Tommi Kokko

Neural Networks for Computationally Expensive Problems

Master's Thesis in Information Technology

May 30, 2013

University of Jyväskylä

Department of Mathematical Information Technology

Author: Tommi Kokko

Contact information: tommi.kokko@gmail.com

Supervisors: Jussi Hakanen, and Karthik Sindhya

Title: Neural Networks for Computationally Expensive Problems

Työn nimi: Neuroverkkojen käyttö laskennallisesti raskaissa optimointitehtävissä

Project: Master's Thesis

Study line: Vaativien järjestelmien optimointi ja hallinta

Page count: 101+22

Abstract: Optimization often involves usage of a simulator, which can be computationally expensive to use. Simulators can be replaced by surrogate models, which are computationally cheaper and can be almost as accurate as the simulators. In this thesis we consider closer a surrogate model, namely neural networks. We prepare surrogate assisted optimization by building, training and validating different neural network models for a surrogate model. In addition we compare how different training data sets, which are generated by different data sampling techniques, effect the generalization accuracy of neural networks.

Keywords: surrogate model, MLP, recurrent MLP, RBF network, data sampling

Suomenkielinen tiivistelmä: Optimointiin tarvitaan usein simulaattoria, jotka voivat olla laskennallisesti raskaita. Simulaattorit voidaan korvata sijaismalleilla, jotka ovat nopeampi laskea ja voivat olla lähes yhtä tarkkoja kuin simulaattorit. Tässä työssä tarkastelemme tarkemmin yhtä sijaismalli, neuroverkkoja. Valmistelemme sijaismalli avusteista optimointia rakentamalla, opettamalla ja validoimalla erilaisia neuroverkkoja sijaismalliksi. Lisäksi vertailemme eri data samplaustekniikoilla generoitujen opetusdatojen vaikutusta neuroverkkojen approksimointitarkkuuteen.

Avainsanat: sijaismalli, MLP, takaisinkytketty MLP, RBF verkko, data samplaus

Preface

This thesis was written for practitioners, who want to apply neural networks, and need the knowledge base in a brief package. It was written in Jyväskylä University, Department of Mathematical Information Technology.

I want to thank my supervisors Ph.D. Jussi Hakkanen and Ph.D. Karthik Sindhya for guidance, support and valuable comments. Special thanks for M.Sc. Vesa Ojalehto for running the simulator for me and the rest of optimization group for great support.

Extra special thanks for my partner, she has been extra patient during my studies.

Enjoy!!

Tommi Kokko

Glossary

NN	neural network
SLP	single layer perceptron
MLP	multilayer perceptron
RBF	radial basis function
RMLP	recurrent multilayer perceptron
TF	transfer function
x	input of neural network
n	number of inputs
y	output of neural network
k	number of outputs
w	number of synaptic weights
b	bias
tp	training pattern
ep	epoch
t	center
η	learning rate
α	momentum
$\mathbb{R}$	Euclidean space
Lhs	Latin hypercube
Ham	Hammersley sampling
OA	Orthogonal array

List of Figures

List of Tables

Contents

1 Introduction

In this thesis we are studying effectiveness of different surrogates and in addition we study the dependence on different data sampling techniques. Surrogates can be used in computationally expensive optimization problems. Optimization is a systematic search for minimum or maximum values of a problem. The problem may be e.g. an industrial process or a product. Optimization is often used to improve the performance, the cost efficiency or the design of a problem. The process to be optimized may be for example, a plant [Fahmi and Cremaschi 2012; Sindhya et al. 2013] or a product to be optimized for example microprocessor [Marianik et al. 2009] or airfoil shape [Park et al. 2009].

For optimization we need a simulation model for evaluating the performance with different settings. For example a simulator, which is built to describe the process as accurately as possible. As simulators have become more and more accurate they have also become more and more expensive to operate. Here computationally expensive means computational time, which is needed for a simulator to evaluate a single performance. Hence it might take several days to complete the optimization process, because typically a large number of evaluations is needed [Park et al. 2009].

To handle computationally expensive simulators researchers have started to find out ways to decrease the computational time required for optimization. One way of doing it is to use surrogate models to replace the expensive simulators. Different surrogates models are commonly used e.g. artificial neural networks, kriging and support vector machines [Jin 2005], as they are very fast to compute. There are numerous successfully completed surrogate assisted optimization problems in literature (see e.g. [Benedetti, Farina, and Gobbi 2006; Kusiak, Zhang, and Li 2010; Simpson et al. 2001; Marianik et al. 2009; Fahmi and Cremaschi 2012].

We have chosen one surrogate model, namely artificial neural networks and let's call those just as neural networks, for a closer look, because they can be used for function approximation, classification, pattern recognition and control, among other purposes as well [Haykin 1999; Hassoun 1995]. Neural network consists of layers, namely input, hidden and output layers, which consists of computation units called neurons. Neural network takes input val-

ues and after several operations it produces an output, these are called approximation values. If the neural network is properly trained, approximation values will match the real function values. For training we need a training data, which consists of the input values and the real function values of the problem, which we want to approximation. Then neural network is trained by using the training data. In Chapter 2, we present a broad introduction to neural networks and their features.

In this thesis, our focus is in function approximation properties of the neural networks. Our research problem is how to build an accurate neural network model for a function approximation problem and how does different data sampling techniques affect the accuracy of the neural network. Building a neural network model, training it and validating it are essential phases before it can be used in optimization. In this thesis, we do not deal with surrogate based optimization, but instead concentrate on preparing surrogate model, that is neural network, to be used. Our secondary goal is to give practitioners a practical introduction to neural networks and show how to use them. Hence a reader would not need to spend months of studying neural networks literature, before acquiring the knowledge to use them.

Structure of the thesis is as follows. In the following sections we firstly introduce single objective optimization and how surrogates can be used in single objective optimization. Secondly we discuss about multiobjective optimization and give an example of wind turbine optimization. In Chapter 2 we discuss about neural networks theory. In Chapter 3 we introduce some heuristics found from literature, which can be useful when neural networks are implemented. In Chapter 4 we give a numerical experiment about neural network design using some heuristics and comparing different data sampling techniques and data sizes effect on neural networks training. In Chapter 5 we provide conclusions about this work.

1.1 Single objective optimization

In this thesis our motivation is to study the surrogates and sampling techniques to be able to finally use them for computationally expensive optimization problems. Hence, we first provide a brief background on optimization to bring the thesis into the context. In this section we discuss about basic concepts of single objective optimization. In addition we give an

example of surrogate assisted single objective optimization study, found from literature.

1.1.1 Basic concepts

The goal of solving optimization problem is to find the best possible solution optimizing a given objective with respect to given constraints. The problem can be basically anything, whether we want to improve some existing feature of e.g. controls, machineries, plants, distribution channels, etc. or we want to find out different designs for new creations. The objective is illustrating the features, design parameters, performance parameters, etc. of the optimization problem, which we want to improve. Modeling of the objective or the optimization problem is not considered in this thesis. Hence the optimization problem is, in general, formulated as

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{subject to} \quad & g(x) \leq 0 \\ & h(x) = 0 \\ & \alpha \leq x_i \leq \beta,\ \alpha, \beta \in \mathbb{R}, \end{aligned} \tag{1.1}$$

where f is the objective function, g is inequality function constraint, h is equality function constraint and α, β are bounds for variables $x = [x_1, \ldots, x_n]$. To avoid confusion, we are only considering minimizing as maximizing is the same as $min\frac{1}{f(x)}$ *or* $-f(x)$. Although all of the problems might not look like this and they may contain only some or none of the constraints. Let's call the variables x as solutions and they belong to a decision space. Corresponding objective (function) values $f(x) \in \mathbb{R}$ belong to an objective space, which, in here, is same as the $\mathbb{R}$. If problem consists of constraints, then feasible solutions are the ones that are satisfying all of the constraints. Feasible solutions belong to a feasible decision space $S \in \mathbb{R}^n$. The objective value of the optimal solution is better than other objective values. We have basically two different types of optimal solutions (see Figure 1)

- Local optimum: The solution (x^*) is a local optimum if $f(x^*) \leq f(x)$ for all $x \in S$, who $||x - x^*|| \leq \delta$, $\delta > 0$. We can have multiple local optima.
- Global optimum: The solution (x^*) is a global optimum if $f(x^*) \leq f(x)$ for all $x \in S$. We have only one global optimum objective value, but multiple optimum solutions,

consider e.g. $f(x) = sin(x)$.

