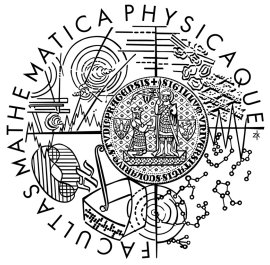
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

DIPLOMOVÁ PRÁCE



Marek Kukačka

Artificial neural networks for pattern recognition

Katedra teoretické informatiky a matematické logiky

Vedoucí diplomové práce: Doc. RNDr. Iveta Mrázová, CSc.

Studijní program: Informatika, obor Teoretická informatika

2007

Rád bych poděkoval především vedoucí mé diplomové práce, paní Ivetě Mrázové, za cenné rady a připomínky, které mi při realizaci této práce velice pomohly. Děkuji také knihovně MFF UK za poskytnutí nezbytných studijních materiálů.

Dále chci poděkovat své rodině a přátelům za poskytování podpory a motivace, jak při psaní této práce, tak i po dobu celého mého studia.

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce.

V Praze dne Marek Kukačka

Contents

1 Introduction **1**

2 Artificial Neural Networks **3**

2.1 History 3

2.2 Taxonomy of neural networks 4

3 Pattern Recognition **6**

3.1 Theory of classification 6

3.2 Preprocessing and Feature Extraction 7

4 Multi-layered perceptron network **10**

4.1 Model description 10

4.2 Notation 11

4.3 Learning algorithm 12

4.4 Faster and better learning 15

4.4.1 Momentum 15

4.4.2 Altering the derivative 16

4.4.3 Altering the learning rate 16

4.4.4 Avoiding expensive calculations 17

5 Kohonen's neural network **19**

5.1 Definition 19

5.2 Learning algorithm 20

5.3 Properties of Kohonen's networks 21

6 Convolutional network model **23**

6.1 Background 23

6.2 Model description 24

6.3 LeNet-5 25

6.4 Learning algorithm 27

6.5 Properties of convolutional networks 28

6.6 Implementation 29

7 RBF hybrid network model **31**

7.1 Model description 31

7.2 Learning algorithm 33

7.3 Winner takes all 35

7.4 Properties of RBF hybrid networks 36

8 Implementation **37**

8.1 Library description 37

8.2 Encountered problems 38

9 Generalization and training speed **42**

9.1 Over-fitting and early stopping 43

9.2 Test proposal 44

9.3 Results 47

10 Invariance to transformations **53**

10.1 Test proposal 54

10.2 Results 55

11 Conclusion **73**

A Script documentation **77**

Název práce: Rozpoznávání vzorů pomocí neuronových sítí
Autor: Marek Kukačka
Katedra (ústav): Katedra teoretické informatiky a matematické logiky
Vedoucí bakalářské práce: Doc. RNDr. Iveta Mrázová, CSc., Katedra softwarového inženýrství
e-mail vedoucího: Iveta.Mrazova@mff.cuni.cz

Abstrakt: V této práci jsou popsány možnosti, výhody a nevýhody využití neuronových sítí při rozpoznávání vzorů.
Je představeno několik modelů neuronových sítí použitelných pro tuto úlohu. Standartní vícevrstvá perceptronová síť je porovnávána se sofistikovanější konvoluční sítí. Práce také představuje nový model, inspirovaný konvolučními sítěmi, jehož účelem je odstranit některé jejich nedostatky.
Práce popisuje výsledky testů porovnávajících výsledky popsaných neuronových sítí na úloze rozpoznávání ručně psaných číslic.

Klíčová slova: neuronové sítě rozpoznávání vzorů

Title: Artificial neural networks for pattern recognition
Author: Marek Kukačka
Department: Katedra teoretické informatiky a matematické logiky
Supervisor: Doc. RNDr. Iveta Mrázová, CSc., Katedra softwarového inženýrství
Supervisor's e-mail address: Iveta.Mrazova@mff.cuni.cz

Abstract: This work describes the advantages and disadvantages of using neural networks for pattern recognition.
Several neural network models are described and their use for pattern recognition is demonstrated. Standard multi-layered perceptron model is compared to a more sophisticated convolutional network model. A new network model is introduced, which is inspired by the convolutional networks and aimed at rectifying some of their shortcomings.
The work describes results of tests performed with the described network model on the problem of recognizing hand-written digits.

Keywords: neural networks pattern recognition

Chapter 1

Introduction

Artificial neural networks are mathematical models, simulating systems of biological neurons found in animals' and humans' brains. They exhibit some interesting properties, like the ability to learn or adapt, which hints on how these abilities could arise in biological networks of interconnected computation units. In practical applications, artificial neural networks are being used for solving various problems that are not solvable by conventional means (i.e. by an optimal deterministic algorithm). These problems include data mining, image processing and pattern recognition, control of robotic manipulators, prediction of behavior of complex systems, and many other.

In this work, I shall focus on the pattern recognition problem. It is a well-known fact that humans still exceed computers by far in the ability to recognize patterns, whether in visual, aural or other type of data. There are even tests for discriminating between humans and computers based on this fact, for example the popular CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) test. Even though there are many areas, in which neural networks are widely used for pattern recognition (i.e. ZIP code reading, face recognition), there are still problems where automation of pattern recognition would be profitable, but the efficiency of the available techniques is not sufficient for them (i.e. hand-written text reading).

The aim of this work is to summarize some of the models of artificial neural networks most widely used for pattern recognition. I demonstrate their use and compare their success rates on a number of various pattern recognition

tasks. Furthermore, I propose a hybrid model of neural network, combining the standard multi-layered perceptron networks with other types of adaptive computation units. Hopefully, the comparison will expose the advantages of various properties of the introduced network models, and show new directions for devising even better models.

Chapter 2

Artificial Neural Networks

2.1 History

The beginning of the research in the field of artificial neural networks dates back to the first half of 20th century, when various computation models were being considered and tested. The first model inspired by the working of biological neurons were the McCulloch-Pitts neurons, proposed in 1943 by Warren McCulloch and Walter Pitts, and a more general model introduced by Frank Rosenblatt, the *perceptron*. Since then, a large number of artificial neural network models was introduced and studied thoroughly. The inspiration for these models were the processes in biological brains, the communication between densely interconnected neural cells and the ability of these networks to adapt to external stimuli.

Artificial neural networks are tightly tied with many other areas of research. The computational power of networks of simple units is being studied in the area of electronic circuit design. Neurology is often a source of inspiration for new models of artificial neural networks, which strive to imitate newly discovered principles of biological information processing. The fields of artificial intelligence and cybernetics are proving the usefulness of the theoretical results in practical applications.

2.2 Taxonomy of neural networks

Over the years of research of the neural networks, many models were proposed and studied. Some of them were improvements of older models, while other used new and unique methods of computation. Naturally, there are many ways to differentiate the neural network models into separated groups. Usually, a network is classified by its topology, the type of computation it performs, and the learning algorithm that is used to adapt the network for the task at hand.

The topology of a neural network strongly influences the way the network processes its inputs. Generally, networks are divided in depence on the occurence of cycles in the graph of their neurons' connections. If there are no loops in the topology, neurons in the network can be ordered into layers, starting from the ones that accept the input values. The neurons in the second layer then process the output of neurons in the first layer, and so on. Such computation model is then called *synchronous*, as opposed to the *asynchronous* model, in which neurons to be computed are chosen stochastically, independently of their position in the network. An example of an asynchronous model is the Hopfield network.

There are two main types of learning algorithms for the neural networks. The first one, called *supervised learning* or *learning with a teacher*, is used when for each training input pattern we know what the desired output pattern is. This can be viewed as using the neural network to a function for which we have a set of input-output value pairs. The learning consists of iterative adaptation of network parameters aimed to minimize the difference between real and desired network output for all training input patterns. Gradient descent optimization is usually used for these adaptations, implemented in the form of *backpropagation algorithm* or one of its many variations. This algorithm will be described in detail in later chapters.

The other type of learning of neural network is called *unsupervised learning* or *learning without a teacher*. Here, no desired behavior is presented to the network. Network parameters then adapt to presented input patterns to describe their distribution in input space. The network itself decides how to react to various stimuli, therefore this adaptation strategy is also sometimes calles *self-organization*. This kind of learning is usually used to find natural clustering of similar inputs and teach the neural network to react in the same

way to the patterns in the same cluster. This kind of adaptation is used for the Kohonen's network model, described later in this work.

There are few other ways to classify neural networks. Layered networks can be differentiated by their number of layers, because this number strongly influences computational power of the network. Networks can be distinguished by the activation function used by their neurons, most common of which are the signum function for binary networks and a sigmoid function for analog networks.

Some neural network models are composed of sub-networks of various types, and therefore cannot be easily classified into one of the above mentioned categories. These are called *hybrid networks*. Examples of such networks will be described with more detail in later chapters.

Chapter 3

Pattern Recognition

Pattern recognition problem consists of assigning an input vector to one of target classes. Such a classification of real-world data is seldom straightforward, even though humans perform such tasks all the time without effort. For computers, however, some of these problems prove to be extremely difficult to solve effectively, i.e. character recognition or speech recognition. In some simple cases rule-based systems can be manually constructed and used with success, but usually it is more effective to use one of Artificial Intelligence techniques which can be trained from a set of examples. Such classifiers include Bayesian networks, Markov models, few "nearest neighbors" methods, and various kinds of artificial neural network models, which are the focus of this work.

3.1 Theory of classification

Statistical pattern recognition is a well-studied field, providing a very general point of view on the problem. Here, input patterns and their classification are considered to be probabilistic variables and classification is made by finding the most probable class for the given input. This is formalized by the *Bayes' theorem*

$$P(C_k|X) = \frac{P(X|C_k)P(C_k)}{P(X)} \tag{3.1}$$

where C_k refers to k-th target class, X refers to the given input, and function $P(\cdot)$ represents the probability of occurence of its argument. *Conditional* probability function $P(X_1|X_2)$ represents the probability of occurence of X_1 *under the condition* that X_2 has been observed. Left-hand side of the equation 3.1 represents the so-called *posterior* probability, since it describes the likelihood for the observed input X to belong to the class C_k. This value is expressed in terms of prior probability $P(C_k)$ and *class-conditional* probability $P(X|C_k)$. The denominator $P(X)$ is only normalization factor and, since it depends only on the input, can be ignored when comparing posterior probabilities for a single input X.

In the optimal case, we would know all the quantities on the right-hand side of the equation 3.1 and classification of given input would then be trivial. Unfortunatelly, in real-world problem we can only estimate these values. Various means are used to make these estimation, and their fidelity then influences the precision of the resulting classification. It is usual to make use of a set of manually classified input patterns, called the *training set*. Apriori probability $P(C_k)$ of class C_k can then be estimated from rate of occurence of this class in the training set. Estimating the class-conditional probability, which can be also viewed as probability density function for the given class, is more difficult. There is a number of methods for estimating this function, and they can be separated into groups according to assumptions they take about the estimated function. *Parametric* methods take a certain functional form of the density function (i.e. Gaussian distribution), and then strive to find its parameters that fit the function to the training data set with minimal deviation. Non-parametric methods, on the other hand, do not make any assumptions about the function's form, rather they let the data determine the resulting density's form - the method called "K-nearest-neighbours" belongs into this category. Finally, there are methods that take the best from both of these approaches. These are called *semi-parametric*, and feed-forward neural networks, described later, can be regarded as belonging to this category. For more details about density estimation methods, see chapter 2 of Bishop [2].

3.2 Preprocessing and Feature Extraction

The process of classification is usually divided into several stages. The reason for this is the fact that it is seldom beneficial to present the raw data

straight to the classifier. It is also usually advisable to transform the output of the classifier into a form usable for the rest of the application. Therefore, preprocessing and postprocessing stages usually precede and follow the classification itself.

There are many good reasons to introduce a preprocessing stage into the classification process. One of them is the decrease in dimensionality of the data the classifier has to process. Rather than feeding all of 1024 pixels of 32x32 image into the classifier, we can pick only the important information from the picture, and present it to the classifier. For example, if we want to determine whether an image contains a character "A" or "B", we can either feed all of its pixels into the classifier, or preprocess the image to figure the number of "holes" in the character, which will then make the classification trivial. Also, lower dimensionality of the input increases the speed of learning of the classifier.

Such pieces of information, describing the object we want to classify, are commonly called *features*, and preprocessing aimed at finding values of selected features is then denominated as *feature extraction*. Classifying features instead of raw data is a lot easier, because this way it is possible to incorporate *prior knowledge* about the problem in the classification process without the need to modify the classifier itself. Generally, we want features to be *invariant* to transformations that do not influence the classification (i.e. translation and scaling in character recognition) and *discriminable*, which means they should be different for objects of different classes (i.e. the number of "holes" mentioned earlier, as opposed to, for example, color of the character). Selecting a suitable set of features to be extracted in the preprocessing stage can greatly increase the effectiveness of the classification.

However, there are recognition problems which do not allow for a easy construction of the preprocessing stage. The reading of character-based CAPTCHA codes used to protect WWW forms from automatic processing can be used as an example of such a problem. The usual preprocessing in case of visual object recognition consists of separating the object from the scene, normalizing its size, position and orientation, followed by detecting characteristic features. When the characters in the image are arbitrarily positioned and oriented, possibly even overlapping, it is very difficult to design an algorithm which would produce their standardized form. In case the characters are subjected to arbitrary deformation, automatic remedying of this

deformation is practically impossible.

It would therefore be a great advantage if the used classifier was able to work on the raw data and adapt to the possible input deformations during its training. In this thesis, I describe and test two neural network models designed to answer the need for such classifiers. For comparison, I also test a very simple neural network model commonly used for classification and function approximation.

Chapter 4

Multi-layered perceptron network

Multi-layered perceptron network is perhaps the most frequently used network architecture when supervised learning is applicable. This neural network model is applicable to a wide variety of problems, and, of course, to the pattern recognition problem as well. In this chapter, I descibe this model in detail and consider its advantages for pattern recognition.

4.1 Model description

A multi-layer perceptron network is composed of computational units called *neurons*, that are organized into ordered layers. Each neuron receives its input from all neurons in the preceding layer but from no other ones. Input patterns are presented to the network in the form of a virtual input layer with the index equal to zero. This input is processed by neurons in the first layer. Afterwards, their output is taken as the input for the neurons from the second layer, and so on. Finally, the network's output is composed of the output values of neurons in the last layer.

A network of this type maps the vectors from its input space to vectors in its output space. Its function is encoded in the strengths of connections between network's neurons, also called *weights*. The network is trained by

adjusting the weights of its neurons in such a way to produce the desired output values in reaction to the presented input patterns.

4.2 Notation

Below, we will specify the notation used when describing the details of the multi-layered perceptron model.

Each perceptron unit computes the dot product of its weights and its input values, adds a bias, and applies its *activation function* to the resulting potential value. Formally, this yields the equation:

$$y_{n+1,i} = \Phi(\sum_{j=1}^{J} w_{n+1,i,j} y_{n,j} + b_{n+1,i}) = \Phi(\zeta_{n+1,i}) \qquad i \geq 1 \tag{4.1}$$

where $y_{n,j}$ is the output of j-th neuron in the n-th layer, Φ is the activation function, $b_{n+1,i}$ is the bias of the i-th neuron in the layer $n+1$, and $\zeta_{n+1,i}$ is the *potential* of the neuron i. Network's input values are represented by $y_{0,j}$ in this notation. The output values are represente by $y_{L,j}$ where L is the number of network's layers. We can simplify the equation 4.1 by defining $y_{i,0} = 1$ for each layer i, and then storing the bias value in $w_{i+1,j,0}$. The resulting equation will have the following form:

$$y_{n+1,i} = \Phi(\sum_{j=0}^{J} w_{n+1,i,j} y_{n,j}) \qquad i \geq 1 \tag{4.2}$$

If not stated otherwise, I will generally work with N layers in the network. I will also denote the number of the output neurons as I and index the individual neurons in this layer by i. For the layer below the last one, I will use the symbol J for its size and j for the index, and so on. When considering the set of all weights in the network, I will use only one subscript, e.g. w_i instead of $w_i j$.

For the activation function Φ, logistic sigmoid is used. This function has several useful properties, which will be mentioned later in this chapter. The formula for the sigmoid can be expressed as follows:

$$\Phi(x) = \frac{1}{1 + \exp(-\lambda x)} \tag{4.3}$$

The parameter α determines the tangent of the sigmoid around the origin. Although it can be included as a further parameter into network training and adjusted by the learning algorithm, I have kept it constant in my implementation to keep the learning algorithm simple and fast.

It is a common practice to denote the first layer of neurons, to which the input vector is presented, as the *input layer*. The last layer, from which the network's output is read, is called the *output layer*, and any intermediate layers are called *hidden layers*.

4.3 Learning algorithm

To train multi-layered perceptron networks, the standard back-propagation algorithm can be used. In order to keep this thesis self-contained, I will describe the algorithm. Later on, I will refer back to various steps of the algorithm.

