to j}$. We can do that by expressing the partial derivative mostly in the terms of computations that to be done anyway, these are with the weights closer to the output layer of nodes. This is done by using the intermediate variable, o_j, the output of the jth node. The new derivative of P becomes (using the chain rule):

$$\frac{\partial P}{\partial w_{i \to j}} = \frac{\partial P}{\partial o_j} \frac{\partial o_j}{\partial w_{i \to j}} = \frac{\partial o_j}{\partial w_{i \to j}} \frac{\partial P}{\partial o_j} \qquad (1)$$

Now consider $\frac{\partial o_j}{\partial w_{i\to j}}$. o_j is computed by adding all the inputs to node j and passing the result through a threshold function, $o_j = f(\sum_i o_i w_{i\to j})$, where f is the threshold function. Take the sum as an intermediate variable, $\sigma_j = \sum_i o_i w_{i\to j}$, and applying the chain rule again:

$$\frac{\partial o_j}{\partial w_{i\to j}} = \frac{df(\sigma_j)}{d\sigma_j}\frac{\partial \sigma_j}{\partial w_{i\to j}} = \frac{df(\sigma_j)}{d\sigma_j} o_i = o_i \frac{df(\sigma_j)}{d\sigma_j} \qquad (1.1)$$

Now substituting this result back into the equation 1 yields the following equation:

$$\frac{\partial P}{\partial w_{i\to j}} = o_i \frac{df(\sigma_j)}{d\sigma_j}\frac{\partial P}{\partial o_j} \qquad (1)$$

Now try to express the partial derivative, $\frac{\partial P}{\partial o_j}$, in terms of the partial derivative, $\frac{\partial P}{\partial o_k}$, in the next layer to the right. The effect of o_j on P is through the outputs of the nodes in the layer, o_k, $\frac{\partial P}{\partial o_j}$, can be expressed as follows:

$$\frac{\partial P}{\partial o_j} = \sum_k \frac{\partial P}{\partial o_k}\frac{\partial o_k}{\partial o_j} = \sum_k \frac{\partial o_k}{\partial o_j}\frac{\partial P}{\partial o_k} \qquad (1.2)$$

Hence $o_k = f(\sum_j o_j w_{j\to k})$, where f is the threshold function.Treating the sum as a intermediate variable, σ_k, and applying the chain rule:

$$\frac{\partial o_k}{\partial o_j} = \frac{df(\sigma_k)}{d\sigma_k}\frac{\partial \sigma_k}{\partial o_j} = \frac{df(\sigma_k)}{d\sigma_k} w_{j\to k} = w_{j\to k}\frac{df(\sigma_k)}{d\sigma_k} \qquad (1.2.1)$$

Substituting this result back in equation 1.2, yields:

$$\frac{\partial P}{\partial o_j} = \sum_k w_{j\to k}\frac{df(\sigma_k)}{d\sigma_k}\frac{\partial P}{\partial o_k} \qquad (1.2)$$

If we take a closer look at equations 1 and 1.2, we can see two important properties of the partial derivatives. First $(equation 1)$, the partial derivative of performance with respect to a weight depends on the partial derivative of performance with respect to the following output; and second $(equation 1.2)$, the partial derivative of performance with respect to one output depends on the partial derivative of performance with respect to the outputs in the next layer. From these two remarks, we can conclude that the partial derivative of P with respect to any weight in the ith layer can be given in terms of computations already required one layer to the right in the jth layer.

To complete this forward computation, we have to determine the partial derivative of performance with respect to each output in the final layer:

$$\frac{\partial P}{\partial o_z} = \frac{\partial}{\partial o_z} - (d_z - o_z)^2 = 2(d_z - o_z)$$

To make things easy, we chose the threshold function as:

$$f(\sigma) = \frac{1}{1+e^{-\sigma}}$$

$$\begin{aligned} \frac{df(\sigma)}{d\sigma} &= \frac{d}{d\sigma}\left[\frac{1}{1+e^{-\sigma}}\right] \\ &= (1+e^{-\sigma})^{-2}e^{-\sigma} \\ &= f(\sigma)(1-f(\sigma)) \\ &= o(1-o) \end{aligned}$$

So by choosing the right threshold function, the derivative is expressed in terms of each node's output, $o = f(\sigma)$, rather than the sum of the inputs, σ. This is exactly the way we wanted, this equation is also expressed in values in terms of other values to its left.

At last, weight changes depends on a rate parameter r, the learning parameter. This determines the step size of each weight change.

Write $\frac{\partial P}{\partial o}$ as β, the factor 2 (from the derivative of the threshold function) absorbed in the learning rate r, and the derivative of the threshold function as o'. The backpropagation formulas are written:

$\Delta w_{i\to j} = r o_i o'_j \beta_j$,

$\beta_j = \sum_k w_{j\to k} o'_k \beta_k$ for nodes in the hidden layers,

$\beta_z = d_z - o_z$ for nodes in the output layer.

According to the chain rule, one first compute the changes for each sample input combination, then adds up the weight changes, and finally make the actual changes to the weights.

The backpropagation procedure:

- Pick a learning parameter r,
- until performance is satisfactory,
  - For each sample input:
    - Compute the resulting output,
    - Compute β for nodes in the output layer using:

$$\beta_z = d_z - o_z$$

- Compute β for all other nodes using:

$$\beta_j = \sum_k w_{j \to k} o'_k \beta_k$$

- Compute weight change for all weights using:

$$\Delta w_{i \to j} = r o_i o'_j \beta_j$$

– add up the weight changes for all sample inputs, and change the weights

3.3 Problems with learning

3.3.1 Converge problems

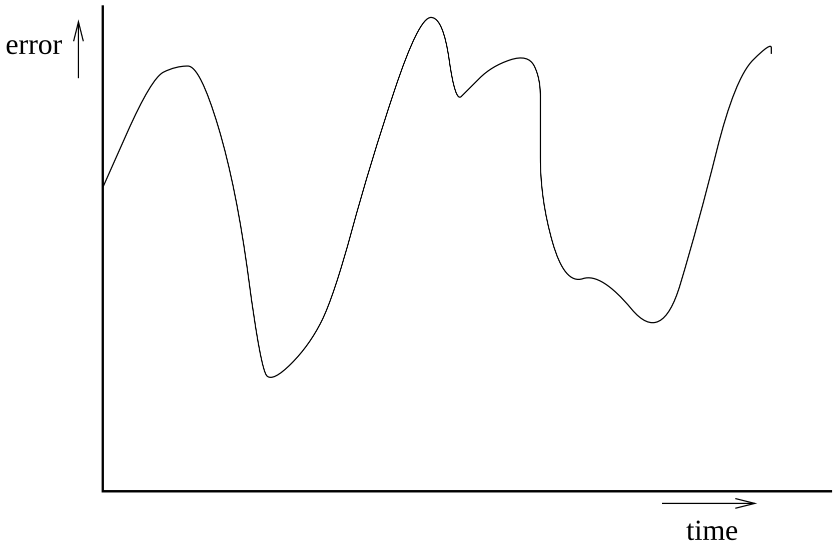


Figure 3.2: Oscillation during training

The neural nets has problem to converge to a minima, the error has a high oscillation during the training stage (see figure 3.2). This can occur when the learning rate is too large, so the weight changes will be large too, resulting in fluctuation in the weight-error space. But if one starts with one small learning rate, the learning process will take too long.

3.3.2 Local minima

Consider figure 3.3 which shows a simplified error-weight in 2D. The neural

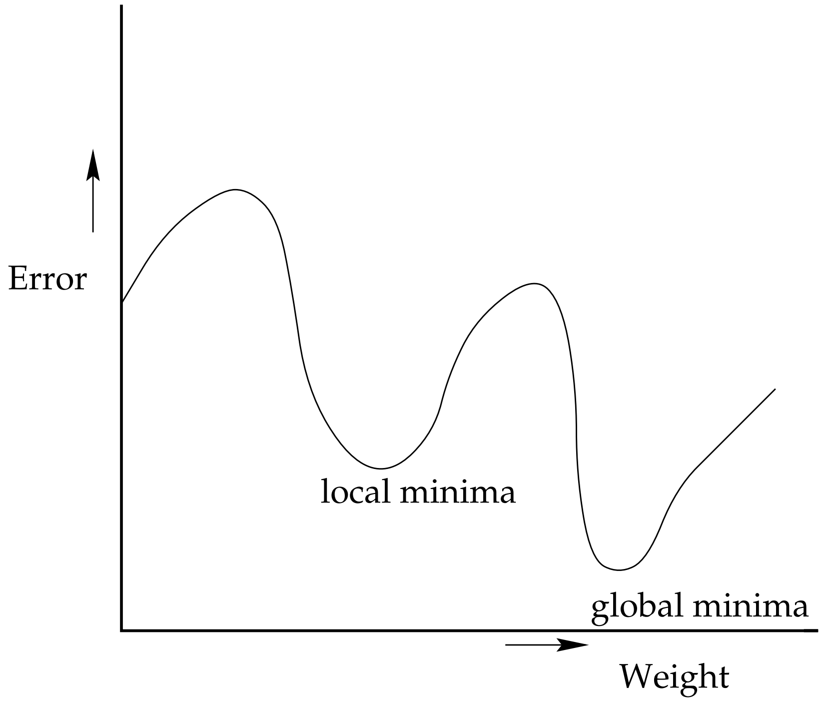


Figure 3.3: Local and global minima

net can get stuck in the local minima, without knowing that there is a better setting for the weights in the global minima. Measures should be taken to escape from the local minima, for example by introducing noise in the data.

3.3.3 Overfitting

One might think, if a network does well, then if we add more weights to it, it might become more accurate. Well, this is not the case when we have too many weights. See figure 3.4 , line I gives the decision line with less weights, while line II is done with more weights. The separation in classes by the net with more weights is too accurate, this network will fail when fed with test data. So the number weights is related to the dataset size and its complexity.

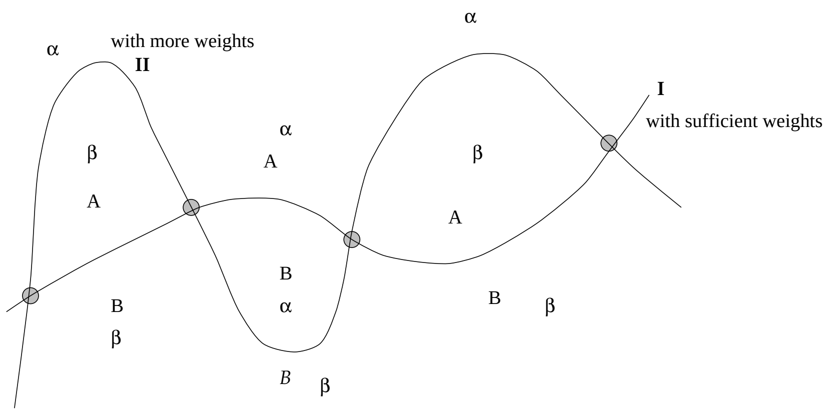


Figure 3.4: Overfitting the network

3.3.4 Speeding up learning: the momentum term

The learning rate is determined by the r, the rate parameter. If this is too large, the net will oscillate too much. On the other side, if r is too small, the learning will become too slow. One way too overcome this limitation is to include a term which includes a proportion of the last weight change.

$$\Delta w_{i\to j}(n) = r o_i o'_j \beta_j + m \Delta w_{i\to j}(n-1)$$

Where m is the momentum term, and $\Delta w_{i\to j}(n-1)$ is the previous weight change. So if the previous weight change was large, so too will the new weight change. This has an effect that the small fluctuations in the error-weight space will be smoothed out.

When the output is 0, the derivative $o' = o(1-o)$ is also equal to 0, which results in no weight change. In this case a speed-up is achieved by adding a constant to the derivative.

Chapter 4

Dataset

For our research a large and complex dataset is needed at low cost or free if possible. The dataset should also available to the general public. In this case the dataset from NIST is well suited. This copy(SROCR Rel 19-1.1, August 1994) contains 168.365 segmented character images of 128 by 128 binair pixels. In the following sections the NIST database is further described[Ni94].

4.1 NIST database

The National Institute of Standards and Technology(NIST) has developed a standard reference form-based handprint recognition system for evaluating optical character recognition (OCR). NIST is making this recognition system freely available to the general public on CD-ROM in the ISO-9660 data format.

The system is designed to run on a UNIX workstations and has been successfully compiled and tested on hosts of UNIX workstations of several manufacturers. The source code for the system is written primarily in C with two additional utilities having FORTRAN components. Source code is provided for form registration, form removal, field isolation, field segmentation, character normalization, feature extraction, character classification and dictionary based postprocessing.

The NIST *Special Database* 1(SD1) contains 2100 full page images of handwriting examples printed by 2100 different writers geographically distributed across the Unites States with a sampling roughly proportional to population density. The forms are scanned at 12 pixels per millimeter (300 dots per inch - bpi) binary and contain entry fields demarcated by boxes, one box for the entire field value.

Each of the 2100 forms in SD1 is an image of a structured form filled in by a unique writer. An image of a empty form is shown in appendix B. The form is comprised of :

- 3 identification boxes containing name, date and residence,
- 28 digit boxes of varying length,
- a randomly ordered lower case alphabet,
- a randomly ordered upper case alphabet,
- a handprinted text paragraph containing the Preamble to the U.S. Constitution

Note that the first field, the name field, will be covered with black pixels making the writer of each form anonymous.