The optimal solutions are said to be true optima if $\leq \rightarrow <$. For further information see e.g. [Deb 2001; Branke et al. 2008].

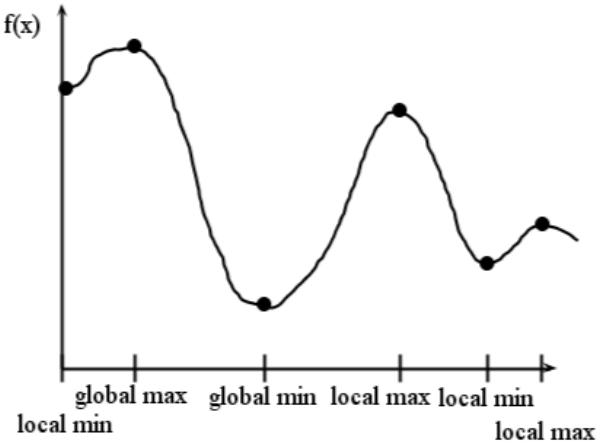

Figure 1: Local and global optima for single function.

For solving single objective optimization we have basically two classes of methods. The classes are

- Classical methods. These methods are evaluating surrounding of the current objective value. According to surrounding the method then decides, which direction it continues.
- Population based methods. These methods generate a population and evaluate the solutions. Then different operators are performed to alter solutions. Operators are usually stochastically. Then the best objective values of the altered solutions are picked to next generation. This is done as long as the fixed number of generations is obtained.

Classical methods are iterative and follow, roughly, the next algorithm [Branke et al. 2008]:

1. Generate a starting point x^0 and set $h = 0$.
2. Generate direction d^h
3. Calculate step size λ_h
4. Set $x^{h+1} = x^h + \lambda_h d^h$
5. Stop if termination condition is fulfilled or go back to step (2).

Classical methods can be divided into two subclasses according how it generates the direc-

tion. Direct search methods are evaluating surrounding of the current objective value and by that decide, which way to continue. This kind of methods are e.g. Hooke and Jeeves [Hooke and Jeeves 1961] and Powell [Powell 1964]. Gradient based methods evaluate gradients of the current objectives. According to gradients the method decides, which direction it continues. If we are minimizing, the direction is towards negative gradient value and vice versa when maximizing. This kind of methods are e.g. Newton and conjugate-gradient [Haykin 1999].

Population based methods are also iterative and follow, roughly, the next algorithm [Deb 2001]:

1. Generate a starting population x^0.
2. Evaluate all solutions in population $f(x^0)$.
3. Pick the best solutions for operators.
4. Generate next population x^1, using altered solutions and no altered solutions.
5. Stop if termination condition is fulfilled or go back to step (2).

Different population based methods are e.g. Differential Evolution [Price, Storn, and Lampinen 2005] and Genetic Algorithms [Mitchell 1999]. In single objective optimization surrogates can be used to replace expensive objective function evaluations done by a simulator, since we can point out the global optimum objective value. In the next subsection we give an example how surrogate have been used in single objective optimization.

1.1.2 Example: A biodiesel plant design

Single objective optimization example [Fahmi and Cremaschi 2012] is about determining an optimal flow sheet for a biodiesel production plant, which minimizes the costs of the plant in 10 years. In this study they used neural networks to replace simulation models of different processes. The problem is to choose the best option of three different reactors, determine the optimal flow sheet and operating conditions, which are minimizing the total costs of the biodiesel plant in 10 years when production goal is 8000 tons of biodiesel per year. In previous studies biodiesel plant optimization is shown to be computationally expensive e.g. one process simulation with exact model took 200-500 CPU seconds, although this was done

using Pentium III 667 MHz processor so computation time is not comparable using todays machinery, but later studies have shown that even with a faster CPU the problem is expensive to solve.

The training data for neural networks was produced with simulator. The number of needed training data points for different processes was very diverse, some processes needed only 700 training points as some needed 8000 training points to form an accurate surrogate model. Input dimension reducing technique, namely principal component analysis, was tested but because of independent nature of inputs it could not be applied. Mathematical modification was done to inputs so that it got uniform distribution. Different neural network models were built and the best models measured via sum of squared error were chosen. It is noted that building and choosing neural network models can be very time consuming.

For optimization they used GAMS - DICOPT, which can be used for mixed-integer nonlinear programming. As a result 48 local optimum objective values (top3 see Table. 1) were found and computational time was between 5 and 23 CPU seconds. The best solution was built and simulated with simulator and result was 41,45 m$ (vs. 41,07m$), hence the difference was 0,96%.

No.	Total cost (m$)	Solution time (CPU s)
1	41.068	5.302
2	42.811	9.992
3	44.905	5.240

Table 1: Optimal results from various initial guesses for a biodiesel production plant. [Fahmi and Cremaschi 2012]

1.2 Multiobjective optimization

When compared to single objective optimization, multiobjective optimization has more than one objective that should be optimized simultaneously. Usually those objectives are conflicting and the best solution might also be the matter of opinion, hence we might need a decision maker for further information about the problem. [Branke et al. 2008]

1.2.1 Basic concepts

The multiobjective optimization problem is, in general, formulated as

$$\begin{aligned} \min_{x} \quad & F(x) = \{f_1(x), \ldots, f_m(x)\} \\ \text{subject to} \quad & g(x) \leq 0 \\ & h(x) = 0 \\ & \alpha \leq x_i \leq \beta,\ \alpha, \beta \in \mathbb{R}, \end{aligned} \tag{1.2}$$

where F is the set of objective functions and m is the number of objectives. Solutions, which are satisfying the constraints, are creating a feasible decision space S. The objectives, which are calculated from the feasible solutions, are creating an image of the feasible decision space $F(S) = Z \in \mathbb{R}^m$. The Z is not usually known in explicit, but only through S.

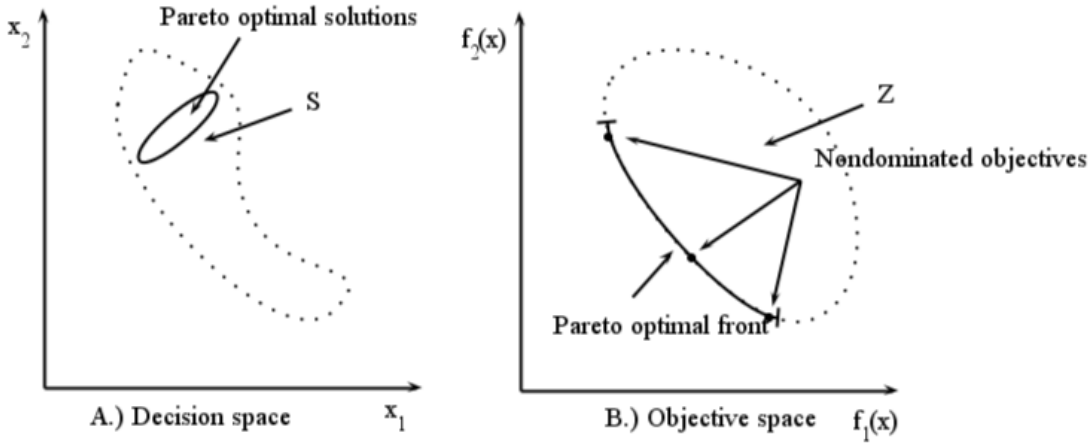

Figure 2: Feasible decision space S, which contains all feasible solutions. B.) Objective space Z by the feasible solutions. Pareto optimal front is a set of nondominated objectives, which are created from nondominated solutions. Only a few nondominated objectives are drawn, but Pareto-optimal front contains numerous amount of them.

Defining the optimal solutions in multiobjective optimization we can use concept of partial ordering [Deb 2001]. For partial ordering we need to define *domination* of solutions. A solution x^1 is dominating solution x^2 ($x^1 \prec x^2$) when

- $f_i(x^1) \leq f_i(x^2)$, for all $i = 1, \ldots, m$
- and at least one $f_k(x^1) < f_k(x^2)$, for some $k \in \{1, \ldots, m\}$.