When training neural networks to approximate a function, a set of input-output pairs is needed, which roughly covers areas of interest of the function's input space. Let us denote this set of training patterns as a training set τ of size T and its members as oriented pairs $< \vec{i}_i, \vec{o}_i >$. Here, $\vec{i}_i$ is the input vector to be presented to the network, and $\vec{o}_i$ is the vector of the desired output values after processing the input $\vec{i}_i$.

During the optimization of network's performance, we need an objective function measuring the error rates of the trained system, which we then try to minimize. This function is commonly called the *error function*, denoted E. It is defined as

$$E = \frac{1}{T} \sum_{i=1}^{T} E_i \tag{4.4}$$

$$E_i = \frac{1}{2}(\vec{y}_i - \vec{o}_i)^2 \tag{4.5}$$

where $\vec{y}_i$ is the vector of actual output values obtained from network after processing the input $\vec{i}_i$. The function E_i can take various forms, here we use

the standard Mean Square Error (MSE) function. Vector $\vec{o_i}$ is the desired, or *target* vector. We can also remark that the value $\vec{y}i$ is in fact a function

$$\vec{y} = \psi(\vec{w}, \vec{x}) \tag{4.6}$$

which for a fixed network topology, vector of network's weights and input vector produces the network's output.

We can see that the function ψ is differentiable, because it is composed of differentiable functions Φ (see 4.3), addition and multiplication by a constant. Therefore, the function E is also differentiable, and we can use the well-known *gradient descent* optimization technique to minimize E. The key idea of gradient descent lies in determining the partial derivatives $\frac{\partial E}{\partial w_j}$ for every weight w_j from $\vec{w}$, thus determining the direction of the gradient ∇E, and adjusting $\vec{w}$ in the reverse direction. Written formally:

$$\Delta w_j = -\alpha \cdot \frac{\partial E}{\partial w_j} \tag{4.7}$$

where α is a *learning parameter*, determining how much the weight will be changed in a single learning step. The choice of this parameter influences the speed and quality of learning, and will be discussed with other parameters later.

Calculating the partial derivatives for all weight parameters separately can be very demanding. The chosen model is allows us to reuse values previously computed in higher layers of the network. This is the main idea of *back-propagation algorithm*, or BP-algorithm for short.

First, we will compute the adjustment coefficients for the weights in the output layer of network.

$$\begin{aligned} \Delta w_{ij} &= -\alpha \cdot \frac{\partial E}{\partial w_{ij}} \\ &= -\alpha \cdot \frac{\partial E}{\partial y_i} \cdot \frac{\partial y_i}{\partial \zeta_i} \cdot \frac{\partial \zeta_i}{\partial w_{ij}} \\ &= -\alpha \cdot (y_i - t_{o,i}) \cdot y_i \cdot (1 - y_i) \cdot y_j \end{aligned} \tag{4.8}$$

We make use of the equation $y' = y \cdot (1-y)$ for the logistic sigmoid function defined earlier. It is extremely easy to evaluate this derivative, considering

that the output values of the neurons are usually stored for the next step when calculating the network's output.

Next, let us compute the adjustment coeficients for the weights in the layer below the last one, denoted as Δw_{jk}.

$$\begin{aligned}\Delta w_{jk} &= -\alpha \cdot \frac{\partial E}{\partial w_{jk}} \\ &= -\alpha \cdot \sum_{i=1}^{I} \frac{\partial E}{\partial y_i} \cdot \frac{\partial y_i}{\partial \zeta_i} \cdot \frac{\partial \zeta_i}{\partial y_j} \cdot \frac{\partial y_j}{\partial \zeta_j} \cdot \frac{\partial \zeta_j}{\partial w_{jk}} \\ &= -\alpha \cdot \sum_{i=1}^{I} (y_i - t_{o,i}) \cdot y_i \cdot (1 - y_i) \cdot w_{ij} \cdot y_j \cdot (1 - y_j) \cdot y_k\end{aligned} \tag{4.9}$$

Computing these values would be time-consuming, more so for even lower layers. However, we can use already obtained values, in particular the $\frac{\partial E}{\partial y_i} \cdot \frac{\partial y_i}{\partial \zeta_i} = (y_i - t_{o,i}) \cdot y_i \cdot (1 - y_i)$ part from 4.8. This value is commonly denoted δ_i and called the *error term* of neuron i. By reusing these values from 4.8, we can simplify 4.9 to

$$\Delta w_{jk} = -\alpha \cdot \sum_{i=1}^{I} \delta_i \cdot w_{ij} \cdot y_j \cdot (1 - y_j) \cdot y_k \tag{4.10}$$

Further optimization can be achieved by reusing the δ_i value, since it can be calculated only once and then used for determining changes to all weights connecting the neuron i to the lower layer.

If we consider the equation for Δw_{jk} in the next lower layer, we can again reuse previously obtained values, which will be

$$\delta_j = \sum_{i=1}^{I} \delta_i \cdot w_{ij} \cdot y_j \cdot (1 - y_j) \tag{4.11}$$

At this point, we have determined the values Δw_{jk} using error factors of neurons from the layer N and the outputs of neuron at the layer N-1. At the same time, we have determined the error terms for neurons at layer N-1. we can therefore repeat this step for layer N-2 etc., eventually determining the weight adjustments for all weights in the network. We proceed from the output layer of the network, in the opposite direction than the input processing was carried out. Hence, the name *back-propagation algorithm*.

4.4 Faster and better learning

Training a neural network with the basic Back-propagation algorithm is generally very time-consuming. Learning simple problems can take seconds or minutes, while for more complex problems can take even hours to find an acceptable network parameters. Furthermore, learning often has to be performed several times to determine the optimal parameters of the network. Over the years of research in the field, many methods have been devised to speed up the learning process. Some of these methods can be easily added on top of the basic BP algorithm, other alter the learning process in more substantial way.

4.4.1 Momentum

One of the first adjustments to consider when looking for an improved learning performance of a neural network is *learning with momentum*. Momentum can be added to the BP-algorithm very easily, and usually helps to speed up convergence of the network's weights. The aim of this method is to minimize the perturbation of the weights caused by inconveniently shaped error function. If the error function forms a shape of a narrow "valley", then the learning gradient doesn't point to the function's minimum, but rather to the other side of the valley. This leads to oscilations of the weight vector, thus slowing down the convergence of network.

Momentum constitutes a simple solution to the problem of avoiding these oscilations. Instead of relying solely on the gradient of the error function when computing change of a weight w_i in step s, we add the change of this weight computed in the previous step $s-1$. Formally, we rewrite the equation 4.7 to

$$\Delta w_i^s = -\alpha \cdot \frac{\partial E}{\partial w_i} + \gamma \cdot \Delta w_i^{s-1} \tag{4.12}$$

where γ is the *momentum rate*, or the rate of mixing the current and the previous change of the weight. This causes attenuation of oscilations, and helps the network to converge faster towards minimum in a narrow valley. Also, it helps the network to get out of flat plateaus of the error function, where the error function's derivative is close to zero.

4.4.2 Altering the derivative

The change of the weights of a neuron depends on the derivative of the neuron's output function, as can be seen in 4.8 and 4.9. If the neuron's output gets close to 0.0 or 1.0, derivative of the output function comes close to zero, which causes learning of this weight to slow down. Because this derivative is included in the error term δ_i of the neuron, it also influences all neurons in lower layers, in the worst case scenario causing the learning process to freeze for a large part of the network.

One of the solutions to this problem, proposed by Fahlman in [3], is very simple - we can alter the derivative's value to stop it from going to zero. This can be done either by clipping, where we demand the derivative to be always larger than a certain value, setting it to this value if it gets lower, or by adding a small offset factor to the derivative. This prevents the learning from freezing in the flat areas. Drawback of this technique is that it changes the direction of the gradient vector slightly. However, despite this drawback, it has been shown in [3] that this simple alteration to BP algorithm can speed it up by a factor of magnitude.

4.4.3 Altering the learning rate

In the basic version of the BP algorithm, a constant learning rate α is used to determine the step size of all weights. It is difficult to find an optimal value of α, as for too small α the learning slows down considerably, while too big α may cause the learning algorithm to step over the minimum of the learning function. Moreover, different problems and different network topologies have different optimal values of learning rate. There are also algorithms that use a different learning rate for each weight, several of them are examined in this section.

The general idea of the algorithms for adjusting the learning rates is the same as the idea of learning with momentum. We want to make larger steps if we are moving in the right direction. At the same time, we want to slow down when there are indications we are going too fast, like oscilations. Algorithm proposed by Almeida is a simple implementation of this principle. The algorithm works with separate learning rates for each weight. Let $\alpha_i^{(m)}$ be the learning rate for weight w_i in step m, and $\nabla_i E^{(m)} = (\partial E/\partial w_i)^{(m)}$ be

the partial derivative of the error function with respect to the weight w_i in step m. In step $m+1$, learning rate α_i will be adjusted according to the following rule:

$$\alpha_i^{(m+1)} = \begin{cases} \alpha_i^{(m)} \cdot u & \text{if } \nabla_i E^{(m)} \cdot \nabla_i E^{(m-1)} \geq 0 \\ \alpha_i^{(m)} \cdot d & \text{if } \nabla_i E^{(m)} \cdot \nabla_i E^{(m-1)} < 0 \end{cases} \tag{4.13}$$

where u is the acceleration rate and d is the deceleration rate, satisfying the conditions $u > 1$ and $d < 1$. At the begining, the learning rates $\alpha_i^{(0)}$ are initiated with small values. It is obvious from these equations that both the acceleration and deceleration is exponential, and this fact can present a problem if too many acceleration steps are taken.

The delta-bar-delta algorithm, proposed by Jacobs, addresses this issue by accelerating linearly while keeping the deceleration exponential. Learning rates adjustments are in this case carried on according to the following formulas:

$$\alpha_i^{(m+1)} = \begin{cases} \alpha_i^{(m)} + u & \text{if } \nabla_i E^{(m)} \cdot \beta_i^{(m-1)} > 0 \\ \alpha_i^{(m)} \cdot d & \text{if } \nabla_i E^{(m)} \cdot \beta_i^{(m-1)} < 0 \\ \alpha_i^{(m)} & otherwise \end{cases} \tag{4.14}$$

where $\beta_i^{(m)}$ is a floating average of partial derivatives $\nabla_i E$:

$$\beta_i^{(m)} = (1-\phi)\nabla_i E^{(m)} + \phi\beta_i^{(m-1)} \tag{4.15}$$

The constant ϕ determines the rate of deterioration of the floating average. The differences compared to Almeida's algorithm are meant to help avoid oscilations. However, this modification forces the user to set up yet another constant influencing the learning speed.

4.4.4 Avoiding expensive calculations

It is also possible to speed up the BP algorithm by avoiding time-expensive floating point operations. The main source of these operations is computing the activation function of the neurons. One way to avoid these calculations is creating a look-up table with the values of the activation function, and

using these values to compute piecewise linear approximation according to formula

$$f(x) = s(x_i) + \frac{s(x_{i+1}) - s(x_i)}{x_{i+1} - x_i} \cdot (x - x_i) \tag{4.16}$$

where x_i are the points for which the function's values are stored in the table, $s(x_i)$ are the stored values, and $x_i \leq x \leq x_{i+1}$. By this, we have reduced the computation of the activation function to several table lookups, multiplication and addition. The approximation is more exact if the points x_i are more densely distributed. Since the commonly used activation functions converge to their limits fairly quickly, the x_i points are most important in the around 0. Areas far away from 0 can be approximated by the function's limits.

Chapter 5

Kohonen's neural network

Since I am using Kohonen's neural networks for training certain stages of other models, I will describe this model in more detail. Also called *self-organising map* (SOM), this model was proposed by Teuvo Kohonen. It processes input patterns without having any desired output assigned, and adjusts its nodes so as to describe the distribution of these patterns in the input space.

5.1 Definition

The network consists of a set of nodes, or neurons, each of which is characterized by its *weight vector*. These vectors are of the same size as the input patterns presented to the network. The reason for this is that the neurons' weight vectors represent points in the input space. Furthermore, the neurons are organized into a grid of an arbitrary but fixed form. The most common form is probably a 2D lattice. However, grids with more dimensions or with a different topology can be used, too. This grid defines the so-called topological neighborhood of each neuron, and these neighborhoods are then used by the learning algorithm.

We denote the weight vector of the i-th neuron as w_i. Since a neuron is defined by its weight vector, I will use the terms "neuron" and "weight vector" interchangeably. The notation $N(w_i, n)$ will be used to specify the neigborhood of the neuron w_i with the width n, where the width is the

maximum difference in grid coordinates along any grid axis. For example, in a 2D square grid the neigborhood $N(w_i, 1)$ contains 9 neurons including w_i, considering w_i is not on the edge of the grid.

The learning algorithm used with Kohonen network uses a *neighborhood function* $\nu(w_i, w_j, t)$, which determines the strength of the connection between neurons w_i and w_k. The function's value depends on their position in the grid and on the number of performed learning steps, specified by the function's third argument t. The value of ν gets smaller with growing distance between the neurons in the grid and with t. Another parameter of the learning process is the learning rate $\alpha(t)$, also getting smaller with rising number of performed learning steps.

5.2 Learning algorithm

The learning algorithm of the Kohonen network can be described in the form of a few simple steps:

1. initialization: set the n-dimensional weight vectors $\vec{w}_1$ .. $\vec{w}_m$ randomly, set $t = 0$
2. present a randomly chosen vector $\vec{x}_t$ from the training set to the network
3. find a neuron with the weight vector $\vec{w}_i$ closest to the input vector, i.e. such a $\vec{w}_i$ that $\|\vec{w}_i - \vec{x}_t\| \leq \|\vec{w}_k - \vec{x}_t\| \forall k \in 1..m$
4. update neurons' weights using the neighborhood function, learning rate and the following rule:

$$\vec{w}_k^{(t+1)} = \vec{w}_k^{(t)} + \alpha(t) \cdot \nu(\vec{w}_i^{(t)}, \vec{w}_k^{(t)}, t) \cdot (\vec{x}_t - \vec{w}_k^{(t)}), \forall k \in 1..m \quad (5.1)$$

5. test stop conditions, if they do not apply, increment t and continue with step 2

This way, the neurons in the affected neigborhood around the "winning neuron" determined in step 3 are drawn towards the presented input vector. As

the learning proceeds, a neurons converges to the area which it will later represent. Presumably, vectors in this area form a cluster distinct from other vector clusters in the input space.

The functions $\nu(w_i, w_j, t)$ and $\alpha(t)$ can have arbitrary arbitrary form, as long as they have the following properties. They have to have a decreasing tendency, and they have to yield positive values for $t \in \Re^+$. Furthermore, the speed of their declining should not be too fast, to give the network the time necessary to fully unfold and fit to the data. The declining speed should not be too slow, to force the network to converge in reasonable time. In my implementation, I use the following form of these functions:

$$\alpha(t) = \sqrt{e^{-P \cdot log^2 t}} \tag{5.2}$$

$$\nu(w_i, w_j, t) = \begin{cases} 1 & \text{if } D_{Chebyshev}(w_i, w_j) < d_{avg} \cdot \alpha(t) \\ 0 & \text{otherwise} \end{cases} \tag{5.3}$$

where d_{avg} is the averaged dimensions of the grid of neurons. The metric $D_{Chebyshev}$ is the Chebyshev distance, defined as $\max_k(|c(w_i)_k - c(w_j)_k|)$ where $c_(w_i)$ is a vector of coordinates specifying the position of the neuron w_i in the grid. The parameter P then controls the speed at which the functions diminish, in all of my tests the value of P was set to 0.25. This value, as well as the form of the functions described above, were determined heuristicaly, after trying out different variations.

5.3 Properties of Kohonen's networks

The key property of Kohonen's networks is *self-organisation*. It is also the main cause of their popularity, and the reason I am using this network model for data processing.

Using only the presented data samples, the network is able to adapt itself to describe the samples' distribution in the input space. For example, if the data samples are distributed evenly in the input space, then the network's grid will also evenly cover the input space, with each neuron representing an area of roughly the same size. On the other hand, if the data samples are generated by different distribution, like Gaussian normal distribution, then the network will accomodate to this fact. The grid of neurons will then be

deformed to cover the areas with high sample density more thickly. Those areas will then be more "described" by the reaction of network's neurons.

Different situation occurs when there is a larger number of sample clusters in the input space, and a small Kohonen's network is used. In this case, each cluster will attract one or more neurons. These neurons will then have the strongest reaction to samples from this cluster, thus representing the cluster. The Kohonen's networks can therefore be used for clustering the input space.

In either case, the network creates a chart of the input space, dividing it into regions represented by a single neuron. If we have a large set of unlabeled data and only a small set of labeled samples, we can use the Kohonen's network to create such a chart. From this chart, we can then obtain information about similarity between samples.

Another interesting property of Kohonen's network is the *topology preserving*. Two data samples that are close to each other in the input space are likely to be represented by neurons that are close to each other in the network's grid. Similarity of two samples can therefore be hinted at by the distance of neurons representing these samples in the network's grid. However, this implication cannot be reversed, because the network can converge to a state with knots in the lattice or any other pathological state.