After the installation of the CD-ROM (see [Ni94] for more detail), in the directory *train* there are 168.365 segmented and labeled handprint characters. In all there are 119.740 images of handprint digits, 24.205 lower case letters, and 24420 upper case letters. The images of segmented characters are stored in the Multiple Image Set (MIS) file format. Each MIS file ends with the extension *mis*. For each MIS file in the training set, there is an associated classification file containing the identity of each character contained in the MIS file. These classification file ends with the extension *cls*.

4.2 IHead file format

The images are scanned and saved in IHead file format. IHead is an attempt to design an open image format which can be used universally implemented across heterogeneous computer architectures and environments. It has been successfully ported and tested on several systems including: UNIX workstations and servers, DOS personal computers, and VMS mainframes. IHead has been designed with an extensive set of attributes in order to: adequately represent both binary and gray level images; represent images captured from different scanners and cameras; and satisfy the image requirements of diverse applications; including but not limited to image archival/retrieval, character recognition, and fingerprint classification.

The IHead structure definition written in C is listed below:

```
/************************************************************/
```

```
 /* File Name:  IHead.h */
 /* Package:    NIST Internal Image Header */
 /* Author:     Michael D. Garris */
 /* Date:       2/08/90 */
/***********************************************************/
/* Defines used by the ihead structure */
 #define IHDR_SIZE    288  /* len of hdr record (always even bytes) */
 #define SHORT_CHARS  8    /* # of ASCII chars to represent a short */
 #define BUFSIZE      80   /* default buffer size */
 #define DATELEN      26   /* character length of date string */
typedef struct ihead{
 char id[BUFSIZE];              /* identification/comment field */
 char created[DATELEN];         /* date created */
 char width[SHORT_CHARS];       /* pixel width of image */
 char height[SHORT_CHARS];      /* pixel height of image */
 char depth[SHORT_CHARS];       /* bits per pixel */
 char density[SHORT_CHARS];     /* pixels per inch */
 char compress[SHORT_CHARS];    /* compression code */
 char complen[SHORT_CHARS];     /* compressed data length */
 char align[SHORT_CHARS];       /* scanline multiple:  8|16|32 */
 char unitsize[SHORT_CHARS];    /* bit size of image memory units */
 char sigbit;                   /* 0->sigbit first | 1->sigbit last */
 char byte_order;               /* 0->highlow | 1->lowhigh*/
 char pix_offset[SHORT_CHARS];  /* pixel column offset */
 char whitepix[SHORT_CHARS];    /* intensity of white pixel */
 char issigned;                 /* 0->unsigned data | 1->signed data */
 char rm_cm;                    /* 0->row maj | 1->column maj */
 char tb_bt;                    /* 0->top2bottom | 1->bottom2top */
 char lr_rl;                    /* 0->left2right | 1->right2left */
 char parent[BUFSIZE];          /* parent image file */
 char par_x[SHORT_CHARS];       /* from x pixel in parent */
 char par_y[SHORT_CHARS];       /* from y pixel in parent */
}IHEAD;
```

The description of the fields:

id: This field uniquely identifies the image file, typically by a file name.

created: This is the date on which the image was captured or digitized.

width, height, depth: A binary image has a pixel depth of 1 whereas a gray scale image containing 256 possibles shades of gray has a pixel depth of 8.

density: This field contains the scan resolution of the image, in this case *300 dpi.*

compress, complen: An Ihead image can be uncompressed or compressed. The header itself is always uncompressed. The compress field is an integer flag which signifies which compression technique, if any, has been applied to the raster image data which follows the header. If compress is zero, then the image is not compressed. If compress is nonzero, then the complen is used to load the compressed block data into main memory.

align: In general, images are not an even multiple of 8 pixels in width, in order to minimize the overhead of ending a previous scanline and beginning of the next scan line within the same byte, a number of padded pixels are provided in order to extend the previous scan line to an even byte boundary.

unitsize, sigbit, byte_order: These three fields are used due interchanging issues among heterogeneous computer architectures and displays. The unitsize specifies how many contiguous pixel values are bundled into a single unit by the digitizer. The sigbit field specifies the bit-order of each unit; most significant bit first or least significant bit first. The byte_order specifies the order in which bytes occur within the unit if unitsize is a multiple of bytes.

pix_offset: The pix_offset attribute defines a pixel displacement from the left edge of the raster image data to where a particular image's significant image information begins.

whitepix: The whitepix attribute defines the value assigned to the color white.

issigned: This attribute specifies the range of a pixel. If an image has depth of 8, then the range will be -128 to 127 or 0 to 255, depending on this attribute.

rm_cm, tb_bt, lr_rl: These fields specifies the orientation of the raster scan of a digitizer. So rm_cm specifies whether the digitizer captured the image in row-major order or in column-major order. The tb_bt specifies whether the scan lines of an image were accumulated from top to bottom, or from bottom to top, and lr_rl determines left to right or right to left.

parent, par_x, par_y: These final attributes in Ihead provide a historical link from the current image to its parent image. The parent field contains the full path name to the image from which this image was extracted. The par_x and par_y fields contain the origin from where the extraction took place from the parent image.

4.3 MIS file format

Storing character images in individual files has proven to be very inefficient, especially when manipulating databases containing hundreds of thousands of characters. Rarely do recognition system components process only a single character image in isolation, rather to process a large sample of characters.

A new memory structure called Multiple Image Set (MIS) is created. The MIS structure and file format have been developed to manage large volumes of segmented character images. The MIS format allows multiple images of homogeneous dimensions and depth to be stored in one file. MIS is actually a simple extension or encapsulation of the IHead format described in the previous section.

An MIS file contains one or more individual images stacked vertically within the same contiguous raster memory, the last pixel row or scanline of the previous image is concatenated to first scanline of the next image. The individual images that are concatenated together are referred to as MIS *entries*. The resulting raster memory is referred to as the MIS *memory*. The MIS memory contains one or more entries of uniform width, height, and depth is stored using the IHead file format.
The MIS structure definition written in C is listed below:

```
/*********************************************************/
 /* Filename:      Mis.h */
 /* Adapted from:  Calerahdr() */
 /* Author:        Michael D. Garris */
 /* Date:          7/18/90 */
/*********************************************************/
typedef struct misstruct{
IHEAD *head;
unsigned char *data;
int misw;
int mish;
```

```
int misd;
int entw;
int enth;
int ent_alloc;
int ent_num;
} MIS;
```

The description of the fields:

head: The IHead structure contained by this MIS structure.

data: The MIS memory.

misw, mish: These attributes specify the width and the height of the MIS memory.

entw, enth: These attributes specify the uniform width and the height of the MIS entries.

ent_alloc: The attribute ent_alloc specifies how many MIS entries have been allocated to the MIS memory data.

ent_num: This attribute specifies how many entries out of the possible number allocated are currently and contiguously contained in the MIS memory data.

4.4 File format for Neural Networks

The segmented character images from NIST have 128 as width and height. The outer part of the images are blank, they do not contain any important information for the neural network. Instead of the original images, which contains 128*128 = 16384 pixels, new images are formed by cutting off the borders. The remaining image has 64 as height and width. The new image contains 4096 bits of data. This has a reducing factor of 4 compared to the original images.

When an image is fed to the neural network, an input layer of 4096 neurons will be needed, one neuron for each pixel of the image. A thought was given to pack four neighboring bits into one neuron, in that case only 1024 input neurons will be needed. This is not done, since the goal of this project is to experiment with complex datasets, and packing the pixels will probably accelerate the training speed, but will also reduce the complexity. Finally these smaller images are converted to ASCII format.

For our neural network, the following file format in ASCII is used:

header is the first line of the file, this line consists of four integers:
format number-of-patterns number-of-input-neurons number-of-output-neurons
where:

- format == 0 : input and target values as floats
- format == 1 : input in floats, output as integer class value (0...#outputs-1)
- format == 2 : input is binary image, output as integer class binary (0 or 1) values are packed in hexadecimal byte values

body ; we use format 2 so the rest of the file contains one pattern per line. A line has the following the format:

- pattern data in hexadecimal format, without the 0x prefix.
- pattern class in hexadecimal format, without the 0x prefix.

The raster image data of the characters are stored in MIS format. A conversion has to be made so the data are available for the neural network. Several conversion utilities are written for this purpose:

ConvMis: This program converts a MIS file to the format which is needed for the neural network.

SumCls: This program reads the total number of patterns from several class-files, then it writes these individual totals in a output file, including the total of all the patterns.

SplitClassData: This program reads a datafile and writes each class to a separate file. After this process one can *cat* the individual data files together. In the resulting file the patterns wont appear in random order.

Concate: When the training is not done in one session, the training proceduces several output files, all output file starts with iteration number 1. Actually the second output file should start with a number following the last iteration number of the previous output file. This program concatenates the files and fixes the iteration numbering problem.

For more detail about these utilities see appendix G.

Chapter 5

The comparison

5.1 Neural network implementation

The existing implementation of the backpropagation neural network runs on a set of PowerPC chips. It consists of four nodes, each has four PowerPC chips. The following resource allocations are possible: 2, 4, 6, 8 , 10, 12, 14 and 16 PowerPC microprocessors (see appendix A). The implementation of online and offline versions of the backpropagation algorithm are based on network and dataset decomposition[Vu95].

- dataset decomposition
  With dataset decomposition every processor has exactly the same copy of the neural network. The dataset is equally divided between the processors and each processor processes its own dataset. After the computation, the computation results are collected from all the processors and each network on each processor is updated according to the global error accumulated. This technique is especially efficient for training with large dataset, since during the computation no communication is necessary.

- Network decomposition
  Here the network is divided over several processors. For the backpropagation algorithm, every layer itself is divided over the processors. So during every propagation, communication is needed between these processors. This is only useful when the time gained by using multiple processors is larger than penalty one gets from the communication needed between the processors.

5.2 Topology

A network of three layers is chosen for this purpose, since a **three-layer** network implements decision surfaces of arbitrary complexity. This is also a commonly used network for pattern recognition. The neural network used is a fully interconnected feed-forward network, which means every neuron from a certain layer receives input from all the neurons from the preceding layer.

As described in section 4.4, an image contains 4096 bits. The **input layer** contains 4096 input neurons, each neuron for one pixel of the image, every pixel is represented by a floating number, where 0.8 represents a black pixel and -0.8 represents a white pixel.

The images are 64 by 64, a convenient choice could be a hidden layer of 64 units. If the number of units in the **hidden layer** becomes too large, the training time will increase accordingly, this is not realistic in the scope of this project. In a later section hidden layers of different sizes are tried to find a suitable hidden layer.

The **output layer** contains 10 output units, since only the images of the digits are used. A digit is assumed as identified if the node with the largest output value satiefies the following condition:

$abs(output - class\_value) < confusion\_delta$

$where:$

$$class\_value = 0.8$$
$$confusion\_delta = 0.5$$

5.3 Parameters

The **initial weights and biases** are randomly distributed over the interval of $[-0.005, +0.005]$. If the initial weights are too large, the chance for a network to learn is smaller. With small weights however, the learning process will take too long. But then again, if the network is able to learn, with small weight it will eventually lead to success.

The current implementation offers the choice between *tanh* and *elliot* **activation rules**. A few tests are done to determine the best activation function for this project.

As determined by experiments, a **momentum** of 0.8 is used. A momentum will speed up training in very flat regions of the error space and suppresses weight oscillation in step valleys or ravines. In this experiment, the training is much slower without a momentum.

Several **learning rates** will be tested to see which one will result in best learning speed. The learning rate depends on the dataset size. If the learning rate is set too large, it will lead to heavy oscillating weights, then the network can not be trained. During a training session the learning rate is constant.

All other parameters not mentioned above are in file in appendix C.1

5.4 Description of the experiments

All networks have the same topology, concerning to layers, neurons and the connectivity. The epoch size is 50%, the weights are updated twice per epoch for offline learning. The network is trained with several datasets of different sizes. All datasets have equal portion of patterns of all classes. The learning rate is changed according to the size of the dataset. A training session is terminated when a recognition rate of 95% is achieved, which means that the 95 percent of the patterns are classified correctly. The training is also terminated when the network is oscillating too much.
Then a comparison is made between the online and the offline version of the neural nets when trained with dataset of the same size. The comparison depends on the following components:

- number of iteration needed
- training time: computation time + communication time
- result with the test dataset, only done with the last large dataset

5.5 Session 0, trying out

To justify some choices made in the preceding sections, several tests are done to determine the best activation function and the size of the hidden layer to be used. The training dataset used for the following networks contains 1139 patterns, which were equally distributed over the ten classes. After the network reached a recognition rate of 95 percent, the network is then tested with a test dataset of 10613 patterns. This is the same test dataset that will used with the final neural networks.