The solution, which is not dominated by any other solution, is a nondominated solution and objective value of that is a nondominated objective value. A nondominated objective value of Z is a Pareto optimal (PO) objective value, hence the set of nondominated objective values is a PO front and corresponding nondominated solutions are PO solutions (see Figure 2). PO objectives are mathematically equally good. A feature of PO objectives is that if we want to improve value of one objective then at least one objective value needs to be weakened. There can be a number of numerous of PO objective values. Hence without knowledge from the application area Pareto-optimal objectives cannot be put in order. For more formal definition of Pareto optimality see e.g. [Deb 2001; Branke et al. 2008].

For finding PO solutions in multiobjective optimization we have four different method classes. In most of those a decision maker (DM) is needed, who is the domain expert of the optimization problem. The methods [Branke et al. 2008] are

1. *No-preference methods*: This is used when we do not have DM or DM has no preference about solutions. Hence we only need to find some PO solution. The solution can be used as a starting point in an interactive method, which is discussed later.
2. *A posteriori methods*: We are generating the whole set of PO solutions or approximation of it. After the set has been completed the DM picks the solution, which is the most satisfactory for him.
3. *A priori methods*: DM gives some a priori preference information about the solutions, which are most suitable for him. Then the search of PO solutions is performed by using those preferences.
4. *Interactive methods*: The most preferred PO solution is found by using iterative interaction between a method and a DM. Starting from some PO solution, the DM evaluates the solution(s) presented to him. If he is satisfied with some PO solution, the search is stopped. Otherwise, the DM expresses preferences on how solution(s) should be improved and new solution(s) are computed based on the preferences.

Our motivation is to build a surrogate model to be used in optimization, but optimization itself is not done in this thesis, although some preliminary work is done towards the optimization. In the next subsection we introduce how surrogates can be used in multiobjective optimization and what optimization method might benefit the most of from it.

1.2.2 Surrogate based multiobjective optimization

When compared to single objective optimization we have multiple choices to use surrogate models in multiobjective optimization. In multiobjective optimization the surrogate models can be used for four different purposes.

1. They can be used to approximate the objective functions (see e.g. [Kusiak, Zhang, and Li 2010]).
2. They can be used to approximate the Pareto front, when multiple conflicting objectives are involved (see e.g. [Wilson et al. 2001]).
3. They can be used to approximate the decision space, which produces the Pareto front (see e.g. [Fahmi and Cremaschi 2012]).
4. They can be used to replace the decision maker in multiobjective optimization (see e.g. [Sun, Stam, and Steuer 1996]).

Firstly using surrogate model to approximate the objective functions is quite obvious way, since by that we can directly replace the computationally expensive simulator. Secondly we can use a simulator to generate a PO front, when we obtain it we train a surrogate model to approximate the PO front. Hence we do not need to repeat the optimization process to generate new PO objective values. Thirdly as shown in Figure 2 the PO solutions might be gathered in some part of the feasible decision space, hence we may train the surrogate model by using those solutions. Then we may explore the surrounding of PO solution to see the objective values behavior in that area. Fourthly, when we are using an interactive method, the DM is giving us preferences, then he might become tired, he might not be able to apply or he might got some other reason, which is disturbing his preferences. Hence we can provide PO solutions to DM and he can decide, which he prefers, then we can train a surrogate model to approximate those preference solutions. Thus the surrogate model can give us consistent preference and replace the DM.

Our opinion is that population based methods e.g. evolutionary optimization algorithms (EO) would benefit the most when using surrogate models. Since they involve a number of numerous of objective value evaluations as population size can be hundreds and the number of generations can be hundreds of thousands. The EOs have found to be effective and they

have become popular [Deb 2001], hence there is no reason not to use them.

When using surrogate models for multiobjective optimization, practitioners have to consider about model management, which means techniques to make sure that the optimization process converge to right optimum, as a surrogate model might not be as accurate as the original model. According to [Jin 2005, 2011] there are three different ways to do it.

- *No model management*. The surrogate is assumed to be accurate enough and the original objective function is not used.
- *Fixed model management*. For model management there are three different techniques to do it, namely individual-based, generation-based and population-based. In individual-based technique some fixed individuals in the fixed generation are evaluated with the original objective function. In generation-based technique a fixed generations are evaluated with the original objective function. In population-based technique more than one sub-population co-evolves, every population is using its own surrogate.
- *Adaptive model management*. The frequency of using the original objective function is determined by the fidelity of the surrogate model.

In the next subsection we give an example of surrogate based multiobjective optimization.

1.2.3 Example: Wind turbine optimization

This study [Kusiak, Zhang, and Li 2010] presents a surrogate based optimization of a wind turbine. In this study there were three objectives, maximization of the power output, and minimization of the vibrations in the turbine's drive train and in the tower. Authors claim that numerous studies have been reported but those have fallen into parametric and physics-based models. Therefore surrogate based approach is tried, which has been successfully applied to industrial optimization. To present vibrations, drive train acceleration and tower acceleration are selected to be modeled and the output power of the tower is modeled also as a third objective. All of the objectives are replaced with a neural network surrogate. For training two different datasets are used, 10 second length and 60 second length. Data was collected from supervisory control and data acquisition (SCADA) and sample frequency was 0.1Hz. Data is preprocessed, incorrect values are deleted, and data is denoised using wavelet

analysis and normalized. Both datasets are split into training set $\frac{2}{3}$ of the data and testing set $\frac{1}{3}$ of the data. Surrogate models trained with 10 second dataset are achieving accuracies, drive train acceleration 98% accuracy, tower acceleration surrogate 94% accuracy and power output surrogate 96% accuracy. Training with 60 second dataset achieves accuracies, in order, 99%, 97% and 97%. The Strength Pareto Evolutionary Algorithm [Zitzler and Thiele 1998] was employed for optimization. The optimization problem gets form

$$\begin{aligned}
\text{min} \quad & F(y_1,y_2,y_3) = w_1 y_1(t) + w_2 y_2(t) + w_3 \frac{1}{y_3(t)} \\
\text{subject to} \quad & y_1(t) = f_1(y_1(t-1), v_1(t), v_1(t-1), x_1(t), x_1(t-1), x_2(t), x_2(t-1)), \\
& y_2(t) = f_2(y_2(t-1), v_1(t), v_1(t-1), x_1(t), x_1(t-1), x_2(t), x_2(t-1)), \\
& y_3(t) = f_3(v_1(t), v_1(t-1), x_1(t), x_1(t-1), x_2(t), x_2(t-1)), \\
& max\{0, currentsettings - 50\} \leq x_1(t) \leq min\{100, currentsettings + 50\}, \\
& max\{-5, currentsettings - 5\} \leq x_2(t) \leq min\{15, currentsettings + 5\},
\end{aligned} \tag{1.3}$$

where w_1, w_2 and w_3 are weights for optimization and in this study only three different weights sets were used $w_1 = 1, w_2 = w_3 = 0$, $w_2 = 1, w_1 = w_3 = 0$ and $w_3 = 1, w_1 = w_2 = 0$. Results for optimization are shown in Figure 2. Author conclude that with a more accurate data the results might have been even better and this method will be employed again, when it is available. "The objective of this paper, building accurate data-driven models to study the impact of turbine control on their vibrations and power output and demonstrating the optimization results of wind turbine performance, was accomplished" [Kusiak, Zhang, and Li 2010].