Chapter 6

Convolutional network model

The convolutional neural network model, proposed by Yann LeCun [8], is a hybrid network model designed specifically for visual data processing. It exhibits some exceptional property in the field of 2D pattern recognition, which is why I have decided to incorporate it into this thesis.

6.1 Background

Since the publication of the article by Hubel and Wiesel [6] about the structure of the visual cortex in a cat's brain, numerous artificial neural network architectures inspired by this article were proposed. Among the first was Fukushima's *cognitron* architecture [4], which inspired many others. LeCun's convolutional network model is one of them, embracing the idea of several types of neurons in the network, each type of neuron having its function in the pattern processing.

Another network architecture that is inspired by the Hubel's and Wiesel's discoveries is Sven Behnke's *Neural Abstraction Pyramid Architecture* [1]. This model incorporates self-organisation, backward and lateral connection between neurons, much like the original Fukushima's cognitron and advanced *neocognitron* models. The architecture is very complex and gives promising results in the area of pattern recognition.

However, when considering all of the mentioned network models, I have decided to include the convolution networks in my thesis. It involves using

interesting techniques of data processing and promising very good performance, and moreover, it is very well described in [8], thus easy to implement in comparison with the neocognitron or Behnke's Pyradim architecture. That is why I chose this model for analysis and comparison with other neural network models.

6.2 Model description

There are several key ideas behind the construction of a convolutional network. First of all, neurons in the network's layers are organized in a 2D-grid, much like pixels in the visual data presented to the network. Every neuron is connected only to a small area of the lower layer instead of the whole layer, which preserves the locality of the information. These *local receptive fields* may or may not overlap, depending on the type of operation the neuron performs.

Every neuron detects features in its receptive field, the reaction to a feature depending on the weights of the neuron's connections. We would like a feature to be detected in the same way across the whole input of the network. Therefore, we duplicate the neuron across the whole layer, forming a *feature map*. This brings us the second inovation of convolutional networks, which is called *weight sharing*. The feature map can be thought of as a grid of neurons that share one common weight vector. During the training of the network, changes that would normally be made to weights of the neurons in a feature map are summed up and applied to the shared weight vector. Every layer of the resulting network architecture consists of several feature maps, each feature map detecting a certain local feature at every position of the underlying layer.

A convolutional network utilizes several types of neurons. There are *convolutional neurons*, which perform a dot product of its weights and values in its receptive field, add a bias and apply an activation function to the result. The local dot product performed over the whole input grid is equivalent to the operation of convolution with the weight vector as its kernel, hence the name of the network model. The convolutional neuron act as feature detectors. Receptive fields of adjacent convolutional neurons overlap and differ in position by one pixel, the feature detected by this neuron is therefore looked

for on every position of the input grid.

Once the feature is detected at a certain place of the receptive plane, a layer of *sub-sampling neurons* performs local averaging to blur the exact position of the feature. While preserving the relative position of the detected features, this operation blurs their exact location, making the network less sensitive to the exact form of the processed patterns. Usually, the first few layers of a convolutional network are alternating convolutional and sub-sampling feature map layers. This composition works as an input preprocessor, where higher layers detect more complex features put together from simpler features detected at the lower layers. Moreover, this preprocessor is trained on the training data with the rest of the network.

Convolutional networks also contain layers of neurons equivalent to neurons used in multilayered perceptron networks. These neurons process all of the outputs of the previous layer. Layers of these neurons are put on top of convolutional and sub-sampling layers, functioning as a classifier of lower layer's outputs.

6.3 LeNet-5

The LeNet-5 network is a concrete example of the convolutional architecture. Its structure is showed on the Figure 6.3.

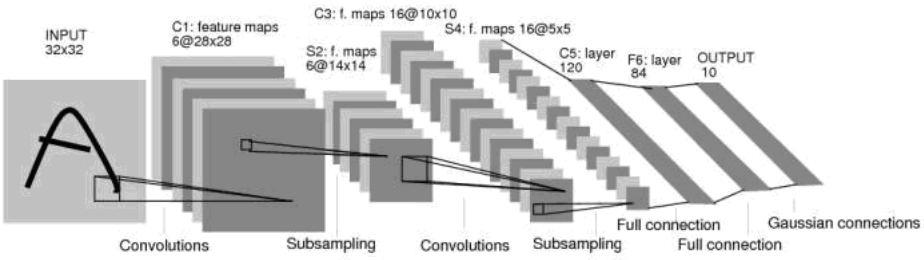


Figure 6.1: Structure of the convolutional neural network LeNet-5. The image taken from the article [8] by Yann LeCun.

The first layer of the network, denoted C1 on the image, consists of six convolutional feature maps. The receptive fields of convolutional neurons in

these maps are 5 pixels high and 5 wide. Since neighboring receptive fields overlap and size of the input is set to 32 by 32 pixels, each feature map in the first layer consists of grid of 28 by 28 neurons.

The second layers S2 contains six sub-sampling feature maps, one for each of the convolutional feature maps in the first layer. The sub-sampling neurons are connected to 2 by 2 non-overlapping input areas of the underlying feature map. Therefore, the sub-sampling feature map is half the size of its input feature map. The input values of each sub-sampling neuron are summed, multiplied by a trainable coeficient, added to a trainable bias and passed through a activation function. Since the input areas are non-overlapping, each of the sub-sampling feature map in the S2 layer has the size 14 by 14 neurons.

The third layer C3 is composed of 16 convolutional feature maps, which extract 16 different features from the layer S2. These feature maps are connected to subsets of maps from the layer S2, the subsets being of size 2 to 6. For schema of these connections, see Figure 6.2. This makes their receptive fields three-dimensional, and the asymetric connections force the feature maps to extract different features. The following layer, S4, contains 16 sub-sampling feature maps, which perform the sub-sampling operation on the maps from layer C3. Feature maps in the layer C3 have the size 10 by 10 neurons, maps in the layer S4 have therefore the size of 5 by 5 neurons.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

Figure 6.2: Schema of connections of feature maps in the layer C3 to the maps in the layer S2. The rows represent the C3 maps, column represent S2 maps, X marks a connection. The image taken from the article [8] by Yann LeCun.

The final convolutional layer C5 contains 120 convolutional feature maps, each of them connected to all sub-sampling maps in the layer S4. Because

the maps in S4 have the same size as the receptive fields of neurons in C5, each of the 120 feature maps has only one neuron.

The layer F6 contains 84 units, each connected to all 120 neurons in the layer C5. These units function in the same way as neurons in a multilayered perceptron network - they count the dot product of their inputs and their weights, add a bias value, and process the result by the activation function.

Finally, the output layer is composed of RBF (Radial Basis Function) units. Their weight vectors are set to values taken from images of the characters, and are not allowed to adapt during training. These vectors play the role of target vectors for layer F6. The used images have the size 7 by 12 pixels, which is the reason for the number of units in the layer F6. The activation of a RBF unit in the last layer is determined in the following way:

$$y_i = \sum_{j=1}^{J} (y_j - w_{ij})^2 \tag{6.1}$$

The input is then classified based on which RBF unit is activated *the least*, which means that the outputs of neurons in the F6 layer are closest to its weight vector. The reason LeCun chose this output code instead of classical "1-of-N" code is that for large number of classes, it is difficult to keep the output units zero most of the time, activating them only for the one particular class. Also, using RBF units, which are activated only in a small area of their input space, makes it easier to discard inputs which do not belong to any of the target classes.

6.4 Learning algorithm

The standart back-propagation learning algorithm, described in Section 4.3, is used to train a convolutional network. There are, however, several modifications to incorporate the weight sharing and the RBF functions with fixed weights used in the last layer of the network. First of all, the MSE error function as described by 4.5 is replaced with the following:

$$E_i = y_{D^p} \tag{6.2}$$

where y_{D^p} is the activation of the RBF unit representing the correct class for the input sample i. LeCun proposes further improvement of this error function, which also penalizes activation of the incorrect classes, and which can be used in case the weights of the RBF units are trainable.

The incorporation of the weight sharing technique is very simple. Weight changes are computed in the standard way for all virtual units in a feature map, using the equations described in Section 4.3. Then, the changes of weights which share the same parameter are added to produce the change to this parameter.

The learning algorithm for the LeNet-5 network can be upgraded with all of the learning acceleration techniques described in Section 4.4. However, the techniques must be adjusted to work with the shared weights.

6.5 Properties of convolutional networks

The utilization of techniques introduced above makes the network exhibit interesting properties when processing visual patterns. Because of the weight sharing, when a pattern is translated, the response of the first convolutional layer will be translated accordingly, but otherwise will stay the same. This, in combination with sub-sampling, which distorts the exact location of a detected feature, makes the network resistant to transformation of the input patterns.

Another remarkable feature of the convolutional networks is their ability to process data with minimum or no preprocessing. The network can be trained directly on image samples, and it learns to detect the necessary features of the presented patterns by itself. A network with fully-connected layers, such as a multilayered perceptron, applied to a real-life problem with large data samples, will have to have a tremendous number of connection in its first layer alone. Because of this, the network will need a very large training set and many epochs to be trained. To solve this problem, data undergo preprocessing prior to their presentation to the network. This helps to reduce the number of values submited to the network. In case of pattern recognition, the preprocessing usually takes form of feature extraction.

The convolutional networks manage to avoid this problem. By using weight sharing and local receptive fields they keep down the number of train-

able parameters in the network. The first few layers function as a feature detector, and since they are adapted during the training, there is no need for implementing a problem-dependent preprocessing stage. The same network architecture can therefore be used on various problems without a need to make any major changes.

6.6 Implementation

When implementing the convolutional networks, I have followed closely the model's description, with a few minor exceptions. These changes to the proposed model might result in slightly worsened performance of the resulting network. In several cases I have the possibility to compare the original and the altered implementations, and it will be interested to compare the advantages and disadvatages of the adjustements.

I have changed the operation performed by the sub-sampling neurons. The original function calculates the dot product of the neuron's inputs and its weights, adds a trainable bias, multiplies by a coeficient and then applies the activation function to the result. This is done to achieve averaging the values in the receptive field of the neuron. Instead of such a complicated operation, I implemented the sub-sampling in the form of average of the input values, without any trainable parameters or an activation function. The inputs are simply summed and the result is divided by the number of the inputs, producing the output of the neuron. Although these neurons do not adapt during the learning, they still propagate the δ_i values defined in 4.10 to lower layers. This change speeded up the learning considerably, although as a result of lowering the number of adaptable parameters in the network, it can lead to slightly worse generalization properties.

A have also exchanged the last layer of the LeNet-5 network, composed of RBF neurons, for an fully connected layer containing 10 neurons. This allowed me to use the standard "1-of-N" output coding and the MSE error function defined in 4.5. The reason for this change was to simplify the network's architecture and thus speed up the processing of the data. Also, I found that since I am testing the network on tasks with only a small number of classes (e.g. recognizing digits), I am not endangered by the problems that large number of classes may cause with sigmoid neurons. I will compare

both versions of the output layer.

Chapter 7

RBF hybrid network model

When I was researching the convolutional networks, I was intrigued with their ability to adapt very well to raw data without any need for preprocessing. For some pattern recognition problems, e.g. breaking the CAPTCHA protection, it is very difficult to design a good preprocessing by hand. It is therefore very helpful if a part of the used neural network model is trained to function as a preprocessing stage. Inspired by the convolutional network model, I tried to experiment with various ways to design a neural network model capable of having inbuilt feature detection. The RBF hybrid neural network is the result of these experiments.

7.1 Model description

The RBF hybrid model is very closely based on the convolutional neural network model introduced earlier. It also uses weight sharing and alternating layers for feature detection and sub-sampling. However, the operation used for feature extraction is different from the one used in the convolutional networks. This difference is the cause of all the divergence between the RBF hybrid model and convolutional networks.

The operation used for feature detection in the RBF hybrid model is inspired by the working of Radial Basis Function (RBF) neural networks, as described in Chapter 5 of Bishop [2]. This network model consists of two layers, the first layer occupied by RBF neurons, the second layer consists

of neurons which perform weighted sums of the first layer's outputs. The neurons in the first layer use a *radial basis function* as their activation function. Such a function uses only the distance between the input vector and the function's *center* without regard of the vector's exact value. The closer the vector is to the function's center, the greater is the function's output. Usually, the Euclidian distance is used for the norm and the Gaussian bell function for the basis function. The exact form of the used Gaussian function is described by the following equation:

$$\begin{aligned} \Gamma(\vec{x}, \vec{c}) &= exp(-\frac{\|\vec{x} - \vec{c}\|^2}{2\vec{\sigma}^2}) \\ &= exp(-\sum_k \frac{(x_k - c_k)^2}{2 \cdot \sigma_k^2}) \\ &= exp(-\zeta) \end{aligned} \tag{7.1}$$

where $\vec{x}$ is the input vector, $\vec{c}$ is the center of the Gaussian function (or the neuron's weight vector), and $\vec{\sigma}$ is the parameter controlling the exact shape of the gaussian "hat". The neuron's potential is denoted by ζ.

In the RBF hybrid network model, the RBF function replaces the convolution operation in the feature-detecting layers. Just like in the convolutional networks, the neurons in the RBF hybrid network form feature maps by duplicating one neuron with a small receptive field over the whole input plane. Such a neuron, using a radial basis function, measures the distance between the pattern in its receptive and its weight vector, and reacts stronger if its input is close to its weights. This way, the neuron functions as a feature detector, responding only to patterns similar to its *archetype*, or the pattern stored in its weight vector.

The assymetric connection scheme, used in the third layer of the LeNet-5 network and described by the Figure 6.2, is not used in the RBF hybrid networks. Introducing this connection scheme was done to break the network's symetry and force the feature maps in the third layer to recognize diverse features. This is not an issue in the RBF hybrid model, as the RBF layers are trained using the Kohonen's networks, as described in more detail in the following section. Use of the Kohonen's networks ensures that different features will be detected at the feature maps in the RBF layers of this model.

In other respects, the RBF hybrid networks have the same architecture and process input data in the same way as the convolutional networks. The first few layers of the network are alternating RBF and sub-sampling feature maps, followed by full-connected layers of perceptron units.

7.2 Learning algorithm

The function evaluated by a RBF hybrid network is differentiable, which allows the standard back-propagation algorithm to be used to train the network. The application of the Gaussian function presents a disadvantage here, since it can make the learning time-consuming due to its specific properties. The function is very sensitive to alterations of its parameters, and big changes in behavior of the network's lower layers slow the learning down considerably.

However, here lies one of the advantages of using the RBF neurons as feature detectors. Tthe weight vectors of these neurons are in direct relation to the data in the input space, in contrast to the weight vectors of convolutional neurons. Moreover, for extracting the most information from the data presented to them, the weight vectors of the RBF neurons should be placed at the points in the input space which best describe the sample data distribution. Ideally, the input samples would form clusters in the input space and the weight vectors of the RBF neurons would be placed in centers of these clusters. With this intent, a clustering algorithm can be used to find the weights for the RBF neurons.

Because of the advantages of *Kohonen's network*, I use them to find the values for the weight vectors of RBF neurons. Kohonen's networks are not aimed directly at finding the ideal complete clustering the data like the *k-means* algorithm frequently used for this purpose. They rather try to chart the data distribution, and they do it very quickly with satisfactory results. Their speed of convergence is in this case the key advantage over various clustering methods. They can be used to find the weights for the RBF neurons in the RBF hybrid networks in time which is insignificant compared to the rest of the network's training. Thus trained RBF neurons with local receptive fields are then able to react to various features of input data. Further improvement of this feature detection can be expected after applying new variants of self-organising networks, example of which is described in

Kohonen's article [7].

After determining suitable values of weights for the RBF neurons, we have two options. The values can be considered to be only heuristic initialization for further training by conventional algorithm, in this case by the back-propagation algorithm. We can also declare these values as final and keep them from being trained further with the rest of the network. This narrows down the parameters to be trained to the weights in the full-connected layers above the last RBF layer, speeding up the learning considerably. Initial experiments showed the same resulting generalization properties in both of the mentioned cases, thus making the faster method of fixing the RBF weights more favourable. As a part of this thesis, I will make further measurements to find out if this hypothesis holds true.

Because I implemented training of the RBF neurons with the back-propagation algorithm as well as with the Kohonen's networks, I needed to alter the back-propagation algorithm to work with the RBF neurons. The RBF neurons use different activation function and calculate their potential in a different way than the perceptron neurons. Namely, the Gaussian function and Euclidian distance, as described in Figure 7.1, are used. The partial derivatives used in the Figures 4.7 through 4.9 have to be altered to work with the RBF neurons. Using the chain rule and the definition 7.1, I can change the equation 4.10 in the following way:

$$\begin{aligned}
\Delta w_{jk} &= -\alpha \frac{\partial E}{\partial y_j} \cdot \frac{\partial y_j}{\partial \zeta_j} \cdot \frac{\partial \zeta_j}{\partial w_{jk}} \\
&= -\alpha \sum_{i=1}^{I} (\delta_i \cdot w_{ij}) \cdot \frac{\partial y_j}{\partial \zeta_j} \cdot \frac{\partial \zeta_j}{\partial w_{jk}} \\
&= -\alpha \sum_{i=1}^{I} (\delta_i \cdot w_{ij}) \cdot (-1) \cdot y_j \cdot \frac{\partial \zeta_j}{\partial w_{jk}} \\
&= -\alpha \cdot \delta_j \cdot \frac{\partial \zeta_j}{\partial w_{jk}} \\
&= -\alpha \cdot \delta_j \cdot (-1) \cdot \frac{(y_k - w_{jk})}{\sigma_k^2}
\end{aligned} \tag{7.2}$$

where y_j is the output of the j-th RBF neuron in the layer, produced by the Γ function defined by 7.1, y_k is the output of k-th neuron in the lower layer,

and δ_i is the error term of i-th neuron in the higher layer. The parameters σ_k can be also adapted by the back-propagation algorithm, but I did not introduce this functionality to the networks I tested.

The change of the activation function also influences the way the error terms are propagated to the lower layers. The error terms are no longer multiplied simply by the weights of the neuron when propagated, but a more complicated equation has to be used:

$$\delta_k = \sum_{j=1}^{J} (2 \cdot \delta_j \cdot (y_k - w_{jk})) \cdot \frac{\partial y_k}{\partial \zeta_k} \tag{7.3}$$

which results from integrating the Gauss function into the back-propagation equations.

7.3 Winner takes all

The convolutional layers in a convolutional network describe the measure of each detected feature at each position by a value between 0 and 1. When I was experimenting with RBF hybrid networks and had the chance to try out various alterations of the networs' behaviour, I decided to test a variation of this feature detecting method. Inspired by Kohonen's network, I implemented *winner-takes-all*(WTA) functionality into the RBF layers. Roughly described, features detected at one location compete between each other. The feature with the strongest support is marked by value 1 in the corresponding feature map at the location in question. The other features' presence is evaluated to 0. Therefore, at each location of the input plane, only one feature is detected, instead of every feature being detected to a certain degree.

This functionality is optional for the RBF hybrid networks. Preliminary experiments hint at the possibility that using the WTA technique speeds up the network's convergence during training. But there are also drawbacks of using the technique. If I was to create a RBF network with the same layer composition as the LeNet-5 network described earlier, then using the WTA would cause only one neuron in the fifth layer to fire. This is of course too little information for the higher full-connected layers to work with. I have therefore changed the type of neurons in the fifth layer to full-connected,

and added one more full-connected layer before the last to make up for the change in functionality. I use this architecture for all tests of RBF hybrid networks, whether they use the WTA technique or not, to ensure that only the configuration changes are responsible for the difference in the tests' results.

Another effect caused by using the WTA is the necessity to use the Kohonen's networks to train the weights of the RBF neurons. This is caused by the network's function no longer being differentiable, which prevents us from using the back-propagation algorithm to train these weights. However, the layers above the last RBF layer are still trainable by the back-propagation algorithm.

7.4 Properties of RBF hybrid networks

The RBF hybrid networks share many properties with the convolutional networks, and hopefully, this will lead to adequately good performance of these networks. Because of the weight sharing and bluring the position of detected features with sub-sampling, the RBF hybrid networks should have the same tolerance to transformation as the convolutional networks. Because of the feature-detecting layers working as a preprocessing stage for the higher full-connected layers, the networks are capable of working with image data with minimal preprocessing.

The main advantage of the RBF hybrid model over the convolutional model is the faster training when the Kohonens' networks are used to train the weights of the RBF layers. On the other hand, the RBF hybrid model might have worse generalization ability caused by the different operation used for feature detection. This, as well as other properties of the proposed network model, will be shown by the results of the performed tests.

Chapter 8

Implementation

As a part of my thesis, I implemented a library containing the code necessary to build neural networks with arbitrary architecture and perform various pattern recognition tests on these networks. My language of choise was Java, therefore the resulting library has the form of a JAR file. It can be found on the CD enclosed with this thesis, along with source code and scripts for running sample tests on the network models described in earlier chapters. The library is called NNL (short for Neural Network Library).

8.1 Library description

The library is divided into three packages. The package *nnl.classifiers* contains classes implementing the tested neural network models, as well as some experimentary models not described in this thesis. The key element of this package is the class hierarchy starting with the class ConvolutionNetwork and ending with the class ConvolutionNetworkFinal. The first class implements the basic functionality for building a neural network composed of layers of feature maps. Each of the layers in the network can be of a different type, the available layer types are *Convolutional*, *Subsampling*, *LinearSubsampling*, *RBF* and *FullConnected*. These layer types allow the user to compose all of the tested models of neural networks, as well as any other model which respects the requirements of the various layer types (i.e. the Convolutional layer can be put only above a layer with large enough feature maps to fit

the receptive field of convolutional neurons). Other classes in the hierarchy, i.e. *ConvolutionNetWithDeltaBarDelta* and others, add various functionality to the networks, for example learning with the delta-bar-delta technique or transformations of the presented samples. The last class, *ConvolutionNetworkFinal*, works as a wrapper for this class hierarchy, providing access to all of the necessary functionality. This package also contains the class *Logger*, which facilitates the training procedure and records all the important values measured during the training.

The second package, *nnl.datasources*, provides classes for standardized work with visual samples for recognition tasks. These samples can be loaded from various formats. Currently implemented is the IDX format, used by the MNIST database, and loading the datasources from a directory with BMP image files and a file containing the images' labels.

The third package, *nnl.tasks*, contains applications for performing test of the neural network models and various other supporting programs. The most important are classes *ScriptedTrainingTask* and *ScriptedTransformationTask*, which allow the user to perform the two types of tests with parameters loaded from a script file. Such a script file, described in detail in the Appendix A, contains all the necessary parameters for the given test. In case of the training test, the parameters include the data source specification for the training and testing sets, network's architecture specification, parameters controlling the training process and settings of the learning algorithm variations. The transformation test script contains a path to the file with saved trained neural network and ranges of transformation parameters to be applied to the presented samples.

8.2 Encountered problems

Implementing a general neural network model able to support architectures with varying layer types proved to be a chalenge, especially when the efficiency and running speed of the resulting program was important. I avoided a fully objective implementation, because my experience has proven that the overhead of frequent method calls can slow the program down considerably. For that reason, I tried to keep the code as compact and procedural as possible.

When the code complexity reached a certain level, I encountered problems with sudden drops in the speed of the program. When new code was added, the program slowed down with factor of up to five, even when the new code was not executed at all. It seemed that the compiler was having problems with optimizing the code. The IDE I was using at the time had its own implementation of Java compiler, and I was not aware of this fact prior to these troubles. After switching to an IDE which uses the original Java compiler produced by Sun Microsystems, the problem seemed to be solved.

After adding another piece of rather complex code (it was the training method using the delta-bar-delta technique), the program slowed down again, even when the code was not executed. This time, the problem was solved by splitting the main training method *learn()* into four separate methods, containing the four separated stages of a training step. My hypothesis about the cause of this problem is that the original method was too large to be held in the CPU's cache memory at once, and swapping to slower storage ensued, bringing the program's speed down. The lesson taken from these obstacles is therefore the following: while pure objective design can cause a considerable overhead, avoiding this overhead at all costs will result in troubles as well.

Graphical interface The library does not contain any graphical interface for assembling a neural network nor for setting up a test. Both of these features are planned for future development of the library, but were not necessary for the purposes of this thesis and were left out. The only graphical interface provided by the library is a tool for viewing the outputs of the network's neurons during training.

The *Network Viewer* is demonstrated on Figure 8.1. It can be opened during the training by setting the *displayViewer* option in the training script to *true* (for detailed documentation of the training scripts, see Appendix A). This viewer was implemented purely for practical purposes. I found it is a great advantage to review the network's behavior during its training. It allows the user to diagnose problems in the network architecture or implementation that are not revealed by other sources of information.

In the window of Network Viewer, you can see the sample presented to the network, followed by layers of feature maps organised into columns and ordered from left to right. The displayed activity of neurons in the feature maps is encoded in gray scale, value 1 is displayed as black, 0 as white. In the last layer, you can see the correct classification of the sample as a character

9 in the standard 1-of-N encoding. Additional information about any feature map can be viewed in the text area in the lower part of the window after clicking on the map. Aside from the feature map's type, I planned to display the map's weights. I refrained from the idea after trying it out and finding the weight vector's extract too chaotic to give any usable information by itself.

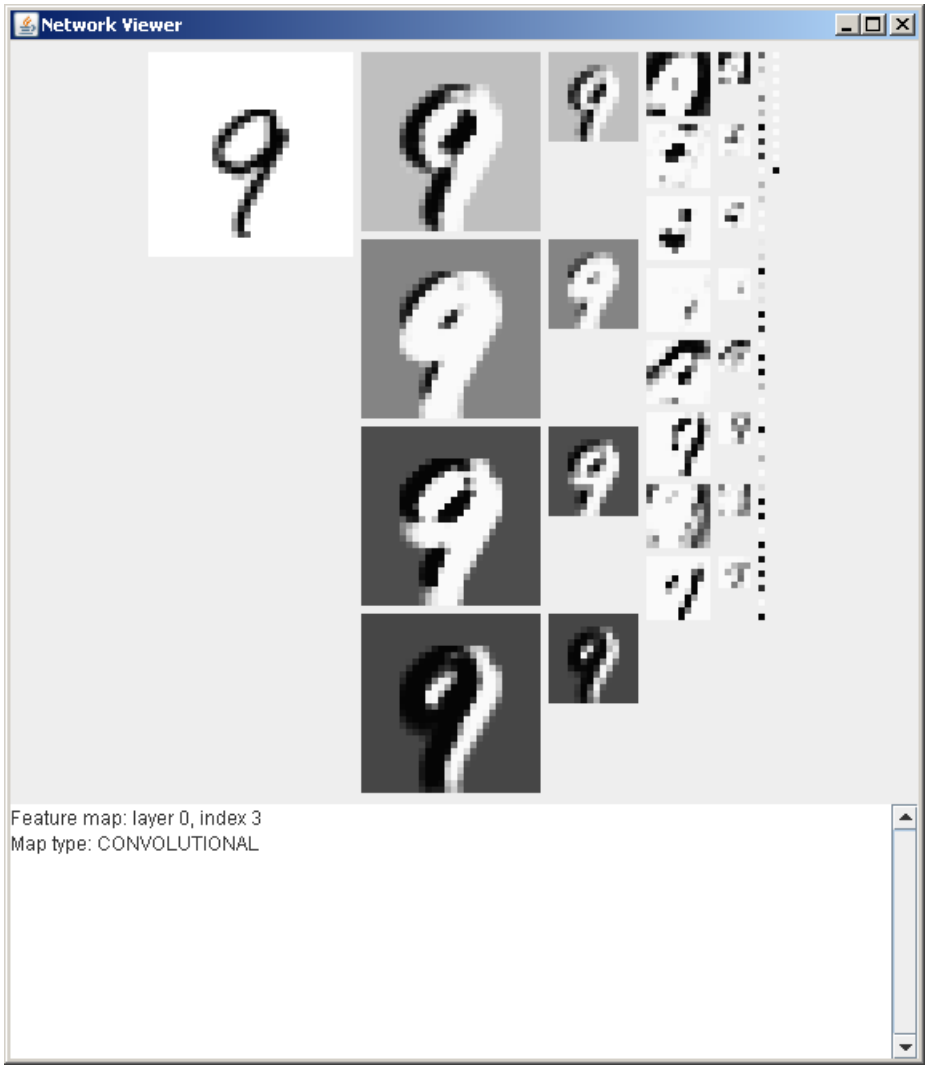


Figure 8.1: The component for viewing the outputs of the network's neurons during its learning. Only a very small network is displayed to make the image fit onto the page. The input sample is on the left, the layers of feature maps are organized into columns and ordered from left to right. The output layer is displayed as the 10 squares in the last column.

Chapter 9

Generalization and training speed

One of the first things to consider when selecting a neural network model for a pattern recognition task is the quality of *generalization* this model is capable of. Because the size of the validation set is usually very small in comparison with all possible inputs the network may encounter, we would like the network trained on the validation set to react similar samples in the best possible way. The generalization property encompasses reacting to samples with similar features in the same way, regardless of the exact form of the samples. In most of the pattern recognition tasks, and especially in the area of visual pattern recognition, we do not care about the exact values of the data samples. We rather aim to classify the samples based on their general properties, on the features they exhibit and their relative organization. A neural network for pattern recognition should learn to detect these features and to differentiate the samples based on these features.

Because of the importance of this property of neural network models, I want to compare the presented models with respect to how well they generalize the information they learn. The standard way to measure the generalization is to find the network's error rate on a set of samples the network has not been trained on. The need for this *validation set* puts an additional demand on the number of labeled data samples, but luckily the set can be much smaller than the size of the training set, as long as it consists of representative samples.

9.1 Over-fitting and early stopping

Another interesting quality the network models vary in is the speed with which they can be trained to a particular problem. This is usually measured in *epochs*, where one epoch represents one presentation of the whole training set to the network. There are many factors which influence the number of epochs the network needs to converge. Among the most important ones is the number of adaptable parameters in the network and the complexity of the problem. But there is also the topology of the network, the initial values of the adaptable parameters, used activation function, the values of the constants that directly influence the learning process, and many more. Some network models converge at a faster rate than others, and if they offer the same resulting performance, then the faster converging model will certainly be preffered.

One complication of finding out the speed of convergence of a network is determining the appropriate stopping condition. The training can stop when all of the samples from the training set have been classified correctly by the network, or when the error rate accumulated over the training set in one epoch reaches a low enough value. These conditions are widely used, however, in case of more complex problems they can have undesirable effects. One of the negative effects is called *over-fitting*, which occurs when the network is forced to reach extremely low values of the error rate. The generalization of the network then gets worse, because the network is trying to learn the exact form of the training samples. This causes the network to behave unexpectedly when presented with samples even a little different from the ones in the training set.

A much better stopping condition is based on watching the quality of the network's generalization during the training. As the network's error rate on the training set converges to zero, the error rate on a validation set decreases at first, then stops at a cerain value and fluctuates around it. The validation error rate functions as a rough estimate of how well does the network generalize. If the training set error rate gets sufficiently low, the error rate on the validation set starts to increase, signaling that over-fitting has occured. At that time, it is vital to stop the training. The technique using this stopping condition is called *early stopping*. This technique is very useful especially when the training set is very small in comparison with the number of parameters of the network, in which case the danger of over-fitting

is very high.

In my opinion, one of the most effective and perhaps the most simple way to ensure the quality of the trained network is to watch the error rate of the network when applied to the validation set of samples. Then, when the error rate has reached a new minimum, the network is saved. This approach helps is a simple solution to avoid the over-fitting problem, and does not influence the training time in a significant way. I have used this technique to get the best possible results out of this experiment for later comparison of invariance to transformations.

9.2 Test proposal

As a part of this thesis, I compare the earlier defined network models with regard to the networks' qualities described above. This test is aimed at determining which neural network model will exhibit better training performance with a pattern recognition task. The key values to be compared is the generalization error of a trained network and the number of epochs required for the network to be trained. Beside these values, I will also watch the total time required for the training to finish.

I use a handwritten digit recognition as a exemplary pattern recognition problem. Data samples used for this task are taken from the MNIST database, provided on the Internet by Yann LeCun. Several samples from this database are shown in the Figure 9.1.

The test will consist of several training sessions with fixed network model and parameter values, the results of these sessions will then be averaged to produce the final result. The parameter values and network settings will be chosen to with respect to the following point of focus:

- Considering networks of various models and their configuration containing approximatelly the same number of adjustable parameters, how well do they generalize the learned information?
- How do networks of different models and their configuration, again with approximatelly the same number of parameters, compare as to the speed of convergence during the training?

Figure 9.1: Size-normalized samples from character database MNIST.

- What is the total time required for the training to finish? While some models take less epochs to converge, the training may take longer because of more time-demanding calculations carried out by the network.

For the training and validation sets, I have used subsets of the MNIST database containing 1000 and 500 samples, respectively. These sets contain representants of the ten target classes in roughly balanced amounts. The samples are 28 by 28 pixels in size, with the character aligned in the center of the sample. The samples had to be padded by 2 pixels on each size before their presentation to the hybrid networks, to conform to the requirements of the original LeNet-5 network. The compositions of convolutional and subsampling layers in this network requires the samples to be at least 32 by 32 pixels in size, otherwise the fifth layer's input feature maps would be smaller than its receptive field.

The following network configurations were tested:

- A multi-layered perceptron model, with 4 layers, containing 120, 84, 30 and 10 neurons, respectively. The numbers of neurons in each layers were chosen to be the same as in the higher fully connected layers of

tested convolutional and RBF hybrid networks. The aim of this choice was to find out if the lower layers of the hybrid networks present any advantage over this simple classificator.

- A convolutional network similar to the LeNet-5 network, with a few minor adjustments. The sub-sampling layers were altered to perform simple averaging of values in their receptive fields, without any trainable parameters. The last layer of the network, constituted by RBF neurons with target patterns in the original network, was replaced by 10 perceptrons for easier comparison with the other models.

- A convolutional network similar to the last one, with the exception of the fifth layer, where perceptron neurons were used instead of convolutional units. Also, one more fully-connected layer was added before the last one, with 30 perceptron neurons. These adjustments were done to improve the comparison between the tested models.

- A RBF hybrid network with the same topology as the tested convolutional network described above, with the exception of using RBF neurons instead of convolutional neurons in its first and third layer. The fifth layer is composed of perceptron units fully connected to the neurons in the fourth layer, because this network model uses the winner-takes-all (WTA) functionality. Using RBF neurons in this layer would result in only one neuron in this layer to be activated, which would greatly limit the capabilities of the network. The RBF layers of this network are trained with help of the Kohonen networks, and then their weights are frozen when the higher layers are trained.

- A RBF hybrid network similar to the previous one, without the WTA functionality. Again, the Kohonen networks are used to find suitable weights for the RBF layers, and the higher layers are then trained using the standard back-propagation algorithm.

- A RBF hybrid network with topology similar to the previous two. This time, the whole network is trained with the standard back-propagation algorithm, including the RBF layers. Comparing the results of the three variations of the RBF hybrid model will show the advantages and weaknesses of each configuration.

The networks were trained by the standard back-propagation algorithm, if not stated otherwise in the above configuration description. Learning with momentum with coeficient 0.5 was used in all cases, to help avoid local minima of the error function. Each of the network was trained five times, the error rate values were then averaged over the five runs to produce the final results presented below.

9.3 Results

The results of testing the speed of adaptation and the level of generalization of the introduced networks brings only minor surprises. Since the main component of each of the networks is the multi-layered perceptron classifier and the back-propagation algorithm was used, all of the networks adapted to some extent to the presented training set. The initial speed of adaptation was high and slowed down when the network converged to a minimum of the error function. The use of the learning momentum helped the networks to escape from some of the local minima, which can be seen from the jumps of the error rate on the initial slope.

More interesting are the error rates measured on the validation set. These error rates follow the decrease of the training set error rate and converge to a certain higher value. This is very well shown on the Figure 9.2, where the validation error rate converges to a value of approximatelly 0.01. This value is a rough estimate of the network's ability to generalize, although it is not very representative of the overall success rate, as will be demonstrated in the tests described in the next chapter. This is caused by the fact that the value of error function, averaged over the set of samples and over the output neurons of the network, is only weakly related to the actual number of correctly classified samples. In spite of the lower error rate on the validation set, the number of samples correctly classified by the multi-layered perceptron is actually higher than the number of samples correctly classified by the LeNet convolutional network. The error rate of the LeNet network on the validation set converged to the value of 0.016, as can be seen on the Figure 9.3.

When examining the individual results, we can make several assumptions about the properties of the networks when applied to the pattern recognition tasks. The Figure 9.2 demonstrates the importance of using a good stopping

condition. The multi-layered perceptron network converged fairly well at first, then after about 25 epochs the over-fitting phenomenon occured and the error rates increased quickly. The early stopping technique, if used, would stop the learning at this moment. I was saving the state of the network frequently, so I was able to retrieve undefiled state of the network usable in the next experiment.

The tested convolutional networks passed the test very well. The change of type of the neurons of the fifth layer from convolutional to perceptrons did not cause much of a difference in the results, the next test will show if the networks differ in their invariance to transformations.

Much more interesting situation arose among the RBF hybrid networks. The network with RBF weights trained with Kohonen networks and then refrained from further adaptation, without the WTA functionality, behaved very wildly during the adaptation. The resulting error rates, however, were not as bad as I would expect after such a chaotic adaptation process. I believe the sudden jumps of the error rate values were caused by stepping out of local minima of the error function.

The other two configurations of the RBF hybrid model behave well during the adaptation and finished with good results. The Figure 9.5 is very similar to the Figure 9.2, this is caused by basically the same classifier being adapted. The difference is caused by the presence of the preprocessing stage in the form of RBF and sub-sampling layers in the case of the RBF network with WTA functionality. The next test of invariance to transformations will show if this preprocessing stage, created automatically from the training data, represents any advantage in the pattern recognition process.

The presented graphs of error rate convergence show only the number of epochs necessary for the training of the networks. For calculating the actual duration of training, the time taken by one epoch is needed. The epoch durations were measured on one computer used to perform all of the tests, producing the following results:

- multi-layered perceptron: 6 seconds per epoch
- convolutional network: 20 seconds per epoch for both configurations
- RBF hybrid network trained with back-propagation: 32 seconds per epoch

- RBF hybrid network with frozen RBF weights: 22.5 seconds per epoch
- RBF hybrid network with WTA: 23.5 seconds per epoch

The training of the RBF weights in case of the RBF network with the WTA functionality took about 10 seconds. The RBF layers can then be used as the preprocessing stage for creating a preprocessed training set. The rest of the network is then reduced to training a multi-layered perceptron, which would take even shorter time than training of the stand-alone multi-layered perceptron network due to the lower dimensionality of input.

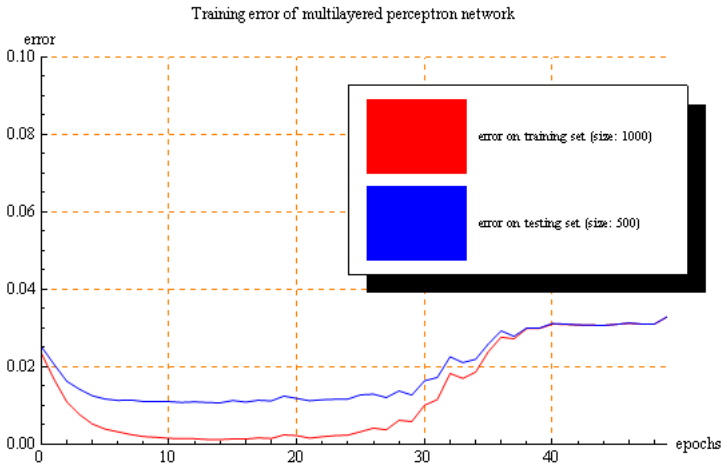


Figure 9.2: Error rates of the multi-layered network on the training and the validation sets during the training.

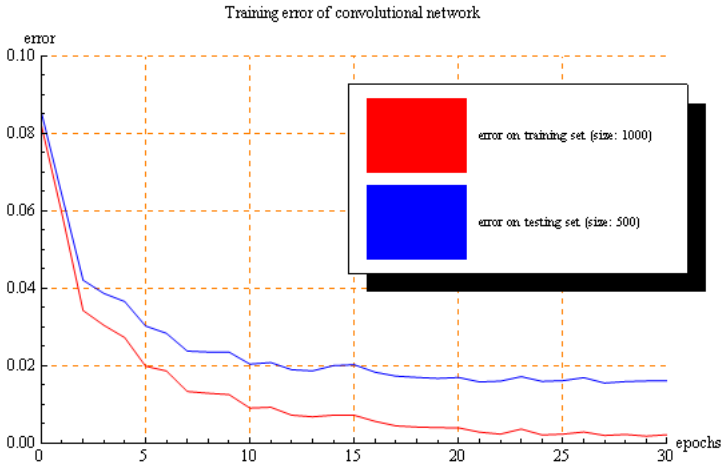


Figure 9.3: Error rates of the convolutional LeNet-like network on the training and the validation sets during the training.

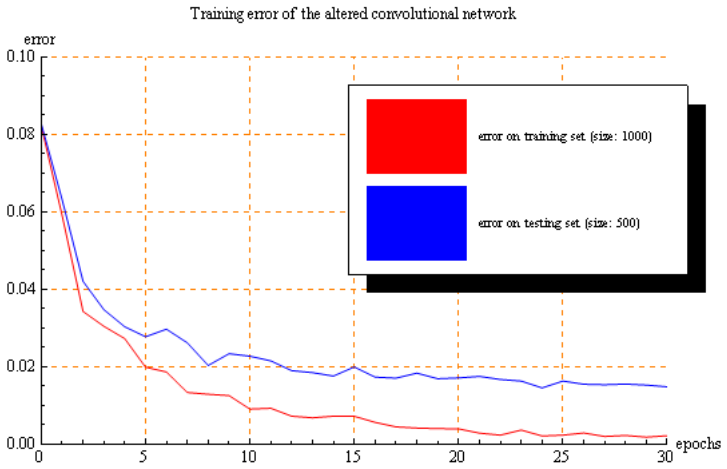


Figure 9.4: Error rates of the convolutional network with the altered architecture on the training and the validation sets during the training.

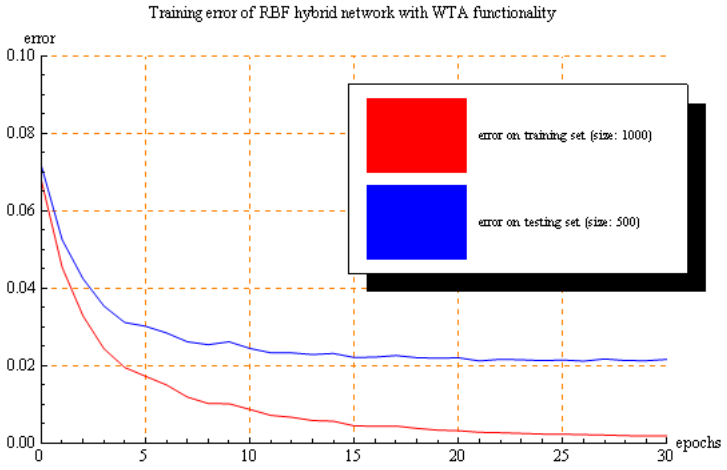


Figure 9.5: Error rates of the RBF hybrid network using the WTA functionality on the training and the validation sets during the training.

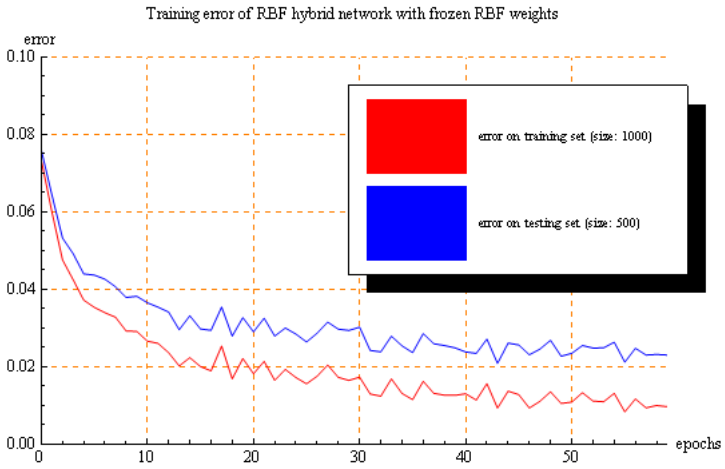


Figure 9.6: Error rates of the RBF hybrid network with frozen RBF weights on the training and the validation sets during the training.

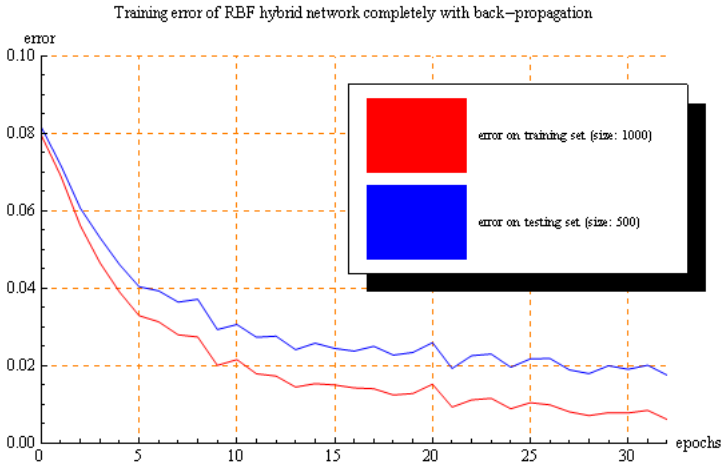


Figure 9.7: Error rates of the RBF hybrid network on the training and the validation sets during the training using only the back-propagation algorithm.

Chapter 10

Invariance to transformations

One of the most important properties of a classifier for a pattern recognition task is its invariance to transformations of input. The input patterns to be classified are usually subject to many influences which are hard to predict or to handle in the preprocessing stage. The classifier should be insensitive to these influences, which can alter the exact form of the presented pattern without changing the class it belongs to.

When considering a visual pattern recognition task, the usual set of transformations include translation, rotation and scaling of the classified samples. In addition, the samples are often distorted by noise. A good classifier for such a task should therefore classify a sample without regard to these influences, as long as their effect on the sample is inside some reasonable boundaries.

The preprocessing stage of the classification is aimed at minimizing the effect of transformations on the input presented to the classifier itself. In some cases, this can be done very effectively, and complete removal of the effects of undesirable transformations can be ensured. In other cases, removing the effects of transformations and turning the sample to a standardized form is next to impossible. It is then up to the classifier to generalize the learned information and classify the sample correctly despite its differences from the learned patterns.

I expect this test to reveal the real value of the introduced models of neural networks for the task of visual pattern recognition. When it is possible

to design a good preprocessing stage, even a simple classifier can be used with acceptable results. On the other hand, when designing a quality preprocessing constitutes a very difficult problem, for example when redeeming the sample's transformation is not possible, then using a sofisticated classifier is necessary. I expect the hybrid networks described in earlier chapters to prove their quality in such a case by having high success rates in this test. This should be shown by the comparison with multilayer perceptron, which is expected to have very low success rates when presented with the transformed samples.

10.1 Test proposal

Testing the neural network models for their invariance to transformations consists of measuring their success rate when presented with appropriatelly deformed samples. I took representatives of all introduced network models, trained as a part of the previous test. Then I presented to them a set of 1000 samples from the MNIST database, processed by one of the tested transformations, and measured the success rate of the network. The following sample alterations were applied, one at a time:

- Translation by up to 3 pixels along both X and Y axes
- Rotation ranging from -20 to 20 degrees
- Scaling with factor ranging from 0.7 to 1.3
- Additive Gaussian noise with amplitude of up to 0.4
- Salt-and-pepper noise with probability of up to 0.4

The results of these tests are displayed by graphs in the following section. The translation tests are displayed in the form of a 2D colored matrix, where position in the matrix specifies the applied vector of transformation. The zero transformation is in the center of the grid. Failure rate for each tested translation vector is expressed by a color on the gray scale, darker color indicates a lower failure rate.

The results of the other experiments are demonstrated by conventional function graphs. The zero transformation is at the point 0.0 in graphs describing the rotation and noise experiments. In the scaling experiment, the zero transformation is at the point 1.0.

The transformation tests differ from the training tests in the fact that while the results presented in the training tests are the values of the error function, the results of the transformation tests are the numbers of misclassified samples from the tested set, divided by the number of samples in the set. For that reason, these values are much more representative of the quality of the tested classifiers. To distinguish between the two ways of measuring the network's success, I use the terms "error rate" for the error function measurements and "failure rate" for the ratios of misclassified samples.

10.2 Results

The multi-layered perceptron is used as a measurement standard in these tests, showing how a very simple classifier performs at the pattern recognition task. The tested hybrid networks have fulfilled the expectations to withstand the transformations better than the multi-layered perceptron. The comparison of the various configurations of the hybrid networks are far more interesting.

The convolutional networks performed very well in all the transformation tests. They managed to keep low failure rates even in spite of the high rates of rotation, scaling or noise. The translation invariance is profitable, but this transformation can be easily remedied either by centering the inspected character, or by applying the network to all possible positions in the inspected image and then selecting the most supported result. Such a technique is described by LeCun in his article [8].

The configurations of the RBF hybrid model passed the transformation tests quite well, too. Their failure rates are comparable to the results of the convolutional networks. A big surprise was the invariance to the Gaussian noise of the network using the WTA functionality in its RBF layers, as shown by the Figure 10.19. The network was completely insensitive even to the noise of amplitude of 0.3, with which other networks showed rising failure rates. This is clearly caused by the use of the WTA functionality, which hides the

random alterations of the input's pixels by marking the same neurons as the winning ones in the output of the RBF layers. The salt-and-pepper noise causes the failure rate of the network to rise in the same way as seen with the other tested networks, the WTA functionality is therefore an advantage only in case the input image contains noise which is similar to the Gaussian noise.

The lowest reached failure rate was measured on the original LeNet network, which misclassified only 73 out of 1000 samples. This result was measured when no transformations were modifying the samples. The second best success rate was measured on the RBF hybrid network fully trained by the back-propagation algorithm - only 96 misclassified samples. The other networks had the failure rate on samples without any applied transformation of approximately 11 %.

All of these failure rates can be improved by using larger set of samples for training. Tests performed with larger training sets showed improvement in the number of correctly classified samples for all of the tested models, but such tests were too time-demanding to be performed thoroughly. Furthermore, enlarging the training set does not necessarily improve the network's invariance to transformations. It is also possible to train the network with randomly transformed samples, where the basic training set is used and each sample is transformed with small random coefficients right before it is introduced to the network. This has a significant effect on the number of epoch required for the network to converge to an acceptable values of the error function, as each sample presented to the network is in fact unique. The convergence is slowed down by factor of two or more, however, the trained network should be less sensitive to transformations.

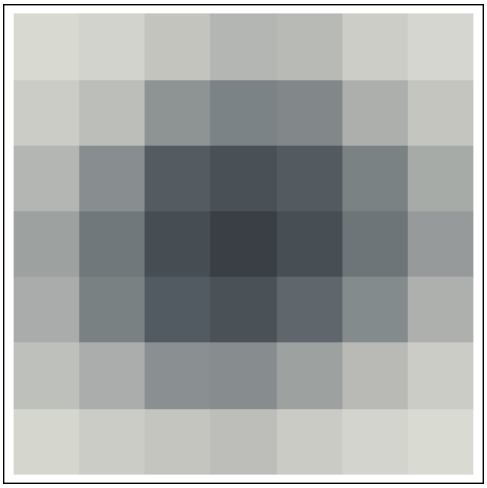


Figure 10.1: Failure rate of the multi-layered perceptron network on a set of translated samples. Darker color means lower failure rate. The value in the center is 0.116 (116 misclassified samples out of 1000).

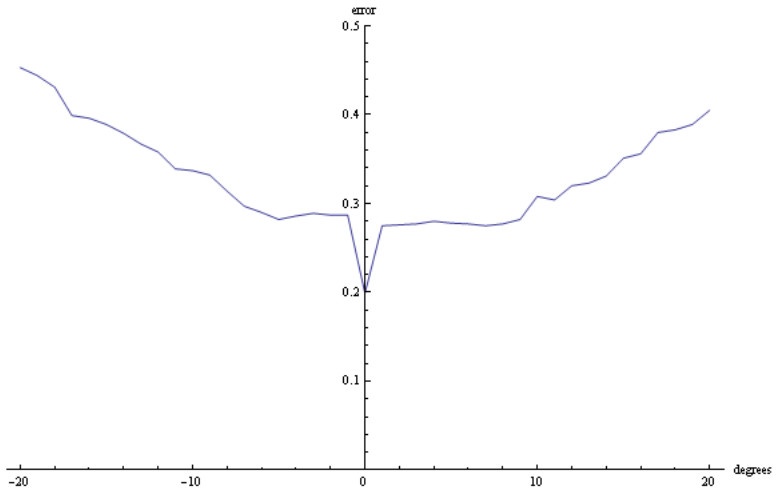


Figure 10.2: Failure rate of the multi-layered perceptron network on a set of rotated samples.

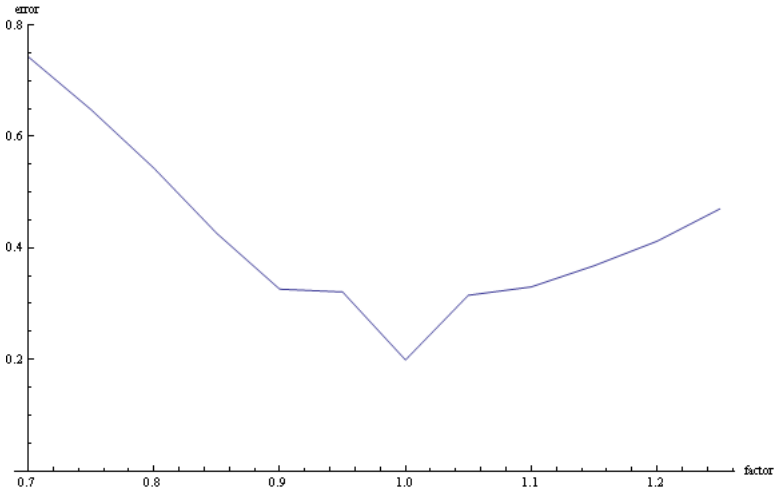


Figure 10.3: Failure rate of the multi-layered perceptron network on a set of scaled samples.

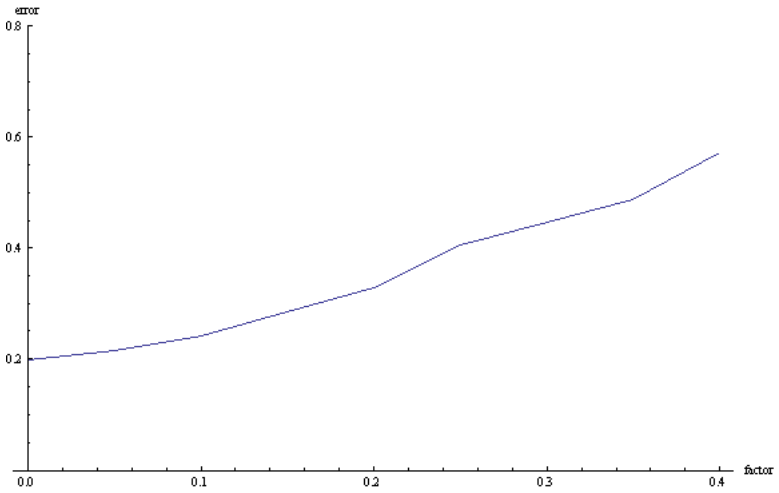


Figure 10.4: Failure rate of the multi-layered perceptron network on a set of samples with added gaussian noise of various amplitude.

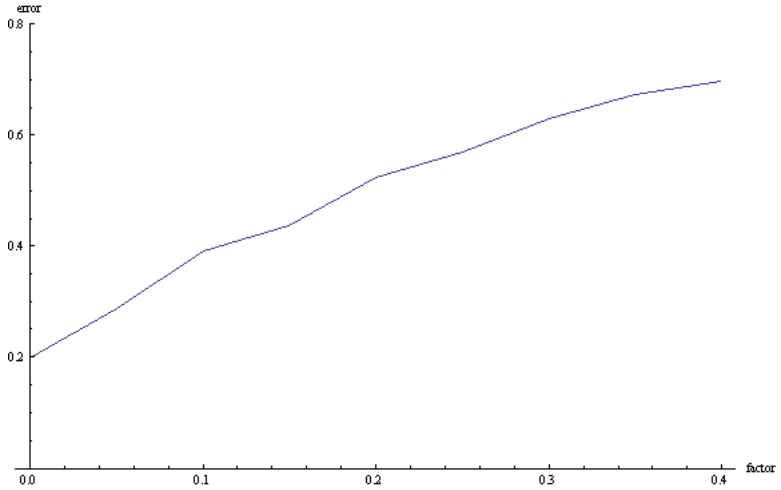


Figure 10.5: Failure rate of the multi-layered perceptron network on a set of samples with salt-and-pepper noise of various probability.

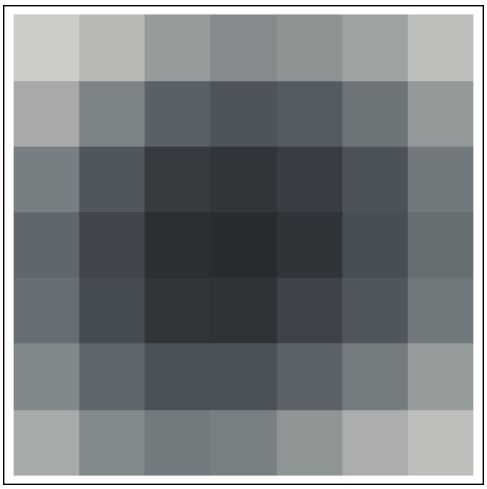


Figure 10.6: Failure rate of the LeNet network on a set of translated samples. Darker color means lower failure rate. The value in the center is 0.101 (101 misclassified samples out of 1000).

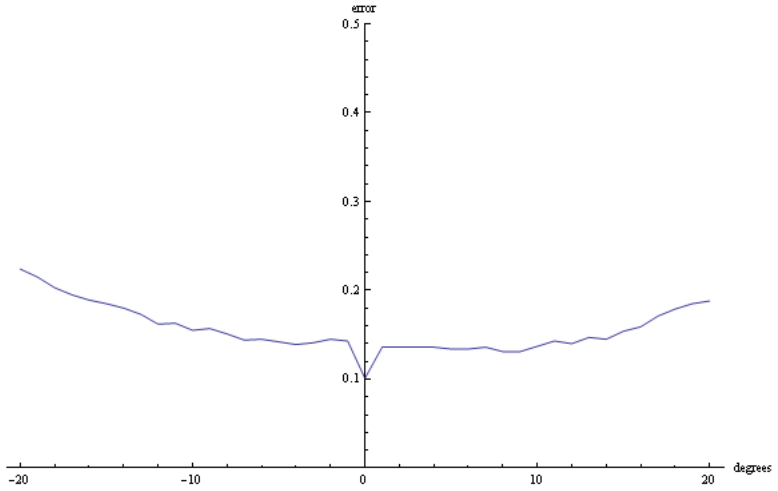


Figure 10.7: Failure rate of the LeNet network on a set of rotated samples.

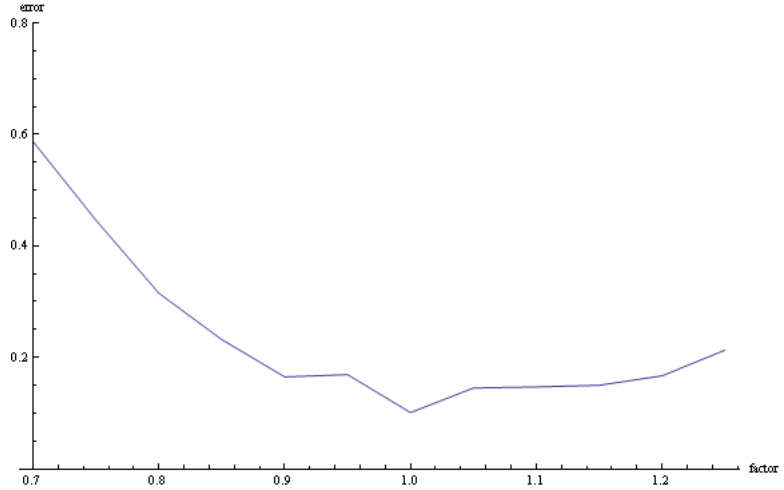


Figure 10.8: Failure rate of the LeNet network on a set of scaled samples.

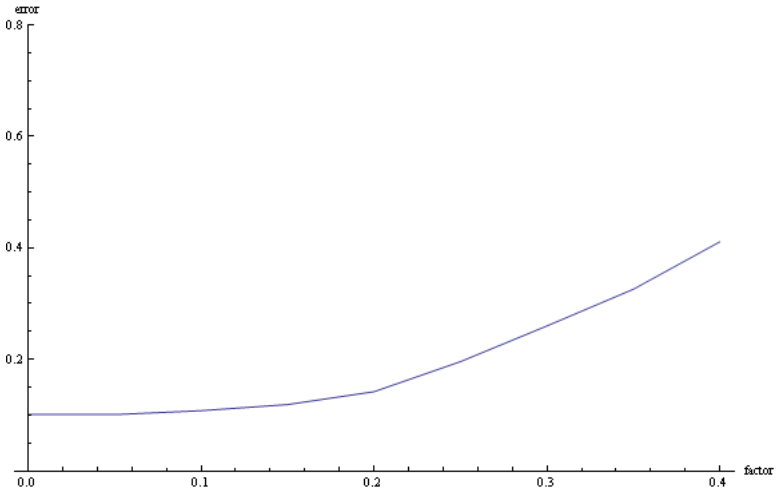


Figure 10.9: Failure rate of the LeNet network on a set of samples with added gaussian noise of various amplitude.

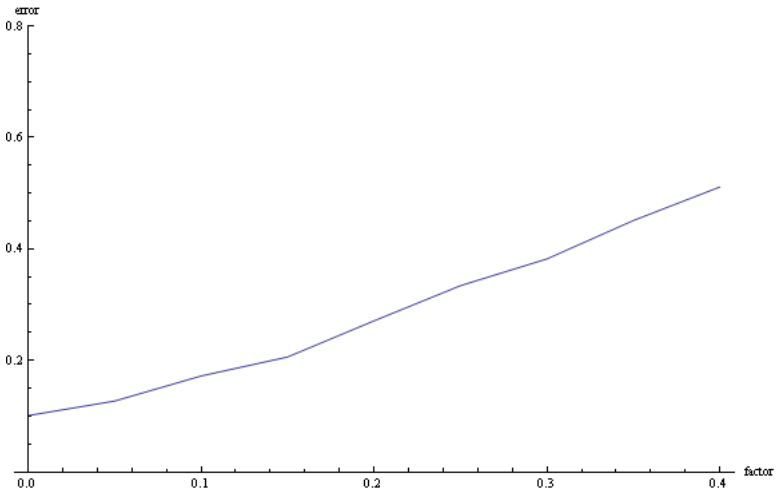


Figure 10.10: Failure rate of the LeNet network on a set of samples with salt-and-pepper noise of various probability.

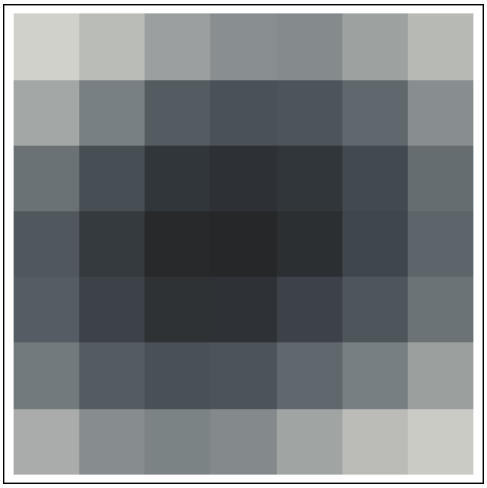


Figure 10.11: Failure rate of the altered convolutional network on a set of translated samples. Darker color means lower failure rate. The value in the center is 0.073 (73 misclassified samples out of 1000).

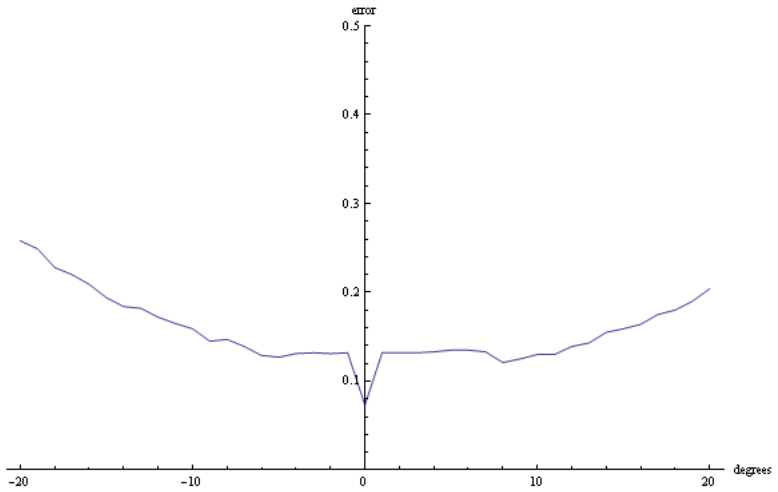


Figure 10.12: Failure rate of the altered convolutional network on a set of rotated samples.

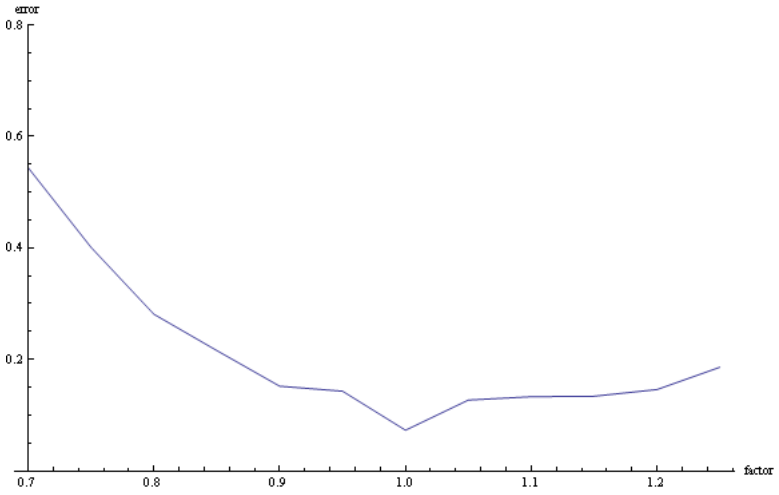


Figure 10.13: Failure rate of the altered convolutional network on a set of scaled samples.

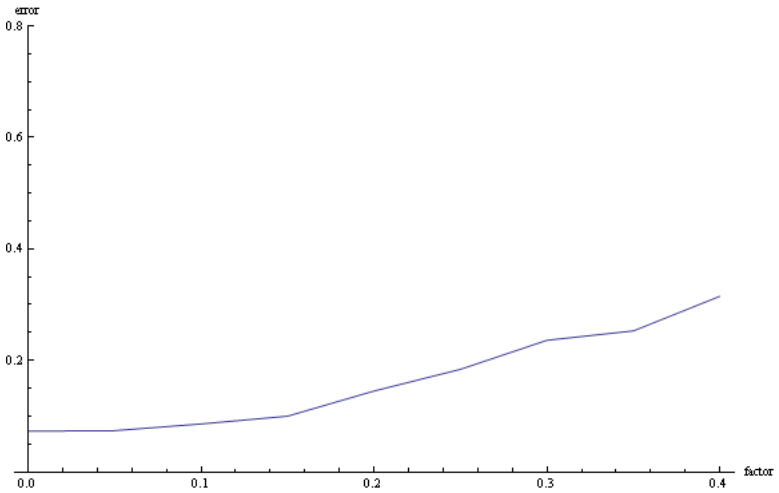


Figure 10.14: Failure rate of the altered convolutional network on a set of samples with added gaussian noise of various amplitude.

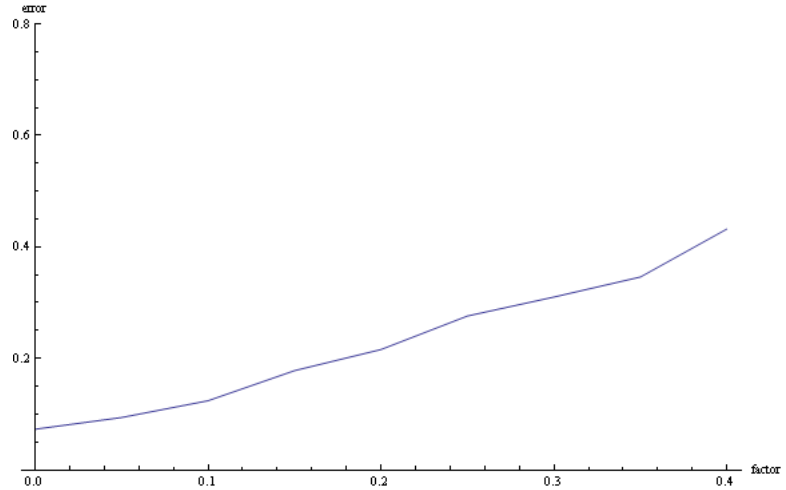


Figure 10.15: Failure rate of the altered convolutional network on a set of samples with salt-and-pepper noise of various probability.

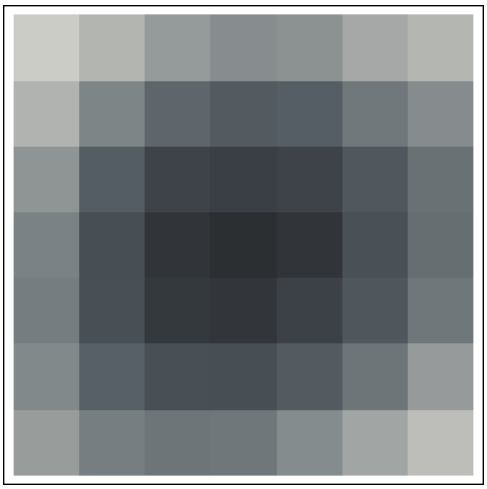


Figure 10.16: Failure rate of the RBF hybrid network with WTA on a set of translated samples. Darker color means lower failure rate. The value in the center is 0.117 (117 misclassified samples out of 1000).

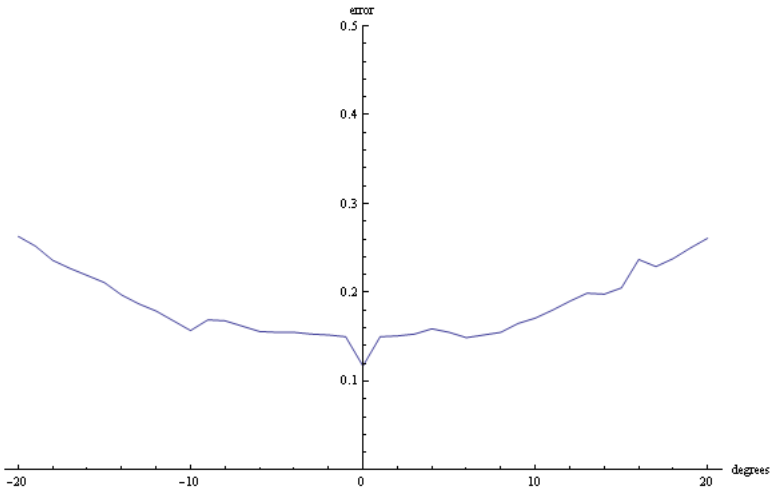


Figure 10.17: Failure rate of the RBF hybrid network with WTA on a set of rotated samples.

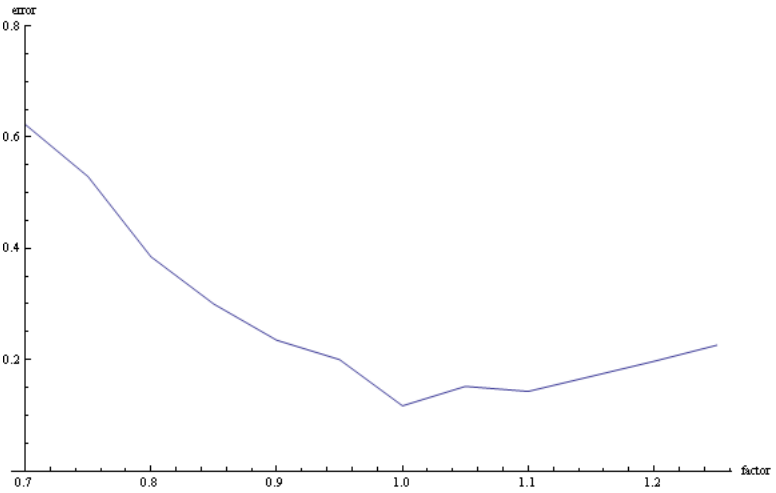


Figure 10.18: Failure rate of the RBF hybrid network with WTA on a set of scaled samples.

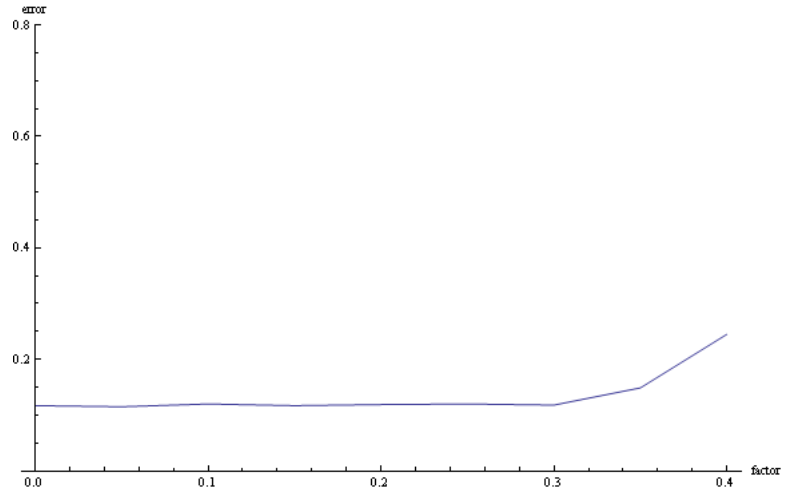


Figure 10.19: Failure rate of the RBF hybrid network with WTA on a set of samples with added gaussian noise of various amplitude.

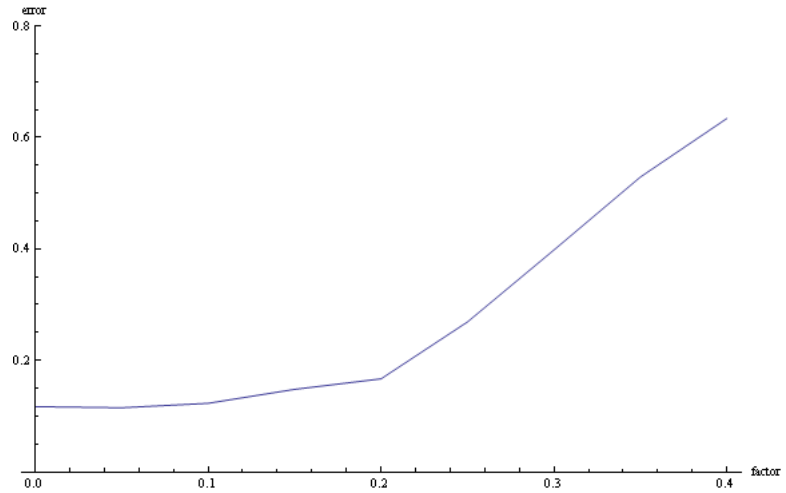


Figure 10.20: Failure rate of the RBF hybrid network with WTA on a set of samples with salt-and-pepper noise of various probability.

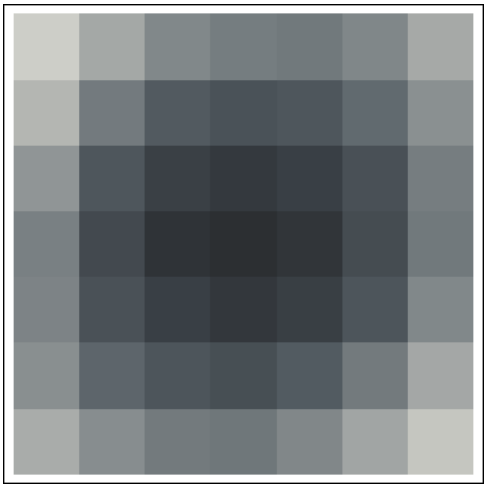


Figure 10.21: Failure rate of the RBF hybrid network with frozen RBF weights on a set of translated samples. Darker color means lower failure rate. The value in the center is 0.116 (116 misclassified samples out of 1000).

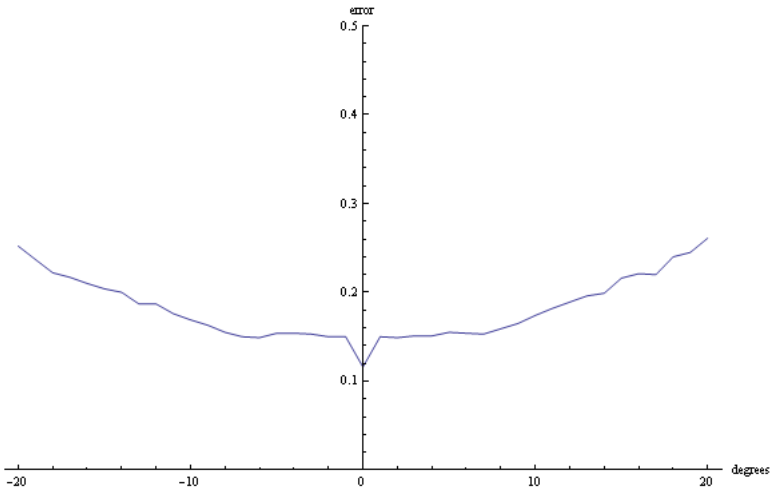


Figure 10.22: Failure rate of the RBF hybrid network with frozen RBF weights on a set of rotated samples.

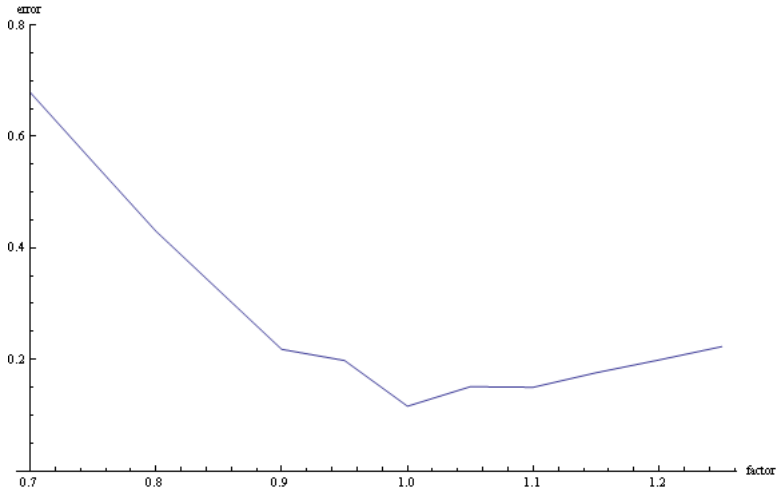


Figure 10.23: Failure rate of the RBF hybrid network with frozen RBF weights on a set of scaled samples.

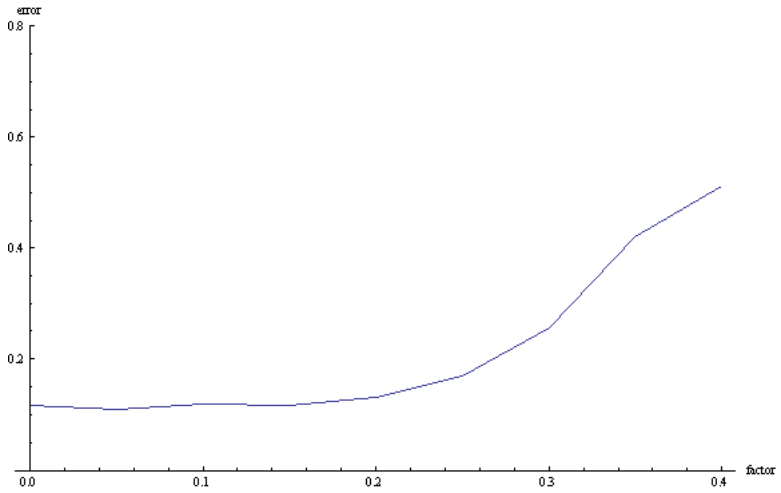


Figure 10.24: Failure rate of the RBF hybrid network with frozen RBF weights on a set of samples with added gaussian noise of various amplitude.

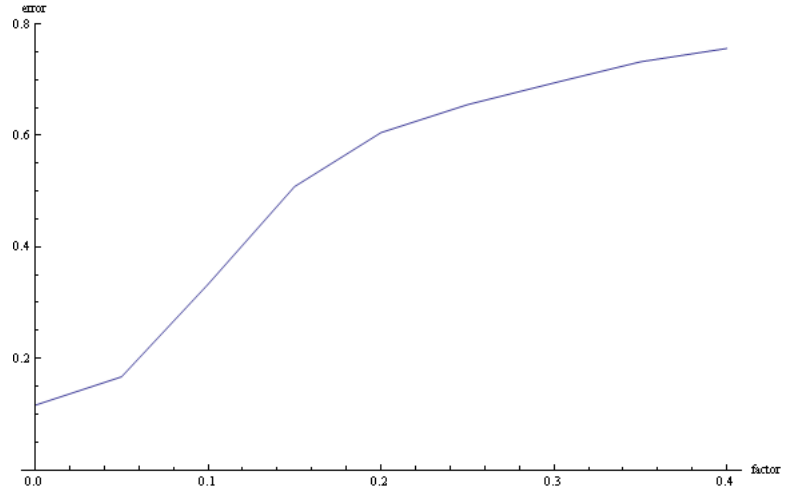


Figure 10.25: Failure rate of the RBF hybrid network with frozen RBF weights on a set of samples with salt-and-pepper noise of various probability.

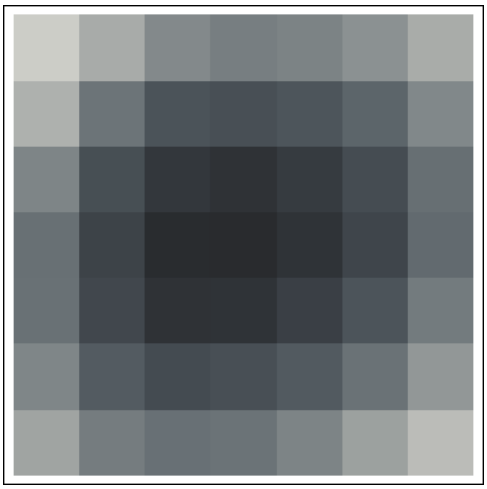


Figure 10.26: Failure rate of the RBF hybrid network trained fully by back-propagation on a set of translated samples. Darker color means lower failure rate. The value in the center is 0.096 (96 misclassified samples out of 1000).

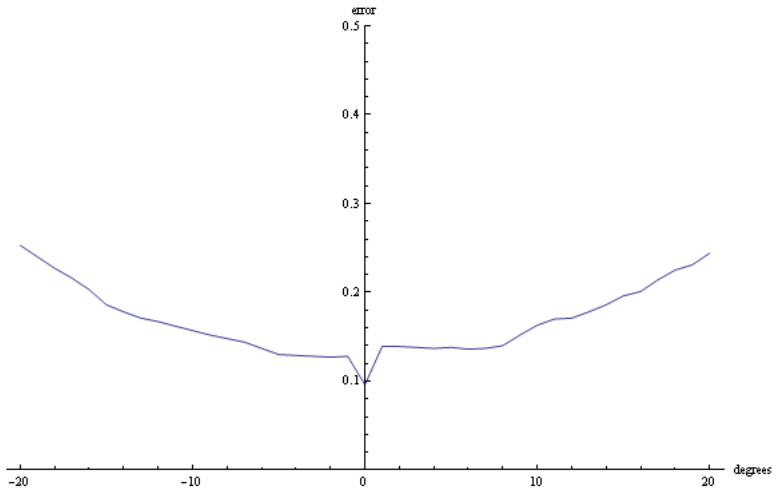


Figure 10.27: Failure rate of the RBF hybrid network trained fully by back-propagation on a set of rotated samples.

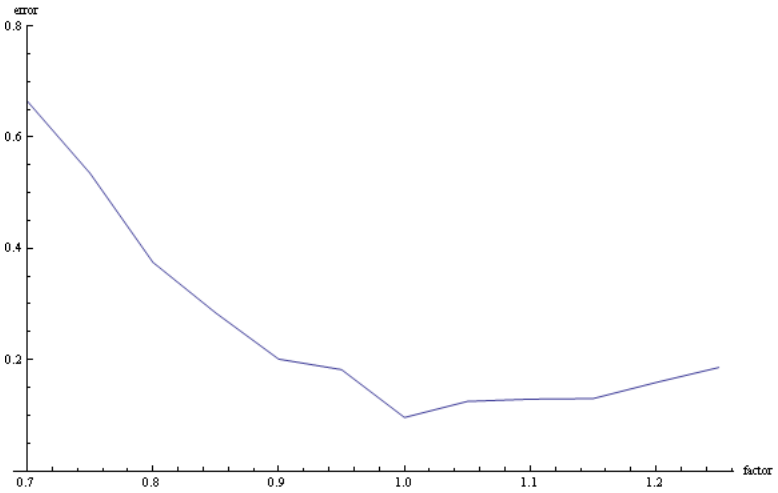


Figure 10.28: Failure rate of the RBF hybrid network trained fully by back-propagation on a set of scaled samples.

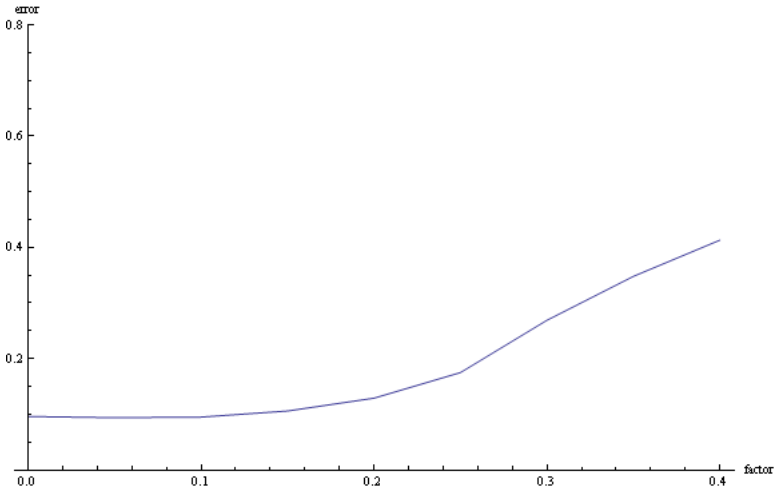


Figure 10.29: Failure rate of the RBF hybrid network trained fully by backpropagation on a set of samples with added gaussian noise of various amplitude.

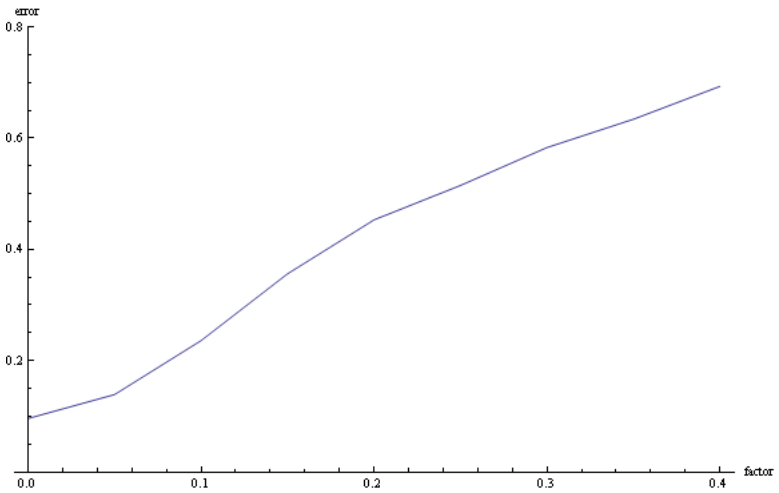


Figure 10.30: Failure rate of the RBF hybrid network trained fully by backpropagation on a set of samples with salt-and-pepper noise of various probability.

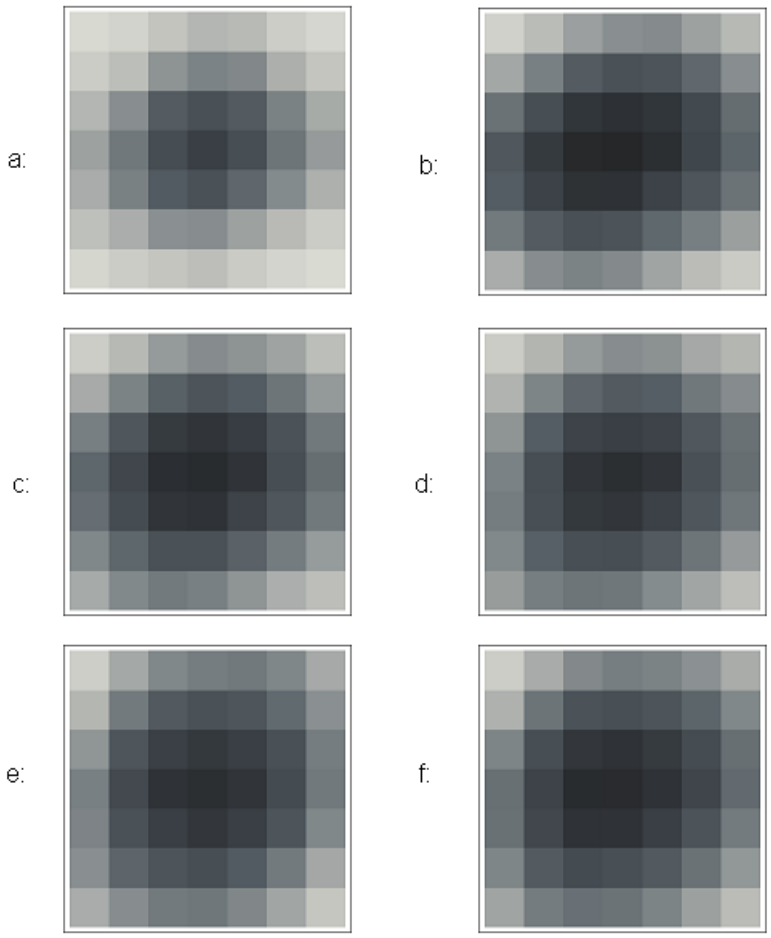


Figure 10.31: The visualization of the failure rates of the tested neural network models when the classified samples were translated. a: multi-layered perceptron, b: altered LeNet network, c: the original LeNet network, d: RBF hybrid network with WTA, e: RBF hybrid network with frozen RBF weights, f: RBF hybrid network fully trained with back-propagation.

Chapter 11

Conclusion

The aim of this thesis was to compare the suitability of several neural network models for pattern recognition tasks. To this end, the introduced models were trained to recognize hand-written digits and then tested for invariance to transformations. These tests were supposed to reveal advantages and disadvantages of neural network models of varying complexity. The samples presented to the networks were not preprocessed in any way, to simulate a recognition problem for which a custom preprocessing cannot be designed easily. The networks were therefore also tested with regard to their ability to deal with raw image data.

The problem of pattern recognition spreads over many areas of importance. It is used for automatic document processing in banks and post offices, for analysis of medical data, even for car identification in traffic control systems. Many recognition systems were designed for these purposes, varying from simple classifiers based on clustering to elaborate systems using advanced algorithms and custom data preprocessing. I have focused on the set of problems which do not allow for a effective preprocessing stage to be built. Reading hand-written postal codes or breaking a character-based CAPTCHA protection of web forms are examples of such problems.

The main obstruction of solving these problems is the unpredictability of deformation of the input data. A perfect classifier for these problems would be completely insensitive to these deformations, classifying a sample correctly in spite of any perturbations to its form. Standard classifiers do not cope with such complications very well. In this thesis, I tried to test the

insensitivity to deformations of several classifiers based on neural networks, hoping to discern a way to a better solution for this class of recognition problems.

The presented results show that hybrid networks perform much better on the raw data than a simple classifier, embodied by a multi-layered perceptron network. Parts of the presented hybrid networks function as a preprocessing stage. It can either be trained with the rest of the network by a standard learning algorithm, or automatically from the data by a data mining algorithm. Use of the Kohonen's networks is demonstrated for configuring the preprocessing stage. The hybrid networks have higher success rates and are less sensitive to transformations of the input samples when compared to the perceptron network.

Several models of hybrid networks were implemented and tested. The convolutional networks, introduced by Yann LeCun, were tested in two configuration. These networks gave the best overall performance and are recommended for general visual pattern recognition tasks. Inspired by this model, I introduced its variant which uses neurons with radial basis activation function instead of the original convolutional units. This model allows for the fast automatic initial training using the Kohonen's networks, which is the main advantage of this model over the convolutional networks. I denoted this model the *RBF hybrid neural network*.

Several configurations of the RBF hybrid network were tested. Although the model did not reach the success rates of the convolutional networks, it still performed considerably well in the tests. One of the configurations, using the winner-takes-all functionality, proved to be almost completely insensitive to Gaussian noise added to the sample. This surprising result shows a great advantage of this network over the other models in case a noise is present in the data to be processed by the network. Along with the very fast training of the model, the RBF hybrid networks present a viable alternative to the convolutional networks in the area of visual pattern recognition.

In summary, I have shown in this thesis that hybrid networks pose an advantage over simple classifiers when a pattern recognition is to be performed on raw image data, especially when the data is subject to various transformations. Moreover, it was shown that a preprocessing stage for a given problem can be designed automatically in only a short amount of time. The use of such a preprocessing improves the precision of classificatoin and trans-

formation invariance of even a simpler classifier. When the winner-takes-all functionality is used in this preprocessing stage, it renders the recognition process insensitive to considerable amount of Gaussian noise. Other training techniques and hybrid network types which could further improve the precision of pattern recognition remain the object of future studies.

Bibliography

[1] Behnke, Sven: Hierarchical Neural Networks for Image Interpretation, Springer 2003

[2] Bishop, C. M. (1995): Neural networks for pattern recognition. Oxford University Press, New York.

[3] Fahlman, S. E. (1988): Faster-Learning Variations on Back-Propagation: An Empirical Study. *Proceedings, 1988 Connectionist Models Summer School*, Morgan-Kaufmann, Los Altos CA.

[4] Fukushima, K. (1975). Cognitron: A self-organising multilayered neural network. Biological Cybernetics, 20, 121-136.

[5] Fukushima, K.(1998) "A Neural Network for Visual Pattern Recognition," *Computer*, vol. 21, no. 3, pp. 65-75, Mar., 1988

[6] Hubel, D.H. and Wiesel, T.N. : "Receptive Fields, Binocular Interaction and Functional Architecture in the Cat's Visual Cortex", *J. Physiology*, Vol. 160, No. 1, London, 1962, 106-154

[7] Kohonen, Teuvo: "Self-organixing neural projectinos", *Neural Networks 19 (2006), 723 - 733*

[8] LeCun Y., Bottou L., Bengio Y., and Haffner P. (1998): Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, vol. 86, no. 11, 2278-2324.

[9] Rojas, R. (1996): Neural networks: a systematic introduction. Springer-Verlag, Berlin, New-York.

Appendix A

Script documentation

The implemented NNL library allows the user to run test for training a neural network and for testing a trained network's invariance to transformations. To describe the possible configurations of the training procedure in more detail, I will document the script used to control the training process. Such scripts were used to set up all of the training tests described in the thesis, and can be found on the CD enclosed with this thesis.

```
trainSetFormat: IDX
trainSetFile: data/t10k-images.idx3-ubyte
trainSetLabels: data/t10k-labels.idx1-ubyte
trainSetSize: 1000
trainSetStart: 0
trainSetPadding: 2

testSetFormat: IDX
testSetFile: data/t10k-images.idx3-ubyte
testSetLabels: data/t10k-labels.idx1-ubyte
testSetSize: 500
testSetStart: 1000
testSetPadding: 2

architecture:  custom
```

```
numLayers: 6

layer1type: convolutional
layer1size: 4

layer2type: linear_subsampling
layer2size: 4

layer3type: convolutional
layer3size: 8

layer4type: linear_subsampling
layer4size: 8

layer5type: convolutional
layer5size: 40

layer6type: fullconnected
layer6size: 10

learnParam: 0.2
maxWeightInit: 0.6
sigmoidLambda: 2.0

useMomentum: true
momentParam: 0.5

useDeltaBarDelta: false
dbdUpParam: 0.1
dbdDownParam: 0.85
dbdDerivationWeight: 0.5

enableRandomTransform: false

enableTranslation: false
enableRotation: false
enableScale: false
enableGaussNoise: false
```

```
enableSapNoise: false
setTranslationX: 3
setTranslationY: 3
setRotationDgr: 20
setScale: 0.7
setGaussNoise: 0.1
setSapNoise: 0.1

useSigmoidLookup: true

enableLogging: true
loggingInterval: 1
targetErrorRate: 0.01
exactEpochs: 20
maxEpochNumber: 60

saveOnExit: true
displayViewer: true
logDirPrefix: \\testNetworks
```

Detailed description of the configuration parameters follows. To keep this chapter within reasonable size limits, I will not describe in detail the parameters for turning on the transformations of the samples presented to the network and the parameters specifying properties of these transformations.

- *trainSetFormat, testSetFormat*: specifies the format of input data. Currently supported types are IDX and IMAGE.
- *trainSetFile, trainSetLabels, testSetFile, testSetLabels*: for the IDX format, specifies the locations of files containing the sample data and sample labels. For the IMAGE format, the values contain the path to the image files and the name of the file containing samples' labels.
- *trainSetSize, trainSetStart, testSetSize, testSetStart*: specifies the size of training and testing sets, along with index of the first image used for these sets from the whole sample database found at the location described by the previous parameters.

- *trainSetPadding, testSetPadding*: specifies the size of additional sample padding. Enlarging the samples is necessary for some of the more complex architectures like the original LeNet network.

- *architecture*: this parameter was originally meant to allow the user to specify either a particular network's architecture (i.e. LeNet), or the value "custom" followed by list of layer sizes and types. Currently, "custom" is the only supported value.

- *numLayers*: specifies the number of layers in the network.

- *layerXtype*: specifies the type of the network's layer, X is substituted for index of the layer, starting from 1. The supported types are *full-connected, convolutional, rbf, subsampling* and *linear_subsampling*.

- *layerXsize*: specifies the size of the layer with index X.

- *learnParam*: the parameter controlling the size of learning steps, denoted α in the equation 4.7.

- *maxWeightInit*: the value controlling the range for random initialization of the network's weights. Where applicable, the weights will be initially set to random values from the interval between -*maxWeightInit* and *maxWeightInit*.

- *sigmoidLambda*: the used steepness of the sigmoid function, denoted λ in the equation 4.3.

- *useMomentum*: set to "true" or "false", this parameter controls the use of the learning with momentum technique

- *momentParam*: in case learning with momentum is used, this parameter controls the portion of the last change of the weight to be added to the current change. It is denoted γ in the equation 4.12.

- *useDeltaBarDelta*: this parameter specifies whether the delta-bar-delta algorithm for altering the learning rates will be used. It can be set to values "true" or "false".

- *dbdUpParam*: denoted u in the equation 4.14, this parameter controls acceleration of change in the delta-bar-delta algorithm.

- *dbdDownParam*: denoted d in the equation 4.14, this parameter controls deceleration of change in the delta-bar-delta algorithm.
- *dbdDerivationWeight*: denoted ϕ in the equation 4.15, this parameters is used to control the speed of change of the learning rates by the delta-bar-delta algorithm.
- *enableRandomTransform*: if this parameter is set to "true", the transformation set by the following parameters are applied at random within the specified limits to the samples presented to the network. If set to "false", the enabled transformations are applied with the specified values only.
- *useSigmoidLookup*: specifies whether values of the sigmoid function will be directly evaluated, or read from a look-up table constructed beforehand.
- *enableLogging*: turns on recording important values during the training of the network.
- *loggingInterval*: specifies the interval, in epochs, in which the recorded values are read. The usual value is 1.
- *targetErrorRate*: one of the ending conditions controlling the training. If the error rate of the network on the training set is below the specified value, the training is ended.
- *exactEpochs*: another ending condition. The training will last at least the specified number of epochs.
- *maxEpochNumber*: the last ending condition. The training will be cut off after the specified number of epochs.
- *saveOnExit*: If this parameter is set to "true", the network will be saved when the training is finished or interrupted.
- *displayViewer*: If this parameter is set to "true", the Network Viewer window will be displayed during the training, showing the reaction of the network to a random sample after each epoch.

The script used to control the transformation test contains only the parameters specifying the location of the testing set, the applied transformations, and the file containing the saved network to be tested.