5.5.1 Online backpropagation

Two three layered network were trained with the patterns. The hidden layer

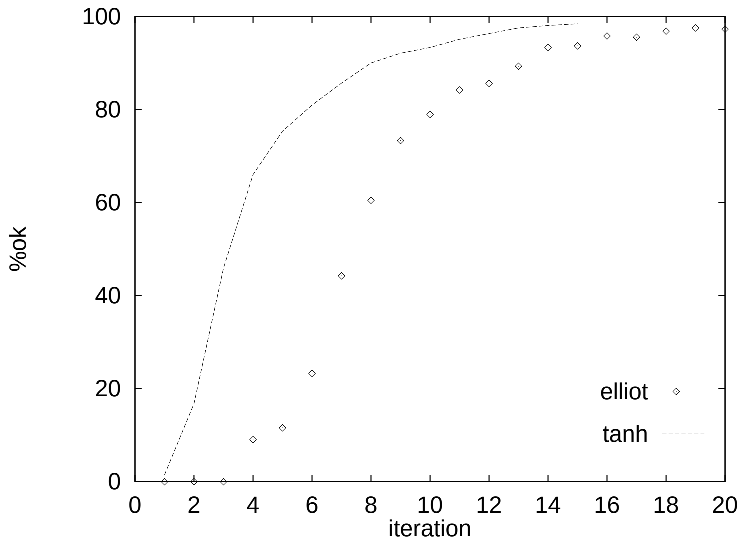


Figure 5.1: Online, Tanh vs Elliot

of both neural networks contains 64 neurons. The difference is the *activation rule* used in both network, *tanh* and *elliot*. The result is shown in figure 5.1.
As figure 5.1 shows, the *tanh* neural network converges faster. In the following table the test result is shown.

Activation	iteration	Network	Test result		
Function	(95%)	trained	%Ok	%Bad	%Und
Tanh	11	15, 98.4	59.0	6.7	34.3
Elliot	16	20, 97.2	64.8	10.6	24.6

The *Network trained* column shows the network that is saved when the training session is terminated. In this case the saved networks have almost the same recognition rate. The *Test result* column shows the percent *Ok, bad* and *undetermined* of the test dataset. The conclusion is that *tanh* converges faster. But *tanh* has a lower percent 'ok' compared to the *elliot* activation rule. Then again *elliot* has a higher 'bad' rate.

Tanh needs less iteration to train, but this is not the same as the computation time. To see which activation rule needs less time, the iteration-times are shown in the next table.

Activation rule	Iteration time(sec)
Elliot	66.6
Tanh(approximation table)	67.8
Tanh(built-in)	69.0

The iteration-time are almost the same for all the three functions. The implementation of backpropagation uses two versions of *tanh*, one using a look-up table and the the other one using the run-time library provided by the programming language.

Since *tanh* has a higher training speed and has almost the same performance as the *elliot* network. The *tanh* is studied further. Several new networks are trained, they have different number of neurons in their hidden layer.

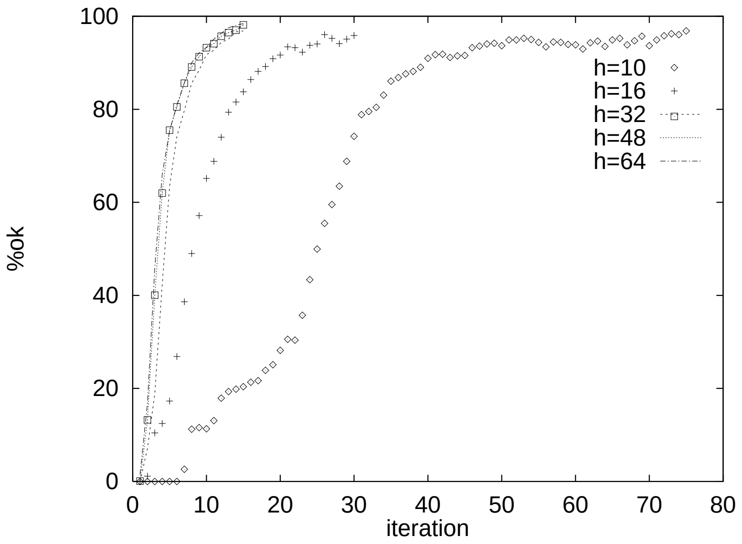


Figure 5.2: Online, Network with different hidden layer(1)

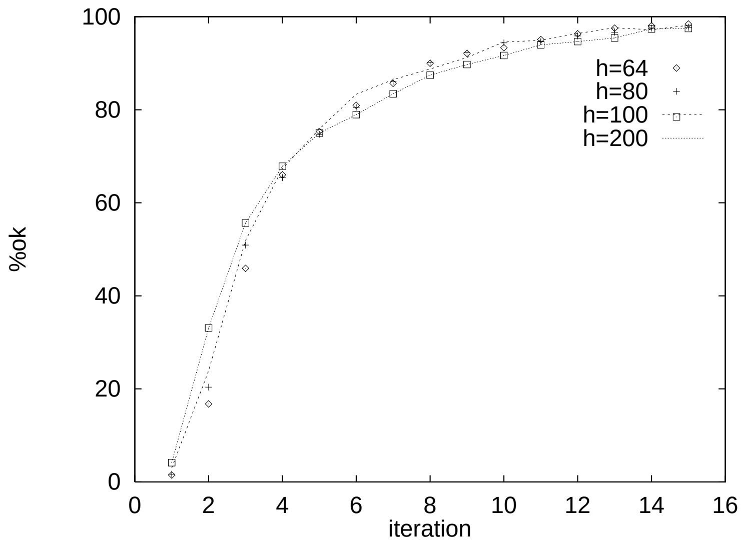


Figure 5.3: Online, Network with different hidden layer(2)

Size	iteration	Network	Test result		
hidden layer	(95%)	trained	%Ok	%Bad	%Und
10	53	75, 96.8	49.0	33.9	17.1
16	26	30, 95.9	55.5	15.4	29.1
32	13	15, 96.8	56.3	7.3	36.4
48	12	15, 98.1	59.0	6.2	34.8
64	11	15, 98.4	59.0	6.7	34.3
80	12	15, 97.7	59.8	6.4	33.8
100	13	15, 97.5	59.7	5.9	34.4
200	12	15, 97.7	56.3	7.4	36.3

As shown in figure 5.2 and 5.3, The neural networks with hidden layer size larger than 48 neurons do not differ much. All these nets have 59%ok when tested with the test dataset. The choice is between 48 and 64, since other nets with larger number of neurons require more computation resource. The net with 64 neurons needs less iteration, just one actually, compared with the net with 48 neurons as hidden layer. This does not justify the choice for 64, the choice is more relied on the fact that the image is 64 by 64.

5.5.2 Offline backpropagation

For offline backpropagation different activation rules are studied. The hidden layer of both neural networks contains 64 neurons.

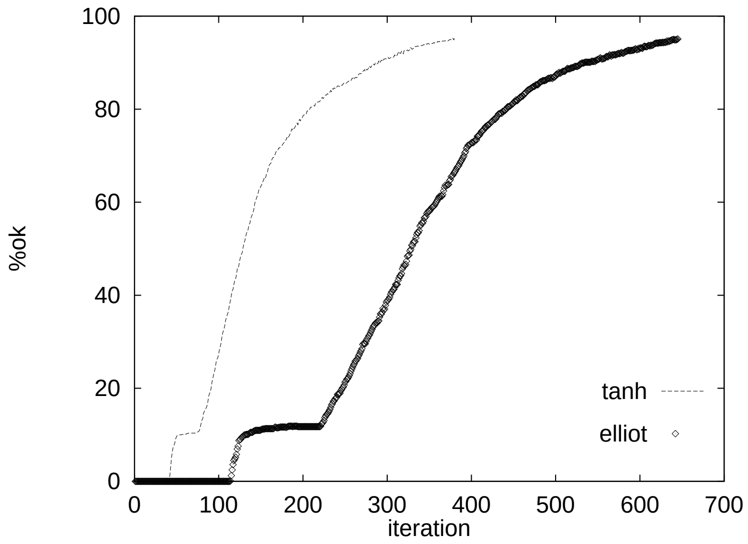


Figure 5.4: Offline, Tanh vs Elliot

Activation	iteration	Test result		
Function	(95%)	%Ok	%Bad	%Und
Tanh	377	53.7	4.6	41.7
Elliot	645	58.6	6.0	36.4

In figure 5.4, the result with the offline version is very similar with the online version. $Tanh$ needs less iterations, the iteration time is almost the same, which results $tanh$ being the faster activation rule. However, *elliot* has a better test result.
The final choice is made for $tanh$ as activation rule and 64 as the size of the hidden layer. $Tanh$ is chosen above *elliot* for the faster training speed. *Elliot* performs better with the test dataset, this is for both offline and online neural network. So it does not matter that the $tanh$ is chosen.

5.6 Session I, a small dataset

To get an idea how the network will perform with a small dataset, a dataset with 1023 patterns is used to try it out. The online backpropagation uses 4 processors for its training. The offline type uses 12 processors for its computation. The epoch size is 50%.

5.6.1 Result

The graphics in appendix D show the recognition rate during the training. The next table summarizes the result. An '-' behind the iteration number means that the network didn't converge. An 'na' means that the result is not available because it is not important any more. The best training speeds are accentuated with bold font type.

Network		Learning Rate						
type		0.1	0.05	0.01	0.005	0.001	0.0005	0.0001
Online	iteration	615-	na	585-	8	13	18	60
	time(min)				**8**	13	18	60
Off-line	iteration	348	83	292	700	2695	na	na
	time(min)	150	**36**	126	303	1167	na	na

5.6.2 Conclusion

Several learning rates are tried. With $r = 0.1$ and $r = 0.01$ the online version of backpropagation does not succeed in achieving 95 percent recognition rate, the training is stopped because the network oscillates too much. When the learning rate is smaller, the network is trained successfully, but it requires more iterations, since the weight changes are smaller every time.
With offline network, training is successful with every r tried. With $r = 0.1$, the network is oscillating, but it finally reached the 95 percent goal. Starting from $r = 0.01$, the error-space is a smooth line. Using even smaller learning rate has no use with offline network, because the result is predictable: the network will learn and the learning time is increased.
As they are compared, the online version is four times as fast as the offline opponent. Even when the offline version used more processors. Even with this small dataset, it seems that the online network is superior to the offline network. The online has the disadvantage that not all the learning rates tried here will converge, in contrary to the offline version.

5.7 Session II, a mid-sized dataset

In this second session the training is done with a mid-sized dataset. It contains 5823 patterns. The online backpropagation uses 4 processors for its training. The offline type uses 12 processors for its computation.

5.7.1 Result

The graphics in appendix E show the recognition rate during the training. The next table summarizes the result. An '-' behind the iteration number means that the network didn't converge. An 'na' means that the result is not not available because it is not important any more.

Network		Learning Rate				
type		0.1	0.01	0.001	0.0001	0.00001
Online	iteration	na	230-	24	48	375
	time(hour)			**2.3**	4.5	35.5
Off-line	iteration	1058	1137	11492	na	na
	time(hour)	**15.4**	16.5	167.3	na	na

5.7.2 Conclusion

Again $r = 0.1$ and $r = 0.01$ failed for the online version of backpropagation to learn. When the learning rate is smaller, starting from $r = 0.001$, the network is trained successfully.
With offline network, training is successful with all r. With $r = 0.1$, there are sharp declines in the graphics, after the declines the network recovered very fast to it is original level. Which finally leads to successful learning. With smaller learning rates, the recognition rate is smooth.
From the comparison it can be concluded again that the online version performs better, even with a factor of seven.

5.8 Session III, a large dataset

In this last session the training is done with a large dataset. It contains 32019 patterns. The online backpropagation uses 4 processors for its training. The offline type uses 12 processors for its computation.

5.8.1 Result

The graphics in appendix F show the recognition rate during the training.

The next table summarizes the result. An '-' behind the iteration number means that the network didn't converge. An 'na' means that the result is not not available because it is not important any more.

Network		Learning Rate			
type		0.01	0.001	0.0001	0.00001
Online	iteration	220-	530-	80	495
	time(day)			**1.7**	10.7
Off-line	iteration	8000	na	na	na
	time(day)	**18.1**	na	na	na

5.8.2 Conclusion

With $r = 0.1$, both online and offline failed to learn.
For online backpropagation, the network starts to learn from $r = 0.0001$. When the learning rate is smaller, starting from $r = 0.001$, the network is trained successfully.
With offline network, training is successful with with $r = 0.01$, This means with smaller learning rates, the network will also learn, but it needs more iterations to learn.
For this dataset, the online version is ten times as fast as the offline version. The recognition rate-iteration graph of both network is almost the same, only the offline needs more iterations to accomplish the same result.

5.8.3 Test result

Two test datasets are used to determine the efficiency of the network trained with backpropagation. These test dataset are disjunct from training dataset. The first test dataset contains 3217 patterns, which is about 1/10 of the training dataset. The second dataset contains 10613 patterns, which is about 1/3 of the training set.

	Test result with 3217 patterns				
Network	Learning	iteration	Test result		
type	rate	(95%)	%Ok	%Bad	%Und
On-line	0.0001	80	82.4	3.0	14.6
Offline	0.01	8000	81.1	3.3	15.6

	Test result with 10613 patterns				
Network	Learning	iteration	Test result		
type	rate	(95%)	%Ok	%Bad	%Und
On-line	0.0001	80	83.2	2.8	14.0
Offline	0.01	8000	82.0	2.7	15.3

In the above tables, the test result of the online network with $r = 0.00001$ is not included, it has the same result as the the network with $r = 0.0001$. The test results of both network with the two different test dataset are almost similar. Although the online version performs slightly better.

5.9 Online and pattern sequence

During online learning, the patterns are in random order at each iteration. To determine the influence of the order of the patterns two tests are done.

1. Only one permutation at the beginning of the training session, every iteration after this will use the same order of the patterns.

2. No permutation at all, which means the patterns are in fixed order. First all the patterns of digit 0, then digit 1, and so on.

The training and test result of the first experiment is similar as its online variant with permutation at every iteration.
The second experiment needs only one 1 iteration to reach the 95% limit. This is very surprising. When the network is examined further with test data, the mystery is solved. All patterns are classified as digit 9 by the network. Every class is fed to the network continuously, this causes the network to learn classifying one class at a time. The final classification the network learned is class 9.

Chapter 6

Conclusion

6.1 Comparison result

As shown in the previous chapter, online learning clearly outperforms offline learning with respect to the training speed. As the dataset increases in size, the comparison is more in favor of the online version. When a test dataset is used to validate the networks, the test results are almost similar. This is not a surprise, since both network have the same error function.
The comparison result is only valid with the current implementation and the available resources. In this experiment 12 processors are used for training the offline network. If more processors had been available, the number of iteration needed remains the same, but the computation time will be reduced according to the number of processors. This is because only one communication per iteration between the processors is needed during the training. The advantage of the offline version is the possibility to execute the training procedure parallelly. However, in the scope of the current technology and the price for the processors, online backpropagation remains the favorite algorithm for learning with large complex dataset.

6.2 Further research and suggestions

6.2.1 Epoch size

Online and offline learning are two extremes. The epoch size determines the network type. For example, if the epoch size is *one* pattern, then the network is obviously a online network. Research can be done to find the optimal epoch size for the backpropagation network with respect to the complexity and the size of the dataset.

6.2.2 Pattern order

In section 5.9 it is shown that an online network can be trained within one iteration. But the trained network only *remembers* the last class being learned. This is caused by offering too many patterns of the same class after each other at the same time. One can study the correct number of patterns of the same class to be fed to the network for reaching fast training time.

Appendix A

PowerXplorer Hardware Configuration at CSI

This appendix is derived from `http://www.cs.kun.nl/~ths/PowerXplorer/hardware.html`

Content:

- Communication topology
- Block diagram of 1 node
- Block diagram BBK-SCSI
- PARIX partitions

Communication topology

The internal switches of the cabinets are set in the default mode and the 1/2 and int/ext switches on the cabinets are all set in the down position. The reset signal for the nodes is thus passed only in the positive X direction starting from the links.

The links are via a BBK-SCSI box connected to POSEIDON, a SUN SPARCstation 20 workstation with two 75MHz SuperSPARC-II CPU's and 64 MByte memory.

PARIX partitions

```
#file nrm.conf.16linear
# nrm.conf for 16 nodes PowerXplorer
```

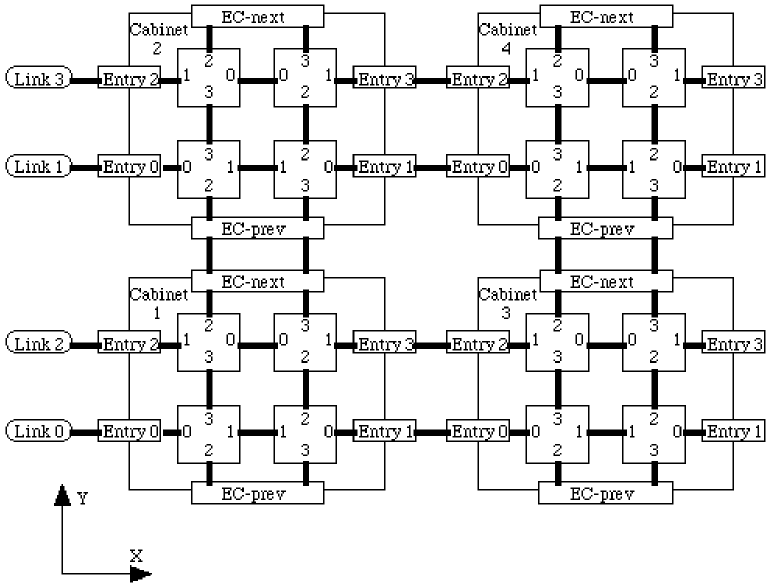


Figure A.1: Communication topology

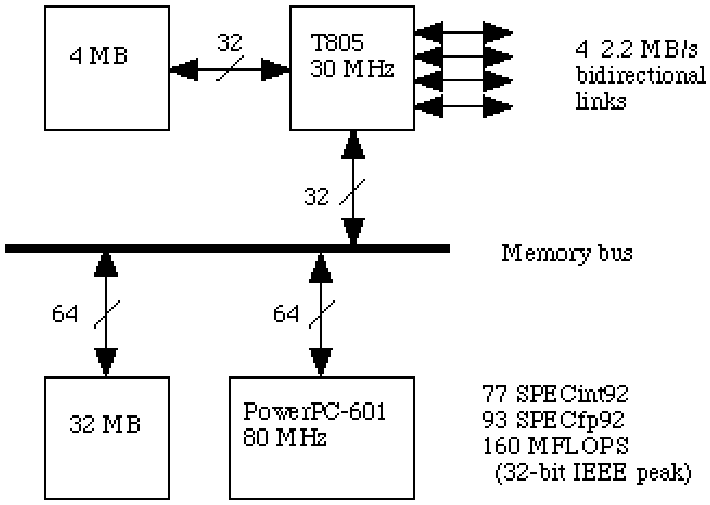


Figure A.2: Block diagram of 1 node

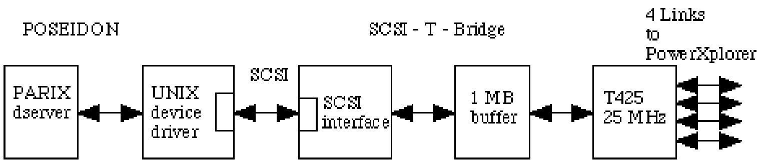


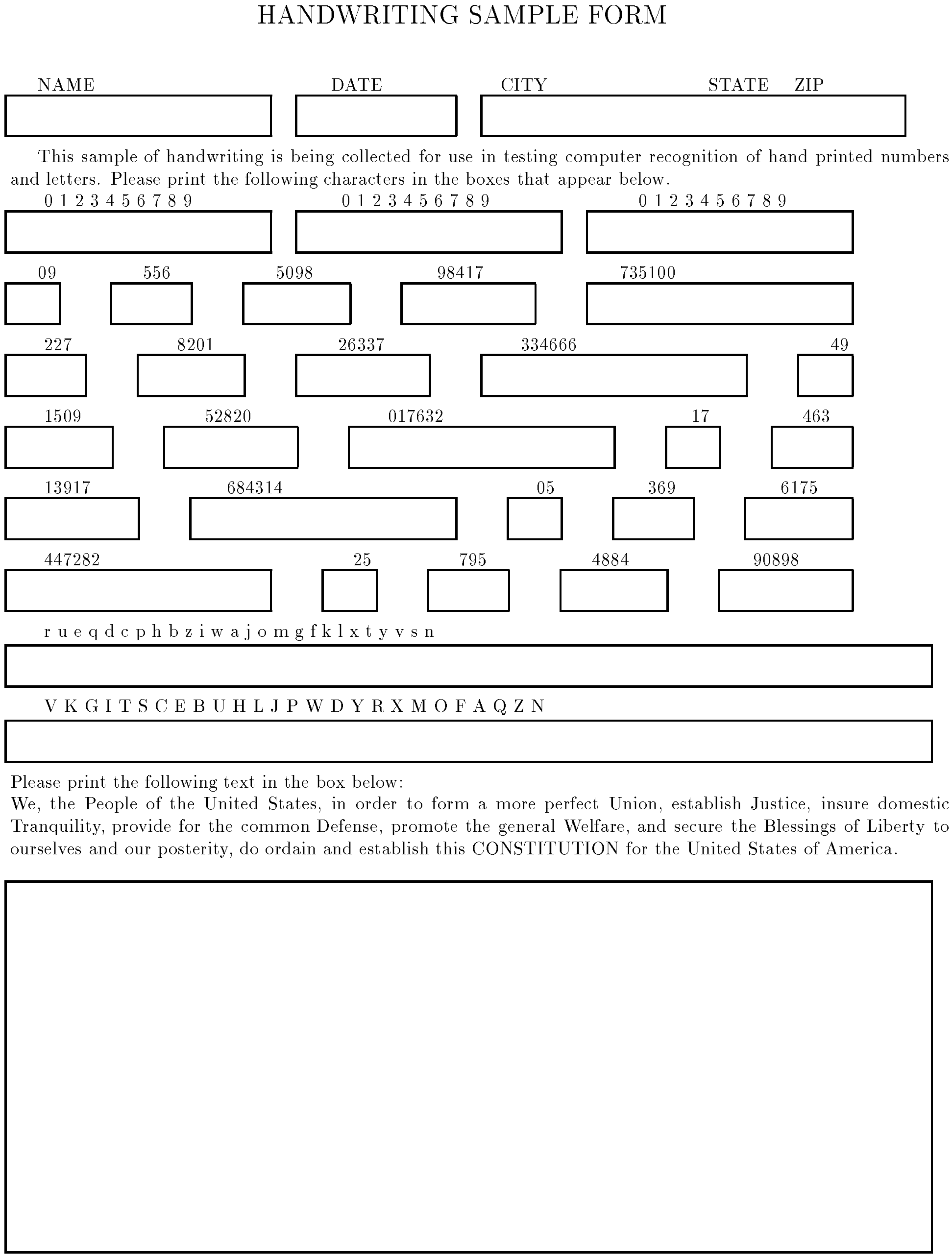
Figure A.3: Block diagram BBK-SCSI

```
# default settings of internal switches:
# reset in +x direction
# wired with all PSS links to cabinets 1 and 2
# thus 4 users
# 1/2 and int/ext switches down (resets not connected)
#
# Theo Schouten nov 1996
#
FT PXpl 4,4,1 OLDGC
#
ENTRY entry0:  0,0,0 0 poseidon,0,-1
ENTRY entry1:  0,2,0 0 poseidon,1,-1
ENTRY entry2:  0,1,0 1 poseidon,2,-1
ENTRY entry3:  0,3,0 1 poseidon,3,-1
#
PARTITION p2a:  pp2 0,0,0 1,0,0 entry0
PARTITION p2b:  pp2 0,1,0 1,1,0 entry2
PARTITION p2c:  pp2 0,2,0 1,2,0 entry1
PARTITION p2d:  pp2 0,3,0 1,3,0 entry3
#
PARTITION p3a:  pp3 0,0,0 2,0,0 entry0
PARTITION p3b:  pp3 0,1,0 2,1,0 entry2
PARTITION p3c:  pp3 0,2,0 2,2,0 entry1
PARTITION p3d:  pp3 0,3,0 2,3,0 entry3
#
PARTITION p4a:  pp4 0,0,0 3,0,0 entry0
PARTITION p4b:  pp4 0,1,0 3,1,0 entry2
PARTITION p4c:  pp4 0,2,0 3,2,0 entry1
PARTITION p4d:  pp4 0,3,0 3,3,0 entry3
#
PARTITION pb4a:  ppb4 0,0,0 1,1,0 entry0 entry2
PARTITION pb4b:  ppb4 0,2,0 1,3,0 entry1 entry3
```

```
PARTITION pb4c:  ppb4 0,1,0 1,2,0 entry1 entry2
#
PARTITION p6a:  pp6 0,0,0 2,1,0 entry0 entry2
PARTITION p6b:  pp6 0,2,0 2,3,0 entry1 entry3
PARTITION p6c:  pp6 0,1,0 2,2,0 entry1 entry2
#
PARTITION p8a:  pp8 0,0,0 3,1,0 entry0 entry2
PARTITION p8b:  pp8 0,2,0 3,3,0 entry1 entry3
PARTITION p8c:  pp8 0,1,0 3,2,0 entry1 entry2
#
PARTITION p9a:  pp9 0,0,0 2,2,0 entry0 entry1 entry2
PARTITION p9b:  pp9 0,1,0 2,3,0 entry2 entry1 entry3
#
PARTITION p12a:  pp12 0,0,0 3,2,0 entry0 entry1 entry2
PARTITION p12b:  pp12 0,1,0 3,3,0 entry2 entry1 entry3
#
PARTITION p16:  pp16 0,0,0 3,3,0 entry0 entry1 entry2 entry3
#
```

Appendix B

Empty HSF form



HANDWRITING SAMPLE FORM

NAME DATE CITY STATE ZIP

This sample of handwriting is being collected for use in testing computer recognition of hand printed numbers and letters. Please print the following characters in the boxes that appear below.

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9

09 556 5098 98417 735100

227 8201 26337 334666 49

1509 52820 017632 17 463

13917 684314 05 369 6175

447282 25 795 4884 90898

r u e q d c p h b z i w a j o m g f k l x t y v s n

V K G I T S C E B U H L J P W D Y R X M O F A Q Z N

Please print the following text in the box below:
We, the People of the United States, in order to form a more perfect Union, establish Justice, insure domestic Tranquility, provide for the common Defense, promote the general Welfare, and secure the Blessings of Liberty to ourselves and our posterity, do ordain and establish this CONSTITUTION for the United States of America.

Figure B.1: An empty HSF form

Appendix C

Batch files

C.1 Default configuration file for backpropagation

```
# parameter file for backprop
# implementation on PowerXplorer using PVM
#
#
# a # or / on a line starts a comment field, which is ignored
# empty lines are allowed

netmode DATA_NET / ONLINE_NET: distributed net, on-line learning
                 / DIST_NET: distributed net, epoch learning
                 / DATA_NET: data set parallellism
nlayers 3 / number of layers
neurons 4096 64 10 / neurons per layer

epoch 50 / % of patterns to take per epoch

randomstart 500 / start value of rand()
randomfactor 0.00500 / scale weights to +- this value

act_function 1 / 0:  sigmoid, 1:  tanh, 2:  Elliot's
last_linear 0 / 0:  no, 1:  last layer is linear
act_steepness 1.000000 / steepness of activation function
act_derivativeoffset 0.100000 / added to derivative of act function
learn_errorfunction 0 / 0:  quadratic, 1:  log (sigmoid needed)
```

```
learn_method 1 / 0:  no momentum
               / 1:  with momentum
               / 2:  quickprop
learn_steepness 1 / 1:  learn steepness per neuron, 0:  not
learn_etachange 0 / 0:  not
                  / 1:  divide by #neurons previous layer
                  / 2:  multiply by 2 starting from output layer
learn_minerror 0.000000 / required error on at least 1 output neuron

learn_eta 0.010000 / eta
learn_momentum 0.800000 / momentum or mu factor for quickprop

learn_classfactor 0.0 / to multiply error of same class output neuron
                      / 0.0:  do not multiply
                      / < 0.0:  use #outputneurons/ (-factor)
learn_maxerrfactor 0.0 / to multiply error of worst output neuron
                       / 0.0:  do not multiply
                       / < 0.0:  use #outputneurons/ (-factor)
learn_noise 0.000000 / add random noise between +-learn_noise
                     / for binary inputs (format = 2) flip
                     / learn_noise fraction of input bits

class_value 0.800000 / wanted value of output neuron of input class
class_notvalue -0.800000 / wanted value of other neurons
bin_value 0.800000 / value of 1 bit of binary input
bin_notvalue -0.800000 / value of 0 bit

confusion_method 0 / 0:  1 class only, 1:  more possible
confusion_delta 0.500000 / max difference with class_value
```

C.2 Batch file for training/testing

```
# bp.com   command file used for training/testing the neural net
#
#
set maxiter 5
set fulleval 1
set eta 0.01
```

```
#goto DoTest

#
makenet
loadpat DataTrain.nn
exiterr
open TrainLogFileName
label ContLearn
learn
if percok GE 95
goto EndLearn
if totaliter GE 1000
goto EndLearn
goto ContLearn
label EndLearn
savenet NNFileName&
# where & is the iteration number
close
exit

label DoTest
set maxiter 1
makenet
loadnet NNFileNameIterNr
exiterr
test DataTest.nn TestLogFileName& 0
```

Appendix D

Result session I

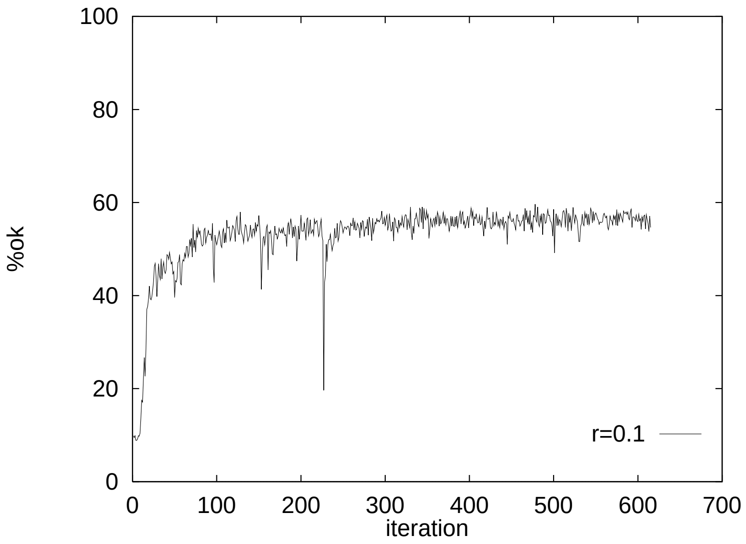


Figure D.1: Online, r=0.1

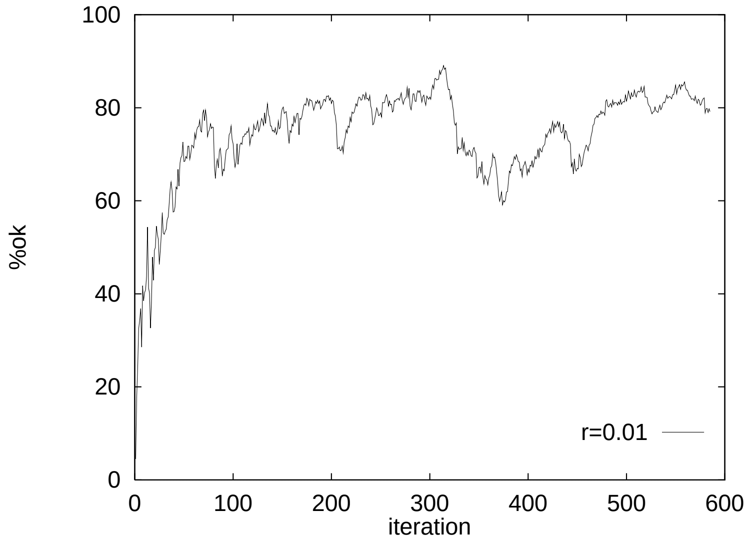


Figure D.2: Online, r=0.01

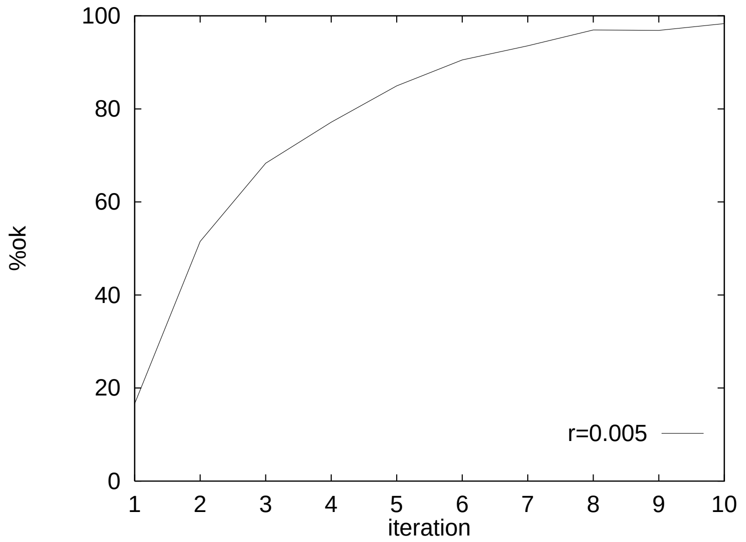


Figure D.3: Online, r=0.005

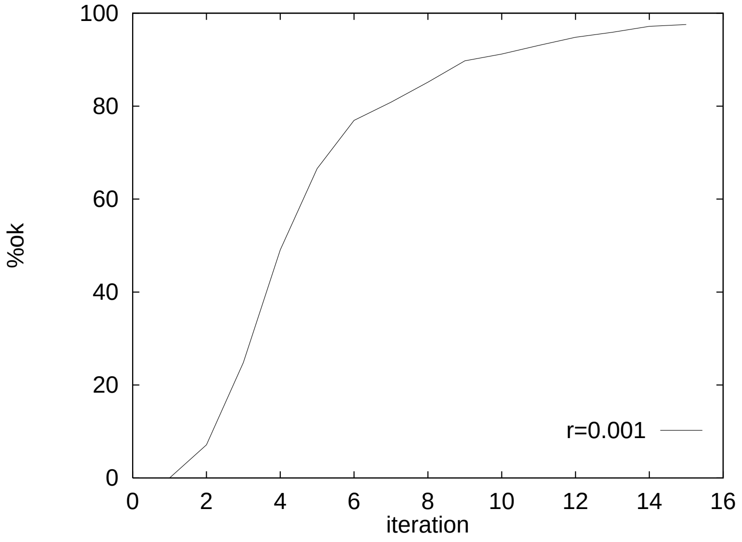


Figure D.4: Online, r=0.001

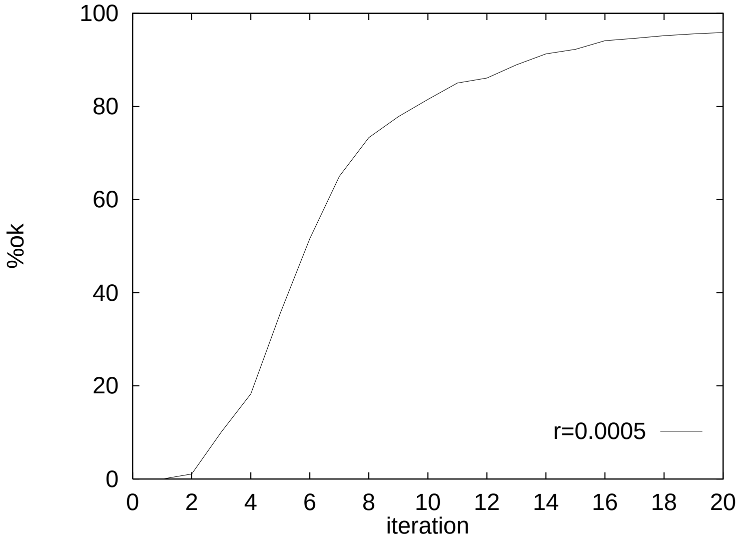


Figure D.5: Online, r=0.0005

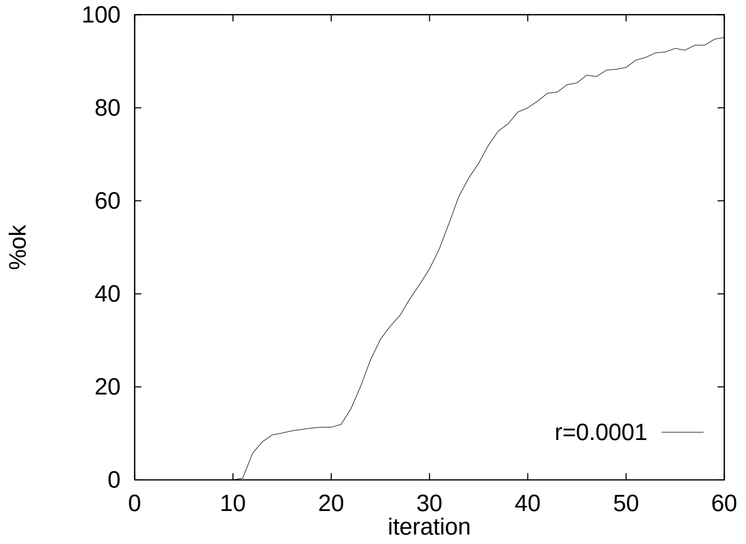


Figure D.6: Online, r=0.0001

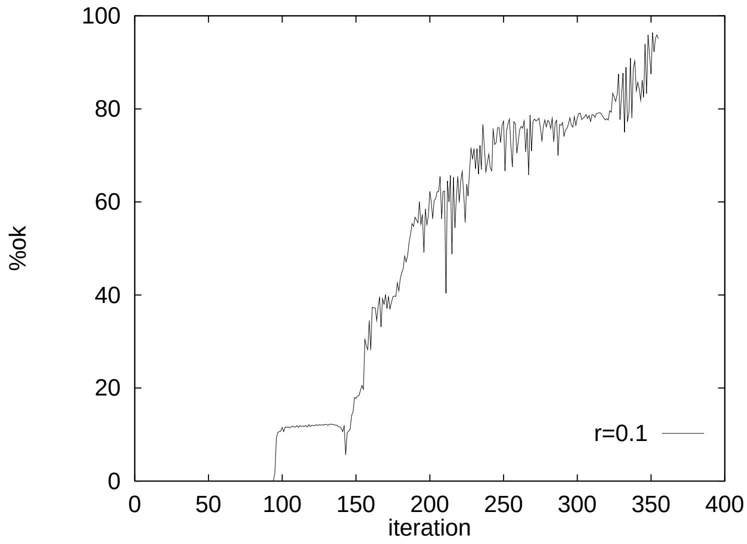


Figure D.7: Offline, r=0.1

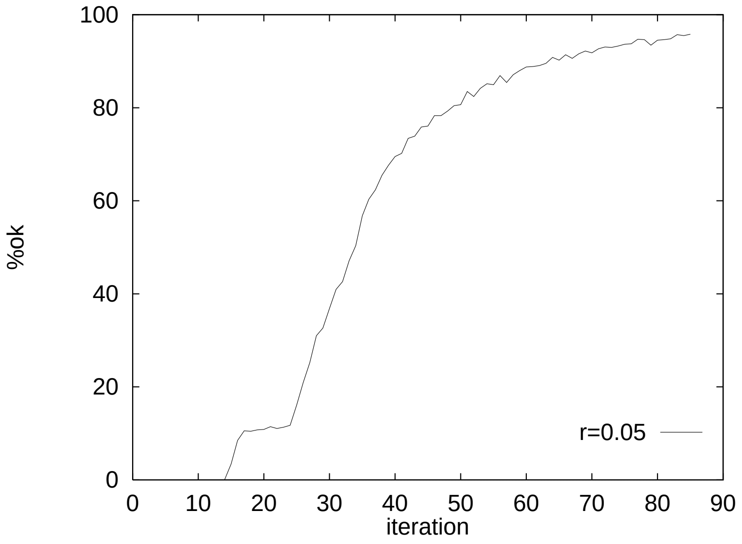


Figure D.8: Offline, r=0.05

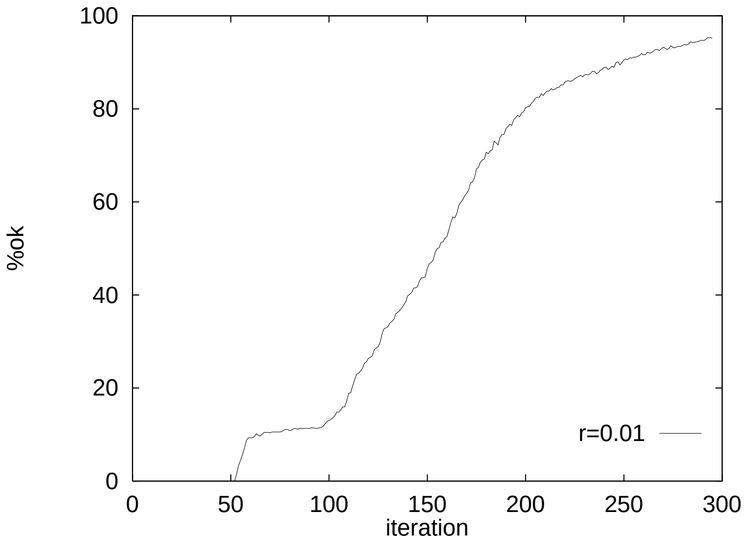


Figure D.9: Offline, r=0.01

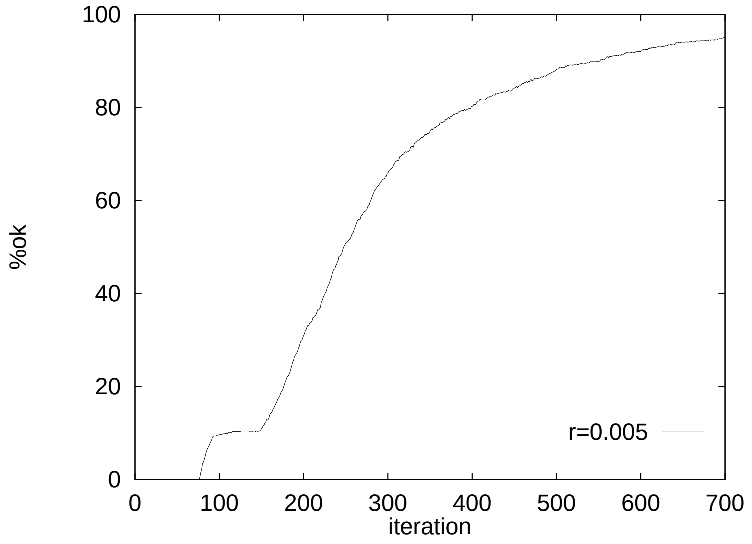


Figure D.10: Offline, r=0.005

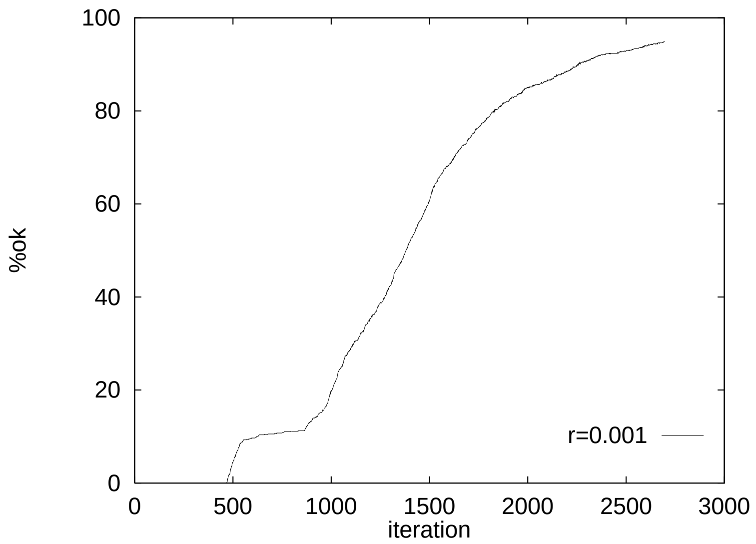


Figure D.11: Offline, r=0.001

Appendix E

Result session II

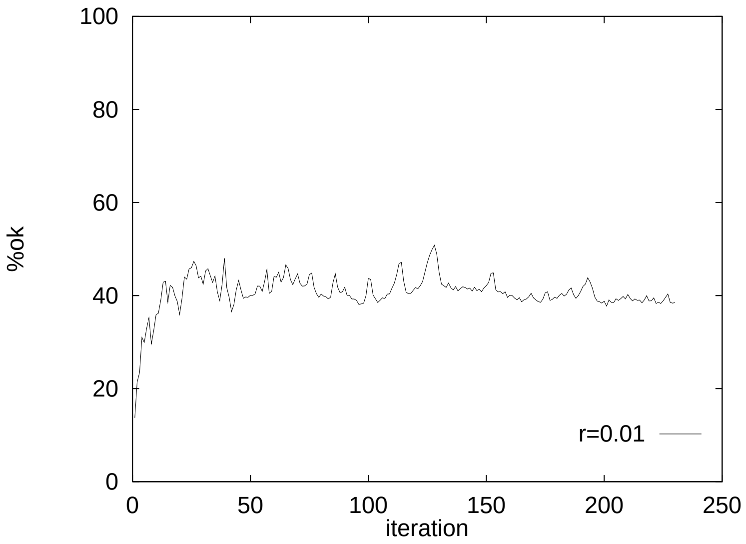


Figure E.1: Online, r=0.01

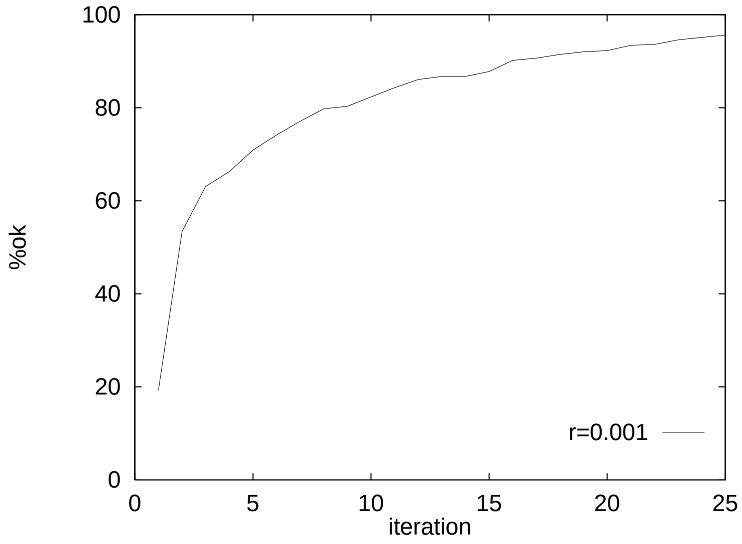


Figure E.2: Online, r=0.001

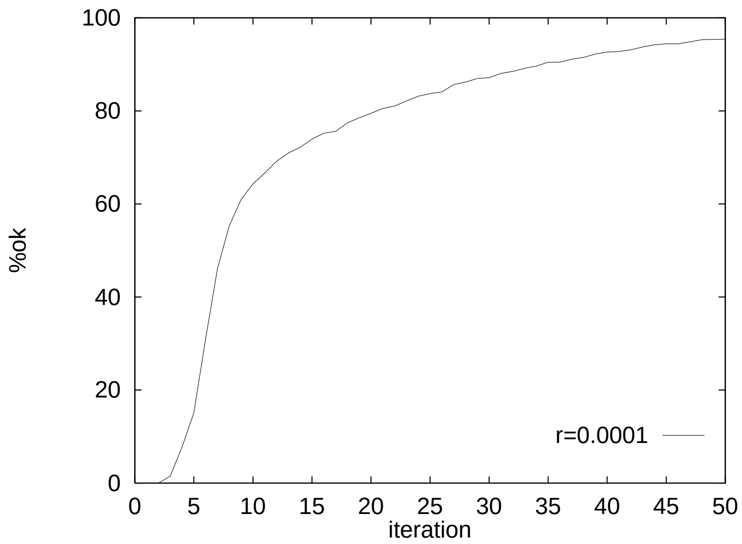


Figure E.3: Online, r=0.0001

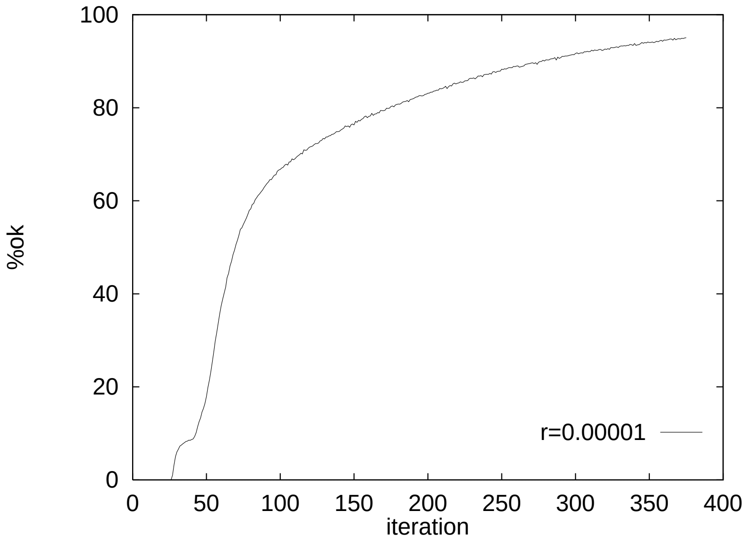


Figure E.4: Online, r=0.00001

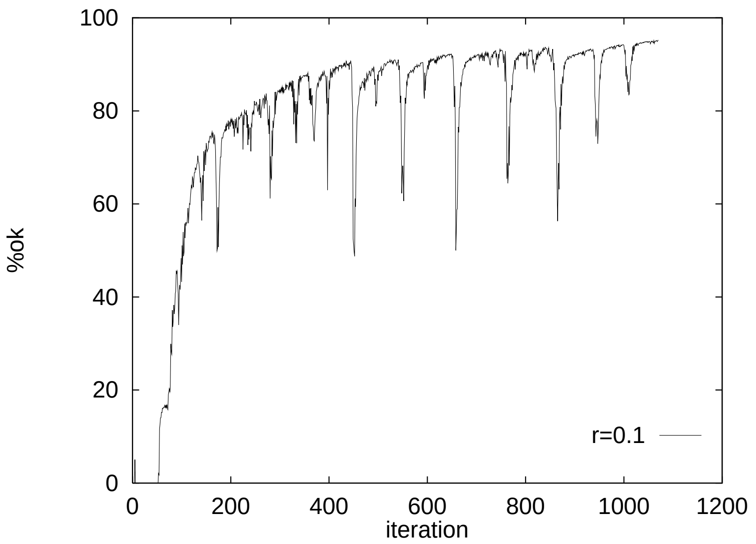


Figure E.5: Offline, r=0.1

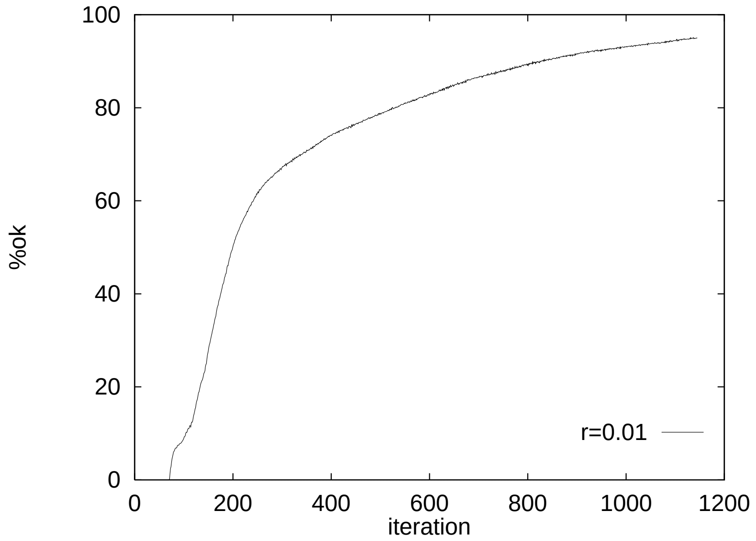


Figure E.6: Offline, r=0.01

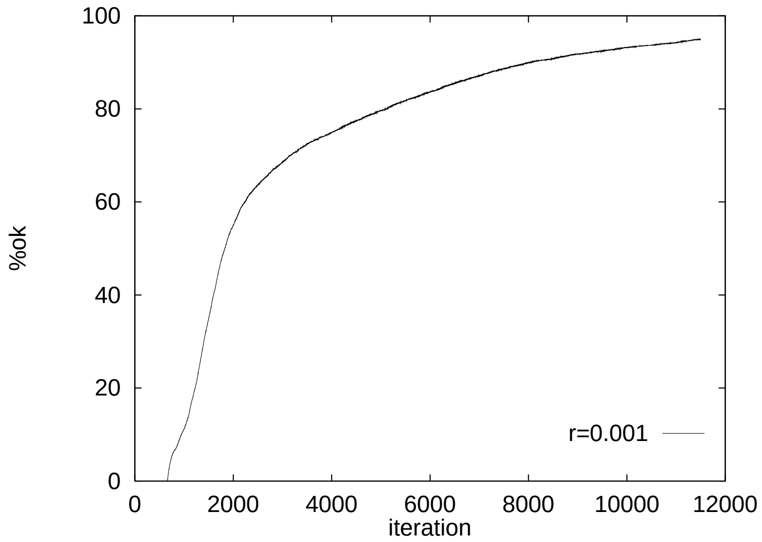


Figure E.7: Offline, r=0.001

Appendix F

Result session III

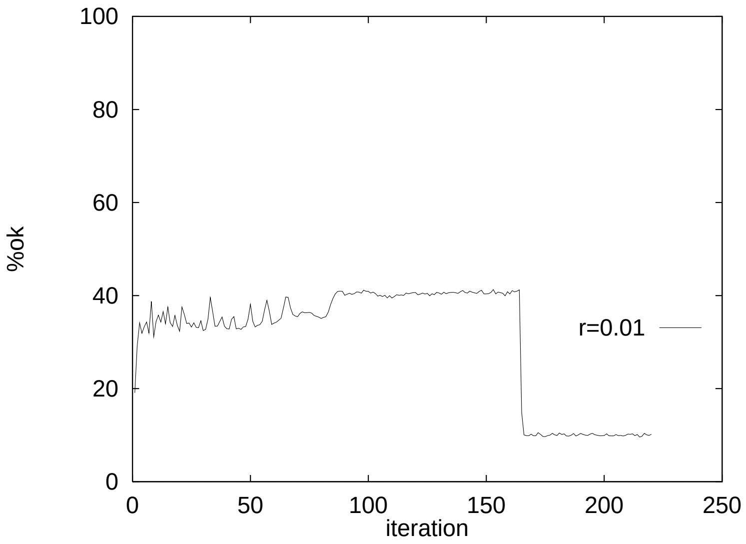


Figure F.1: Online, r=0.01

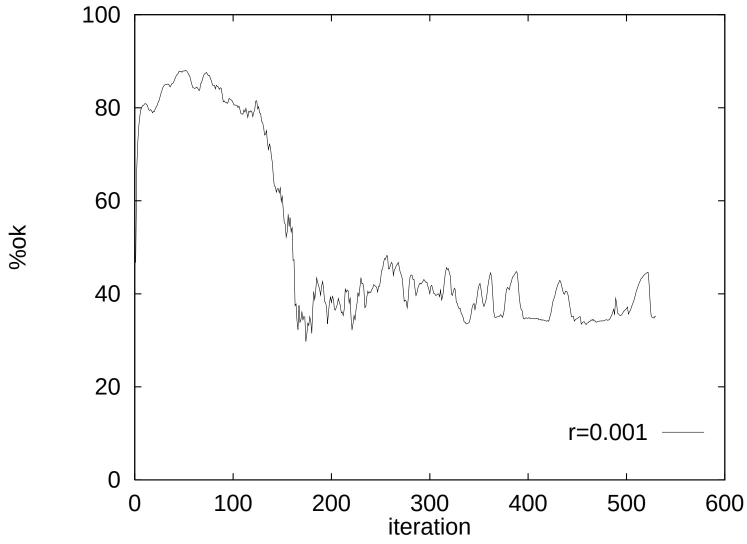


Figure F.2: Online, r=0.001

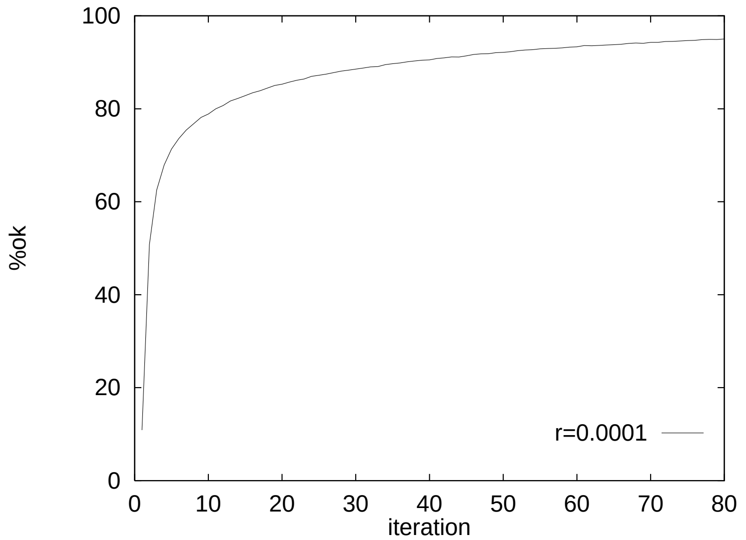


Figure F.3: Online, r=0.0001

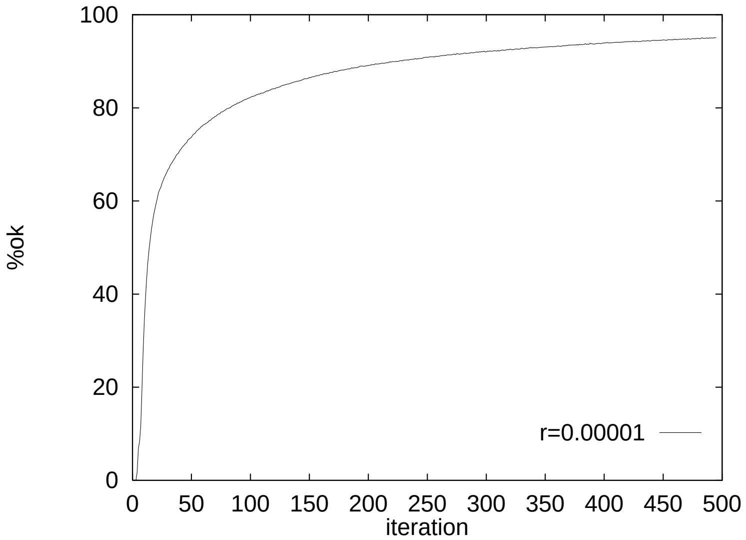


Figure F.4: Online, r=0.00001

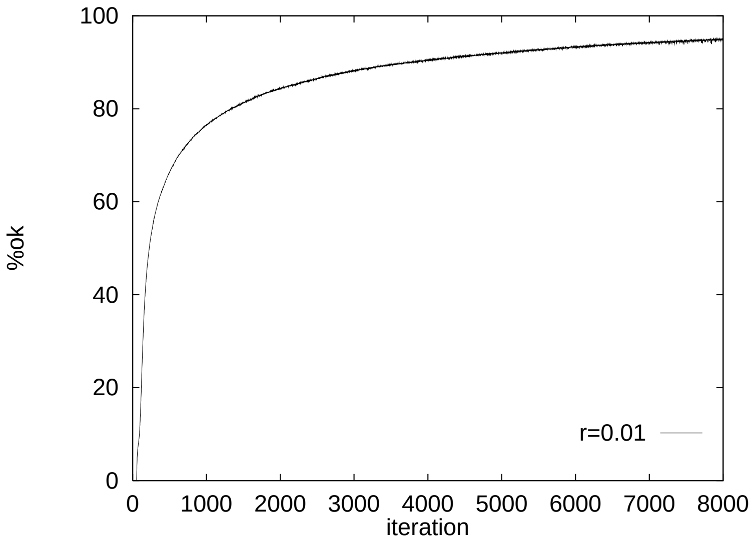


Figure F.5: Offline, r=0.01

Appendix G

Source code utilities

G.1 convmis.c

```
#include <stdio.h>
#include <string.h>

#include <ihead.h>
#include <mis.h>
/*
 * This program reads a mis-file and the corresponding class file,
 * writes the patterns in a format usable for the neural network.
 */

typedef struct
{
  int cls_tot;
  int base;
  char* cls_data;
} MisCls;

void freeMisCls(aMisCls)
MisCls* aMisCls;
{
  if (aMisCls)
  {
    if (aMisCls->cls_data)
      free (aMisCls->cls_data);
```

```
    free(aMisCls);
  }
}

/* width is in bits */
/* this proc shows the pattern image */
void MisToAscii(asc_fn , misdata, width ,height)
  char* asc_fn;
  unsigned char* misdata;
  int width;
  int height;
{
  int w_counter, h_counter, bit_counter ;
  static unsigned char byte_mask[8] = {128, 64, 32, 16, 8, 4, 2, 1};
  FILE* outf;


  /* open output file */
  outf = fopen (asc_fn, "a+");
  if (!outf)
  {
    printf("open err\n");
    return;
  }

  width = width /8;
  for (h_counter=0; h_counter<height; h_counter++)
  {
    for (w_counter=0; w_counter<width;  w_counter++)
    {
      unsigned char misbyte = misdata[h_counter*width+w_counter];
      for (bit_counter=0; bit_counter<8; bit_counter++)
      {
        if (misbyte & byte_mask[bit_counter])
          fprintf(outf, "*");
        else
          fprintf(outf, " ");
      } /* end for bit_counter */
    } /* end for w_counter */
```

```
    fprintf(outf, "\n");
  }/* end for h_counter */

  fprintf(outf, "\n");
  fclose(outf);
}

/* converts the mis-formaat to nn-formaat, also cutting the edge */
void MisToNN(aMis, aMisCls, nn_fname)
  MIS *aMis;
  MisCls *aMisCls;
  char *nn_fname;
{
  int mis_counter;
  int PattrnLen, byte_counter;
  FILE* outf;

  int NewHeightEnd, NewWidthEnd, NewHeightStart, NewWidthStart;
  int WidthCounter, HeightCounter;

  outf = fopen (nn_fname, "w+");
  if (!outf)
    fatalerr("MisToNN", "open", "failed to open the file");

  PattrnLen = aMis->entw/8 * aMis->enth; /* PattrnLen in bytes */

  if ((aMis->entw/8) % 4 || aMis->enth % 4)
  {
    printf("MisToNN hoogte/breedte niet deelbaar door 4\n");
    return;
  }

  NewHeightStart = aMis->enth/4;
  NewWidthStart = (aMis->entw/8)/4;
  NewHeightEnd = aMis->enth - aMis->enth/4;
  NewWidthEnd = aMis->entw/8 - (aMis->entw/8)/4;

  /* each image */
  for (mis_counter=0; mis_counter<aMis->ent_alloc * PattrnLen;
       mis_counter += PattrnLen)
```

```
  { /* data within image */
    for (HeightCounter=NewHeightStart; HeightCounter < NewHeightEnd;
         HeightCounter++)
    {
      byte_counter = mis_counter + HeightCounter * (aMis->entw/8);
      for (WidthCounter=NewWidthStart; WidthCounter<NewWidthEnd;
           WidthCounter++)
      {
        fprintf(outf, "%02x ", (int)aMis->data[byte_counter+WidthCounter]);
      }
    }/* end for HeightCounter */

    fprintf(outf, "%d\n", (int)(aMisCls->cls_data[mis_counter/PattrnLen] -
            aMisCls->base));
  }/* end for mis_counter */

  fclose(outf);
}


//reads a mis data */
MisCls* ReadMisCls(cls_fname)
char* cls_fname;
{
  FILE* inpf;
  int cls_tot, cls_counter;
  int cls_data_digit;
  int cls_base;
  static char inp_str[20];
  static char inp_str_hex[20] = "0x";


  MisCls* aMisCls = NULL;

  cls_base = '0';

  inpf = fopen(cls_fname, "r");
  if (!inpf)
  {
    printf("opening classification file:%s failed\n", cls_fname);
```

```
    return NULL;
  }

  if (fscanf(inpf, "%d", &cls_tot) != 1)
    fatalerr("ReadMisCls", cls_fname,"Can not find the tot_num");
  if (cls_tot < 0 || cls_tot> 1000)
    fatalerr("ReadMisCls",cls_fname ,"illegal tot_num");
  if (!(aMisCls = (MisCls*) malloc(sizeof(MisCls))) )
    syserr("ReadMisCls","malloc","MisCls");
  if (!(aMisCls->cls_data = (char*) calloc(sizeof(MisCls),cls_tot )) )
    syserr("ReadMisCls","calloc","MisCls->cls_data");

  for (cls_counter = 0; cls_counter<cls_tot; cls_counter++)
  {
    if (fscanf(inpf, "%s", inp_str) != 1)
      fatalerr("ReadMisCls",cls_fname,"Can not scanning data from file err");

    strcat(inp_str_hex, inp_str);
    inp_str_hex[5] = '\0';
    if (sscanf(inp_str_hex, "%i", &cls_data_digit) != 1)
      fatalerr("ReadMisCls",cls_fname,"Can not scanning datafrom mem err");
    if (cls_data_digit <= 0 || cls_data_digit > 255)
      fatalerr("ReadMisCls",cls_fname,"illegal  data err");

    aMisCls->cls_data[cls_counter] = (char) cls_data_digit;
    inp_str_hex[2] = 0;
  }

  aMisCls->cls_tot = cls_tot;
  aMisCls->base = cls_base;
  fclose(inpf);

  return aMisCls;
}

/*
mis_input_file: d0000_97.mis
program will read the corresponding .cls file
output: .nn text file with the bitmap-data in hex format
        and at the end a classification value
```

```
*/
void main(argc, argv)
int argc;
char* argv[];
{
  MIS* aMis;
  MisCls *aMisCls;
  int mis_counter;
  char* mis_ext_SP;
  static char inp_root_fname[250];
  static char mis_fname[250];
  static char cls_fname[250];
  static char out_fname[250];

  if (argc < 2)
  {
    printf("usage %s <mis_input_file> [out_root]\n", argv[0]);
    return;
  }

  strcpy(inp_root_fname, argv[1]);
  printf("argv1: %s\n", inp_root_fname);
  mis_ext_SP = strstr(inp_root_fname, ".mis");
  if (!mis_ext_SP)
    fatalerr("Main","parameter",
     "Can not find the mis-extension or\n this is not a digit-mis-file\n");
  *mis_ext_SP = 0;
  strcpy(cls_fname, inp_root_fname);
  strcat(cls_fname, ".cls");

  if (argc >= 3)
  strcpy(out_fname, argv[2]);
  else
  {
    strcpy(out_fname , inp_root_fname);
    strcat(out_fname , ".nn");
  }
  strcpy(mis_fname, argv[1]);
  printf("inp_root_fname: %s\n", inp_root_fname);
  printf("mis_fname: %s\n", mis_fname);
```

```
  printf("cls_fname: %s\n", cls_fname );
  printf("out_fname: %s\n\n", out_fname );
  aMisCls = ReadMisCls(cls_fname);
  aMis = readmisfile(mis_fname);
  if (!aMisCls || !aMis)
    fatalerr("Main", "mis-files", "error reading MIS or CLS file");
  if (aMisCls->cls_tot != aMis->ent_alloc)
  {
    printf("aMisCls->cls_tot != aMis->ent_alloc\n");
  }

  MisToNN(aMis, aMisCls, out_fname);
  freeMisCls(aMisCls);
  freemis(aMis);
}
```

G.2 sumcls.c

```
#include <stdio.h>
#include <iostream.h>

/* This program will read the total number of patterns
 * from several mis-class-files, put put them in a output file,
 * together with the total of all the patterns.
 */
int main(int argc, char* argv[])
{
  if (argc <= 2)
  {
    printf("This program will read the total number of patterns \n");
    printf("from several mis-class-files, put put them in a output file \n");
    printf("together with the total of all the patterns\n");
    printf("usage %s out_file inp_cls_files...\n", argv[0]);
    return 1;
  }

  FILE* outf = fopen(argv[1], "wt");
  if (!outf)
  {
```

```
    printf("opening %s for output failed\n", argv[1]);
    return 1;
  }

  char inpstr[100];
  int curnum;
  int totnum = 0;
  for (int outf_counter = 2; outf_counter < argc; outf_counter++)
  {
    printf("reading %s\n", argv[outf_counter]);
    FILE* inpf = fopen(argv[outf_counter], "rt");
    if (!inpf)
    {
      printf("opening %s for input failed\n", argv[outf_counter]);
      return 1;
    }
    int scanresult = fscanf(inpf, "%d", &curnum);
    if (!scanresult || scanresult == EOF || curnum <=0)
    {
      printf("scanning %s for num failed\n", argv[outf_counter]);
      return 1;
    }
    totnum += curnum;
    fprintf(outf, "%15s %5d\n",argv[outf_counter], curnum);
    fclose(inpf);
  }

  fprintf(outf, "%15s %5d\n","total", totnum);
  fclose(outf);
}
```

G.3 Concate.cpp

```
#include <stdio.h>
//
//When the training is not done in one session, the training proceduces
//several output files, all output file starts with iteration number 1.
//Actually the second output file should start with a number following
//the last iteration number of the previous output file.
```

```
//this program concatenates the files and fixes the iteration number problem.
//example:
//file 1:
//      1  0.779 1.800 1.739   934    9.873   81.720   8.407   9.6   8.7
//      2  0.820 1.800 1.746   881    9.482   81.134   9.384  10.2   9.6
//
//file 2:
//      1  0.827 1.800 1.739    40    8.895   82.600   8.504  10.4   9.0
//      2  0.831 1.800 1.746  1015    8.895   82.502   8.602  10.6   9.0
//
//this will become:
//      1  0.779 1.800 1.739   934    9.873   81.720   8.407   9.6   8.7
//      2  0.820 1.800 1.746   881    9.482   81.134   9.384  10.2   9.6
//      3  0.827 1.800 1.739    40    8.895   82.600   8.504  10.4   9.0
//      4  0.831 1.800 1.746  1015    8.895   82.502   8.602  10.6   9.0
//
//
inline int FindFirstDigitPos(const char* a)
{
  for (int k=0; k < strlen(a); k++)
    if (isdigit(a[k]))
      return k;

  return -1;
}

inline int FindFirstSpacePos(const char* a)
{
  for (int k=0; k < strlen(a); k++)
    if (isspace(a[k]))
      return k;

  return -1;
}

int main(int argc, char* argv[])
{
  if (argc <= 2)
  {
    printf("This program reads the tt-logfiles created by bp,\n");
```

```
  printf("changes the sequence and cancatenate them to the out_file\n");
  printf("usage %s out_file inp_files...\n", argv[0]);
  return 1;
}


FILE* outf = fopen(argv[1], "wt");
if (!outf)
{
  printf("Opening %s for output failed\n", argv[1]);
  return 0;
}

char InpLine[200];
char outstr[200];
int LineCnt = 1;
for (int k=2; k < argc; k++)
{
  printf("Concanating ...\n", argv[k]);
  FILE* inpf = fopen(argv[k], "rt");
  if (!inpf)
  {
    printf("opening %s failed.", argv[k]);
    fclose(outf);
    return 1;
  }

  while(1)
  {
    //read line
    char* fgetsRes = fgets(InpLine, 199, inpf);
    if (!fgetsRes)
    {
      if (feof(inpf))
        break;
      else
      {
        printf("Error: fgets returns null while not EOF\n");
        fclose(outf);
        fclose(inpf);
```

```
          return 1;
        }
      } //end if

      int FirstDigitPos = FindFirstDigitPos(InpLine);
      if (FirstDigitPos < 0 )
      {
        printf("Digit not found\n");
        fclose(outf);
        fclose(inpf);
      }

      int FirstSpacePos = FindFirstSpacePos(InpLine + FirstDigitPos);
      if (FirstSpacePos < 0 )
      {
        printf("Space after digit not found\n");
        fclose(outf);
        fclose(inpf);
      }

      //write to output file
      sprintf (outstr, "%6d", LineCnt);
      strcat(outstr, InpLine + FirstSpacePos + FirstDigitPos);
      fputs (outstr, outf);
      LineCnt++;
    } //end while

    printf("Concanating %s done.\n\n", argv[k]);
  } //end for

  return 0;
}
```

G.4 SplitClassData.c

```
#include <stdio.h>
#include <iostream.h>
#include <string.h>
const MaxClass = 10; //only available for digit images
```

```
/*
* This program will read the patterns from a datafile
* and writes each class to separate files.
*/
int main(int argc, char* argv[])
{
  if (argc != 4)
  {
    printf("This program will read the datafile and writes each class\n");
    printf("to a separate file\n");
    printf("Usage %s inp_file out_root tot_class\n", argv[0]);
    printf("where tot_class is the max num of classes\n" );
    printf("The inpfiles is assumed to have a header,\n");
    printf("which will be skipped\n");
    return 1;
  }

  int TotClass;
  if (sscanf(argv[3], "%d", &TotClass) != 1 || TotClass < 0
      || TotClass > MaxClass )
  {
    printf("invalid TotClass\n");
    return 1;
  }

  // input file
  FILE* inpf = fopen(argv[1], "rt");
  if (!inpf)
  {
    printf("opening %s for input failed\n", argv[1]);
    return 1;
  }

  //ouput files
  FILE* outf[MaxClass];
  int ClsConverted[MaxClass];
  char outfname[256];
  for (int k=0; k < TotClass; k++)
  {
    sprintf(outfname, "%s_%1d", argv[2], k);
```

```
    outf[k] = fopen( outfname, "wt");
    if (!outf[k])
    {
      printf("opening outfname for output failed\n");
      return 1;
    }
    ClsConverted[k] = 0;
  }//end for

  //converting
  int Err=0;
  char InpLine[2048];
  //read the header and skip it
  char* fgetsRes = fgets( InpLine, 2047, inpf);
  if (!fgetsRes)
  {
    printf("Error: header not found\n");
    return 1;
  }
  else
    printf("Header: %s\n", InpLine);

  while (1)
  {
    //read line
    char* fgetsRes = fgets(InpLine, 2047, inpf);
    if (!fgetsRes)
    {
      if (feof(inpf))
        break;
      else
      {
        printf("Error: fgets returns null while not EOF\n");
        return 1;
      }
    }//end if(!fgetsRes)

    char* LastSpacePos = strrchr(InpLine, ' ');
    if (!LastSpacePos)
    {
```

```
    printf("Space not found\n");
    Err++;
    continue;
  }
  int SpacePos =  int(LastSpacePos - InpLine) + 1;
  if (SpacePos % 3) // not a multiple of 3
  {
    printf("Illegal space-pos found\n");
    Err++;
    continue;
  }

  //find class for this pattern
  int ClassNum = -1;
  if (sscanf(LastSpacePos, "%d", &ClassNum) != 1 || ClassNum<0
      || ClassNum >= TotClass)
  {
    printf("Unknown class encountered\n");
    Err++;
    continue;
  }
  //writes to the correct output file
  fputs(InpLine, outf[ClassNum]);
  ClsConverted[ClassNum]++;
}//end while

fclose(inpf);
//show information about conversion
if (Err)
  printf("%d errors occured during conversion !!!\n");
else
{
  printf("conversion ok\n");
  int TotConverted = 0;
  for (int k=0; k < TotClass; k++)
  {
    printf("Class %d : %d\n",  k, ClsConverted[k]);
    TotConverted += ClsConverted[k];
    fclose(outf[k]);
  }
```

```
    printf("Total   %d\n", TotConverted);
  }//end if
}
```

Appendix H

References

Am72 S. Amari, *Learning Patterns and pattern Sequences by Self-Organizing Nets of Threshold Elements*, IEEE Trans electr. Comp. 21, 1197 (1972)

An72 J.A. Anderson, *A Simple Neural Network Generating Interactive Memory*, Math. Biosci. 14, 197 (1972)

Bu77 T.H. Bullock, R.Orkand, and A. Grinnell: *Introduction to Nervous Systems* , Freeman, San Francisco(1977)

Co72 I.M. Cooke and M. Lipkin: *Celluar Neurophysiology: A Source Book*, Holt, Rinehart & Winston, New York(1972)

Ec77 J.C. Eccles: *The Understanding of the Brain*, McGraw-Hill, New York(1977)

Go92 Rafael C. Gonzalez and Richard E. Woods, *Digital Image Processing*, Addison-Wesley UAS(1992)

Gu96 K. Gurney, *Computers and Symbols versus Nets and Neurons*, UCL Press Limited, UK(1996)

He96 Tom Heskes and Wim Wiegerinck, *A Theoretical Comparison of Batch-Mode, On-Line, Cyclic, and Almost-Cyclic Learning*, IEEE Transactions on neural networks(1996), vol 7, page 919-924

Ho82 J.J. Hopfield, *Neuural Networks and Physical Systems with Emergent Collective Computational Abilities*, Proc. Natl. Acad. Sci. USA 79, 2554 (1982)

Je93 T.T. Jervis and W.J. Fitzgerald, *Optimization Schemes for Neural Networks*, Cambridge University Engineering Department, England(1993)

Ka85 E. Kandel and J.H. Schwartz, *Principles of Neural Science*, Elsevier-North Holland, San Francisco(1985)

Ko72 T. Kohonen, *Correlation Matrix Memories*, IEEE Trans. Electr. Comp. 21, 353 (1972)

Na72 K. Nakano, *Associtron - A Model of Associative Memory*, IEEE Trans. Syst. Man Cybern. 2, 381 (1972)

Ni94 *NIST Form-Based Handprint Recognition System*, U.S. Department of Commerce (1994)

Pa85 D.B. Parker, *Learning Logic: Casting the Cortex of the Human Brain in Silicon*, MIT Techn. Rep. TR-47(1985)

Re90 B.Müller and J. Reinhardt, *Physics of Neural Networks, An Introduction*, Springer-Verlag, Berlin(1990)

Ru86a D.E. Rumelhart, G.E. Hinton and R.J. Williams, *Learning Representations by Back-propagating Errors*, Nature 323, 533(1986)

Sh79 G.M. Shepherd, *The Synaptic Organization of the Brain*, Oxford University Press, New York(1979)

Vu95 L. Vuurpijl, T. Schouten, J. Vytopil, *Performance Prediction of Large MIMD Systems for Parallel Neural Network Simulations*, Elsevier Science BV. The Netherlands(1995)

Wi92 P.H. Winston, *Artificial Intelligence*, Addison-Wesley, USA(1992)