Table 2: Wind turbine optimization results. [Kusiak, Zhang, and Li 2010]

Mean Value			
Minimize Drive Train acceleration	Optimized Drive Train acceleration	Original Drive Train acceleration	Gain
10-s data set	119,53	131,49	9,10 %
1-min data set	124,06	131,79	5,87 %
Tower acceleration	Tower acceleration	Tower acceleration	Gain
10-s data set	87,22	127,82	31,76 %
1-min data set	106,26	130,32	18,46 %
Power Output	Power Output	Power Output	Gain
10-s data set	1497,99	1481,72	1,10 %
1-min data set	1497,99	1482,57	1,03 %

2 Neural Networks

Brains are very complex, nonlinear and parallel computing units. Brains are built from neurons. A neuron contains cell body and different types of synaptic branches, which are connected to other neurons, reception or response organs. A different type of knowledge is stored to different parts of neuron. It is estimated that, in brain, there are about 10 billion neurons and 60 trillion synaptic connections. The structure of neurons connected to other neurons is called a neural network, hence our brains are basically just a huge neural network with huge computational capability. Hence imitating brain is very ambitious goal to achieve. [Haykin 1999]

In 1943, McCulloch a psychiatrist/self-trained neuroanatomist and Pitts a mathematician decided to do a study together. In their study they united neurophysiology and mathematical logic to build an artificial neuron, which was similar to ones in brain [McCulloch and Pitts 1943] and was called McCulloch and Pitts -model. They also proved, that with an adequate number of neurons and properly set synchronously operating synaptic weights can approximate any function. This proof led to the birth of artificial neural networks and artificial intelligence. [Haykin 1999]

Our main objective is to build a framework for practitioners to choose a satisficing neural network surrogate model. Neural networks (NN) are interconnected groups of artificial neurons. NNs are used to model inputs to corresponding outputs. Those inputs and outputs can be obtained from some function $y = f(x)$, some process, some physical phenomenon and so on. Then NN is trained to match those input/output relationships, training a NN means that synaptic weights are altered to give the best result. Therefore training NN we need data containing inputs and corresponding outputs. Trained NN can be as accurate as the data, thus it is important that the training data is as diverse as possible, since the NNs are good in interpolation, but not very good in extrapolation [Haley and Soloway 1992]. A well trained NN can represent the input-output relation very accurately [Cybenko 1989; Kusiak, Zhang, and Li 2010].

Some definitions is needed before we start. A *training pattern* is known input/output pair

which illustrates the input/output behavior of the problem. A set of training patterns is a *training set*. An *epoch* is one iteration when all of the training patterns are presented to network. A *time step* is an interval which we use for going through the time. *Feedforward* means that work flow of the network goes from input to output and feedbacks occur. In some literature is referred as feedforward single layer/multilayer neural network, which means same as our single layer neural network, multilayer neural network. Structures discovered here are all feedforward networks, but Recurrent Multilayer Perceptron is not.

In this chapter, we first discuss neurons in section 2.1 and neural network structures in section 2.2. In section 2.3, we discuss about NNs training techniques and data sampling techniques for generating a training data. In section 2.4, we discuss about different error metrics, theoretic accuracy of NNs and one way to measure algorithmic complexity.

2.1 Neurons

In this section we present the model of a neuron and the model of a radial basis function with their important features and applicability.

2.1.1 Neuron

An artificial neuron is an information processing unit and it is the basic building block for NN. It can be presented as

$$y = f(\sum_{j=1}^{n} w_j x_j + b), \tag{2.1}$$

where y is the output, $w = [w_1, ..., w_n]$ is the synaptic weight vector, $x = [x_1, ..., x_n]$ is the input vector, n is the number of inputs, b is the bias and $f(...)$ is the transfer function. The bias is an offset and it determines the output of the neuron when the input is 0, this helps to make affine transformation to the data. By setting the bias to one of the inputs we can represent the neuron as

$$y = f(\sum_{j=0}^{n} w_j x_j), \tag{2.2}$$

where w_0 is value of the bias and x_0 is constant 1.

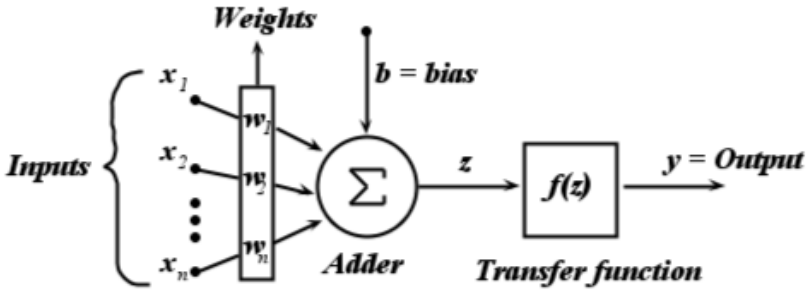

Figure 3: A model of an artificial neuron

An artificial neuron comprises of three components (see Figure 3)

1. Synaptic weight (w): Inputs are connected to the adder and each of them has own synaptic weight. It determines the strength of the connection. Weight can be either positive or negative. For simplicity we will be calling synaptic weight as weight.
2. Adder ($\sum$): This is where the weighted inputs are added together.
3. Transfer function ($f(z)$): This is also called as activation function, but we use transfer function (TF) in this thesis. A TF denotes neurons activation and limits its output value to some finite value.

A few TFs (see Figure 4) are shortly presented next.

Threshold function:

$$f(z) = \begin{cases} 1 & \text{, if } z \geq 0 \\ 0 & \text{, if } z < 0 \end{cases} \tag{2.3}$$

Neuron with this TF is called McCulloch and Pitts -model. It is used when output is needed to be binary or in other words it is $\{0, 1\}$.

Linear function:

$$f(z) = mz + c, \tag{2.4}$$

where m and c are constants and z is the variable. This TF is usable when we are trying to fit a straight line to match some data, we can vary lines slope by altering c and gradient by

altering m. Linear function can be bound to some minimum and maximum values.

$$f(z) = \begin{cases} mz + c & \text{, if } -bo < z < bo \\ -bo & \text{, if } z < -bo \\ bo & \text{, if } z > bo \end{cases} \tag{2.5}$$

, where $bo \in \mathbb{R}$.

Log-sigmoid function:

$$f(z) = \frac{1}{1 + exp(-az)}. \tag{2.6}$$

Log-Sigmoid function is the most commonly used TF. It is an S-shaped function and can take values between 0 and 1. This function is nonsymmetric, nonlinear, differentiable and monotonically increasing. The function of a in $exp(-az)$ is that by change in the log-sigmoid function can obtained slope shapes from threshold function to linear function. Nonlinear feature is needed for capturing nonlinear behavior of a problem.

Hyperbolic tangent function:

$$f(z) = atanh(bz), \tag{2.7}$$

where $a, b > 0$, a determines the maximum and minimum values of sigmoid. Hyperbolic tangent function has the same features as log-sigmoid function but it can take values between -1 to +1 and it is antisymmetric.

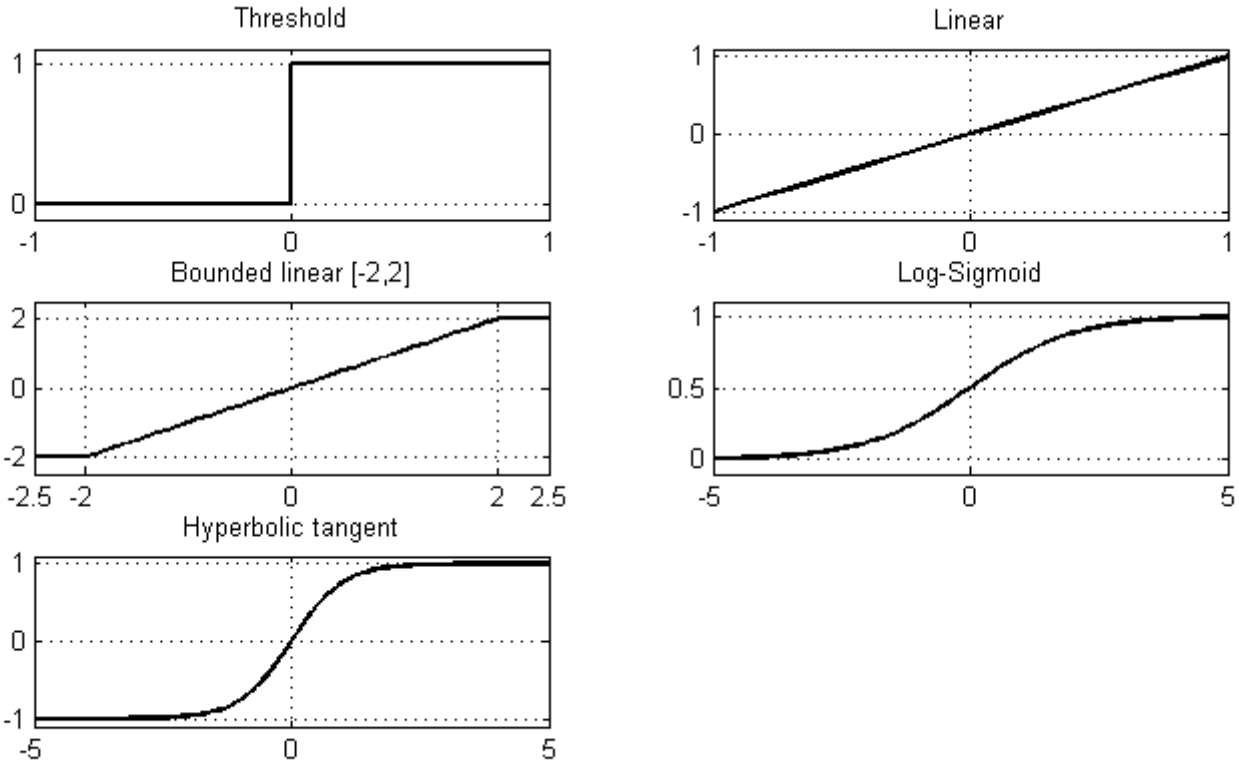

Figure 4: Transfer functions for an artificial neuron

2.1.2 Radial Basis Function

A neuron based on a radial basis function (RBF). It differs from the McCulloch & Pitts -model in a way that here we measure the distance of an input to a fixed center and the inputs are not weighted. The center can be one of the inputs. The distance metric is usually Euclidean. The mathematical formulation for the RBF is

$$y(x) = f(\sum_{i=1}^{n} \|x_i - t\|), \tag{2.8}$$

where $f(...)$ is radial basis function, $x = [x_1, ..., x_n]$ is the input vector and t is the center. Later on we need the N-by-N matrix of RBFs to be nonsingular. In [Micchelli 1986] is shown that a set of distinct points $x_{i_{i=1}}^{n} \in \mathbb{R}^N$, where N is the dimension of input space and n is the number of inputs, the N-by-N matrix, whose ji-th elements is $f_{ji} = f\|x_j - x_i\|$, is nonsingular. Micchelli's theorem covers a large class functions, hence a few of them (see

Figure 5) used here as RBFs are multiquadrics

$$f(r) = (r^2 + c^2)^{\frac{1}{2}}, \tag{2.9}$$

inverse multiquadrics

$$f(r) = \frac{1}{(r^2 + c^2)^{\frac{1}{2}}} \tag{2.10}$$

and Gaussian function

$$f(r) = exp(-\frac{r^2}{2c^2}), \tag{2.11}$$

where $r = \|x - t\|$ is the distance between the input $x = [x_1, ..., x_n]$ and the center is t, $c > 0$ and it determines how diverse the function will be. The RBF is used in the hidden layer(s) of the RBF network.

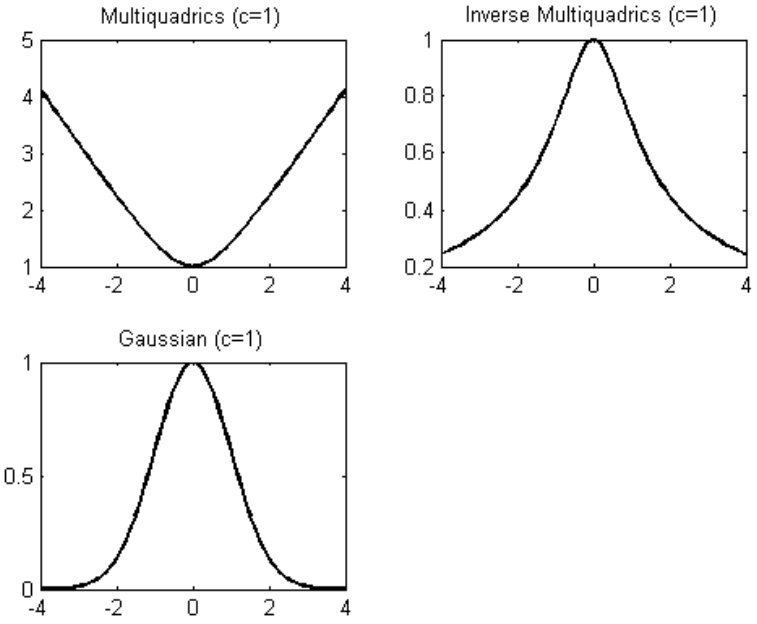

Figure 5: Radial Basis Functions

2.2 Neural Network models

In this section some neural network models are described. We illustrate their structure, how to train them and when to use them. These should help one to choose a proper neural network structure. Structures are chosen so that practitioners have more than one choice for a surrogate model. Three of surrogate models are for (nondynamic) single point approximations, which are single layer neural network, multilayer neural network and radial basis function network. One model is for (dynamic) sequence approximations, namely recurrent multilayer neural network.

2.2.1 Single Layer Neural Network

McCulloch and Pitts (1943) introduced the idea of neural networks. In 1949, Hebb a neuroscientist proposed a first self-organized learning rule [Hebb 1949]. According to the Hebb's learning rule the value of the synaptic weight is increased when it is activated and decreased when it is not activated. Hebb's rule was more from biological point of view and based on assumptions made in there. Rosenblatt introduced the perceptron as the first model, which could be trained [Rosenblatt 1958]. Rosenblatt's perceptron is a neuron with nonlinear TF, weights and bias (see Figure 3). Hence we will be calling single layer neural network as single layer perceptron (SLP).

A SLP contain one layer of neurons and the number of neurons is determined by the number of outputs, as shown in Figure 6. Mathematically it can be formulated as

$$y_k = f(\sum_{j=1}^{n} w_{kj}x_j + b_k), \tag{2.12}$$

where $y = [y_1, ..., y_k]$ is the output vector, w is the weight matrix, $x = [x_1, ..., x_n]$ is the input vector, b is the bias, $f(...)$ is the transfer function. Any TF discovered in previous section can be used here.

Every neural network can be classified either as a fully connected network, where every input and output of the neuron is connected to every neuron or as a partially connected network, where every input and output of the neuron is not connected to every neuron. This implies to other neural networks as well. It is shown in [Hüsken, Jin, and Sendhoff 2005] that a

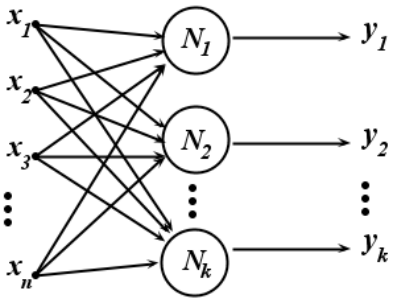

Figure 6: The single layer perceptron with n inputs and k outputs.

fully connected network might not give the best approximation result. The SLP is suitable for classifying linearly separable problems. Linearly separable means that we can draw a straight line between two sets of data points. This network can also be used to estimate linear datasets by fitting a straight line to the set of data (see Figure 7). SLPs outputs are linear combinations of its inputs, hence it is difficult to capture nonlinearity in data.

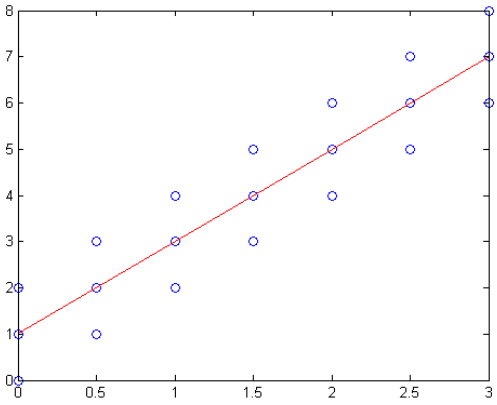

Figure 7: An approximation of linear dataset with single layer perceptron.

Examples, which illustrate SLP applicability, are shown in subsection 2.5.1.

2.2.2 Multilayer Neural Network

In 1985 the first successful realization of a multilayer neural network and it was called the Boltzmann machine [Ackley, Hinton, and Sejnowski 1985]. A popular training algorithm for multilayer neural network, back-propagation algorithm, was developed in 1986 [Rumelhart,

Hinton, and Williams 1986]. They proposed that it could be used for machine learning and demonstrated how it could work. The basic idea of back-propagation algorithm was found much earlier [Haykin 1999].

Multilayer neural network is also referred to as Multilayer Perceptron (MLP) as it is multiple layers of Rosenblatt's Perceptrons. In MLP there are more than one layer and the layers are called hidden layer(s) and output layer. Figure 8 illustrates MLP with two hidden layers containing p and q neurons, and the output layer containing k neurons. The number of neurons in the output layer is equal to the number of outputs. There can be any number of neurons in the hidden layers and such neurons are called hidden neurons. The number of hidden neurons determines network's ability to learn and store knowledge from the input/output - relationships in its weights. In the hidden neurons, it is a common practice to use either the log-sigmoid TF or the hyperbolic tangent TF [Duch and Jankowski 1999]. If a linear TF is used is the hidden layers the MLP reduces to SLP, this is proved in Theorem 1. The mathematical formulation of a MLP is

$$y_k = f^o(\sum_{j=1}^{q} w^o_{kj}(f^{II}(\sum_{t=1}^{p} w^{II}_{jt}(f^I(\sum_{i=1}^{n} w^I_{ti}x_i + b^I_i) + b^{II}_t) + b^o_j))), \tag{2.13}$$

where f^o, f^I and f^{II} are the TFs of layers, w^o, w^I and w^{II} are the weight matrices of layers, b^0, b^I and b^{II} are layers bias vectors and $x = [x_1, ..., x_n]$ is the input vector. Eq. (2.13) can be also formulated in a different way by using eq. (2.2),

$$y_k = f^o(\sum_{j=0}^{q} w^o_{kj}(f^{II}(\sum_{t=0}^{p} w^{II}_{jt}(f^I(\sum_{i=0}^{n} w^I_{ti}x_i))))), \tag{2.14}$$

where only difference is that biases are now included in the weight matrices as one of the inputs. The input for bias is x_0 and its weight is w_{k0}.

Theorem 1. *Multilayer Perceptron will reduce to Single Layer Perceptron if linear transfer functions are used in hidden layers. Let y_k be some arbitrary output of the MLP containing arbitrary number $V \in \mathbb{N}$ of layers and hidden neurons $k, j, t, \ldots, g, a \in \mathbb{N}$,*

$$y_k = f^o(\sum_{j=1}^{q} w^o_{kj}(f^V(\sum_{t=1}^{p} w^V_{jt}(\ldots(f^I(\sum_{i=1}^{n} w^I_{ai}x_i + b^I))\ldots) + b^V)) + b^o) = f^o(\sum_{i=1}^{n} w^X_{ki}x_{ki} + b^X). \tag{2.15}$$

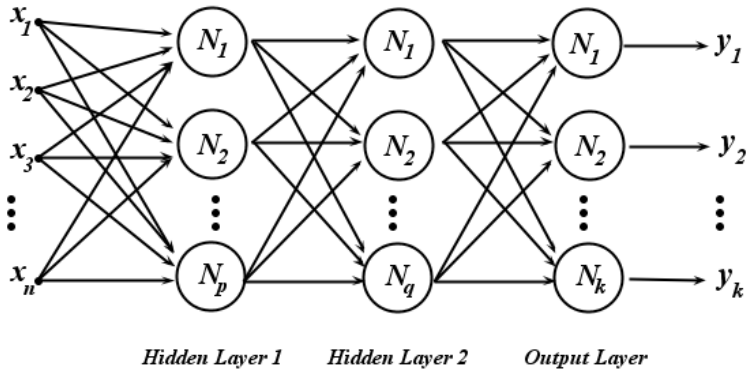

Figure 8: A multilayer perceptron with three layers, n inputs and k output.

Proof. Let the y_k be some output of a MLP with arbitrary number $V \in \mathbb{N}$ of layers and hidden neurons $k, j, t, \ldots, a \in \mathbb{N}$, where hidden neurons contain a linear TF and output layer contains any TF

$$y_k = f^o(\sum_{j=1}^{q} w_{kj}^o(f^V(\sum_{t=1}^{p} w_{jt}^V(\ldots(f^I(\sum_{i=1}^{n} w_{ai}^I x_i + b^I))\ldots) + b^V)) + b^o). \tag{2.16}$$

As all TFs except f^o are linear $f(x) = mx + c$ the y_k can be written as

$$y_k = f^o(\sum_{j=1}^{q} w_{kj}^o(m^V(\sum_{t=1}^{p} w_{jt}^V(\ldots(m^I(\sum_{i=1}^{n} w_{ai}^I x_i + b^I) + c^I)\ldots) + b^V) + c^V) + b^o). \tag{2.17}$$

When we foil input layer, we get

$$y_k = f^o(\sum_{j=1}^{q} w_{kj}^o(m^V(\sum_{t=1}^{p} w_{jt}^V(\ldots(\sum_{i=1}^{n} m^I w_{ai}^I x_i + m^I b^I + c^I)\ldots) + b^V) + c^V) + b^o), \tag{2.18}$$

then by combining $m^I w^I = \hat{w}^I$ and $m^I b^I + c^I = \hat{b}^I$, which are some constant $\in \mathbb{R}$, we get

$$y_k = f^o(\sum_{j=1}^{q} w_{kj}^o(m^V(\sum_{t=1}^{p} w_{jt}^V(\ldots(\sum_{i=1}^{n} \hat{w}_{ai}^I x_i + \hat{b}^I)\ldots) + b^V) + c^V) + b^o). \tag{2.19}$$

Performing such actions for every layer we get

$$y_k = f^o(\sum_{j=1}^{q} w_{kj}^o(\sum_{t=1}^{p} \hat{w}_{jt}^V(\ldots(\sum_{i=1}^{n} \hat{w}_{ai}^I x_i + \hat{b}^I)\ldots) + \hat{b}^V) + b^o). \tag{2.20}$$

When we foil the first hidden layer and the second hidden layer

$$\sum_{z=1}^{a} \hat{w}^{II}(\sum_{i=1}^{n} \hat{w}^{I}_{zi}x_i + \hat{b}^{I}) + \hat{b}^{II} = \sum_{z=1}^{a} \hat{w}^{II} \sum_{i=1}^{n} \hat{w}^{I}_{zi}x_i + \sum_{z=1}^{a} \hat{w}^{II}\hat{b}^{I} + \hat{b}^{II} \tag{2.21}$$

and by combining $\sum_{z=1}^{a} \hat{w}^{II} \sum_{i=1}^{n} \hat{w}^{I} x_{zi} = \sum_{i=1}^{n} \bar{w}^{II} x_{gi}$ and $\sum_{z=1}^{a} \hat{w}^{II}\hat{b}^{I} + \hat{b}^{II} = \bar{b}^{II}$,

hence we may write

$$y_k = f^o(\sum_{j=1}^{q} w^o_{kj}(\sum_{t=1}^{p} \hat{w}^{V}_{jt}(\ldots(\sum_{i=1}^{n} \bar{w}^{II}_{gi}x_i + \bar{b}^{II})\ldots) + \hat{b}^{V}) + b^o), \tag{2.22}$$

where g is the number of neurons in the third hidden layer. By iterating such actions to each layer we finally get $\sum_{i=1}^{n} \bar{w}^{X}_{i} x_{ki} + \bar{b}^{X}$, hence we get may now write the MLP formula as

$$y_k = f^o(\sum_{i=0}^{n} \bar{w}^{X}_{ki}x_i + \bar{b}^{X}). \tag{2.23}$$

□

For a MLP, it is difficult to set the number of hidden layers and the number of hidden neurons. In [Cybenko 1989] it is shown that network with one hidden layer can encode any arbitrary function. However, [Tamura and Tateishi 1997] proved that MLP with one hidden layer and $tp-1$ hidden neurons can approximate tp input/output relations exactly, where tp is the number of training patterns, and a MLP with two hidden layer and $tp/2+3$ hidden neurons can approximate input-output relations with arbitrarily small error. Although in every problem this kind of accuracy is not needed or wanted and the generalization ability is a more important feature. Generalization is NNs ability to approximate input point to output in case that this input/output pair has not been used for training data. The number of hidden layers will affect the training time. Additionally, a higher number of hidden layers can also cause overtraining. On the other hand with too few hidden neurons neural network might not capture the relationships between inputs and outputs. In Chapter 3, we discuss about a few heuristics for the number of layers and neurons.

As said earlier the MLP with one hidden layer can encode any arbitrary function, hence

MLP is suitable for function approximation, pattern recognition and classification, shown in examples (see subsection 2.5.2). Downside of the MLP is that optimal structure is hard to find and it always depends on problem. A few ways find it are trial and error, pruning and growing. The pruning technique means that we start from a large MLP and then we prune it by weakening or eliminating weights e.g. see [Hassibi, Stork, and Wolff 1993]. Growing means that we start from a small MLP and then grown it until it the performance is good enough.

2.2.3 Recurrent Multilayer Perceptron

Hopfield tried to understand the calculation performed by a recurrent network with symmetric synaptic weights via the cost function $E(n)$ [Hopfield 1982]. Doing this he showed the isomorphism between neural networks and an Ising model, which is used in physics. This got the attention of physicists and they started to use neural networks. Hopfield was not the first in the field of the recurrent networks, but he was the first to present it in explicit the way of storing information to dynamic networks, thus the specific class of recurrent networks is called Hopfield networks. In general, recurrent network consists parts of MLP or the whole MLP but they contain at least one feedback loop. The feedback loop is feeding the output of the neuron or the layer back to inputs with some time delay. The feedback loop is local, if it is around the neuron, or global, if it is around the layer(s).

The recurrent network that we present here is a Recurrent Multilayer Perceptron (RMLP) as presented in [Puskorius, Feldkamp, and Davis 1996], where it was used as on-vehicle idle speed controller. Other useful recurrent networks are Nonlinear Autoregressive with eXogenous input (NARX) -model [Narendra and Parthasarathy 1990], State-space model [Zamarreño and Vega 1998] and Second-order network [Pollack 1991]. The RMLP is similar to MLP with an addition of a feedback loop around every layer. The RMLP with three layers is illustrated in Figure 9 and can be mathematically formulated as

$$y_o(t+1) = F_o(y_o(t), F_{II}(y_{II}(t), F_I(y_I(t), x(t)))),$$

where F_o, F_I and F_{II} are layers containing weights and transfer functions, y_o, y_I and y_{II} are the layers output, $x = [x_0, ..., x_t]$ is the input vector through time and t is time step.

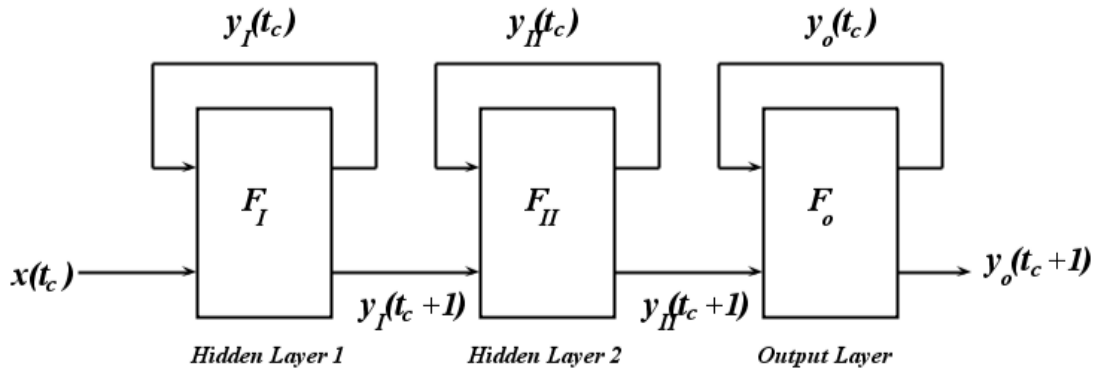

Figure 9: A Recurrent Multilayer Perceptron with three layers

Recurrent network can be used for input-output mapping and associative memory. In input-output mapping we provide an input sequence to the network. Every input corresponds to one time step and they vary from each other. For a given input sequence an output is calculated for every time step and every output in turn affects the next output. As a result we get an output sequence for the input sequence. Recurrent networks can be used for in system identification [Narendra and Parthasarathy 1990], control [Puskorius, Feldkamp, and Davis 1996] and prediction [Xie, Tang, and Liao 2009]. In associative memory we give an input pattern to the network, the input pattern is static over time. The output will change over time but eventually it will convergence to a point and the point is considered as a result.

2.2.4 Radial Basis Function Network

The last network model is a Radial Basis Function network. The basic idea leads back to 1965 when Cover introduced his theorem [Cover 1965]. Theorem says that there are many dichotomies to classify points and as dimension of the function grows the number of dichotomies grows. In Figure 10 it is shown one of those dichotomies, quadratically separable dichotomy and for those five points there are 32 different dichotomies for classification. Two main ideas from Cover's theorem are nonlinearity from input space to hidden neuron space and high dimensionality of hidden space compared to input, thou in study he used polynomial functions but those results implies to RBFs too. And these are saying that we are more likely to find linearly separable problem when input space is mapped to hidden space the XOR-example done with RBFs shows this. In 1988, Broomhead and Lowe introduced the

RBF network as an alternative for MLP [Broomhead and Lowe 1988]. The network consists two layers, a hidden layer and an output layer, in the hidden layer there are RBFs and in the output layer there are linear neurons, so the output is a linear combination of RBFs.

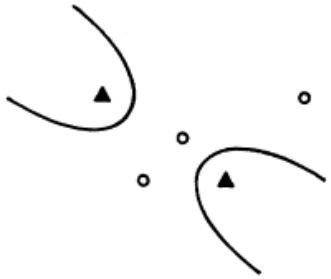

Figure 10: Quadratically separable dichotomy. [Broomhead and Lowe 1988]

A RBF network is a fully connected network. Here only the weights between the RBFs and the output neurons exists as shown in Figure 11 and output neurons may contain bias. The mathematical presentation for the RBF network is

$$y_k(x) = w_{0k} + \sum_{i=1}^{tp} w_{ik} f(\|x_i - t\|), \tag{2.24}$$

where w is the weight matrix between the hidden layer and the output layer, $x = [x_1, ..., x_n]$ is the input vector, $t = [t_1, ..., t_n]$ is the center vector, $f(...)$ is some of the RBFs discovered in subsection 2.1.2 and tp is the number of training patterns.

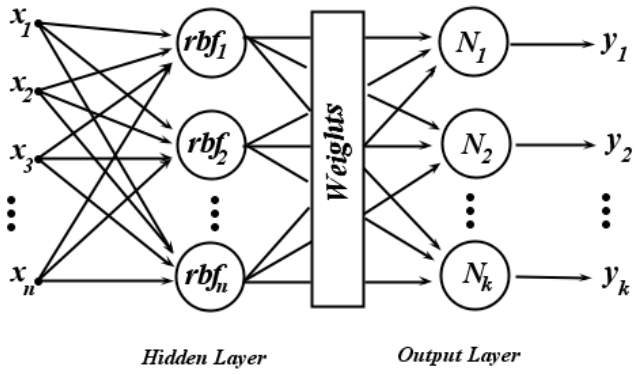

Figure 11: Architecture of Radial Basis Function network.

RBF network is good in interpolation and for interpolation we need, in case we have only one output,

$$F = \begin{bmatrix} f_{11} & f_{12} & \dots & f_{1n} \\ f_{21} & f_{22} & \dots & f_{2n} \\ \vdots & \vdots & & \vdots \\ f_{n1} & f_{n2} & \dots & f_{nn} \end{bmatrix},$$

where $f_{ij} = f(\|x_i - t_j\|)$, so every input is taken to be as a center. Notice that diagonal of F is 1. We also need the weight matrix

$$W = \begin{bmatrix} w_{11} & w_{12} & \dots & w_{1n} \\ w_{21} & w_{22} & \dots & w_{2n} \\ \vdots & \vdots & & \vdots \\ w_{k1} & w_{n2} & \dots & w_{kn} \end{bmatrix},$$

and the desired output vector $Y = [y_1, ...y_n]^T$. Hence we get equation

$$WF = Y \tag{2.25}$$

and because we need to know W, it can be solved as

$$W = F^{-1}Y \tag{2.26}$$

and for RBFs in subsection 2.1.2 the F is nonsingular and therefore F^{-1} exists. We may also start with only one center and add a center as long as the approximation error is desired, hence we do not need all of the tp to be as centers.

If dimensionality of the data set is large and every part is taken for training, in [Broomhead and Lowe 1988] it is shown that a poor generalization performance can be obtained. Additionally, when the quality of the data is unknown, then the problem can ill-posed. To define the ill-posed problem we need to define a well-posed problem first. There are three conditions which we need to satisfy for the well-posed problem [Tikhonov and Arsenin 1977].

1. *Existence*. For every $x \in X$ there is $y \in Y$, when $y = f(x)$, where X is the input space and Y is the feature space.

2. *Uniqueness*. For every input pair $x, z \in X$, we have $f(x) = f(z)$ if and only if $x = z$, where X is the input space.

3. *Continuity*. For any $\varepsilon > 0$ there exists $\delta > 0$ such that the condition $d(x,z) < \delta$ implies that $d(f(x), f(z)) < \varepsilon$, where d implies to distance.

Hence if one of these conditions is violated the problem is ill-posed. Thus in RBF network we want to use distinct inputs as a center even if the training set consists several equivalent inputs. To overcome this problem regularization and generalized RBF networks were introduced by [Poggio and Girosi 1989]. As the regularized network tents to be high dimensional and it is shown by [Wettschereck and Dietterich 1992] that generalized RBF network can be as accurate as MLP. Although in this thesis we will stay in "basic" version of RBF network.

As we have discussed about interpolation with RBF network, almost whole section, we can summary that they are good for approximating function, especially smooth functions, and noise removal [Craven and Wahba 1979]. RBF networks can be used for classification problems as well [Cover 1965]. RBF networks disadvantage is the high dimensionality.

2.3 Training a neural network

Training methods for NN can be separated to two classes:

- *Supervised training*: For supervised training we need the inputs and corresponding outputs, these i/o-pairs are called training patterns. Using training patterns we train the NN so that output produced by NN match to given outputs. This training method is suitable for function approximation.
- *Unsupervised training*: For unsupervised training we need only the inputs. Using certain rules we train the NN to produce some outputs from inputs. This training method is suitable for classification problems.

As we are interested about function approximation in this thesis the main topic will consider about supervised training method. In the first subsection we introduce a few data sampling techniques, which can be used to create an input space for a training data. In the second subsection we introduce an error correction training method as a supervised training and its

applications to structures discovered in section 2.2. In the third subsection we introduce briefly some unsupervised training methods. In the fourth subsection we discuss about presenting training as an optimization problem.

2.3.1 Training Data generation

Before we can begin training, we need to generate training data, which is typically made with a simulator. To ensure that an input space is as diverse as possible so that results from the simulator are as diverse as possible, we need to generate the input space by using some systematic sampling technique rather than randomly sampled inputs [Simpson, Lin, and Chen 2001]. When the inputs are created then those can be given to the simulator, which calculates the corresponding outputs.

We introduce three data sampling techniques. The techniques are Latin Hypercube sampling, Hammersley sampling and Orthogonal array sampling. Basic principle of techniques is pretty much the same. Firstly the sampling area is divided into segments and secondly the segments are filled with a point. In next each of the techniques are introduced briefly and then an example is given.

Latin Hypercube sampling

Latin hypercube sampling in its basic form as shown in [Stein 1987] is

$$X_{jk} = F_k^{-1}(N^{-1}(p_{jk} - 1 + \xi_{jk})), \tag{2.27}$$

where X is the value, F is the cumulative distribution function of X, N is the number of sample points, p_{jk} is the place in matrix P, which size is $N \;\; x \;\; K$, K is the number of variables and ξ is uniformly distributed random variable, which takes values from 0 to 1. The values in P are permutated independently and they are defining the segment of X and ξ defines the value of X. An example of this (see Figure 12), where we can see that there is only one point in each row.

Hammersley sampling

Idea of Hammersley sampling is that every nonnegative integer k can be presented as com-

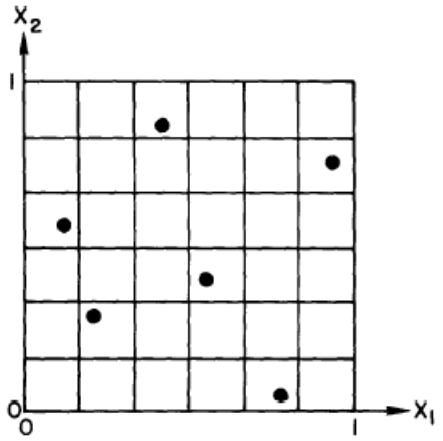

Figure 12: A Latin Hypercube Sample with N = 6, K = 2 for X Distributed Uniformly on the Unit Square. [Stein 1987]

bination of prime p [Wong, Luk, and Heng 1997].

$$k = a_0 + a_1 p + a_2 p^2 + \ldots + a_r p^r, \tag{2.28}$$

where a takes values 0 or 1 according to binary value of k e.g. when $k = 1$ then $a = (0001)$ and when $k = 2$ then $a = (0010)$. For sampling we need a function

$$\phi_p(k) = \frac{a_0}{p} + \frac{a_1}{p^2} + \ldots + \frac{a_r}{p^{r+1}}. \tag{2.29}$$

Now the Hammersley point for k in d-dimension is

$$(\frac{k}{n}, \phi_{p_1}(k), \phi_{p_2}(k), \ldots, \phi_{p_{d-1}}(k)), \text{for } k = 0, 1, \ldots, n-1, \tag{2.30}$$

where n is the number of points. For algorithm see e.g. [Wong, Luk, and Heng 1997]. Example when $d = 2$, $p = 2$ and $n = 4$, which is also a special case of Hammersley sampling called Van der Corput sampling.

k	$d_1 = k/n$	binary	$d_2 = \phi_2(k)$
1	$\frac{1}{4} = 0.25$	001	$\frac{1}{2} = 0.5$
2	$\frac{2}{4} = 0.5$	010	$\frac{1}{4} = 0.25$
3	$\frac{3}{4} = 0.75$	011	$\frac{1}{2} + \frac{1}{4} = 0.75$
4	$\frac{4}{4} = 1$	100	$\frac{1}{8} = 0.125$

Table 3: Example of Hammersley sampling.

Orthogonal array

The orthogonal array is denoted by $OA(n,m,s,r)$, where size of the matrix is $n \ x \ m$, s is the number of elements and r is strength. A $n \ x \ m$ matrix A is an orthogonal array design with strength r with entries forming sets of $s \leq 2$ elements, if each $n \ x \ r$ submatrix of A contains all possible row vectors with the same frequency $\lambda = n/s^r$ [Tang 1993]. Let the elements s be denoted by $1,2,\ldots,s$ and the objects n be denoted by $1,2,\ldots,n$. Now we can map objects n to

$$Z(i) = \begin{cases} 1 & i = 1,2,\ldots,q, \\ 2 & i = q+1,q+2,\ldots,2q, \\ \vdots & \vdots \\ s & i = (s-1)q+1,(s-1)q+2,\ldots,n, \end{cases} \tag{2.31}$$

where $q = n/s$. Then objects in the space Z are permutated so that all possible combinations are shown in rows at λ times. Consider example as $OA(4,2,2,2)$, when we have matrix

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{bmatrix}^T$$

and

$$Z(i) = \begin{cases} 1 & i = 1,2, \\ 2 & i = 3,4. \end{cases} \tag{2.32}$$

We get

$$\begin{bmatrix} Z(1) & Z(2) & Z(3) & Z(4) \\ Z(1) & Z(2) & Z(3) & Z(4) \end{bmatrix}^T = \begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \end{bmatrix}^T$$

and then we perform permutation, we get

$$\begin{bmatrix} Z(1) & Z(2) & Z(3) & Z(4) \\ Z(1) & Z(3) & Z(2) & Z(4) \end{bmatrix}^T = \begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 2 & 1 & 2 \end{bmatrix}^T$$

, hence the matrix

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 1 & 3 & 2 & 4 \end{bmatrix}^T$$

is orthogonal array, but e.g.

$$\begin{bmatrix} Z(2) & Z(1) & Z(3) & Z(4) \\ Z(1) & Z(2) & Z(4) & