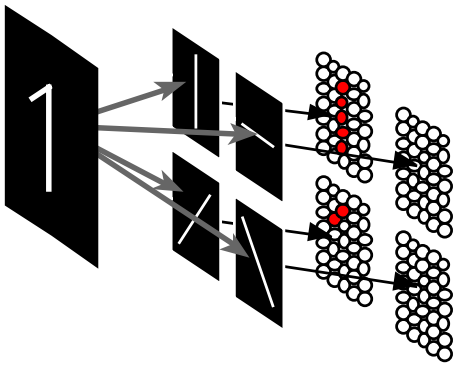
Optical Implementation of a Neural Network for Pattern Recognition



Diploma work for the degree of Master of Science

Jan Lagerwall

April 1997

INSTITUTIONEN FÖR MIKROVÅGSTEKNIK
CHALMERS TEKNISKA HÖGSKOLA

DEPARTMENT OF MICROWAVE TECHNOLOGY
CHALMERS UNIVERSITY OF TECHNOLOGY
GÖTEBORG, SWEDEN

Abstract

THIS REPORT DESCRIBES the construction of a dynamic optical hybrid system for implementing multi-layer neural networks. The communication between neurons is performed by amplitude modulating optical signals with dynamic transmission filters realized with a ferroelectric liquid crystal spatial light modulator (FLC-SLM). A large part of the information processing is thus performed in parallel. The amplitude modulated signals are detected by a CCD-camera and some further processing is done in a conventional computer.

The system should recognize two-dimensional graphic patterns and it has been tested on the ten Arabic digits in different shapes. As neural net algorithm a modified version of the Neocognitron model of Kunihiko Fukushima has been used. The system has been simulated in MATLAB and its ability to generalize and its sensitivity to disturbances have been examined. Furthermore the possibility of using a binary FLC-SLM to perform multi-level amplitude modulation has been verified.

After training on a small number of different series of the ten digits, the simulated network has capability to generalize to shapes that are not part of the training set. Unfortunately the synaptic dimensions of the network are so large that the optical implementation could not be performed with the equipment presently at our disposal. With further refined optical components this hybrid system will probably be highly competitive with systems using entirely digital computation.

Preface

THIS TEXT GIVES an account of my diploma work which I performed at the department of microwave technology in the research group for diffractive optics directed by assoc. prof. Sverker Hård, during the period October 1996 to April 1997. In chapter 2 I give a short introduction to the concept of neural networks while the subsequent chapters are devoted to the specific system I developed in the project.

For stimulating discussions during the course of the work I would like to sincerely thank my supervisors Sverker Hård and my assistant supervisor techn. lic. Björn Löfving at the institution mentioned above, as well as. Dr. Peder Rodhe at FLC Optics AB. I would also like to give my thanks to assoc. prof. Mats Nordahl at the institution of theoretical physics for tips and ideas he has contributed. Furthermore, he was the one who, in the Neural Networks course here at Chalmers, first introduced me to the subject in a most inspiring way.

Contents

1. Introduction

THE BIOLOGICAL brain is today unsurpassed in its ability to identify and distinguish different kinds of patterns. It is an old dream to construct machines capable of the same thing, but this has proved very difficult.

A common algorithm used in this work, is the so called neural network, a structure which takes the biological brain as a model. Normally such algorithms are implemented in a computer, but since neural networks are strongly parallel structures, while today's computers are serial, such solutions suffer from very long computation times.

In this work we present a neural network based system for identification of two-dimensional graphical patterns, in which a large part of the computations are executed simultaneously using an optical setup. The whole system has been simulated in Matlab and some essential features of the optical part have been realized and evaluated in a laboratory setup.

2. Background

2.1 What is a neural network ?

Practically all computers we meet today are built according to the same basic pattern. The central component is the processor which can perform a number of complicated tasks. It is, however, the sole component capable of these tasks, and hence the computations are executed serially, i.e. one operation is performed at a time.

The serial computer is in many situations an excellent tool, but some problems are parallel as to their nature and they become quite difficult for it to solve. An example is pattern recognition of various kinds. If we make small modifications of the input to a normal computational algorithm its response should change thereafter. If the task instead is to identify a letter or a digit, the response should be stable even if we for instance change the font. A machine set to solve such problems must be capable to generalize and extract the essential out of a huge information flow. In order to manage this it needs a so called *associative* memory [1].

The addressing of associative memories differs radically from that of traditional computer memories. A computer has no problems remembering the code to your credit card. It stores the information at a specific address in the memory and when the computer needs the code it reads the exact data from this address. You, on the other hand, do not have this possibility since the brain is an associative memory. In order to bring the code forward you might have to relate it to a word, or maybe it doesn't pop up until you stand with your fingers on the keyboard. While the processor of a computer must give the exact address to the place where the sought information is stored, information from an associative memory is brought up when a fragment of a stored memory reaches it.

It is apparent that the optimal uses for the different types of memory are not the same. Complicated computations which a serial computer evaluates in no time can be very time consuming for a human being, but to understand what a person is saying, or to realize that both oak and birch are trees, are tasks which most people perform without reflecting.

In order to create machines capable of such tasks scientists have taken inspiration from the design of the biological brain. Instead of a single powerful processor our brain has an enormous number of very simple processors, *neurons*, which work in parallel and are connected in a very complex fashion. Neurons are binary units; either they give a signal with a fix strength or they stay quiet.

Each neuron has connections, *synapses*, to a large amount of its neighbors. Through the synapses the neuron receives signals from its neighboring neurons. Since synapses differ in strength, some neighbors will affect the neuron more than others. The weighted signals are added and if the total input exceeds a certain threshold value, the receiving neuron will emit a signal. The strengths of the synapses may be altered during one's lifetime and this is probably what happens when we learn something. Our memory thus consists of a certain set of synaptic strengths.

In a neural network one tries to mimic the structure of the brain. A number of models, differing in complexity and performance, have been developed [1,2,3]. The choice of model depends on the problem to be solved. Common to all models is that the associative memory consists of the set of weighted connections between neurons. The function of neurons, the type of connections, and the method of training the network, are things which often vary from one model to another. Often one combines several different models in one neural network.

The family of neural networks used in this work is the so called *feed forward* network, also called *perceptrons* (see figure 1). These networks have one input- and one outputside and the direction from the former to the latter defines the direction of information flow during evaluation.

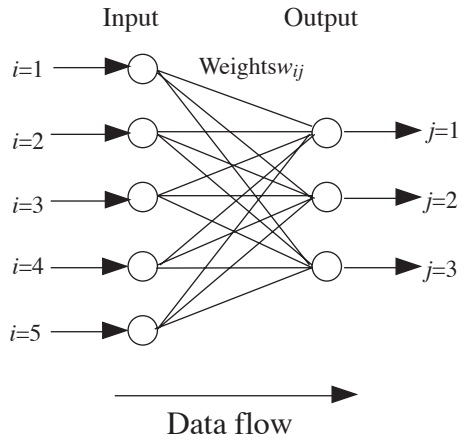


Figure 1. A simple one-dimensional perceptron with one layer of neurons.

In its simplest form a perceptron has only one process stage. It is then said to have one layer of neurons. The network in figure 1 is an example of a singlelayer perceptron. We see, however, that in reality we have two sets of neurons (represented by circles in the figure), but in the first no evaluation of data is performed. These neurons are needed as input receptors. Hence, in a comparison with the human nervous system, the neurons in the first column correspond to the rods and cones of the eye. By layers in neural networks we thus mean the combination of a set of neurons and the weights between them and their inputs.

2.2 Multiple layers

A deeper analysis of the function of the perceptron reveals that there exists a family of problems, the linearly unseparable problems, that are theoretically impossible to solve with only one layer of neurons [1,2]. The classic example of such problems is the logical XOR function.

By connecting two or more layers in series we break this barrier. In a multilayer network the treatment of input is performed in several steps. While the neurons of a onelayer perceptron must achieve a complete answer directly from the external input, each neuron in a multilayer network performs only a partial operation. By dividing the task into several simpler partial problems it is solvable for the network even if it is linearly unseparable.

For the task set up in our project (detection of two-dimensional graphical patterns) a multilayer structure is preferable. Each layer extracts different features in the input to the layer, and we thus get a stepwise abstraction of the problem. In the first layer the input is analyzed concerning very simple features such as straight lines with different inclinations. The signals from this layer then constitute the input to the next, which extracts more complicated features. When the information reaches the output layer it has gone through an extensive preprocessing which renders the response of the system much more reliable.

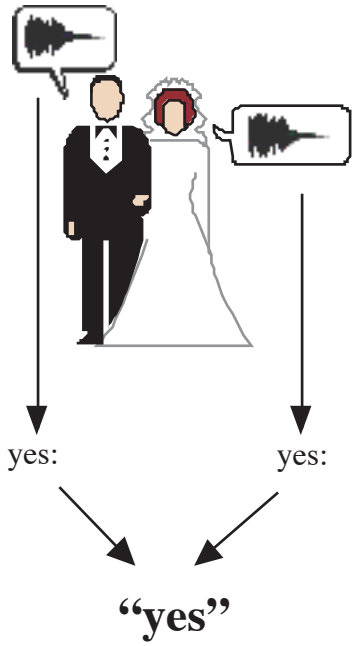


Figure 2. An example of human feature detection. The sounds uttered by the man and woman may be very personal, so in order to understand what they in fact mean the listener must do an analysis with respect to phonetic features, phonemes. By dividing the speech in phonetic constituents, the priest happily realizes that the sounds reaching his ears both express the message ”yes”.

2.3 Hardware

Neural networks are often implemented as a program run in a serial computer. This works reasonably well for simpler problems if the computer is fast. It is, however, an unsatisfactory solution and presently much work is done on developing hardware which works in a parallel manner [1,2,3,5,6,8]. One tempting thought is to implement parts of the system, or even the whole system, optically. For instance one can use an optical correlator to compare the input to stored images [6].

We have in this work chosen an optical implementation as described in figure 3. An input neuron is represented by a light source of which the intensity is proportional to the strength with which the neuron signals. The synapses from the neuron are represented by filters with transmittance values reflecting the synaptic weights.

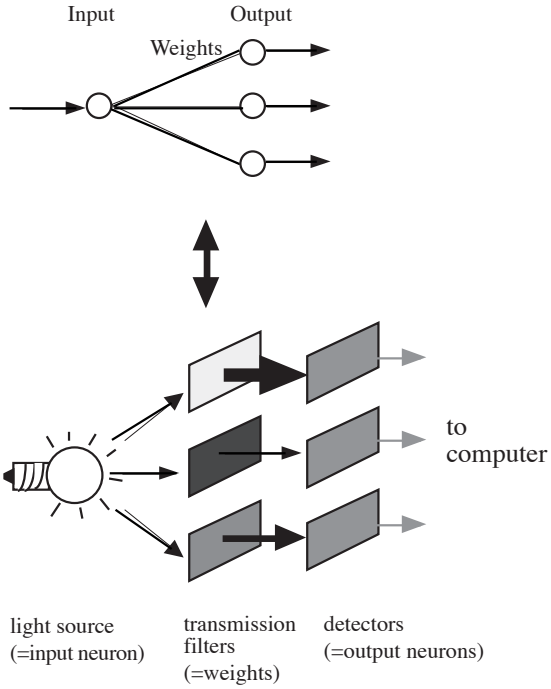


Figure 3. A neural network may be realized by amplitude modulation of light.

With detectors we measure the light intensities after passage through the filters. These values reflect the product between the inputs and the synaptic weights. The point is that a large amount of multiplications may be performed at the same time (and at the speed of light) by using matrices of light sources, weight filters and detectors. I will return with an extensive description of the optical system in chapter 5.

3. The goal of our project

THIS WORK IS in a sense a freestanding sequel to the diploma work of Richard Englund entitled *Optical Neural Networks for Associative Image Processing* [5]. We have both aimed at constructing neural networks capable of recognizing and distinguishing a number of graphical patterns. The optical implementations in our projects are both based on amplitude modulation of incoherent light.

Englund chose to implement a network based on the Ho-Kashyap algorithm, a relatively simple solution with only one layer. Due to the limitations of single-layer neural networks I wanted to go one step further and implement a network with several layers. Since the elements of the Ho-Kashyap algorithm are vectors, the two-dimensional picture to be treated with this algorithm has to be converted into a one-dimensional vector. In doing this we lose the valuable information that lies in the correlation between neighboring picture elements. For my work I searched for an algorithm that preserves such information.

After having searched the literature for suitable alternatives I settled for a model called the *Neocognitron*. This model has a structure that is well suited for the task, and furthermore, the very attractive property for optical implementation, that weights as well as in- and outputs are non-negative. As the model is rather complex I have found it necessary, however, to make some simplifications of it. Partly in order to fit the work within its time limits, and partly in order to adapt the algorithm to the conditions set by the optical implementation.

Englund used a fix photographic mask to perform the weightings in his network. The weights were calculated entirely in a computer. This has two major drawbacks. The first is that no regard is taken to the inevitable distortion of the optical system. The other is related to the dynamics of the system. If one wants to add a pattern to the set of patterns recognized by the system, the weights must be recalculated and the fix mask must therefore be replaced by a new one.

The other area where we wanted to take our system one step further than Englund's, concerned the weight mask. Instead of a fix photographic mask we have chosen to use a Spatial Light Modulator, or SLM, with a ferroelectric liquid crystal (FLC) layer as active medium. The modulator is divided into a large number of picture elements (pixels) which can be switched between a transparent and an absorbing state. FLC-modulators are binary elements, that is they have no gray scale.

The pixel pattern can easily and quickly be changed which gives us the possibility of training the weights of the network with the optical equipment that will be used when the system is in operation. The aberrations introduced by the optical implementation are part of the training set and the neural network can therefore learn to compensate for them.

Furthermore, implementing the weight mask with an SLM gives the system completely different dynamics with respect to change of synaptic weights and thereby the set of stored patterns. The dynamics of the SLM also opens up a way to optically implement a multi-layer network, see chapter 5.

Our aim has not been to construct a very fast or efficient system. From the start we have been limited by the performance of the hardware at our disposal. Our goal has rather been to give a proposal to a technique which could, with better hardware and much further development, prove to be highly competitive with present solutions.

4. The neural network in our system

AS INPUT PATTERNS we have chosen the ten Arabic digits 0 to 9 in different guises, drawn with 27*27 quadratic binary pixels. This picture is first fed through a set of neuron layers which has been trained to extract different features of the picture. Thereafter the by now highly preprocessed information reaches the output layer consisting of 10 neurons with continuous output signals. Each neuron in this layer corresponds to one digit. Since the neurons in this layer give continuous-valued response we can, in addition to see which digit the system has identified the picture with, also get a measure of the reliability of the answer. This is done by comparing the response of the strongest output signal to that of the second strongest.

4.1 The Neocognitron model

Since the task of our system is to detect two-dimensional patterns I have chosen to implement a multi-layer system. I have started from the so called *Neocognitron* model [2,4,9] of Kunihiko Fukushima, but I have made a number of simplifications.

The model features two different types of neuron; S- (Simple) and C- (Complex) neurons. They both have non-negative outputs but their operation and tasks are radically different. Each layer in the Neocognitron contains only one type, but an S-layer is always followed by a C-layer, and together they make up what I call a *complex*. The feature detection is performed in the S-layer, while in the C-layer a simple processing of the S-layer output takes place. The purpose of the latter is to make the system less sensitive to lateral or vertical shifts of the input pattern. Our system features two complexes followed by the output layer which is of a different nature.

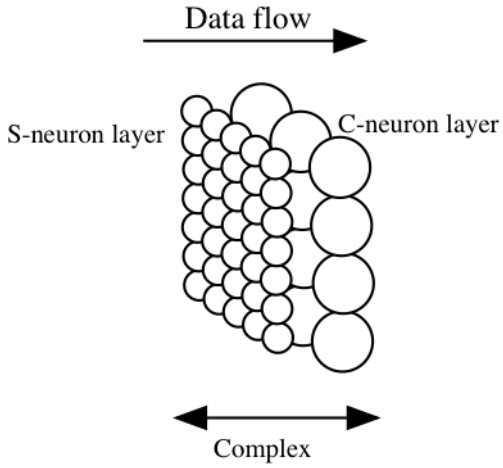


Figure 4. An S- and a C-layer comprise a complex in the Neocognitron structure. When data flows from the S- to the C-layer a reduction of dimensions takes place. Therefore the C-layer has fewer neurons than the S-layer.

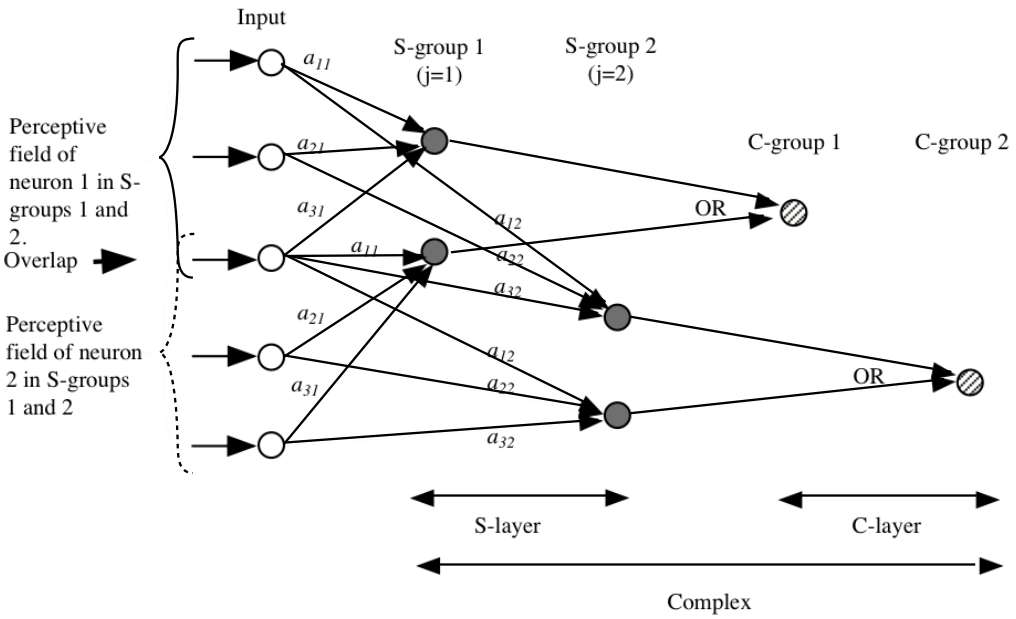


Figure 5. A very simple one-dimensional Neocognitron network. Here we have only two groups of two neurons in the S-layer and accordingly two groups of one neuron in the C-layer. The receptive field of each S-neuron is three neurons large and the two groups have one input neuron in common. Note that the neurons within a group share the same weight setup. The neurons of the C-layer perform a logical OR function of the outputs from the S-neurons they are connected to.

Figure 5 shows a one-dimensional picture of the Neocognitron structure. The neurons of the S-layer are divided into a number of *groups*. The neurons within a group have the same weight setup to the input layer and thereby they become detectors of the same feature. The coupling to the input is locally confined so that each neuron 'sees' only a small part of the image. The neurons of the group are connected to different parts of the input so that each neuron has a unique receptive field. Neighboring neurons have, however, overlapping fields. Together, the neurons of a group scan the whole input image.

The point with this structure is that all neurons within the group search for the same feature (they share the same weight setup) in different, locally confined, parts of the input image. In order to profit maximally from the group structure, we want to avoid that more than one group specializes in one feature. In the training process one therefore sees to it that the weight setup to each group is unique.

Figure 6 illustrates the function of the two-dimensional Neocognitron structure used in our system. In part b of the figure the direction of the receptive field as a function of S-layer position is illustrated.

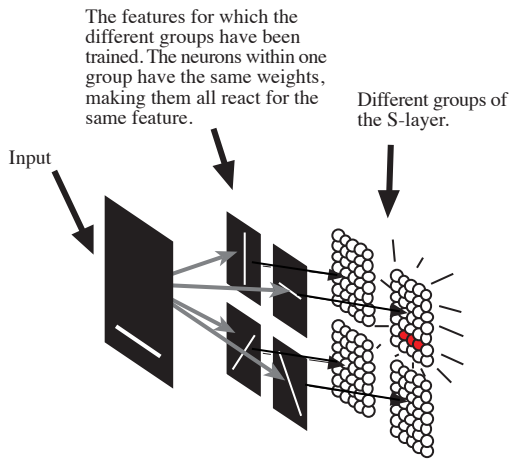


Figure 6a. The S-layer is divided into a number of groups, where the neurons within a group share the same weight setup. Each group scans the whole input image, but its neurons react only for the specific feature they have been trained for.

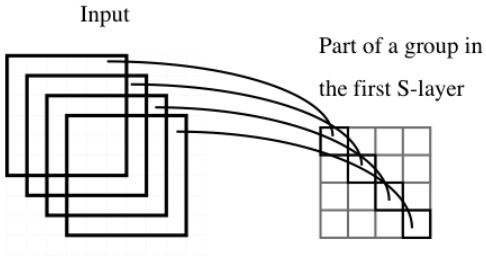


*Figure 6b. Illustration of the local connections between S-neurons and input. Each S-neuron is connected to only 6*6 input neurons, but all in all, the neurons within a group scan the whole input image.*

The C-layers are of a much simpler nature. Each group in an S-layer has a corresponding group in the succeeding C-layer[1]. Each neuron in the C-group is connected to 2*2 S-neurons and there is no overlap of receptive field between C-neurons. If one of the four S-neurons making up the input to a C-neuron signals with a strength above a certain threshold, the C-neuron will fire. The relation is illustrated in figure 7.

Hence, the neurons of the C-layer perform a logical OR function. We get a smoothing out of the outputs from several adjacent S-neurons and thereby the sensitivity to vertical or lateral shifts of the input image is diminished. A small displacement is reflected in the S-group output, but not in the response from the C-group.

The groups of the C-layer thus become binarized and dimensionally reduced images of their respective S-groups, see figure 7. This is a rough simplification of the Neocognitron C-layer but the basic function is the same [2,4,9].

[1]In my system the first complex has a slightly different structure. I will return to this modification in section 4.5.

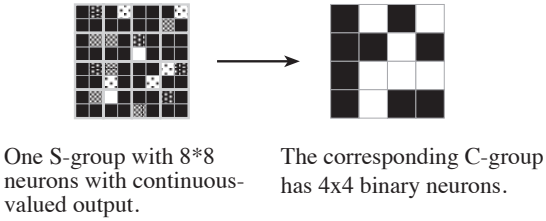


Figure 7. The relation between the neurons of an S-group and those of the corresponding C-group.

4.2 Inhibitory neurons

The fact that most neural networks feature both positive and negative quantities present difficulties when doing an optical implementation. The light intensity reflects the magnitude, but how do we give light a plus or minus sign? The Neocognitron model is in this respect very attractive as its weights as well as its neuron signals are non-negative.

To do without negative weights one adds a layer of so called I-neurons[2] to each S-layer. The name reflects that they are *inhibitory* as opposed to the *excitatory* S-neurons. The latter excite their listeners, that is, a large output U_{ex} from an S-neuron stimulates the succeeding neurons which take U_{ex} as input, to fire. Inhibitory neurons have precisely the opposite effect; they inhibit their listeners, i.e. a large I-neuron output U_{in} suppresses the response of succeeding neurons. The relationship is illustrated in figure 8.

[2]In the original Neocognitron the I-neurons make up one special group of the S-layer, but I think the picture of the network becomes clearer if one treats this group as lying outside the S-layer.

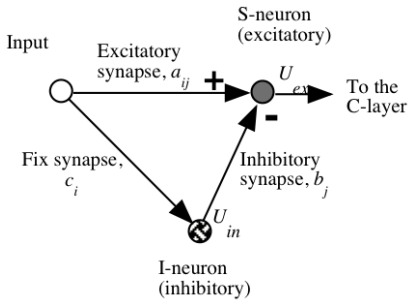


Figure 8. The Neocognitron features two kinds of neurons: excitatory and inhibitory. The latter suppress the response of the former.

The I-layer contains only one group. This resembles the groups of the S-layer as far as dimensions and local connections to input, but its neurons are inhibitory and their weights are not trained. The I-group is from the beginning given a fix symmetrical weight setup c_i constructed such that the weights decrease linearly towards the edges of the receptive field, and their sum equals one, see figure 9. Note that I now use just one letter i to index the input neurons, even though they are divided into rows and columns of the two-dimensional receptive field.

Figure 9. Graphical representation of the weight distribution c_i, i.e. the strengths of the synapses between input and the inhibitory neurons. Each square corresponds to a synapse and its grayshade reflects the strength (the lighter it is, the stronger the synapse). The weight distribution is normalized, i.e. the 36 weights sum up to 1.

The connections between inhibitory and excitatory neurons are illustrated in figure 10. Each I-neuron is connected to all of its sister neurons in the S-layer via synapses with a strength b_j, where j indexes the S-groups. With sister neurons I mean S-neurons with the same geometric location within its group, and thereby the same input, as the I-neuron in its I-group.

Connections leading from input to S-neurons are called excitatory synapses and their strengths are in this text labeled a_{ij}. Connections between an I-neuron and its sister S-neurons are called inhibitory synapses and have strength b_j. As illustrated in figure 10a this quantity is common for all neurons within an S-group.

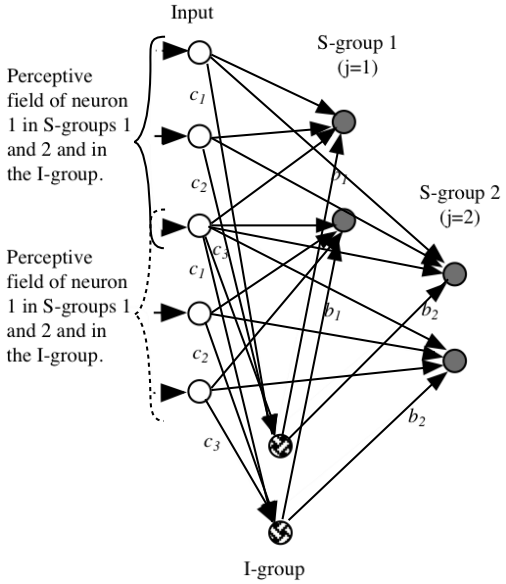


Figure 10a. The structure of figure 5 completed with the inhibitory group. Note that all neurons of an S-group have the same weight to the I-group. In the interest of readability I have omitted the strengths of the excitatory synapses (a_{ij}).

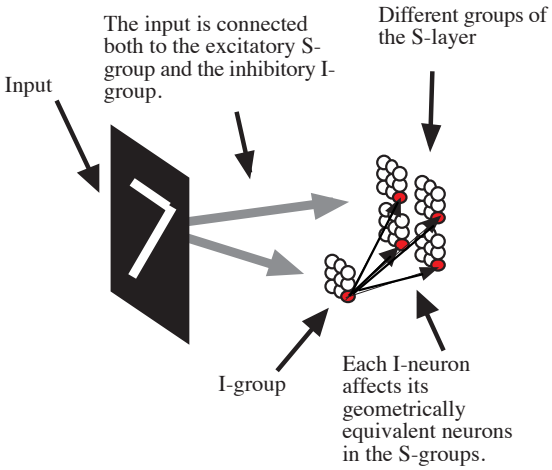


Figure 10b. The connection between inhibitory and excitatory neurons in our two-dimensional architecture.

The inhibitory neurons are needed in order to prevent the excitatory neurons from reacting on incorrect features that *contain* the feature they have been trained for. An extreme example is shown in figure 11. If all the pixels in the receptive field of an S-neuron are lit (case a) the signals through its excitatory synapses will attain their maximum strengths irrespective of the feature they have been trained for. Such an image will, however, also provoke a strong inhibitory response which suppresses the response from the excitatory neuron. On the other hand, if the input corresponds to the specific feature of the group (case b) the excitatory synapses will give exactly the same signals as in case a, but since the inhibitory neuron reacts weakly, we will in this case get a strong S-neuron signal.

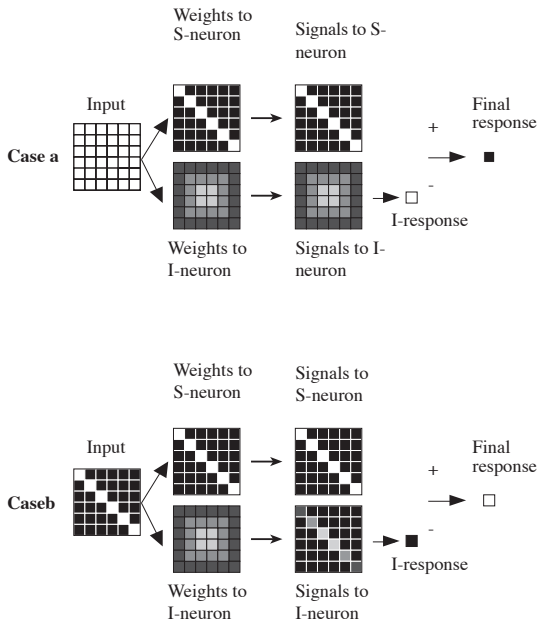


Figure 11. The cooperation between excitatory and inhibitory neurons.

The two complexes of the system are in principal equivalent as far as the S- and I-layers are concerned. They differ only in the number of groups and size of local receptive fields.

4.3 The output layer

In the output layer we leave the Neocognitron model. The ten neurons (one for each output alternative) of this layer are all fully connected to the C_2-layer and each connection is freestanding from the others. Weightsharing is thus not incorporated in this layer, which means we have as many weights as there are connections, i.e. 10*15*16=2400 (number of output neurons * number of groups in C_2 * number of neurons per C_2-group).

The neurons of the output layer are of a very simple kind. The output from a neuron is simply the weighted sum of its input signals, i.e. the scalar product $\mathbf{w} \bullet \mathbf{c}_2$ where $\mathbf{w}$ is a vector containing the weights to the neuron and $\mathbf{c}_2$ is a vector containing the outputs of the C_2-layer. Since no local connections exist between the output layer and the C_2-layer, the vectorization of the latter does not affect the outcome. The correlation between neighboring pixels is in any case not used.

The weights of the output layer are continuous and may be of either sign. In order to implement this layer optically some way of representing negative numbers with the optical system is hence needed. One can for instance displace all values with an offset value such that the most negative value is raised to exactly zero. One extra pixel, representing the offset value, is then needed in the weight matrix, see for instance Englund [5]. In our system the output layer is not implemented optically.

4.4 Summary of central concepts

Before going on to an account of the training process, I would like to give the reader a well needed breathing space, and repeat the definitions of complex, layer and group, since these concepts will appear frequently in the text to follow. Figure 12 gives a comprehensive view of the whole system architecture. The neurons of the network are divided into several

consecutive *layers*. Neurons within a layer take the signals from neurons of the preceding layer as input.

In the Neocognitron model we have three different types of layer. A *complex* consists of one layer of each type; one S-, one I- and one C-layer. These are intimately connected to each other. Each layer is divided into a number of *groups*. My system features two complexes whose S-layers consist of nine and sixteen groups respectively. The two C-layers feature four and fifteen groups respectively. Each I-layer consists of only one group and its function is to inhibit the response of the excitatory groups in the corresponding S-layer, such that these won't respond to incorrect input.

Neurons within a group share the same weight setup but are connected to different parts of the input. Each group has, however, a weight distribution that differs from those of the other groups. Through training we want to specialize each S-group such that its neurons react on a certain feature in the input. Ideally none of the other S-groups should react for the same feature. Every group should be a unique indicator of one certain feature.

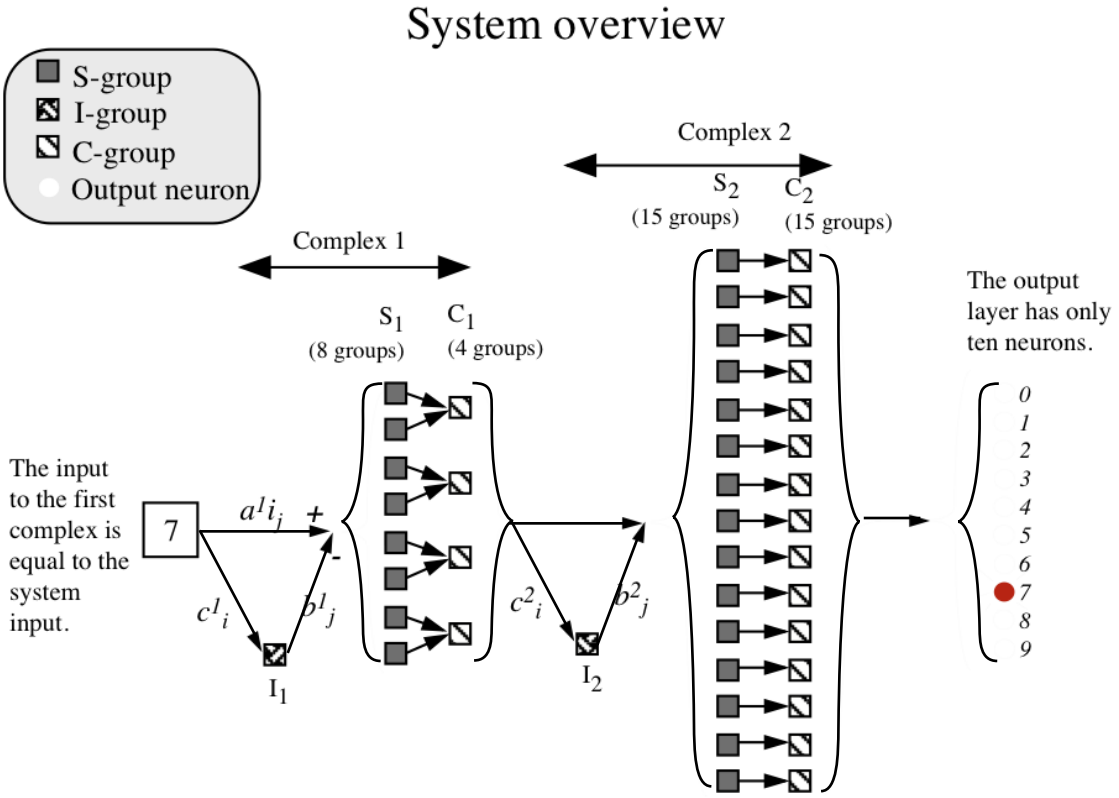


Figure 12. The network architecture of our system. Information flows from left to right (as an example of input we have chosen the digit 7) and during the passage a dimensional reduction is performed. The table below summarizes the dimensions at different stages. In the output layer there are no groups and only 10 neurons; one for each output alternative.

Table 1. Overview of the system dimensions.

Layer	*In*	I_1	S_1	C_1	I_2	S_2	C_2	*Output*
Neuron type	binary exc.	cont. inh.	cont. exc.	binary exc.	cont. inh.	cont. exc.	binary exc.	cont.
Number of groups	1	1	8	4	1	15	15	1
Group dimension	27*27	22*22	22*22	11*11	8*8	8*8	4*4	10*1
Number of neurons in layer	729	484	3872	484	64	960	240	10
Weight-sharing*	-	yes	yes	-	yes	yes	-	no
Perceptive field size	-	6*6	6*6	2*2	4*4*4†	4*4*4†	2*2	15*4*4†
Overlap between perceptive fields of neighboring neurons	-	yes	yes	no	yes	yes	no	no
Number of weights to be set through training	-	0	288	0	0	960	0	2400

*Weight sharing means that several neurons have the same weight distribution. Since C-neurons perform a logical OR-operation on incoming data we can, in their case, not speak of weights in any real sense.
† The neurons are connected to all groups of the preceding layer. The size of the receptive field therefore becomes *n*s*s* where *n* is the number of groups in the preceding layer and *s* is the lateral size of the receptive field.

4.5 Training of the first S-layer

The layers to be trained in the system are the two S-layers and the output system. Since the input of later layers consists of the output from preceding layers, we must start by training the first S-layer separately. During the training of the second S-layer the weights of the first are fixed and its neurons work exactly as they will when the whole system is in use. When both S-layers are sufficiently trained the Neocognitron part of the system is ready and the training of weights to the output layer can begin.

The three layers are trained according to different methods. The S_1-layer is not really trained in the normal sense. Instead we use a rather special method developed by Fukushima [4]. All weights are initially set to zero. Instead of using pictures from the normal system input (Arabic digits) as training set, each group is given a binary picture, with the size of the local receptive field, of the feature it should recognize. Only the connections leading from lit pixels are reinforced. The others maintain zero strength. In this way the neurons of the group are made to react strongly only on this feature.

The weights are calculated in one step through the formula:

$$a_{ij} = q_1 \cdot c_i \cdot u_{ij}$$
$$b_j = q_1 \cdot \sum_i c_i \cdot u_{ij} \qquad (1)$$

The values u_{ij} stand for the 6*6 neuron image that group j is trained with. Each picture represents one feature (see figure 14). With the constant q_1 we may adjust the magnitude of the resulting weights and the weight setup c_i is the fix weight setup from figure 9. The process is illustrated in figure 13. As this part is very simple, it is most easily realized entirely in the computer.

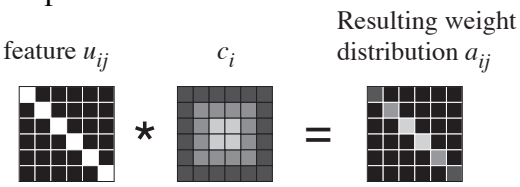


Figure 13. The training of the weights to the first S-layer. Black in the figure corresponds to the value 0, and white to 1. See also figure 14.

The choice of training pictures is by no means self-evident. Since it is difficult to get the neurons to detect curves because of their narrow field of view, we started by using eight pictures of straight lines with different inclinations, see figure 14a. It turned out, however, that this approach made the system too sensitive to different inclinations of the input images. For instance, an italic version of a digit provoked responses from entirely different groups than the straight version.

For the next try we used the images of figure 14b. Half of the groups should still react for straight lines, while we hoped that the others would detect angles and curves. They did in fact have this effect, but unfortunately they also reacted strongly on all horizontal and vertical lines. Hence, they reacted for practically any input, which of course is highly unsatisfactory.

In the final version I returned to only straight lines, but this time with only the four different inclinations shown in figure 14c. As can be seen in the figure each line now appears in two versions: one thin and one thick version. This is because the Neocognitron structure with inhibitory and excitatory neurons (described in chapter 4.2) exhibits a property which turns out to be a major drawback when applied in our system. The feature detectors become sensitive also to the *thickness* of lines. They thus react very differently on two images of the same feature but drawn with pens of different thickness. This drawback is most serious in the first S-layer since its neurons are connected directly to the system input, i.e. the digits to be recognized, which exhibits large variations in line thickness.

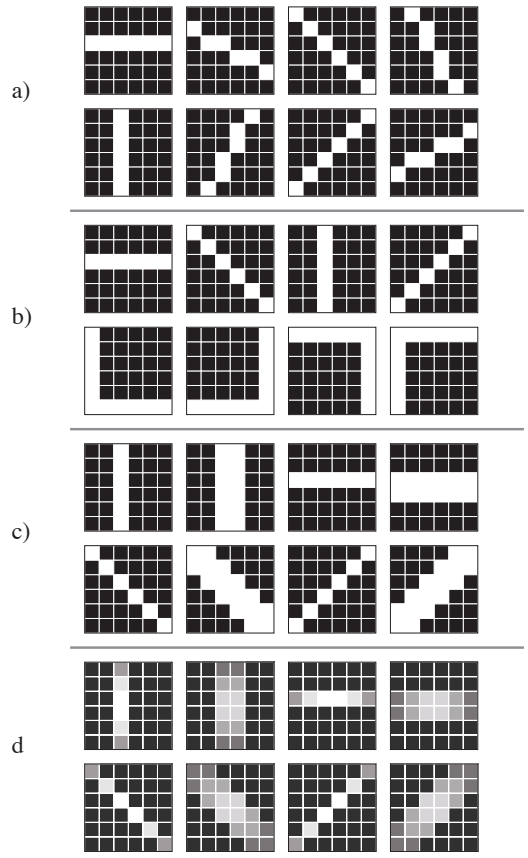


Figure 14. The patterns used to train the weights to the first S-layer. Each group in the layer specializes in one of the patterns. Three different sets of patterns were tried out. The best results were obtained with the set shown in c). Part d) shows the weight distribution after training with this set of patterns.

In order to compensate for this defect I trained the groups pairwise. Each pair specialized in one line inclination, but one of the groups was trained with a thin line while the other one was trained with a thick line. When the system is used the response from the two groups of a pair are superposed and the combination becomes the input to one single C-group. This group thus detects lines with a certain inclination, almost independently of the line thickness.

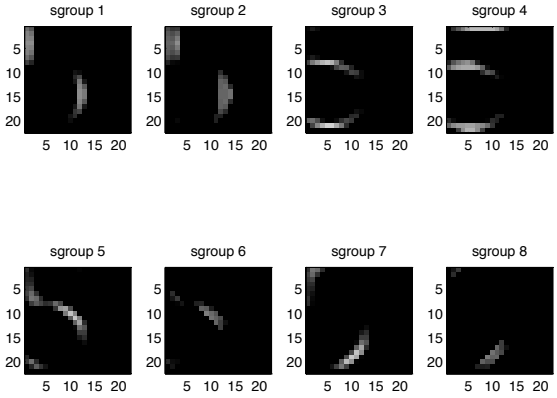


Figure 15a. The response from the eight groups of the first S-layer when the digit 5 in the font Monaco is given as input.

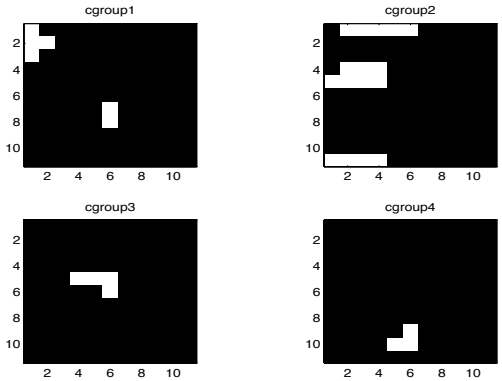


Figure 15b. The response from the four groups of the first C-layer when the digit 5 in the font Monaco is given as input.

4.6 The training of the second S-layer.

The method used to train the S_1-layer has the virtues of being fast and producing groups that are good detectors of different features. However, it requires that we know exactly what these features look like. For the first S-layer it is not too difficult to pick suitable features, but when we get to the second complex, we are on a higher abstractional level and it is much harder to imagine what the features should look like. If we make a bad choice the performance of the whole system will suffer. I have therefore made the second S-layer search for features on its own.

During the training process we present input images showing the ten digits to the system. The first complex performs its feature extraction, and the response from the neurons of the C_1-layer becomes the input to the second S-layer. Instead of having some sort of teacher that corrects the synaptic weights according to how the neurons reacted and how they should have reacted to a certain input, we let the groups compete internally[3] each time an image is presented. The group with the strongest reacting neuron gets its weights reinforced in accordance with the image having provoked the response. A complete account of this process is given in appendix 2.

In order to achieve a good result with such a training method it is important to have a suitable weight distribution at the start. A random start configuration easily leads to the situation that only one group (the one reacting most strongly to the first image) is developed, or that all groups develop in the same way. To avoid this I started by training the S_2-layer in the same way as the S_1-layer, i.e. I gave each group a feature of its own to recognize. The weight distributions of each group were thereafter normalized. The combinations I used for the startout training are shown in figure 16.

The features on this level are different combinations of the four features the C_1-groups were trained four. They can be combined in 15 different ways, and it is these combinations that I have chosen as

[3]Note that all neurons within a group have the same weight setup. Therefore we say that groups, not neurons, compete with each other.

starting point in the training. One could have chosen many other combinations (not only built on the C_1-features), as long as the different S_2-groups differ at the beginning of the self organizing part of the training process.

Since the dimensions of the C_1-groups are 11*11 pixels, an S_2 local receptive field of 6*6 pixels is far too big; it wouldn't be very local. Therefore I have made this only 4*4 pixels large.

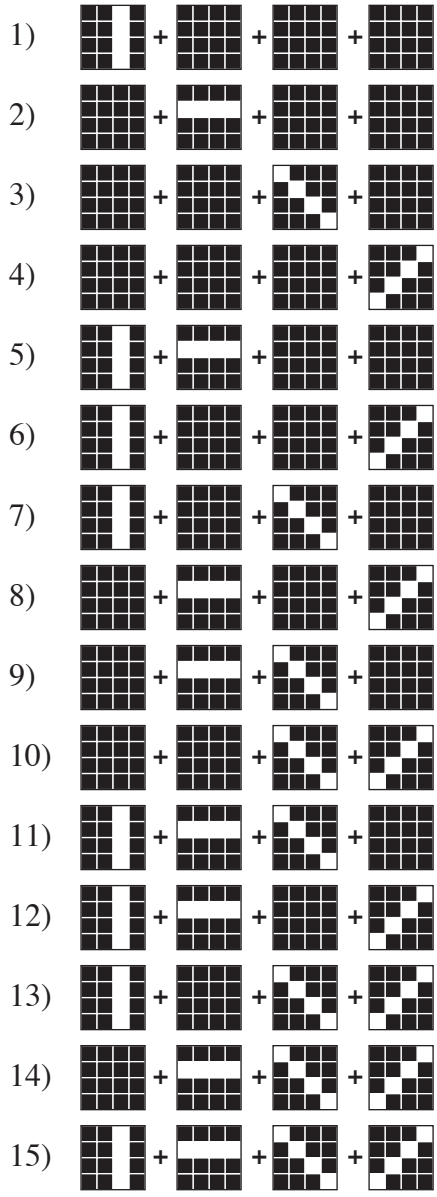


Figure 16. The fifteen training patterns (each pattern is a row of four smaller patterns) that were used to give the S_2-groups a start configuration for the self organization.

The point with the self organization is, however, that the layer by itself should find the most important features on this level, and therefore the pre-training must not be too strong. Hence, I decreased all excitatory synapse strengths, a_2, by a factor of 3, and set the inhibitory ones, b_2, to zero before the self organization started. This training method is similar to the one Fukushima developed in his first version of the Neocognitron [9].

In order to avoid that the weights to one single group grow stronger and stronger, and thereby wins every competition whether its weight distribution fits the input or not, we give the groups a 'bad conscience'. A large bad conscience reduces the group's chances of winning a competition. The winner of a competition gets its bad conscience increased while the others get theirs reduced.

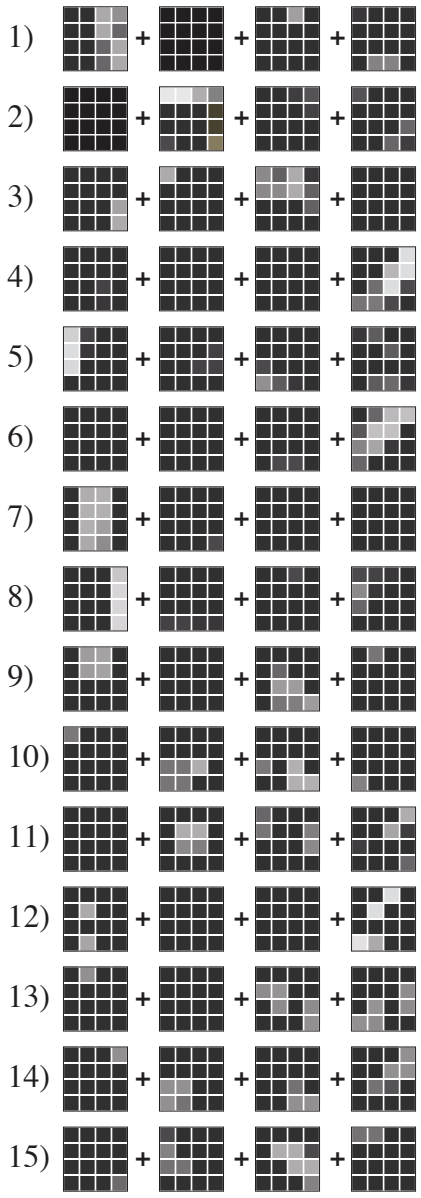


Figure 17. The configuration of the fifteen S_2-groups after self organization.

In order to get every group to specialize on one unique feature we apply so called lateral inhibition between the groups. This means that the response from one group is reduced with a certain percentage of the sum of the responses from all other groups. If a large number of groups respond approximately with the same strength on one type of input, the lateral inhibition will see to it that the signals from all these groups are weakened and none of them wins the current competition. The final weight distribution to the S_2-layer, after self organization, is presented in figure 17.

During the training of the second S-layer we apply the optical system exactly as it will be applied when the whole system is in use for pattern recognition. This leads to the advantage that the system can learn to compensate for possible optical aberrations.

4.7 The training of the output layer

After having completed the training of the two Neocognitron layers, there remains the training of the output layer. All of its ten neurons are fully connected to all the neurons of the C_2-layer. The weights are initially given small random values, and are then trained with a traditional error correction method, see for instance [1]. The different input patterns are presented for the system, and the readytrained Neocognitron complexes perform their abstraction. The result reaches the output layer which calculates a response based on this. The actual response is compared to the correct answer and the error is used to modify the weights.

The neurons of the output layer give continuous output signals, and do not have, as is usual in this type of network, a sigmoidal[4] threshold function. The response of the system is the digit represented by the loudest output neuron. A good measure of the reliability is achieved by comparing its output signal to those of the other neurons. The correct answer is of course that only the neuron representing the digit given as input fires with maximum strength while the others are totally quiet.

The function of the system is highly dependent on the training set used for the output layer. Both the number of training

[4]A sigmoidal threshold function often used is the hyperbolic tangent (tanh) function. It is very close to a step function which switches between -1 and 1 when the input changes sign.

patterns and their types are of large importance. With a small number of training patterns the system becomes very good at detecting these but generalizes badly. If one increases the number of training patterns it takes longer time to train the system, and the error may never be as low, but the generalization capacity is much better.

5. The optical implementation

BY COMBINING TWO spatial light modulators (cf. figure 18) and a CCD camera with a computer, that addresses the SLMs in real time as well as performs the non-optical part of the neural network algorithm, we have constructed a generic neural layer. Both input and weight setup can readily be changed by updating the settings of the SLMs. In this way we can implement a virtually unlimited number of layers.

An advantage of using an extra SLM in presenting the input, instead of directly projecting the input image on the weight matrix, is that the optical neural network system may be placed free from the detector. One could for instance send an image from a camera carried by an airplane, via radio to the rest of the system, placed in a stable and protected environment on the ground. More than one user can then also share the same pattern recognition system.

5.1 Optical representation of continuous-valued weights

Since the weights are continuous-valued while our SLMs are binary, we must simulate grayscale in some way. There exist two apparent alternatives which, however, both give us a certain quantization of the value. The first is to do a spatial multiplexing by representing each weight by several pixels of the SLM. The weight zero corresponds to all pixels set to a light blocking state, while the maximum weight value is represented by all pixels being transparent. In between these limits we can have a number of gray levels, the number of which depends on how many pixels we use for this simulation.

The other solution is to use time multiplexing by reading off the CCD a number of times in a row, with updates of the SLM in between. The different images are then added to each other. A large weight is represented by a pixel being transparent during a large number of exposures, while pixels corresponding to small weights are black most of the time.

We have chosen the former alternative with 4*4 SLM pixels per weight. From now on, when I speak of *gray scale pixels*, I thus mean a kind of macropixel, consisting of 4*4 physical pixels of the SLM.

We did two different measurements to test the performance of this simulation method. An account of the results can be found in chapter 6.

5.2 The optical setup

A sketch of the optical construction is given in figure 19. The first component after the light source is a so called beam expander, i.e. two lenses separated by a certain distance. This component is needed to achieve an even illumination of the whole image area of the input SLM. Normally a light beam has a Gaussian profile which means we have to expand it

to a diameter much larger than the image size of the SLM in order for the intensity variation to be of negligible order.

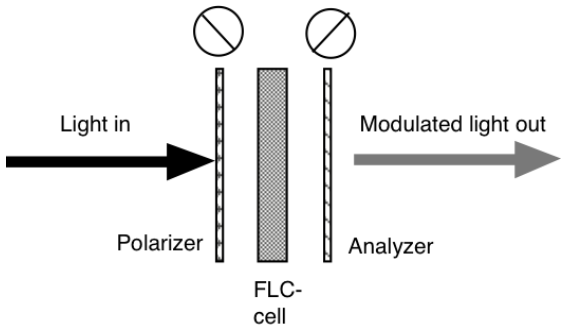


Figure 18. A Spatial Light Modulator (SLM) consists of a liquid crystal cell surrounded by crossed linear polarizers.

Our SLMs consist of the active FLC-cell surrounded by crossed polarizers (polarizer and analyzer), see figure 18. The light from the first SLM is linearly polarized in the direction set by the analyzer of this component. In order not to lose more light intensity than necessary, the polarizer of SLM no. 2 must be turned to this direction. If the retardation of the FLC cell is perfectly adjusted to the light wave length, the function of SLM no. 2 is independent of the direction of the polarizer (as long as analyzer and polarizer are crossed), but if this is not the case the optic axis of the medium must be carefully directed relative to the polarizer, in order to achieve maximum contrast [7].

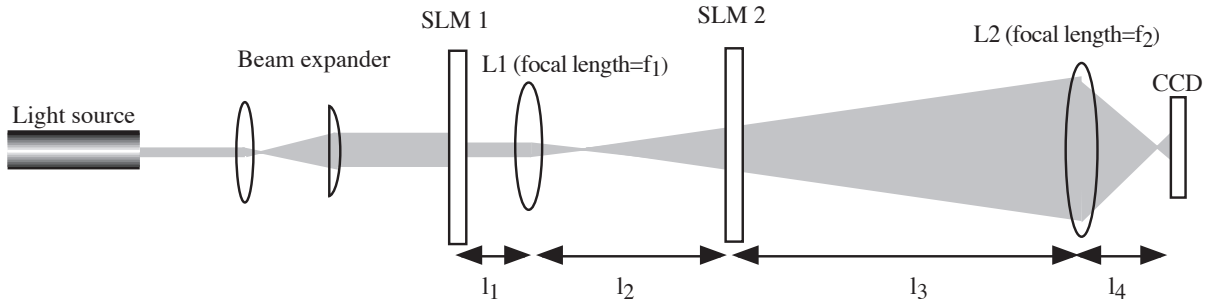


Figure 19. The optical part of the pattern recognition system. The picture generated by SLM 1 is projected onto SLM 2 through the lens L1. The composite picture at SLM 2 is projected onto the CCD-camera through lens L2.

On the first SLM, with a resolution of 128*128 binary pixels, we set a pattern corresponding to the input of the current layer. Since we have limited ourselves in our system to binary input neurons, each neuron can be represented by one pixel in the SLM pattern. If the neuron fires, the pixel is transparent, otherwise it is dark.

The picture produced by illuminating the modulator is projected through a lens on a larger SLM. Its 320*320 pixels are set to represent the weights between the current input and the S-neurons. Another lens projects the combined picture on a CCD camera which records the intensity of each pixel. This is proportional to the product of input and weights. By the virtue of this process the computer is relieved of a large part of its computational burden.

In order to simulate our Neocognitron system of two complexes we start by showing the input of the whole system (a picture of a digit) on the first SLM and the weights to the first S-layer on the second SLM. The output of the camera is transmitted to the computer which carries out the remaining calculations of equation (2) in appendix 3, and when this is ready it calculates the C_1-layer internally. After that the output of the C_1-layer is set on the first SLM, since this constitutes the input

to the S_2-neurons, and the S_2- and C_2-layers are calculated in the same manner.

The pixel size of SLM no. 1 is 220*220 μm^2 and in SLM no. 2 the pixels are 80*80 μm^2. The camera has 500*500 pixels of 8*12 μm^2 area. The difference in pixel size forces us to enlarge and reduce the image in between different stages of the process. Projection of the relatively large images on each other requires lenses of rather large diameter. The tolerance limits for distortion in the system are also very narrow, so the demands on the optics of the system are rather high.

For the calculation of the first S-layer each input pixel should be projected onto 3*3 weight pixels a time, and during the calculation of the S_2-layer the relationship is 1 onto 6*6 (see appendix 1). In using 2*2 pixels of the first SLM for each input neuron during the latter calculation, we can retain the same magnifying ratios and therefore the same physical setup in both cases.

Since each weight pixel uses 4*4 pixels for the grayscale simulation, 3*3 weight pixels take up 12*12 physical pixels of SLM no. 2. Lens no. 1 in figure 19 must thus project a 220*220 μm^2 pixel on an area made up of 12*12 quadratic pixels, each 80 μm wide, that is on 960*960 μm^2. Hence, the lens should magnify the pattern of the first SLM 4.36 times in projecting it on the plane of the other SLM.

When projecting the image of the second SLM on the plane of the CCD camera, the rectangular pixel shape of the latter posed a problem. Either a cylindrical lens may be used to extend the quadratic SLM pixels to the same proportions as the camera pixels, or one can treat several CCD pixels together and in this way get quadratic 'macro pixels'. We chose the latter solution and treated 2*3 physical CCD pixels as a macro pixel of 24*24 μm^2 area.

Each weight pixel in the second SLM (that is 4*4 real pixels) should thus be projected onto a macro pixel in the CCD camera which gives us a size ratio of 320*320 μm^2 to 24*24 μm^2. Hence, the right lens of figure 19 must reduce the image 13.3 times in size.

The distances between different components follow from the set of equations below. The upper row is the Gaussian lens formula which says that if a lens of focus f is placed at a distance s_o from an object, the image of the object will appear on the distance s_i from the lens. The lower row gives the relation between the degree of magnification M_T and the distances s_o and s_i.

$$\begin{cases} \frac{1}{s_o} + \frac{1}{s_i} = \frac{1}{f} \\ s_i = M_T \cdot s_o \end{cases} \qquad (4)$$

Lens no. 1 has a focus of 300 mm and lens no. 2 of 50 mm. With the magnification degrees we desire we get the following geometric relationships (l_2 is expressed as a function of l_1 and M_T by aid of the lower row, and is inserted into the Gaussian lens formula, and so on).

$$l_1 = \frac{5,36}{4,36} * f_1 = \frac{5,36}{4,36} * 300 = 369 mm$$
$$l_2 = 4,36 * l_1 = 1609\ mm$$
$$l_3 = 14,3 * f_2 = 14,3 * 50 = 715\ mm$$
$$l_4 = \frac{14,3}{13,3} * f_2 = \frac{14,3}{13,3} * 50 = 53,75\ mm$$

5.3 Time multiplexing

The local receptive field of each S-neuron is at the most 6*6 pixels large. Hence we get up to 36 neurons per group which are connected to one input neuron (see appendix 1), and from this there are thus at most n*36 synapses, where n is the number of groups in the S-layer under study. If every input pixel could be projected onto n*36 grayscale pixels simultaneously, all multiplications could be carried out in one single step. One way of doing this is illustrated in figure 20. Here we have used the excessive

resolution of the first SLM to produce one input image for each S-group. The weights of each S-group are then gathered in a corresponding region of the second SLM.

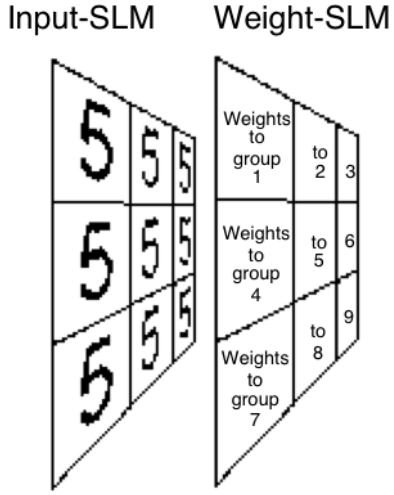


Figure 20. One way of simultaneously carrying out all multiplications optically.

An input image resolution of 27*27 pixels means, with our nine S_1-groups and grayscale simulation taking up 4*4 binary pixels per grayscale pixel, that we need $(27*3*6*4)^2$ (number of input pixels, number of S-groups, number of S-neurons connected to the corresponding input neuron, number of grayscale pixels per side) equal to 1944*1944 pixels in the SLM on which we should produce the weight mask! It is evident that, with our maximum resolution of 320*320 pixels, we cannot realize such a setup. Instead we have to exploit the dynamics of the modulators and use time multiplexing, i.e. during the computation of one layer we update the SLM several times. There are several ways of realizing this. Our choice is described in appendix 1.

6. Results

As a full-scale realization of our pattern recognition system is too large a project to fit within the framework of a diploma project, we chose to simulate the whole system in Matlab. In addition we made measurements verifying the function of two crucial parts of the optical setup. We tested how well our way of simulating grayscale worked and we tried projecting the patterns of the two modulators onto one another and onto the CCD camera.

6.1 The projection system

Basically the setup we constructed is the same as that shown in figure 19, but for the sake of simplicity we sometimes chose solutions which would not work in a real system. For instance, in order to get sufficient light intensity, we used a red He-Ne laser with wavelength 632.8 nm as primary light source. This has some distinct drawbacks in our amplitude modulating setup since the coherence of the laser light gives rise to a number of undesired interference phenomena.

In order to realize our optical system, which in its entire length would not fit on our optical bench, we were forced to use two mirrors. Since dust, fat and dirt on the optical components of the setup have a large impact on the performance of the system, the number of components should be kept at a minimum. In a final version of our system the mirrors should therefore be removed.

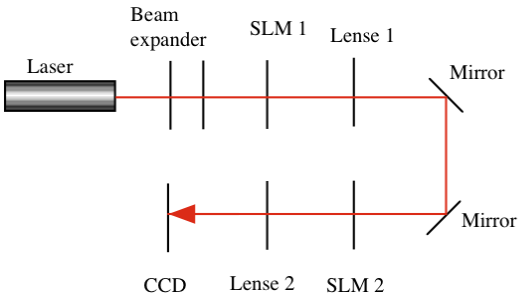


Figure 21. In testing the projection technique we built a modified version of the setup in figure 19.

The lenses of the beam expander were not of the best quality. This gave a distinct effect as aberrations on the final image recorded by the CCD camera.

It turned out to be difficult to achieve just the right degree of magnification between the different steps. All components have a certain thickness which complicates the exact distance measurements needed to realize the theoretically constructed system. For our tests we settled with magnifying powers close to the desired.

In our test of the projection system we showed two different images on the modulators. The result is shown in figure 22. Apart from the image of the two superposed SLM patterns, we also show the image recorded by the camera when one, or both, of the modulators were cleared, as well as when one or both modulators were removed from the setup. These images are added to clearly demonstrate the contributions of each SLM in its different states.

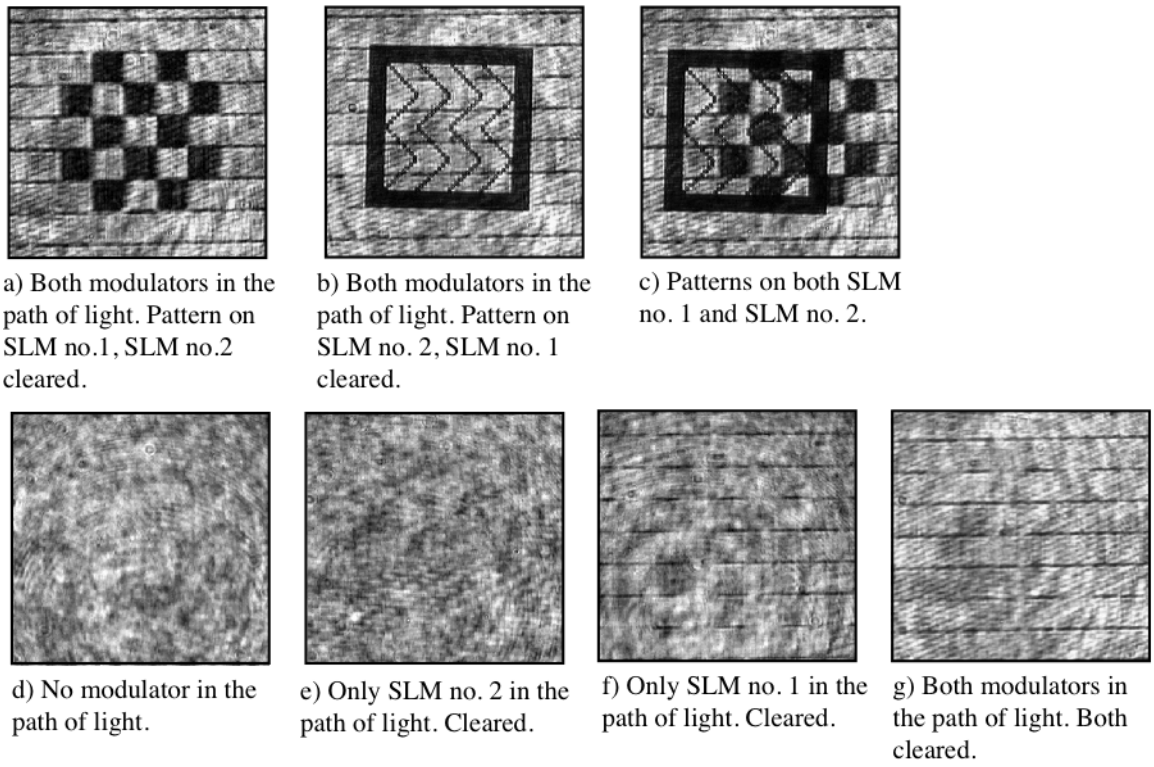


*Figure 22. The upper row shows the superposition of the two modulator patterns. In the first picture the 320*320 modulator is cleared and a pattern is set on the 120*120 modulator. In the next the latter is cleared and the former is set to show a different pattern. The upper right picture shows the image obtained when both SLMs are set with patterns. The ratio in pixel size between the images of the two modulators is approximately the one desired for use in the neural network.*

In order to give an idea of the unwanted contributions to the final image from the different components, we show in the lower row the image obtained without modulators, with only one (cleared) modulator, and with both modulators cleared, respectively. In the first picture we have inserted an extra transmission filter since the CCD camera would otherwise get overexposed. A large part of the undesired effects seen are interference effects and would disappear if the light were incoherent.

6.2 Simulation of grayscale

Since our modulators lack grayscale we have to simulate this in some way when implementing the continuousvalued weight mask. We chose the method described in chapter 5.1.

The number of SLM pixels per grayscale pixel sets the number of levels in which the weights are quantized. To find the lowest number of simulation pixels which gives reasonable performance I made simulations with different number of quantization levels, as well as with continuous weight values, and studied the response from the S_2 groups to the same input. The result is presented in figure 23. It turned out that already with 10 quantization levels, that is 3*3 SLM pixels per grayscale pixel, we got the same answer as with continuous weights. At the start of the project the algorithm was however slightly different, and I got deviations at this level. Hence, I have used 4*4 pixels for grayscale simulations in my system.

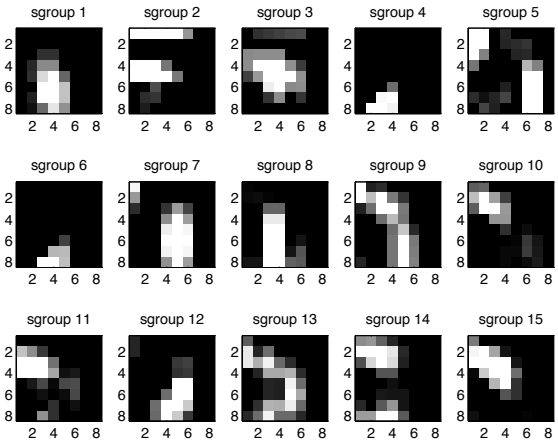


Figure 23a. The output of the S_2-layer when the input is the digit 5 in the font Monaco. The weight mask has no grayscale, i.e. we have binary weights.

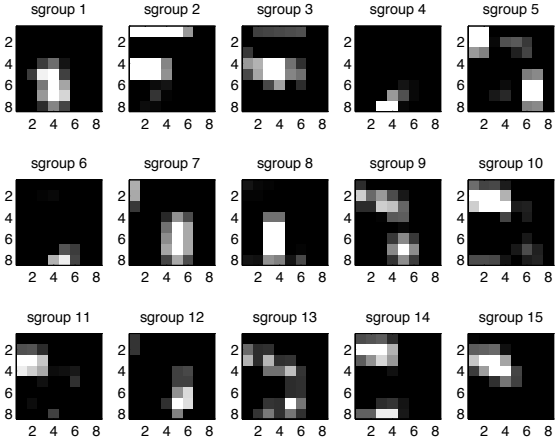


Figure 23b. As in a) but with five level grayscale.

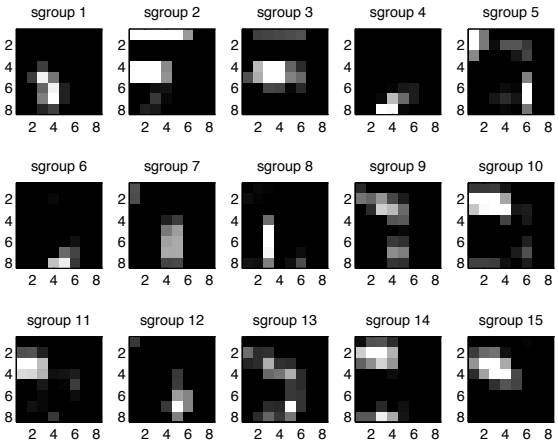


Figure 23c. As in a) but with ten level grayscale.

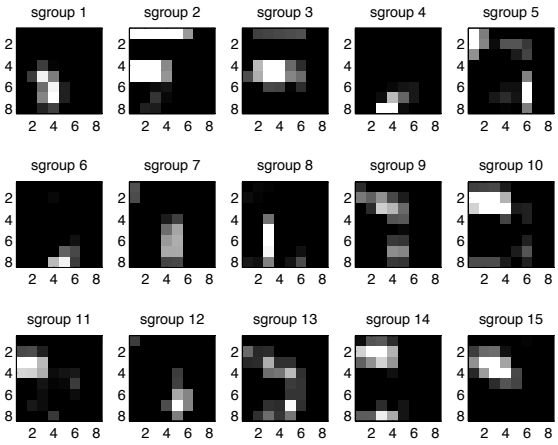


Figure 23d. As in a) but with seventeen level grayscale.

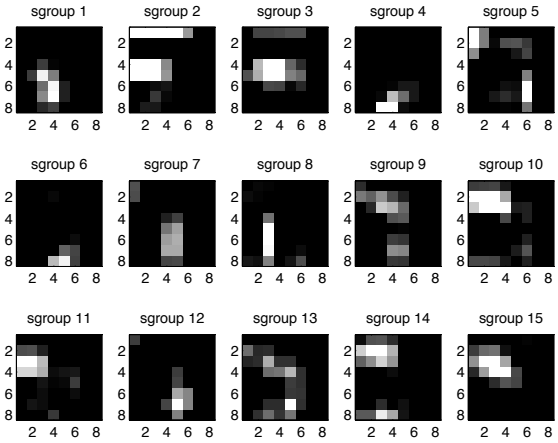


Figure 23e. As in a) but with continuous grayscale.

We thus use 4*4 pixels of SLM no. 2 to represent one weight. These 16 pixels should be projected onto one pixel of the CCD camera[5] and the intensity recorded will attain one of 17 possible levels. If all 16 pixels are black (the weight value is zero) or if the input neuron is silent, no light falls on the camera pixel. For each pixel of SLM no. 2 which is switched to the transparent state, the recorded intensity should increase with a fixed value. Finally, when all pixels are transparent, the maximum value should be attained.

We tested this technique by irradiating a part of the modulator (active area: 25*25 mm) with a 632.8 nm HeNe-laser beam of

[5]In reality we project it upon one 'macro-pixel', consisting of 2*3 physical CCD-pixels. This is due to the rectangular (non-square) pixel shape of the camera. See section 5.2.

Gaussian cross section and a diameter of approximately 3 mm. We wanted to assure ourselves that different combinations with the same number of transparent pixels give the same intensity, as well as that the intensity difference when switching one pixel really is constant, no matter in what part of the grayscale we are.

In the first experiment we let the whole SLM, except for 4*4 illuminated pixels in the center, be constantly non-transparent. We made 17 measurements with an increasing number of transparent center pixels. In the second experiment we repeated the same grayscale pattern over the whole active area of the SLM

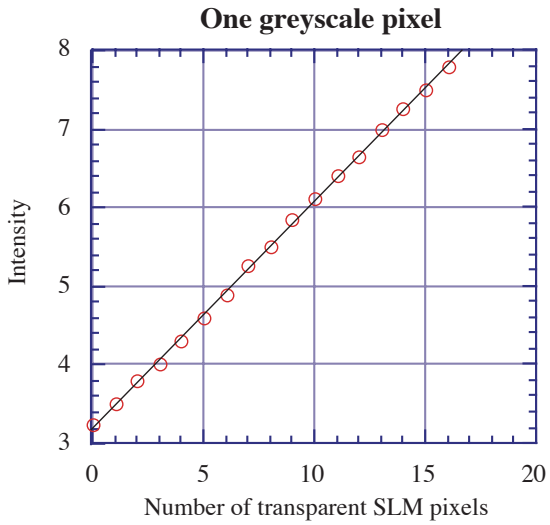


Figure 24a. The intensity variation as a function of the number of transparent SLM pixels in one grayscale pixel.

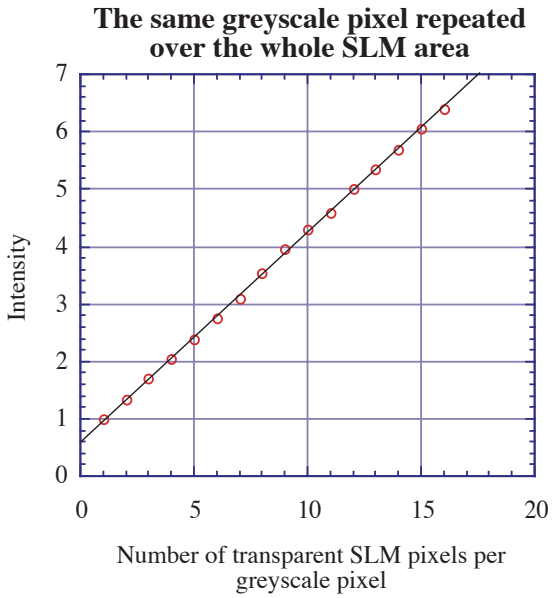


Figure 24b. As above, but here the grayscale pattern is repeated over the whole active area of the SLM.

As can be seen in the graphs the grayscale simulation seems to be working well. The relationship between the number of transparent pixels and the light intensity is practically linear.

For each graylevel (that is, a certain number of transparent pixels) we also tried several different configurations of the sixteen pixels of the grayscale pattern. The intensity turned out to be configuration independent, just as we desired.

6.3 The neural network algorithm

The performance of the neural network algorithm is evaluated from several different points of view. The response of the output layer is in all cases used as the measure. This is strongly dependent of the number of images used to train the layer. Therefore we have compared three different weight setups to the output layer. The first is obtained by training on one font only (Courier), the second by training on three fonts (of which one is italic) and

the third is a result of training with five different fonts (one italic).

In the training of the system I started by evaluating all calculations which are required several times (for instance the output of previously trained layers for each of the input patterns) and the result was saved in a file. When a result was needed during the training it could be loaded from disk instead of reevaluated, thus saving lots of time.

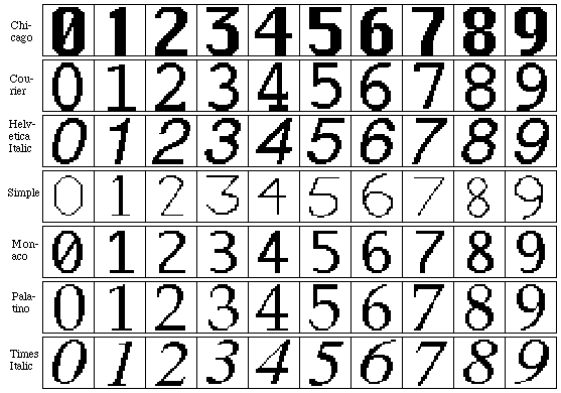


Figure 25. The seven different fonts used for the evaluation of the identification system.

For each output layer weight setup we analyzed the performance of the system with regard to the following abilities:

- How well does the system generalize? Can it identify a digit independent of how it is drawn? Are some digits harder than others to identify?
- How sensitive is the system to displacements of the digit to be identified?
- How does scaling of the digit affect the system performance?
- How sensitive is the system to disturbances like noise or erased pixels?

6.3.1. Generalization

Even though the structure with feature detectors gives good possibilities to find the essential features of each digit and generalize between different fonts, it is quite obvious that the number of training fonts is crucial for the performance of the output layer. Figures 26-28 show the results when we train with one, three and five fonts, respectively. Figure 26 shows how the total error, *i.e.* the sum of the errors from all neurons of the output layer, is changed during the training process. Figures 27 to 29 show the response from the 10 output layer neurons when digits written with the seven different test fonts are presented to the system. The leftmost bar in each histogram shows the outputs of the 0-neuron, the next for the 1-neuron, and so on. The number above each histogram gives the ratio between the highest and second highest output value, and hence it is a measure of how certain the system is in its answer. For the fonts not in the training set of the system, I have given the fraction of correct answers.

Note that in going from figure 27 to figure 29, the number of test fonts is reduced from six, via four, to two. It would of course have been a better comparison if all systems had been tested with the same number of unknown fonts, but as the process of preparing training fonts is quite time-consuming, I had to settle with a compromise in this matter.

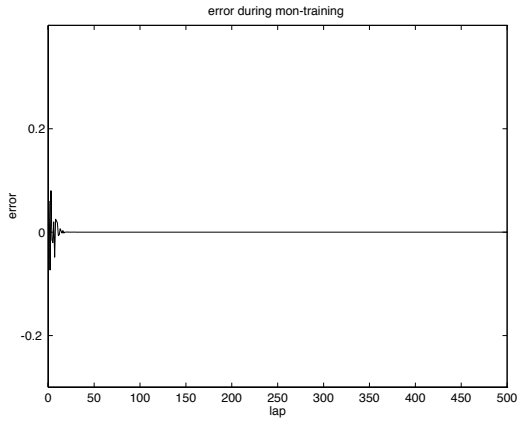


Figure 26a. Error during training on one font (Monaco).

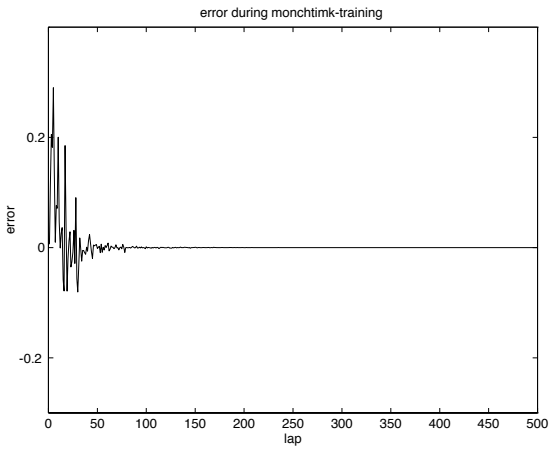


Figure 26b. Error during training on three fonts (Monaco, Chicago, Times italic).

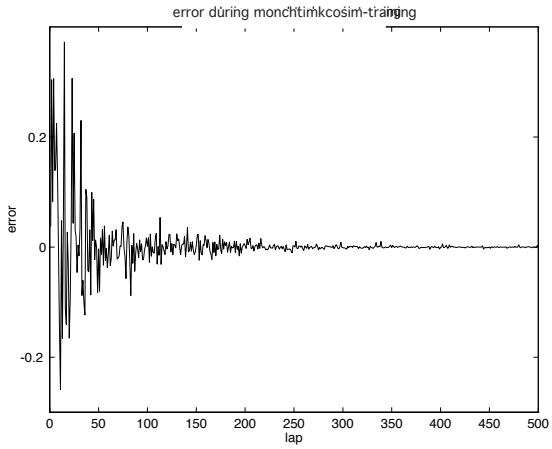


Figure 26c. Error during training on five fonts (Monaco, Chicago, Times italic, Courier, simple handwritten).

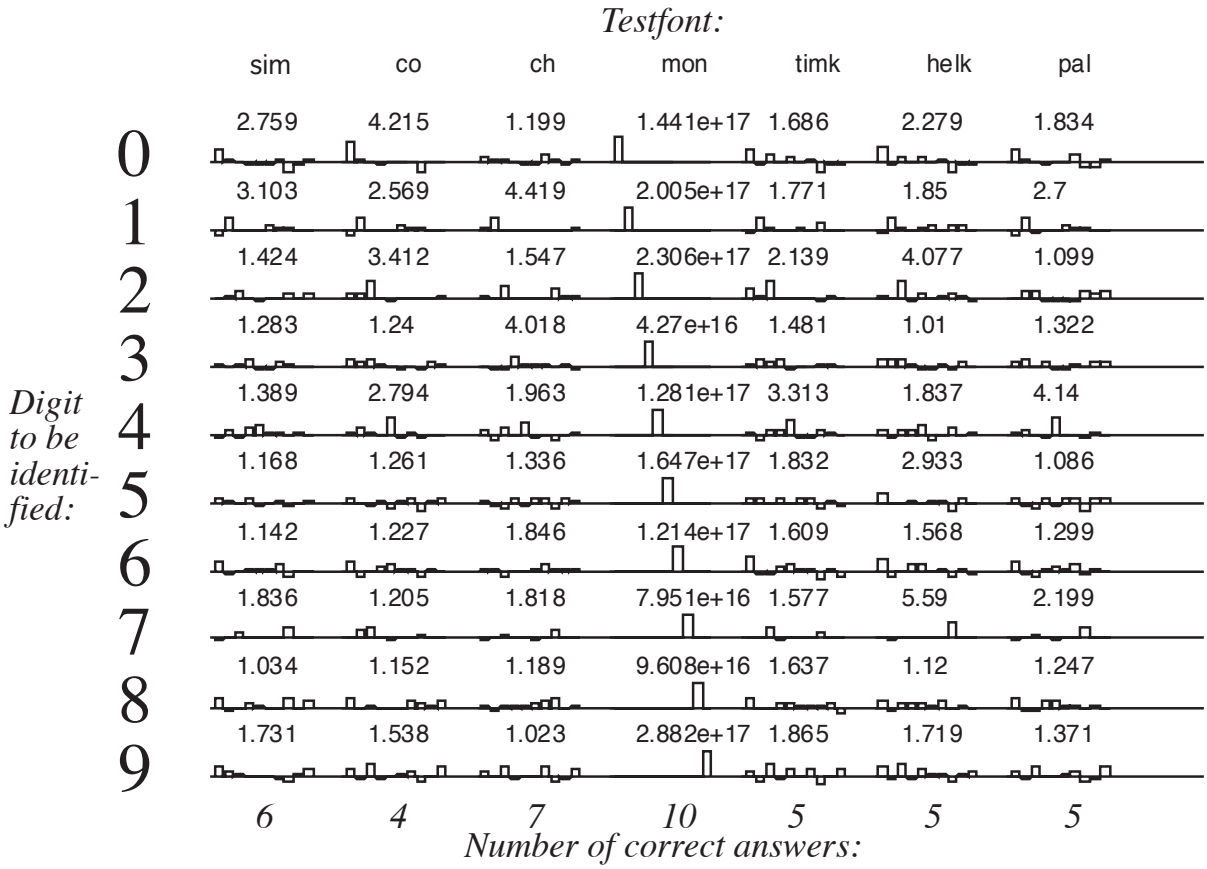


Figure 27. Generalization capability after training the output layer with one font (Monaco). For each combination of font and digit there is a bargraph illustrating the response of the output neurons. The number above each graph is the largest response divided by the second largest response. Note that the font which is always correctly classified (Monaco) is the one used for training. The result on this font does thus not reflect the generalization capability of the system.

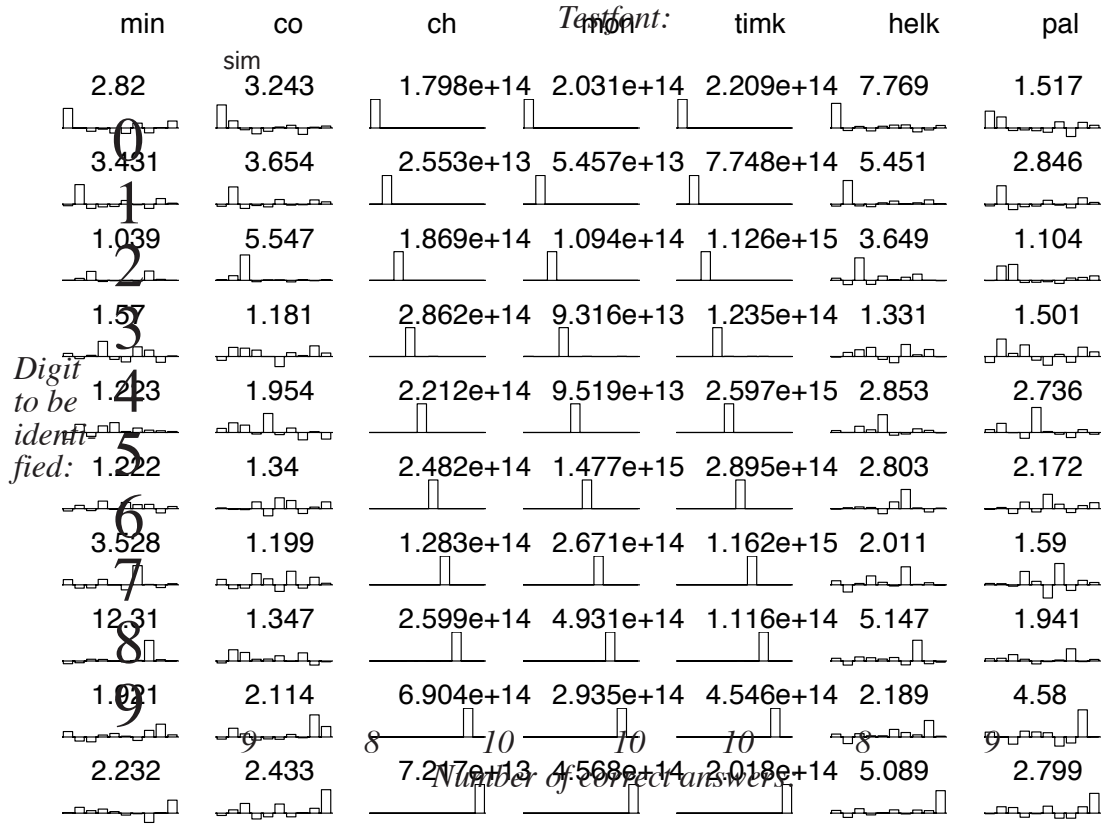


Figure 28. Generalization ability after training the output layer with three fonts (Monaco, Chicago, Times italic). For an explanatory text, see caption to figure 27.

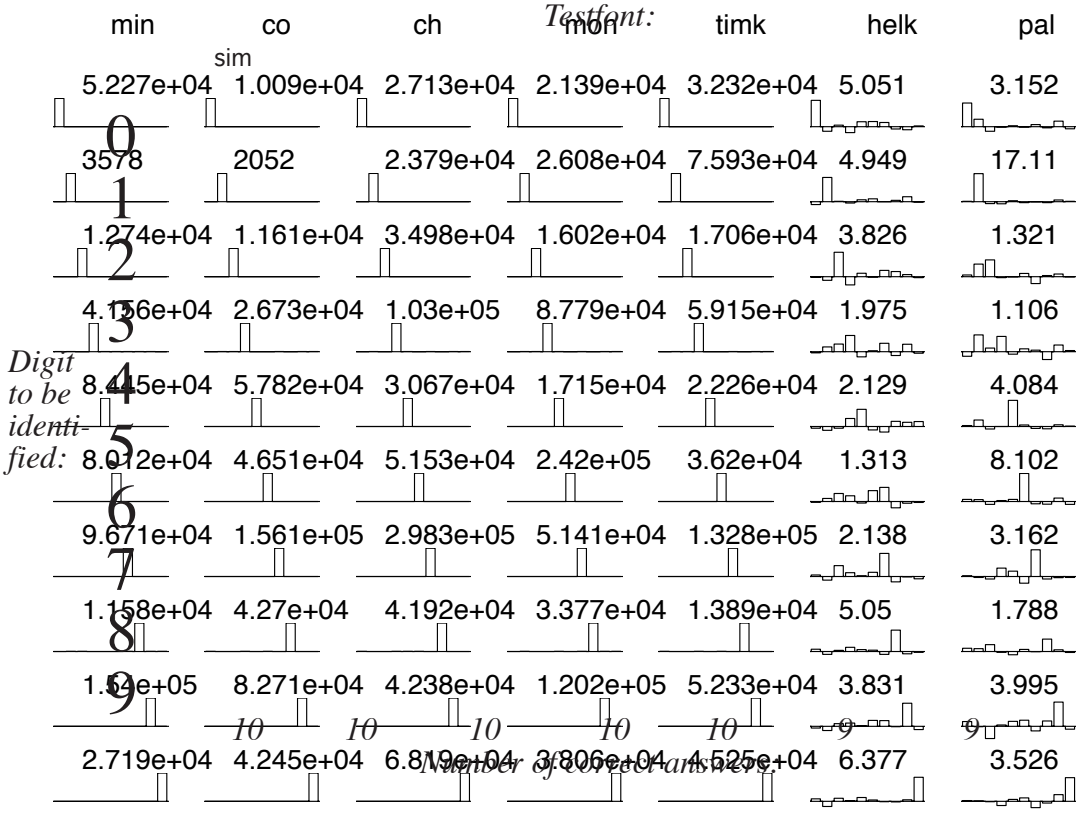


Figure 29. Generalization ability after training the output layer with five fonts (Monaco, Chicago, Times italic, Courier, simple handwritten).

We see from figure 26 that the error very quickly becomes vanishingly small when the number of training fonts is low. The neurons quickly learn to identify the few images used in the training. On the other hand, it is apparent from figures 27 to 29 that the generalization ability is poor after such training. The number of erroneous identifications is much higher in figure 27 than in figures 28 and 29. It is also clear that the system gives more certain and distinct identifications of digits in untrained fonts, as the number of training fonts is raised.

Apart from these general observations some small comments to the figures may be added. First, we see that the digits correctly identified, independent of font, in all three systems are the 1 and 4. A possible explanation is that they both consist almost only of straight lines, and hence suit the feature detectors of the first S-layer well. Against this hypothesis stands the poor performance when the digit 7, also practically free from curves, is given as input. The most evident explanation to this should be the large similarity between the 7 and the 2. When the system fails to identify a 7, the faulty answer is always 2.

It is apparently much more difficult to identify the digits 3 and 5. The many curves of the former is probably the explanation to the poor performance in identifying 3:s. In the case of 5:s this is only partly true. The errors might also to a large extent be due to the similarity of the digit 5 with for instance 3, 6, 8 and 9.

6.3.2. Displacements of the input image

In the remaining evaluation tests we chose one input image (the digit 5 in Monaco) and distorted it in various ways. The general conclusion from these tests is that it is practically only the last system (trained with five different fonts) that has a chance of seeing through the distortions. As is evident from the figures below, the systems trained with fewer training images are extremely sensitive to disturbances of all kinds. The following comments all concern the best system.

First we tried moving the 5 around in the 27*27 pixel input image, and studied how the system reacted. This is in some sense a test of the performance of the C-layer. The result is presented in figures 29-32.

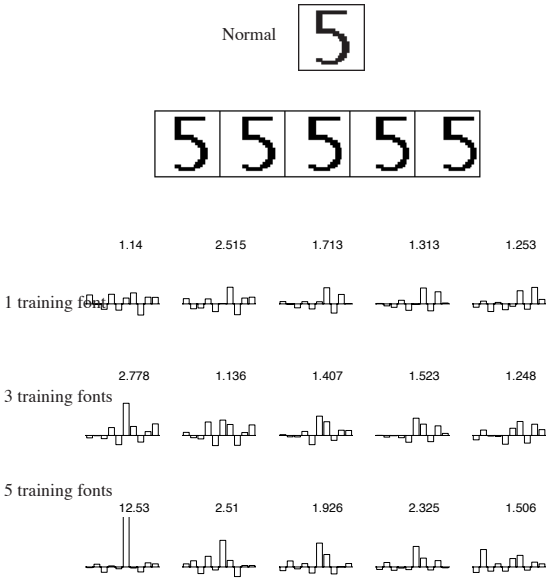


Figure 29. The input digit is shifted stepwise to the right.

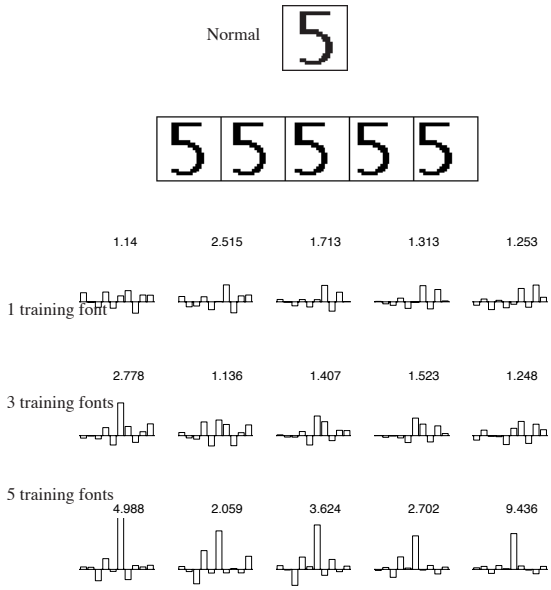


Figure 30. The input digit is shifted stepwise to the left.

Apparently the system is much more sensitive to displacements to the right than to the left. Perhaps this changes if a different digit is chosen as test object.

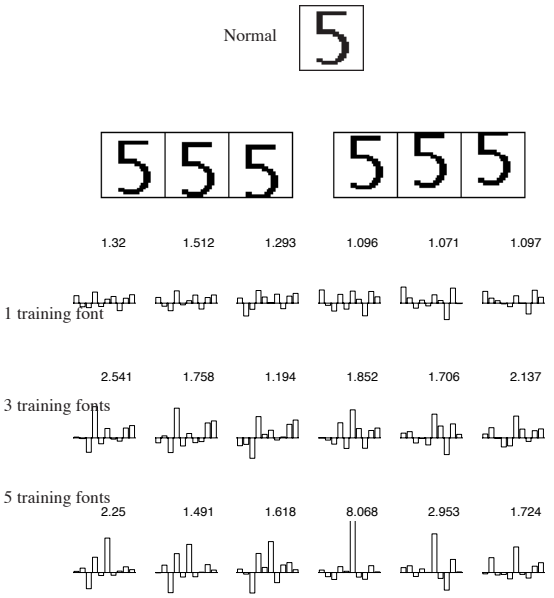


Figure 31. The input digit is shifted stepwise downwards and upwards.

The system seems to be more sensitive to displacements downwards than upwards. Even if the system answer is correct in all cases, in particular the response of the 3-neuron becomes too strong at displacements downwards.

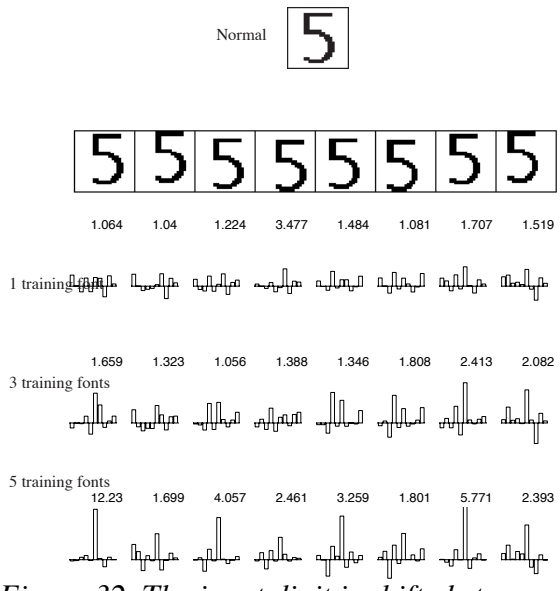


Figure 32. The input digit is shifted stepwise both laterally and vertically.

In diagonal shifts the system initially does quite well for all displacement directions. Already at a shift of two pixels, however, the system becomes unreliable.

6.3.3. Scaling

I tried scaling the 5 in five steps. The system was not markedly disturbed by any of these changes, cf. figures 33 and 34.

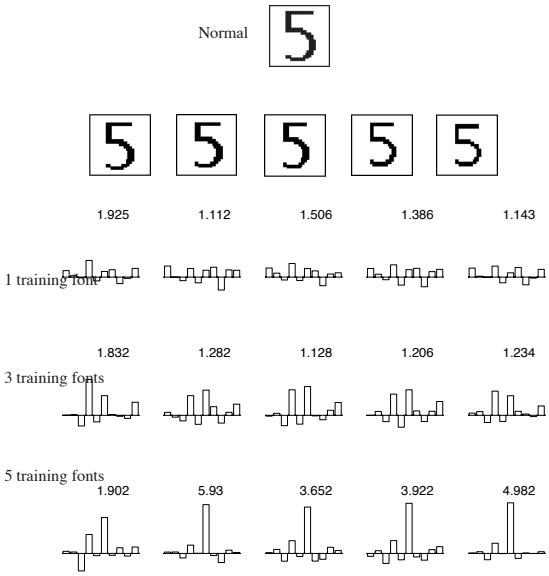


Figure 33. The sensitivity to size reduction of the input digit.

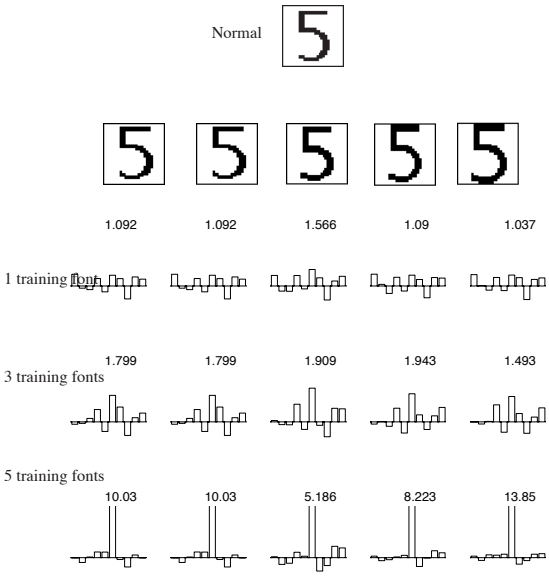


Figure 34. The sensitivity to magnification of the input digits.

6.3.4. Noise

A most interesting test series was the one where we disturbed the input with noise of various kind. I tried adding an increasing amount of random black pixels in the image, I removed an increasing amount of pixels in randomly selected parts of the digit, and finally I tried removing parts of approximately equal size but in different places. The result is shown in the following three figures.

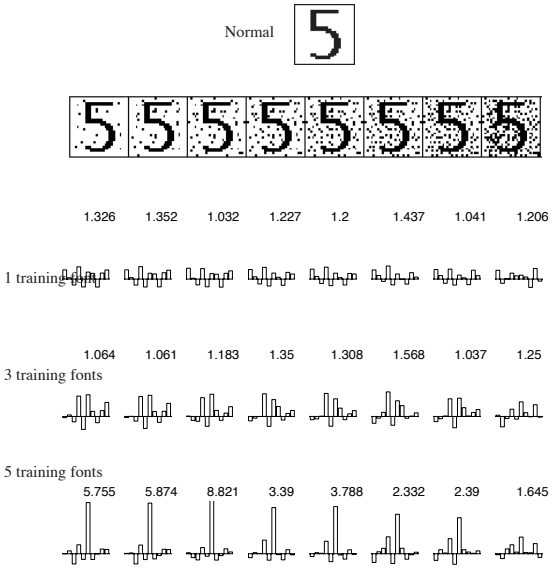


Figure 35. The sensitivity to noise in the input.

The system handles noisy input surprisingly well. The lower system even gives a correct response for the noisiest picture. A curiosity is that the confidence of the system is largest for the third image. With even more noise, however, the confidence decreases drastically.

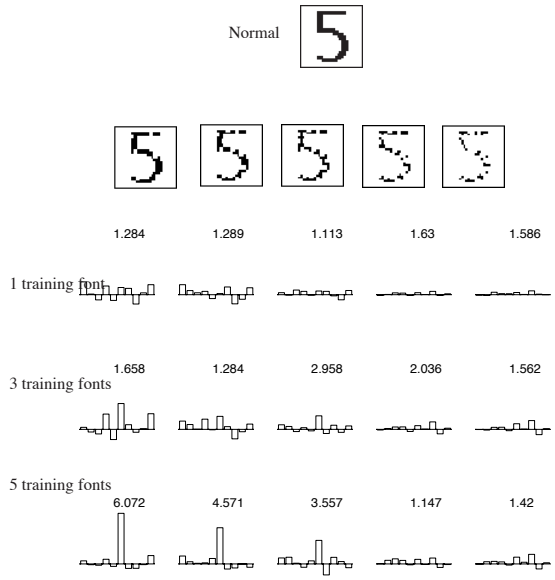


Figure 36. Increasing number of erased pixels in the input digit.

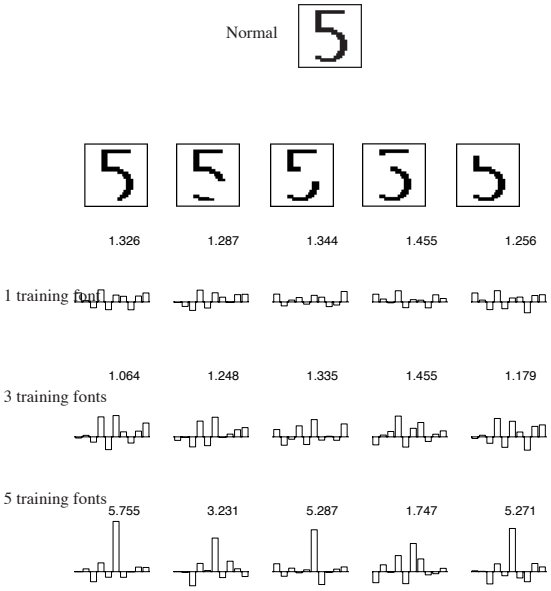


Figure 37. One part of the input digit is missing. In the different images different parts (of approximately the same size) have been removed.

Figure 36 offers no surprises. The confidence decreases continually as we remove more of the digit. The results from the next test are more interesting. We have removed different parts, of approximately the same size, from the digit, and we see that the system reacts very similarly for all images but one. In the fourth image the system hesitates strongly between a 3 and

a 5. If we look at the images the explanation to this is apparent. The part that has been removed from this picture is vital for an unambiguous interpretation of the image. Look at figure 38. If instead of the removed vertical line we insert a diagonal one we get a perfect 3. Hence, we can no longer say whether the digit is a 3 or a 5, and the system responds correctly in hesitating between the two answers.

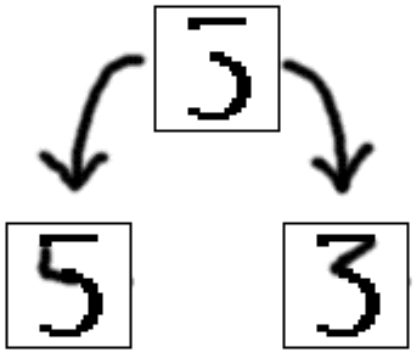


Figure 38. If a certain part of the 5 is removed it is no longer possible to say which digit is shown.

6.3.5. False inputs

As a final test we gave the system letters as input. A better version of the output layer would have 11 neurons, with the eleventh representing non-digit inputs. Our system does not have this possibility, and hence it answers with the digit most resembling the input.

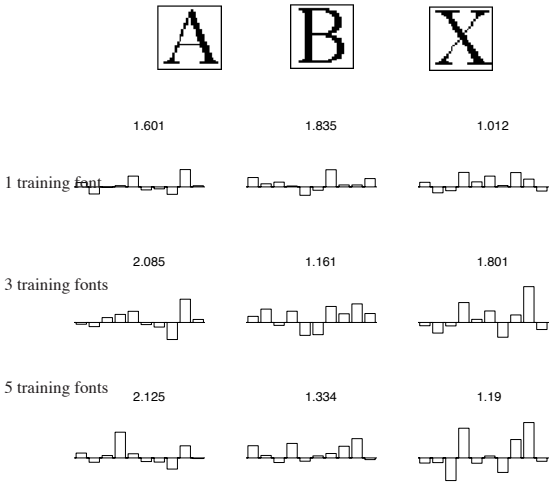


Figure 39. The reaction on letters as input.

The reaction on the letter A is, in my opinion, rather odd, but the response on the other letters is quite understandable. The strong response from the 0-, 3- and 8-neurons on the letter B is not surprising since all these letters can, without too large modifications, be constructed out of the letter. When X is given as input it is the 3-, 7- and 8-neurons which react strongly, which is also quite reasonable.

6.4 Optical implementation versus numerical

Since I have not been able to implement the system optically it is difficult to give a good picture of the efficiency of the system. The simulated system, which simulates the optically implemented system, is most probably a lot slower than the real optical-numerical system, as well as an entirely numerical system, would be. It took approximately 3 hours on a very powerful computer (twin 150 Mhz Hyper-Sparc processors, 224 Mbytes RAM) to perform the training from scratch, and the identification of a digit took more than a minute on the same machine. These times would probably be much shorter had the program been written in a more efficient language than Matlab.

In order to get a picture of the extent to which calculations would be performed optically in a real system, we have to look at the algorithm for calculation of the S-neuron responses (appendix 3). Let's study how the responses from the 3872 neurons of the first S-layer are evaluated.

First the responses of the 484 neurons of the first I-layer have to be calculated. This is done according to equation (5) in appendix 3. Since each I-neuron is connected to 36 input neurons, the evaluation of the formula requires 17424 multiplications and as many summations in total. All these multiplications are performed optically in my system, while the summations are done numerically.

Furthermore, one square root per I-neuron is performed numerically.

For the evaluation of the S_1-layer response, we use equation (4). The number of optically performable multiplications per S-group is the same as for the I-group, which means we get 139 392 optical multiplications in total. The computer performs, apart from the summing of the corresponding products, a small number of summations, multiplications and divisions, as well as one max function per S-neuron.

The result is summarized in table 2.

Table 2. Number of operations of different kind during the evaluation of the responses from the S_1-layer neurons.

	Optical	***Non-optical***
Multiplications:	156 816	15 488
Additions:		168 432
Divisions:		3872
Square root extractions:		484
Search for maxima:		3872

It is apparent from this little study that there a significant number of operations remain to be done in the computer. Hopefully this number can be reduced somewhat. The max function could, for example, be performed by a hardware treating negative numbers as zero, and it is possible that equation (5) in appendix 3 could be modified so that the square root may be avoided.

The profit of the optical system is that the multiplications are performed in parallel outside the computer. The width of this profit depends partly on the system dimensions, and partly on the computer processor. In a modern mathematics processor a multiplication is performed almost as fast as an addition, while in, for instance, an 8-bit Motorola processor it takes 32 clock cycles compared to the 2 needed for an addition. Even in a fast processor, however, the multiplications are performed in series, and since the number of multiplications becomes very large as the dimensions of the system increase, the time gain of an optical implementation might be considerable. A dimensional increase of the system must, however, be accommodated by an increase in resolution of the SLMs. Otherwise we must use time multiplexing and the time gain would decrease.

7. Discussion for further work

THE OBJECTIVE of this work has been to develop a system showing new ways of implementing neural networks for pattern recognition. The ambition has not been to construct a system with a particularly high efficiency. Nevertheless, I think that some valuable conclusions can be drawn from the results.

The equipment I've been using as well as the algorithm I developed, suffers from a number of important shortcomings, and in case of a continuation the system must thus be improved in various ways. This is, however, by no means an impossible task. The SLMs based on ferroelectric liquid crystals, which today are state of the art and commercially available, feature much higher resolution and are much faster than the components I've been using, and even some which also fulfill my demands concerning grayscale are available. Then we have good possibilities of constructing a fast and well performing system.

In this chapter I give some ideas on how one could further develop the system and in this way get a substantial improvement in performance. These developments concern the software as well as the hardware.

7.1 Improvements on the algorithm

Both in the design and in the training of the system a large number of parameters figure. Many of these I have given values more or less randomly since the time scope of the project didn't allow a thorough investigation of their influence. In a continuation it might be wise to make a deeper analysis of them and to try a larger number of different solutions.

The dimensions of my network have been chosen to fit the optical equipment I had access to. It would of course be very interesting to study the performance of a system of different dimensions. A higher resolution would for instance allow us to use a larger local receptive field. Then we could easily define a greater number of interesting features, like for instance curves and angles. These tests can be easily performed with the Matlab simulation I have developed.

7.2 Better Spatial Light Modulators

Doing an optical implementation is not really worthwhile until we have continuous instead of binary inputs to each layer. First then can we talk about performing multiplications optically. With the solution proposed in this work it is more or less an AND-operation between a binary and a continuous matrix that is performed optically. To improve the situation we need either an SLM with grayscale or grayscale simulation using time multiplexing. The latter solution is in principle available with the present equipment, but the low speed of the

modulators would render the system practically useless.

The point in realizing a neural network optically in the way I have proposed is that a large number of multiplications can be performed in parallel and at the speed of light. The Achilles heel of the system is the connections between the computer and the SLMs and the camera. They must be made as fast as possible and used as little as possible. The spatial resolution of the SLMs must therefore be so high that we can perform *all* multiplications for a complex simultaneously, instead of dividing them in a multiplexing scheme as I have been forced to do.

If the pixel number of the modulators is drastically increased we can also afford to increase the resolution of the system, *i.e.* use a larger matrix to describe the images. This will of course improve the possibilities of obtaining a well functioning system.

Whereas the two SLMs at my disposal for this work had 128*128 and 320*320 binary elements respectively, and a typical frame rate of 6 Hz, modern FLC-SLMs are approaching 1000*1000 in resolution and have frame rates lying between 1 kHz and 10 kHz. An FLC-SLM with 256*256 pixels which allow a continuous (analog) gray scale is also commercially available (Boulder Non-Linear Systems, USA).

7.3 A new illumination technique

It is very important that the illumination of the input SLM is uniform. Using a simple beam expander this is hard to achieve, and it is therefore desirable to find another illumination method. An exciting alternative would be to use a so called *kinoform* to obtain a suitable illumination of the modulator. With the aid of the kinoform we can transform an incoming coherent beam into a ray matrix of desired dimensions, see figure 40 [10].

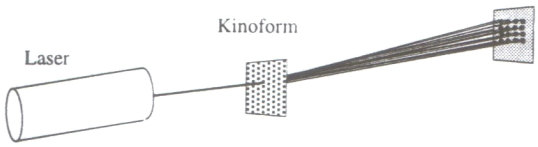


Figure 40. With the aid of a kinoform we can divide an incoming beam into a matrix of rays.

Each ray has in itself a Gaussian cross section and can hence not be used to illuminate several pixels, since this would lead to an uneven illumination. The point is instead to create, with the aid of the kinoform, a ray matrix where the ray cross section is much less than the smallest pixel side of the system, and the distance between two rays is the same size as the pixel side. In this way each pixel of the weight SLM, which has the smallest pixels, is illuminated by exactly one ray. Each input pixel is illuminated by several rays since one input pixel should be multiplied with several weight pixels, see figure 41!

By placing a lens after the kinoform we can collimate the ray bundle so that the rays propagate parallel to each other. If we have a tailor-made system where ray matrix dimensions, pixel number and pixel size are adjusted to each other, we no longer need any projecting lenses! The one dimensional system of figure 41 is an example of this. We see, however, that the diameter of the kinoform rays varies in a Gaussian fashion. This means that the ray matrix is well defined only within a limited space. Both modulators, as well as the CCD-camera, must therefore be fit into this space.

Since the ray must be significantly smaller than a weight pixel (with grayscale) in order not to get cross talk between pixels, we can no longer use spatial gray scale simulations. That method is based on the uniform illumination of several pixels, and is thus ruled out. Therefore we need to use sequential grayscale simulation or get an SLM with real analog grayscale. As the

speed of FLC modulators can be very high, even the former alternative may not be so bad at all. The bistability of the SLM means that only the pixels that should change state need to be readdressed. Using a smart addressing algorithm, possibly inspired from modern techniques for video compression (MPEG, Quicktime, etc.) where only the parts of the picture changing between frames are treated, the time needed to prepare the SLM for the next exposure can thus be further reduced.

If in addition we want continuous valued input, which of course is desirable, the input SLM must be able to show grayscale. For this purpose a nematic SLM could also be suitable. Its drawbacks, mainly the larger pixel size and slower updating, lead to no major consequences since the pixels of the input modulator are large and it is not readdressed as often as the weight SLM.

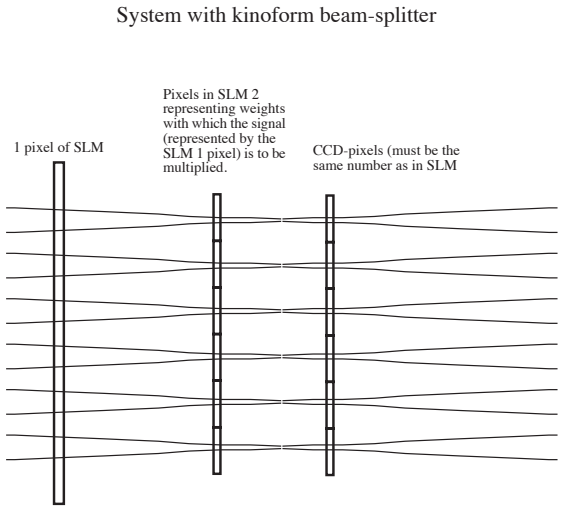


Figure 41. Optical system without projection lenses. With the aid of a kinoform and a lens we have created a matrix of parallel propagating rays. The different components must be adapted to each other concerning dimensions. Each component must also be so thin that the whole sandwich is contained within the depth of field of the Gaussian rays.

7.4 Optical implementation of the output layer

Since the output layer is not of Neocognitron type a different technique must be used to implement it optically. We now run into the difficulty of implementing negative weights with an optical system. On the other hand, both inputs and outputs to this layer are vectors which gives us new possibilities of performing both multiplications and summations optically in a simple way.

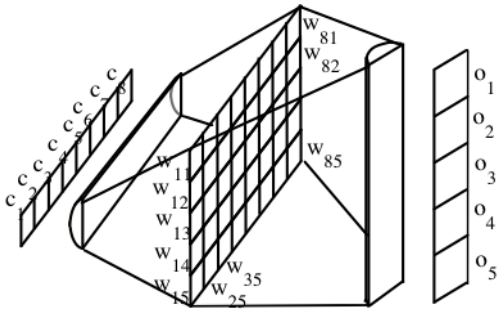


Figure 42. An optical vector-matrix multiplicator. For the sake of clarity in the drawing the dimensions of in- and output spaces in the figure are much smaller (simplified to a small number of pixels) than in our system.

Perhaps the most attractive solution is to use the setup of figure 42. This is called an optical vector-matrix multiplicator and appears frequently in the literature [2,3,5, 8]. The vector in our case consists of the output signals from the C_2 layer, while the elements of the matrix are the weights between the C_2 layer and the output layer. For the sake of clarity the dimensions of the system in the figure are much smaller than in our system.

The input, *i.e.* the C_2 vector, is presented for instance by a one-dimensional horizontal SLM or a LED array, and its image is projected through a cylindrical lens onto a two-dimensional SLM such that each input pixel illuminates one column of the weight-SLM. The two-dimensional weight-SLM must have a

resolution of at least 135*10 grayscale pixels (column *i* in the matrix contains the weights from the C_2 neuron *i* to the 10 output neurons, while row *j* holds the weights from the 135 C_2 neurons to output neuron *j*).

On the other side of the weight matrix we place another cylindrical lens, turned 90 degrees compared to the first. This collects the light from one whole row in the weight matrix to one pixel of the vertical detector furthest to the right in the figure. The light intensity falling on pixel *j* becomes:

$$I_j = \sum_{i=1}^{135} c_i w_{ij}$$

where c_i are the 135 C_2 signals and w_{ij} are the weights between them and the output neuron *j*. The output from detector pixel *j* thus corresponds exactly to the output of output neuron *j*, provided that $w_{ij}>0$. Since we have negative weights between the C_2- and the output layer we must divide the evaluation in two steps. In step 1 we address the SLM with all positive weights while the negative ones are set totally black. When the detector signal is recorded and the result stored away, we instead set all positive weights black while the other pixels are set according to the magnitude of the negative weights. The result from this step is subtracted from the previously stored data.

Appendix 1
Time multiplexing

DUE TO THE LIMITED resolution of the spatial light modulators we cannot simultaneously perform all multiplications needed for the evaluation of one neuron layer. Instead we have to divide the task into several steps. There are several ways of doing this and I will now give an account for the method used in the present system.

1	2	3	4	5	6	6	6	6	6	6	6
2	4	6	8	10	12	12	12	12	12	12	12
3	6	9	12	15	18	18	18	18	18	18	18
4	8	12	16	20	24	24	24	24	24	24	24
5	10	15	20	25	30	30	30	30	30	30	30
6	12	18	24	30	36	36	36	36	36	36	36
6	12	18	24	30	36	36	36	36	36	36	36
6	12	18	24	30	36	36	36	36	36	36	36
6	12	18	24	30	36	36	36	36	36	36	36
6	12	18	24	30	36	36	36	36	36	36	36
6	12	18	24	30	36	36	36	36	36	36	36

*Figure A1:1. The upper left-hand corner of the input image. The square drawn with thick lines shows the size of the receptive field. The number in a pixel indicates the number of synapses connected to the neuron represented by the pixel. These values are a consequence of the way the receptive fields of neighboring neurons overlap, and the size of the fields (6*6 pixels). See figure 5b.*

We chose to perform all multiplications of one group in the S-layer before starting with the next. In this way we minimize the number of readdressing of the input SLM. During the evaluation of the S_1-layer we need no readdressing, while on the other hand, during the evaluation of the S_2-layer we have to readdress the input-SLM once for each group in the C_1-layer, since all of the groups constitute the input for the S_2 neurons.

The image constituting the input to the first S-layer is set on the first SLM. Since each input neuron is connected to between 1 and 36 S-neurons (the number decreases towards the edges of the image, see figure A1:1) every pixel in this SLM must be projected onto a 6*6 pixel area of the weight-SLM. With an input image resolution of 27*27 pixels and a grayscale simulation using 4*4 pixels for each S-neuron we would hence need a weight-SLM with 27*6*4=648 pixels per side. As the SLM used in the present system has a resolution of only 320*320 pixels we have to perform a time multiplexing also for each S-group.

We do this by dividing the 6*6 pixel receptive field into 4 parts of 3*3 pixels and addressing the whole weight-SLM with one such part at a time.[6] The procedure is illustrated in figure A1:2. We must thus readdress the weight-SLM four times per S-group during the evaluation of the S_1-layer. The input-SLM, on the other hand,

[6]In fact, this also puts too high demands on our SLM since we would need a resolution of 324*324 pixels. We do a little bit of cheating, however, and use only 3*3 SLM pixels for the grayscale pixels in two rows along the edges of the image.

needs to be addressed only once, so all in all the evaluation of the S_1-layer comprises 4*9=36 readdressing of SLM no. 2.

When evaluating the S-layer of the second complex the four groups of the C_1-layer constitute the input, and hence we have to display these, one after the other, on the first modulator. For each C_1-group we readdress the weight-SLM once for every group in the S_2-layer since the weights to different groups are not the same. Multiplexing of the synapses to one group is, however, not needed this time since the C_1-groups only have 11*11 pixels. With a receptive field of 4*4 pixels and grayscale simulation using 4*4 pixels we need (11*4*4)* (11*4*4)=176*176 pixels in the weight-SLM, which is well within the capabilities of the modulator at our disposal.

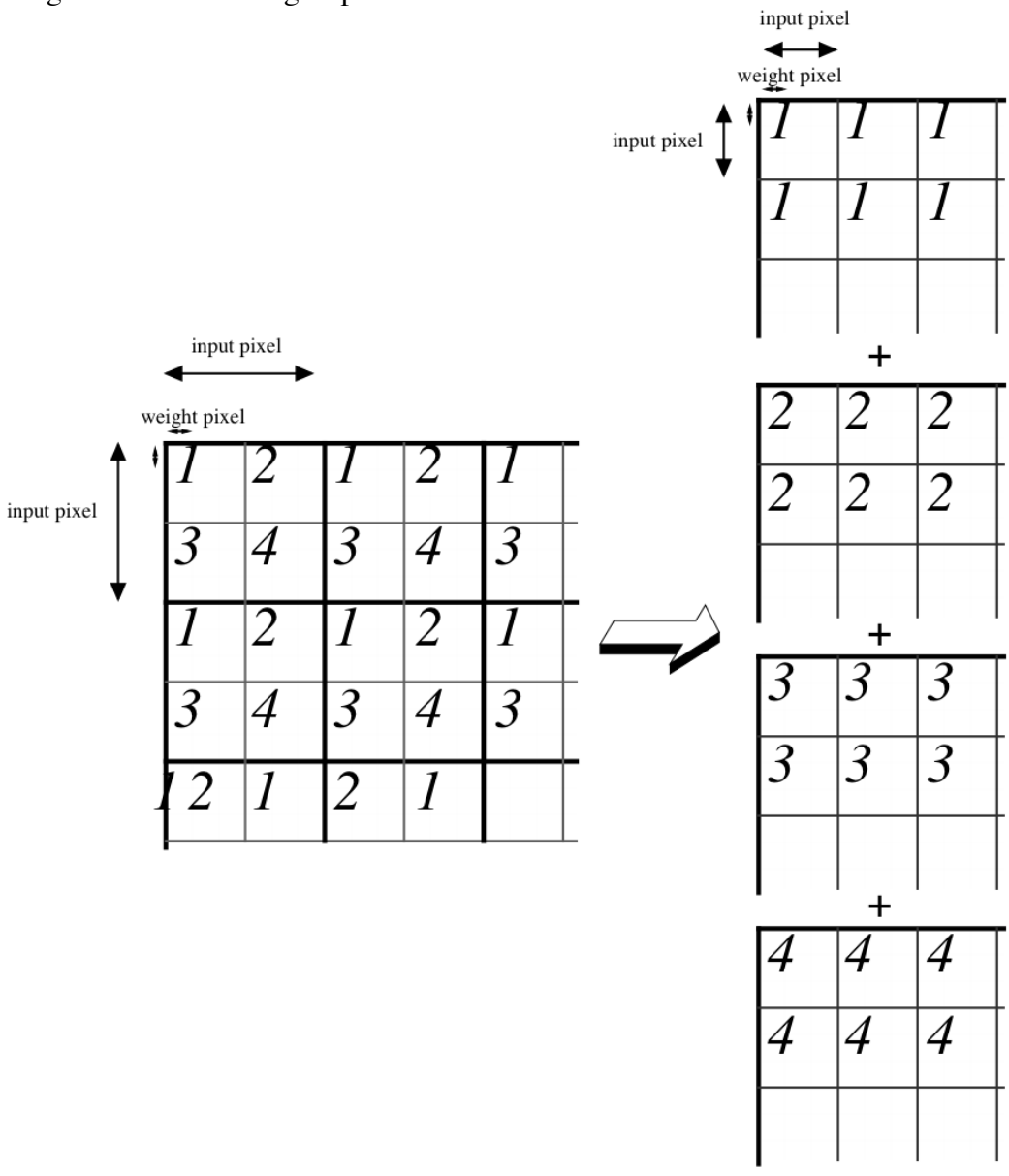


Figure A1:2. Due to the limited resolution of our weight-SLM we must divide the evaluation of each group in the S_1-layer into four timesteps. In each step we display 9 of the 36 weights on the weight-SLM. The partial sums are temporarily stored between the "snap-shots" and are added together at the end to yield the final result.

Appendix 2
The training of the S_2-layer

THE SECOND S-LAYER is divided into 15 groups. As in the first S-layer, the neurons of a group have the same synaptic weight setup, but while the input to the S_1-layer is the system input (i.e. the image of the digit to be identified), the input to this layer consists of the C_1-layer. Since this is divided into four groups, each signaling for one feature in the input image, the features to be recognized by the S_2-neurons are combinations of features extracted by the first complex.

The goal of the training of the S_2-layer is to force each of its different groups to specialize in one such combination. It is, however, no simple task to come up with 15 suitable combinations, and therefore we want the groups to search for them on their own. In this appendix I give a detailed review of how this self-organization is performed.

A2.1 The groups compete against each other

Our version of self-organization is based on the one used by Fukushima in his first Neocognitron version [9]. At the start the S_2-layer is given the weight setup described in section 4.5, that is each group is slightly inclined towards a combination of 'clean' C_1-features (see figure 15). During the training of the groups of the S_2-layer, images of digits, one at a time, are presented to the system. The first complex, for which the training is completed, performs its preprocessing and the outputs of the C_1-neurons are given as input to the S_2-layer. These neurons evaluate a response based on the weights they have for the moment.

Figure A2:1. Each competition is between the 15 neurons which constitute a pile, i.e. those with the same position within their respective groups.

Due to the local connections from the C_1-layer, the input to a neuron will depend on its position within the group; each neuron sees only a fraction of the C_1-layer. Neurons in different groups but at the same position, on the other hand, have exactly the same input. Therefore it is fitting to let the 15 neurons in such a 'pile' (see figure A2:1) compete with one another. The neuron with the strongest response is considered to have the best adapted weights for the input to the pile,

and therefore we update its weight setup, and hence the weight setup of the whole group, such that its predilection for this particular kind of input is further enhanced.

Since we practice weight sharing within the groups this strategy must, however, be somewhat developed. Different piles can have completely different input, but this does not prevent the same group from winning competitions in several piles. Since our objective is to specialize every group in only *one* feature we cannot update the weights every time one of its neurons wins a pile competition. Therefore we also let the neurons within a group compete, such that among the neurons having won pile competitions it is the one with the strongest response that is chosen as representative for the group. The weight setup is updated in accordance with the input seen by this neuron.

A2.2 Lateral inhibition and bad conscience

Lateral inhibition means that competing neurons inhibit each other. I have practiced this during each pile contest in such a way that the response from one neuron is lowered by the sum of the outputs of the other neurons multiplied by a certain factor (0,05 in my case). The purpose is to strengthen the diversification between the groups, i.e. making them specialize in different features. If all 15 groups react with approximately the same strength the lateral inhibition will secure that the neuron winning the pile contest has a small chance of also winning the contest within its group. This is desirable since we want the weight setup of every group to be changed towards one which differs from that of the others. It is then a bad idea to update the weights in accordance with an input which triggers the other groups almost as strongly.

A risk with self-organization is that the two or three groups winning the first few competitions quickly grow so much stronger than the other groups that they win all competitions, no matter what the input is. In order to avoid this I use a trick which is well known in this context – I give the groups a ”bad conscience”. This is implemented by introducing a variable for each group holding the magnitude of its conscience. During the pile contests the response of each neuron is reduced by the bad conscience of the group before being compared to the corresponding values of the other competitors. After a weight update the bad conscience value of updated groups is increased, while it is reduced for all others. A group which has had its weights updated will thus have a harder time winning the next contest since its conscience value has been increased.

A2.3 Updating of the weights

When all competitions centered around one input image are finished it is time to update the weights. The updating algorithm is taken from Fukushima’s self-organizing Neocognitron [9].

I go through the groups one after the other. Every group which has won one pile contest is updated in accordance with the fragment of the input image seen by the group representative. The input matrix describing this partial image is transformed into a row vector by adding the second row to the end of the first, and so on. This vector is then normalized. The excitatory synapses are updated according to the following equations:

$$\begin{cases} a_{ij} \rightarrow a_{ij} + \delta a_{ij} \\ \delta a_{ij} = q_2 \cdot \left(u_i - a_{ij} \right) \end{cases} \qquad (2)$$

The vector a_{ij} contains the weights from the C_1-layer to group no. j of the S_2-layer, and u_i is the normalized input vector. The factor q_2 is the training speed. Since the S-layers are of the same kind I use the same notation as in equation 1. Note, however,

that variables now refer to the weights and input to the *second* S-layer.

The group's inhibitory synapse is of course also updated. This is done according to the following rule

$$\begin{cases} b_j \rightarrow b_j + \delta b_j \\ \delta b_j = q_2 / 2 \cdot \left(\nu - b_j \right) \\ \nu = \sqrt{c_i \cdot u_i} \end{cases} \qquad (3)$$

The vector c_i contains the fixed normalized weight setup to the inhibitory group of the second S-layer. This vector is made such that the weight setup to each C_1-group has the look of figure 9.

Appendix 3

The response algorithm of the S-neurons

THE RESPONSE FROM the neurons of the S-layer is calculated with the following, perhaps a bit deterrent, equation [4]:

$$u_{s_l}(k,\mathbf{n}) = r_l \cdot \Phi(x) =$$

$$= r_l \cdot \Phi\left(\frac{1 + \sum_{\kappa=1}^{K_{C_{l-1}}} \sum_{\nu \in receptive\ field} a_l(\kappa,\nu,k) \cdot u_{C_{l-1}}(\kappa,\mathbf{n}+\nu)}{1 + \frac{r_l}{1+r_l} \cdot b_l(k) \cdot v_l(\mathbf{n})} - 1 \right) \quad (4)$$

The indices *S* and *C* indicate the type of neuron and the subindices indicate which complex (1 or 2) it belongs to. The letter *k* is used to index groups of the S-layer, while κ is used for groups of the input layer. The vector **n** indicates the position within the group of the neuron. Let us spend a couple of lines to motivate the form of this important equation.

The function $\Phi(x)=\max(0,x)$ ensures that the neuron does not give a negative response. The summation in the numerator is a scalar product between the input and the weight setup. In our optically implemented system, the products

$$a_l(\kappa,\mathbf{\nu},k) \cdot u_{C_{l-1}}(\kappa,\mathbf{n}+\mathbf{\nu})$$

are calculated by means of optical processes. The first summation is done over all groups of the preceding C-layer, and the second over the local region of each group to which the neuron is connected. The vector $\mathbf{\nu}$ is changed during the summation in order to cover the whole receptive field (see figure 5b). The equation is general and applies to all S-layers, but for the first complex the first summation sign is unnecessary since the summation is over the groups of layer C_0, i.e. the system input, where, of course, the number of groups is 1.

In the denominator the weight $b_l(k)$ and the value $v_l(\mathbf{n})$ appear. The latter is the output from the I-neuron which is geometrically equivalent to the S-neuron. I will presently discuss how this is evaluated. The weight $b_l(k)$ is the strength with which the inhibitory sister neuron affects the response from the S neuron. The factor r_l sets the feature selectivity of the neurons in the group.

The response from the inhibitory neurons is calculated according to equation (5):

$$v_l(\mathbf{n}) = \sqrt{\sum_{\kappa=1}^{K_{C_{l-1}}} \sum_{\mathbf{\nu} \in receptive\ field} c(\mathbf{\nu}) \cdot u_{C_{l-1}}(\kappa,\mathbf{n}+\mathbf{\nu})} \quad (5)$$

The notations reappearing from equation (1) have the same meaning as there. The weights $c(\mathbf{\nu})$ are those leading to the inhibitory group. In contrast to the weights $a(\mathbf{\nu})$ of the excitatory neurons, the inhibitory weights are not the result of training.

They are fixed from the beginning such that their sum equals one and distributed so that the weight values are greatest in the center of the perceptive field, decreasing towards the edges.

In the original model of Fukushima, where the inputs can be continuousvalued, the C-weights are multiplied with the square of the input signals. In our simplified model we limit ourselves to binary input signals for which taking the square has no effect. We therefore leave it out of our calculations.

Appendix 4
Matlab files

A4.1 The three main training files

A4.1.1 Start.m

%In this part we define all general variables and train the first S-layer.

```
%------------------------
%INITIALIZATION OF VARIABLES

clear
graypix=4;                          %Side of one grayscale pixel
s1viewsize=6;                       %Each S1-neuron is connected to s1viewsize~2
                                    %neurons in the input layer.
s2viewsize=4;                       %Each S2-neuron is connected to s2viewsize~2
                                    %neurons in the C1 layer.
inpside=27;                         %side of input matrix

groupsins1=8;                       %Number of groups in layer S1.
groupsinc1=groupsins1/2;            %Number of groups in layer C1.
groupsins2=15;                      %Number of groups in layer S2.
groupsinc2=groupsins2;              %Number of groups in layer C2.

s1side=inpside-s1viewsize+1;        %Side of a group in S1.
s1=zeros(s1side*groupsins1,s1side);
                                    %The groups in a layer are stored above
                                    %each other in one matrix, that is s1
                                    %is a column of groups.
s1toc1=2;                           %Side of the area in S1 which is compressed
                                    %into one pixel in C1.
c1side=s1side/s1toc1;               %Side of a group in C1.
c1=zeros(c1side*groupsinc1,c1side);
                                    %C1-matrix.
s2side=c1side-s2viewsize+1;%Side of a group in S2.
```

```
s2=zeros(s2side*groupsins2,s2side);
                                        %S2-matrix.
s2toc2=2;                               %Side of the area in S2 which is compressed
                                        %into one pixel in C2.
c2side=s2side/s2toc2;                   %Side of a group in C2.
c2=zeros(c2side*groupsinc2,c2side);

r1=2;                                   %Selectivity of S1-neurons.
r2=3;                                   %Selectivity of S2-neurons.

outputsize=10;                          %Number of possible outputs.
output=zeros(1,outputsize);             %Matrix to store output.

weightstos1=zeros(groupsins1+1,s1viewsize^2+1);
                                        %Each row holds the weights for one group.
                                        %The uppermost row of its field of view comes
                                        %first, followed by the second and so on.
                                        %Last comes the weight to the inhibitory group.
                                        %The weights to this group are stored in the
                                        %last row of the matrix.
weightstos2=zeros(groupsins2+1,groupsinc1*s2viewsize^2+1);
                                        %All groups in C1 are connected to each neuron
in                                      %S2.
weightstooutput=zeros(outputsize,groupsinc2*c2side^2);
                                        %Each row holds the weights from all neurons
                                        %in C2 to one outputneuron.

SLM11=zeros(inpside*3);                 %During computation of S1 only 3x3-parts of
                                        %the 6x6-views can be treated at once.
SLM21=zeros(c1side*6);                  %During computation of S2 the whole 6x6 views
                                        %can be treated at once, but each C1-group must
                                        %be treated separately.

weightstos1(groupsins1+1,:)=getfixweights(s1viewsize,1);
weightstos2(groupsins2+1,:)=getfixweights(s2viewsize,groupsinc1);
                                        %The function we call calculates the weights
                                        %to the inhibitory group.
%-------------------------------------



%-------------------------------------
%TRAINING OF THE WEIGHTS TO S1
%The first S-layer is trained "directly".

getS1set;                               %The trainingset is formed in a separate file.

%Now we can train the weights to S1.
q=10;                                   %Training efficiency.
c=weightstos1(groupsins1+1,1:s1viewsize^2);
```

```
for pic=1:size(trainingset,1)
  u=trainingset(pic,:);
  deltaa=q*c.*u;
  v=sqrt(c*u');
  deltab=q/1.5*v;
  weightstos1(pic,:)=[deltaa deltab];
end %pic
clear trainingset
%---------------------------------------

cd matfiles
save startresult
cd ..
```

A4.1.2 Selforg.m

```
%SELF ORGANIZATION OF WEIGHTS TO S2

clear
cd matfiles
load startresult
cd ..
pretrains2;                                  %Before self organising starts we give
                                             %the weights a good starting point by
                                             %training them in a similar manner to
                                             %how we trained the weights to S1.

%------------------------------
%INITIALIZATIONS
q2=0.8;                                      %Training speed.
genericname2='.raw.tab';
deltaconscience=0.05;
%After winning a contest the conscience of the group is increased =>
%=> its chances of winning the next contest decrease.


%------------------------------
%NOW BEGINS THE SELFORGANIZATION
%In this version we update all groups after each input-image is gone through.

conscience=zeros(groupsins2,1);
sopreproc;

for lap = 1:40
  indexvector=shuffle(outputsize);           %Shuffles the pictures.

  for picturenumber=1:length(indexvector)
    %Read input picture.
```

```
    index=num2str(indexvector(picturenumber));

    r=rand;
    %randomize style of training picture.
    if r<0.33
      style='min';
    elseif r<0.66
      style='ch';
    else
      style='co';
    end %if

    name=[style index 'c1'];
    cd opticresults
    load (name)
    cd ..
    opts2;

    %Time for the contest.
        contest;

    %We now have the representatives for each group. Time to update
        %the weights.
        update15groups;
                                        %This procedure also updates the
                                        %conscience-values.

  end %picturenumber
  q2=q2*0.95;
                                        %After each lap the trainingspeed
                                        %is lowered.
end %lap

cd matfiles
save selforgresult
cd ..
```

A4.1.3 Trainoutput.m

```
%This version trains on the following pictures:
%mon, ch, timk
%It runs 2000 laps.

clear
cd matfiles
load selforgresult
cd ..
```

```
weightmagnitude=0.01;
weightstooutput=(rand(outputsize,groupsinc2*c2side^2+1)-.5)*weightmagnitude;
                                                    %Each output pixel is connected to each
                                                    %pixel in c2 and to a fixed 1-neuron.
eta=.01;                                            %Training speed.
decayfactor=0.999;                                  %Speed is reduced by this factor
                                                    %after each lap.
alfa=.2;                                            %Momentum influence.
mom=zeros(size(weightstooutput));                   %Matrix where we store old weights.


%Do the preprocessing (repeated optical evaluations) first.
outpreproc;

%Now for the outputtraining
for lap = 1:2000
  indexvector=shuffle(outputsize);                  %Shuffles the pictures.

  for curpic=1:outputsize;
        %Randomize style
        r=rand;
    if r<0.33
      style='mon';
    elseif r<0.66
      style='ch';
    else
      style='timk';
    end %if

    name=[style num2str(indexvector(curpic)) 'c2'];
        cd opticresults
        load (name)
        cd ..
    contoutput;

    %Now we evaluate the errors.
    correct=zeros(outputsize,1);                    %Vector to which we compare
                                                    %the output of the system.
    correct(indexvector(curpic)+1)=1;               %Output is indexed by the value
+1,
                                                    %that is zero is stored in
output(1).
    outerror=correct-output;
        errmat(lap)=sum(outerror);                  %The errors during training
                                                    %are stored in this matrix.
    delta=outerror*c2c';
    weightstooutput= weightstooutput+eta*delta+alfa*mom;        %Update.
        mom=eta*delta;
  end %curpic
```

```
  eta=eta*.decayfactor;
end %lap

cd matfiles
save monchtimk2000
cd ..
```

A4.2 Help files in alphabetical order

A4.2.1 Contest.m

```
%Time for the contest.

contestsize=5;
%In each group 5x5 cells compete against each other.
latinhibfactor=.05;
%During the competition all groups inhibate each other laterally.
allwinners=zeros(groupsins2,3);
%Value, row and column of winning neuron for each group.

for row=1:s2side-contestsize+1
  %loop over rows of neurons
  for column=1:s2side-contestsize+1
        %loop over columns of neurons
        %Now the pillar is fixed and the groups compete with each other.
        %The group within this pillar which has the largest output
        %gets its weights updated.
        pillarwinner=[0 0 0 0];
        %value, x, y, and group of the winner within the pillar.
        for group=1:groupsins2
              %loop over the excitatory groups in s2
              onegroup=getgroup(group,s2);
              uplimit=row;
              downlimit=row+contestsize-1;
              leftlimit=column;
              rightlimit=column+contestsize-1;
              groupsample=onegroup(uplimit:downlimit,leftlimit:rightlimit);
              %groupsample contains the neurons of one group in the
              %current contest.
              [m,y]=max(groupsample);
              %m is a vector of max values for each column, y contains
              %the row numbers.
              [m,x]=max(m);
              %m holds the value and x holds the column in the sample.
              y=y(x);                    %y holds the row in the sample.
              %Now let's implement the lateral inhibition.
              for latinhibgroup=1:groupsins2
```

```
          if latinhibgroup~=group
            latgroup=getgroup(latinhibgroup,s2);
                        m=m-latinhibfactor*latgroup(y,x);
                      end %if
        end %latinhibgroup

                %Were the weights the best for this feature,
                %or do other groups have better weights?
                if m-conscience(group)>pillarwinner(1)
          pillarwinner=[m row+y-1 column+x-1 group];
                      %Update winner.
                end %if
      end %group

          %We've gone through the pillar. Time to update the
          %contest protocol.
          group=pillarwinner(4);
          if group>0
        if pillarwinner(1)>allwinners(group,1)
          allwinners(group,:)=pillarwinner(1:3);
        end %if
      end %if
    end %column
  end %row
```

A4.2.2 Contoutput.m

```
%evaluate continuous output.

evaluatec2c;
%Presents c2 in column form.

output=weightstooutput*c2c;
```

A4.2.3 Discreteweights.m

```
%A routine to change a continuous weightmatrix into one with weights quantized
%in 17 discrete steps.
oldmatrix=weightmatrix;
resolution=graypix^2+1
upperlimit=max(max(weightmatrix));
if upperlimit>0
  stepsize=upperlimit/(resolution-1);
  tempmatrix=round(weightmatrix/stepsize);
  weightmatrix=stepsize*floor(tempmatrix);
```

```
end %if
```

A4.2.4 Evaluatec1.m

```
%Evaluate the first C-layer.

%We begin by constructing four matrices holding the 'corners' in all
%the fields of view.
pixel1=s1(1:2:size(s1,1),1:2:size(s1,2));
pixel2=s1(2:2:size(s1,1),1:2:size(s1,2));
pixel3=s1(1:2:size(s1,1),2:2:size(s1,2));
pixel4=s1(2:2:size(s1,1),2:2:size(s1,2));

threshold=0.5;
%Now we compare each pixelvalue with the threshold-value.
pixel1=max(0,pixel1-threshold);
pixel2=max(0,pixel2-threshold);
pixel3=max(0,pixel3-threshold);
pixel4=max(0,pixel4-threshold);

%Now we construct an intermediate C1-layer.
cint=zeros(c1side*8,c1side);
%Time for the OR-function.
cint=pixel1|pixel2|pixel3|pixel4;

%Now we evaluate the final C1-layer. The groups of cint are
%paired two by two.

cone=getgroup(1,cint);
ctwo=getgroup(2,cint);
ctot=cone|ctwo;
c1=putgroup(ctot,1,c1);

cone=getgroup(3,cint);
ctwo=getgroup(4,cint);
ctot=cone|ctwo;
c1=putgroup(ctot,2,c1);

cone=getgroup(5,cint);
ctwo=getgroup(6,cint);
ctot=cone|ctwo;
c1=putgroup(ctot,3,c1);

cone=getgroup(7,cint);
ctwo=getgroup(8,cint);
ctot=cone|ctwo;
c1=putgroup(ctot,4,c1);
```

A4.2.5 Evaluatec2.m

```
%Evaluate the second C-layer.
%We begin by constructing four matrices holding the 'corners' in all
%the fields of view.
pixel1=s2(1:2:size(s2,1),1:2:size(s2,2));
pixel2=s2(2:2:size(s2,1),1:2:size(s2,2));
pixel3=s2(1:2:size(s2,1),2:2:size(s2,2));
pixel4=s2(2:2:size(s2,1),2:2:size(s2,2));

c2threshold=.4;

%Now we compare each pixelvalue with the threshold-value.
pixel1=max(0,pixel1-c2threshold);
pixel2=max(0,pixel2-c2threshold);
pixel3=max(0,pixel3-c2threshold);
pixel4=max(0,pixel4-c2threshold);

%Time for the OR-function.
c2=pixel1|pixel2|pixel3|pixel4;
```

A4.2.6 Evaluatec2c.m

```
%Turns the c2 layer into a column vector.

c2c=ones(groupsinc2*c2side^2+1,1)*(-1);
%The last component is for the threshold value.

for group=1:groupsinc2
  onegroup=getgroup(group,c2);
  for row=1:c2side
    groupstart=(group-1)*c2side^2;
    c2c(groupstart+(row-1)*c2side+1:groupstart+row*c2side)=onegroup(row,:);
  end %row
end %group
```

A4.2.7 Fillmatrix.m

```
function big=fillmatrix (unitmatrix,fillsize)
%unitmatrix is a square matrix which size must
%a factor in fillsize.

small=length(unitmatrix);
for row=1:small:fillsize
  left(row:row+small-1,1:small)=eye(small);
end %row
```

```
halfway=left*unitmatrix;

for col=1:small:fillsize
   right(1:small,col:col+small-1)=eye(small);
end %col

big=halfway*right;
```

A4.2.8 Getfixweights.m

```
function fixweights=getfixweights(viewsize,groups)

%This function calculates the weighs to the inhibitory group.

w=1;
s=viewsize;
total=0;

while s>1
  total=total+w*(s^2-(s-2)^2);
  s=s-2;
  w=w+1;
end %while

if s==1
  total=total+w;
end %if

k=1/(total*groups);

weights=zeros(1,viewsize^2*groups);

weightmatrix=ones(viewsize)*k;
levels=floor((viewsize+1)/2);
for level=2:levels
  index=level:viewsize-level+1;
  weightmatrix(index,index)=weightmatrix(index,index)+k*ones(viewsize-(level-1)*2);
end %for

for group=0:groups-1
  for row=1:viewsize
    goffset=group*viewsize^2;
    weights(1+(row-1)*viewsize+goffset:row*viewsize+goffset)=weightmatrix(row,:);
  end %row
end %group

fixweights=[weights 0];
```

A4.2.9 Getgroup.m

```
function onegroup=getgroup(group,matrix)
%Gets the desired group from the matrix storing all groups.

sidesize=size(matrix,2);
uplimit=(group-1)*sidesize+1;
```

```
downlimit=group*sidesize;
onegroup=matrix(uplimit:downlimit,:);
```

A4.2.10 Getoutput.m

```
function output=getoutput(bild,version, svarsmatris)
rad=bild*3;

output=svarsmatris(rad+version,:);
```

A4.2.11 GetS1set.m

```
%Procedure to form trainingset for S1.

trainingset=zeros(groupsins1,s1viewsize^2);
                                                    %One picture for each group.

%define temporary vectors.
middledot=[0 0 1 1 0 0];
lmiddledot=[0 0 1 0 0 0];
rmiddledot=[0 0 0 1 0 0];
horiline=[1 1 1 1 1 1];
nada=[0 0 0 0 0 0];
left=[1 0 0 0 0 0];
almostleft=[0 1 0 0 0 0];
middleleft=[0 0 1 0 0 0];
middleright=[0 0 0 1 0 0];
almostright=[0 0 0 0 1 0];
right=[0 0 0 0 0 1];
twoandthree=[0 1 1 0 0 0];
fourandfive=[0 0 0 1 1 0];
oneandtwo=[1 1 0 0 0 0];
lefthalf=[1 1 1 0 0 0];
righthalf=[0 0 0 1 1 1];
twothreefour=[0 1 1 1 0 0];
threefourfive=[0 0 1 1 1 0];
fiveandsix=[0 0 0 0 1 1];


%Now we define the training patterns in the following order:
%thick vertical line
%thin vertical line
%thick horizontal line
%thin horizontal line
%thick ul corner to lr corner
%thin ul corner to lr corner
%thick ll corner to ur corner
%thin ll corner to ur corner

f=1.7;                                  %This factor is needed to make the thick and
```

```
                                        %the thin feature detectors respond with
                                        %approximately the same strength.

trainingset=[[middledot middledot middledot middledot middledot middledot]/f
             lmiddledot lmiddledot lmiddledot lmiddledot lmiddledot lmiddledot
             [nada nada horiline horiline nada nada]/f
             nada nada horiline nada nada nada
             [oneandtwo lefthalf twothreefour threefourfive righthalf fiveandsix]/f
             left almostleft middleleft middleright almostright right
             [fiveandsix righthalf threefourfive twothreefour lefthalf oneandtwo]/f
             right almostright middleright middleleft almostleft left];

clear middledot horiline nada left almostleft twoandthree
clear middleleft middleright almostright right fourandfive
clear fiveandsix lefthalf lmiddledot oneandtwo righthalf
clear rmiddledot threefourfive twothreefour
```

A4.2.12 Getsmallmatrix.m

```
function smallmatrix=getsmallmatrix(rowofweights,time)
%Picks out the relevant weights for the current 3x3 matrix.

smallmatrix=zeros(3);
start=(time-1)*3+floor((time-1)/2)*12+1;
smallmatrix(1,:)=rowofweights(start:start+2);
smallmatrix(2,:)=rowofweights(start+6:start+8);
smallmatrix(3,:)=rowofweights(start+12:start+14);
```

A4.2.13 Getweights.m

```
function weights=getweights(ingroup,outgroup,allweights,viewsize)
%Returns a viewsize*viewsize matrix with the weights
%between ingroup and outgroup.

weights=zeros(viewsize);
start=(ingroup-1)*viewsize^2;
for rownumber=1:viewsize
  left=start+(rownumber-1)*viewsize+1;
  right=start+rownumber*viewsize;
  row=allweights(outgroup,left:right);
  weights(rownumber,:)=row;
end %rownumber
```

A4.2.14 Opts1.m

```
%Procedure opts1.

%s1 should be a column of groups with the number of groups being
%groupsins1.

optstart=flops;

groupcolumn=zeros((groupsins1+1)*s1side,s1side);
%In this matrix all groups are temporarely stored below each other.
SLM11=pixelizeinput(input,3);
%multiplies each input-pixel with the appropriate number of SLM-pixels.

for group =1:groupsins1+1
  onegroup=zeros(s1side);

  for time=1:4
    %The computation of each s1group must be performed in 4 parts.
    %time is a number between 1 and 4 which tells us which 3x3 pixel-
    %region is currently on display.
    %1 2
    %3 4

    macrorow=floor((time-1)/2)+1;
    macrocol=rem((time-1),2)+1;

    smallmatrix=getsmallmatrix(weightstos1(group,:),time);
    weightmatrix=fillmatrix(smallmatrix,3*inpside);
    %Each inputpixel is projected onto 3x3 weightpixels at a time.
    discreteweights;
          %The binary nature of the weight-SLM leads to a quantization of values.
    CCD=SLM11.*weightmatrix;
    %In order for this simulative multiplication to work, SLM11
    %must have the same dimensions as weightmatrix. That is, each
    %inputpixel must be composed of 3x3 pixels in the same state.

    for sr=1:s1side
      for sc=1:s1side
        %The CCD is being gone through one pixel at a time.
        rowstart=(macrorow-1)*3+sr;
              colstart=(macrocol-1)*3+sc;
              %Since we are not dealing with complete 6x6-pixels, we have to make
              %jumps in the storage matrix after every 3x3-pixel.
              onegroup(sr,sc)=onegroup(sr,sc)+sumview(CCD,rowstart,colstart,3);
                  end %sc
    end %sr

  end %time
```

```
   groupcolumn=putgroup(onegroup,group,groupcolumn);

end %group

%Now the optical multiplications for all groups are finished and
%the results are stored in groupcolumn.

%Now let's do the non-optical part of the S1-evaluation.
inhibgroup=getgroup(groupsins1+1,groupcolumn);
for group=1:groupsins1
   b=weightstos1(group,s1viewsize^2+1);
   denominator=1+r1/(1+r1)*b*sqrt(inhibgroup);
   result=zeros(s1side);
   currentgroup=getgroup(group,groupcolumn);
   result=r1*max(0,(1+currentgroup)./denominator-1);
   s1=putgroup(result,group,s1);
end %group

totopt=totopt+flops-optstart;
```

A4.2.15 Opts2.m

```
%Procedure opts2.

%s2 and c1 should both be a column of groups with the number of groups being
%groupsins2 and groupsinc1 respectively.

optstart=flops;

clear weightmatrix                                  %In case this has been used earlier,
                                                                    %it is cleared.
sumgroupcol=zeros(s2side*(groupsins2+1),s2side);
%This is to store the CCD-result which will be used in the final algorithm.

for ingroup =1:groupsinc1
%loop over inputgroups, that is, groups in c1.

  c1group=getgroup(ingroup,c1);
  SLM21=pixelizeinput(c1group,s2viewsize);
  %Each input-pixel must be duplicated as many times as there are synapses
  %connected to it.

  for outgroup=1:groupsins2+1
  %loop over outputgroups, that is, groups in s2 + the inhibitory group.

    weights=getweights(ingroup,outgroup,weightstos2,s2viewsize);
    weightmatrix=fillmatrix(weights,s2viewsize*c1side);
        %Each inputpixel is projected onto 6x6 weightpixels at a time.
        discreteweights;
        %The binary nature of the weight-SLM leads to a quantization of values.
        CCD=SLM21.*weightmatrix;
        %In order for this simulative multiplication to work, input
        %must have the same dimensions as weightmatrix. That is, each
        %inputpixel must be composed of 6x6 pixels in the same state.
        sumgroup=zeros(s2side);

        for sr=1:s2side
          for sc=1:s2side
                %We now update all pixels of the current s2-group one by one.
                sumgroup(sr,sc)=sumview(CCD,sr,sc,s2viewsize);
      end %sc
        end %sr

        old=getgroup(outgroup,sumgroupcol);
        sumgroupcol=putgroup(old+sumgroup,outgroup,sumgroupcol);
  end %outgroup

end %ingroup

%Now the multiplications for all groups are finished and the results
```

```
%are stored in sumgroupcol.

%Now let's do the non-optical part of the S2-evaluation.
inhibgroup=getgroup(groupsins2+1,sumgroupcol);
for group=1:groupsins2
  b=weightstos2(group,groupsinc1*s2viewsize^2+1);
  denominator=1+r2/(1+r2)*b*sqrt(inhibgroup);
  result=zeros(s2side);
  currentgroup=getgroup(group,sumgroupcol);
  result=r2*max(0,(1+currentgroup)./denominator-1);
  s2=putgroup(result,group,s2);
end %group

totopt=totopt+flops-optstart;
```

A4.2.16 Outpreproc.m

```
%Repeated optical evaluations are done once and for all, and the results are
%saved into files.

%This version saves C2 for the following pictures:
        %co, min, ch, timk, mon

genericname2='.raw.tab';

picnum=0;

genericname1='bilder/co';
index=num2str(picnum);
input=readpicture([genericname1 index genericname2],inpside);

opts1;
evaluatec1;
opts2;
evaluatec2;

cd opticresults
save co0c2 c2
cd ..

etc,etc....
```

A4.2.17 Pixelizeinput.m

```
function SLM=pixelizeinput(input,pixelsize)
```

```
inpsize=length(input);
colindex=0;
rowindex=0;

left=zeros(pixelsize*inpsize,inpsize);
for index=1:pixelsize:pixelsize*inpsize
  colindex=colindex+1;
  left(index:index+pixelsize-1,colindex)=ones(pixelsize,1);
end %index

right=zeros(inpsize,inpsize*pixelsize);
for index=1:pixelsize:pixelsize*inpsize
  rowindex=rowindex+1;
  right(rowindex,index:index+pixelsize-1)=ones(1,pixelsize);
end %index

SLM=left*input*right;
```

A4.2.18 Pretrains2.m

```
%This routine gives the weights to the S2-layer small startout-values chosen
%so different groups get different styles.

trainingset=zeros(groupsins2,s2viewsize^2*groupsinc1);
                                                    %One picture for each S2-group.
                                                    %Each S2-group takes input from
all C1-groups.

%Temporary vectors.
horiline=[1 1 1 1];
nada=[0 0 0 0];
left=[1 0 0 0];
almostleft=[0 1 0 0];
almostright=[0 0 1 0];
right=[0 0 0 1];


%Now the features we trained C1 for:
vertline=[almostleft almostleft almostleft almostleft];
c1horiline=[nada horiline nada nada];
ultolr=[left almostleft almostright right];
lltour=[right almostright almostleft left];
empty=[nada nada nada nada];

%And now the trainingset for s2.

group1=[vertline empty empty empty];
group2=[empty c1horiline empty empty];
```

```
group3=[empty empty ultolr empty];
group4=[empty empty empty lltour];
group5=[vertline c1horiline empty empty];
group6=[vertline empty empty lltour];
group7=[vertline empty ultolr empty];
group8=[empty c1horiline empty lltour];
group9=[empty c1horiline ultolr empty];
group10=[empty empty ultolr lltour];
group11=[vertline c1horiline ultolr empty];
group12=[vertline c1horiline empty lltour];
group13=[vertline empty ultolr lltour];
group14=[empty c1horiline ultolr lltour];
group15=[vertline c1horiline ultolr lltour];

trainingset= [group1
                        group2
                        group3
                        group4
                        group5
                        group6
                        group7
                        group8
                        group9
                        group10
                        group11
                        group12
                        group13
                        group14
                        group15];

clear  horiline nada left almostleft leftright
clear mittleft mittright almostright right

%Now we can train weightstos2.
q=100;                                  %Training speed.
c=weightstos2(groupsins2+1,1:groupsinc1*s2viewsize^2);
for feature=1:size(trainingset,1)
  u=trainingset(feature,:);
  deltaa=q*c.*u;
  weightstos2(feature,:)=[deltaa 0];
end %feature

nonormfactor=1.6;                       %It turns out that normalization leaves
                                        %with weights that are slightly too small.
                                        %We therefore compensate with this factor.
for row=1:groupsins2
  oldsum=sum(weightstos2(row,:));
  weightstos2(row,:)=weightstos2(row,:)/oldsum*nonormfactor;
end %row
```

```
clear trainingset
```

A4.2.19 Putgroup.m

```
function matrix=putgroup(onegroup,groupnumber,matrix)
%Stores the desired group in the matrix storing all groups.

sidesize=size(onegroup,1);
uplimit=(groupnumber-1)*sidesize+1;
downlimit=groupnumber*sidesize;
matrix(uplimit:downlimit,:)=onegroup;
```

A4.2.20 Readpicture.m

```
function f=readpicture(fnamn,N)

%This function reads a picturefile and transforms the contents into a matlab-
%friendly format. The result is stored in the matrix f. It is used with this
%syntax: "m=readpicture('path/filename',sidesize);"

fid=fopen(fnamn,'r');
v=fscanf(fid,'%d');
v=~round(v/255);
f=reshape(v,N,N)';
fclose(fid);
```

A4.2.21 Shuffle.m

```
function outvector=shuffle(outsize)
a=0:outsize-1;
while length(a)>0
  x=floor(rand*length(a)+1);
  outvector=[outvector a(x)];
  a=[a(1:x-1) a(x+1:length(a))];
end
```

A4.2.22 Sumview.m

```
function total=sumview(CCD,rowstart,colstart,pixelsize)

CCDrowstart=(rowstart-1)*pixelsize+1;
```

```
CCDcolstart=(colstart-1)*pixelsize+1;
total=0;

step=pixelsize+1;
stop=step*(pixelsize-1);
for rowoffset=0:step:stop
  for coloffset=0:step:stop
    total=total+CCD(CCDrowstart+rowoffset,CCDcolstart+coloffset);
  end %coloffset
end %rowoffset
```

A4.2.23 Update15groups.m

```
%TO UPDATE ALL GROUPS AFTER ONE IMAGE, THIS IS USED.
u=zeros(1,s2viewsize^2*groupsinc1);
for group=1:groupsins2
  %Go through all groups and check if they won a contest. Then they should be updated.
  if allwinners(group,1)>0
    x=allwinners(group,3);
    y=allwinners(group,2);

    %The input to the winning neuron has to be transformed
        %into a row vector.
    for c1group=1:groupsinc1
      onegroup=getgroup(c1group,c1);
              uplimit=y;
      downlimit=y+s2viewsize-1;
      leftlimit=x;
      rightlimit=x+s2viewsize-1;
      temp=onegroup(uplimit:downlimit,leftlimit:rightlimit);
      tempvector=reshape(temp',1,s2viewsize^2);
              clear temp
      u((c1group-1)*s2viewsize^2+1:c1group*s2viewsize^2)=tempvector;
              clear tempvector
    end %c1group

    c=weightstos2(groupsins2+1,1:groupsinc1*s2viewsize^2);
    norminput=u/sum(u);
    aweights=weightstos2(group,1:s2viewsize^2*groupsinc1);
    deltaa=q2*(norminput-aweights);
    aweights=aweights+deltaa;
    aweights=aweights/sum(aweights);
    v=sqrt(c*norminput');
    deltab=q2/2*(v-weightstos2(group,s2viewsize^2*groupsinc1+1));
        rightpart=weightstos2(group,s2viewsize^2*groupsinc1+1)+deltab;
    weightstos2(group,:)=[aweights rightpart];
        clear aweights deltaa rightpart
```

```
            conscience(group)=conscience(group)+deltaconscience;
            %Increase the conscience of the winner and decrease the others'.

    else
      %The group won't be updated => lower its conscience.
            conscience(group)=max(0,conscience(group)-deltaconscience);
    end %if
  end %group
  clear allwinners          %This is no longer needed.
```

References

1. S. Haykin, “Neural Networks” (Macmillan College Publishing Company, 1994)
2. P. D. Wasserman, “Neural Computing” (Van Nostrand Reinhold, 1989)
3. R. Hecht-Nielsen, “Neurocomputing” (Addison-Wesley Publishing Company, 1990)
4. K. Fukushima, S. Miyake, T. Ito, “Neocognitron: A Neural Network Model for a Mechanism of Visual Pattern Recognition”, IEEE Transactions on Systems, Man, and Cybernetics, vol SMC-13, no. 5, 1983
5. Richard Englund, “Optical Neural Networks for Associative Image Processing”, Chalmers University of Technology, Department of Physics, Göteborg, Sweden, 1995
6. T. Chao, W. W. Stoner, “Optical implementation of a feature-based neural network with application to automatic target recognition”, Appl. Opt. vol. 32, no. 8, 1993
7. B. Löfving, “Dynamic Light Modulation by Ferroelectric Liquid Crystal Devices”, Technical Report No 259L, Thesis for degree of Licentiate of Technology, Chalmers Univ. of Techn. Department of Microwave Techn. , Göteborg, Sweden 1997
8. K. M. Johnson, G. Moddel, “Motivations for using ferroelectric liquid crystal spatial light modulators in neurocomputing”, Appl. Opt. vol. 28, no. 22, 1989
9. K. Fukushima, “Neocognitron: A Self-organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position”, Biol. Cybernetics 36, 193-202 (1980)
10. F. Nikolajeff, ”Diffractive Optical Elements: Fabrication, Replication, and Applications and Optical Properties of a Visual Field Test”, Technical Report No 300, Doctoral Thesis for the degree of Doctor of Philosophy, Chalmers Univ. of Techn. Department of Microwave Techn. , Göteborg, Sweden 1997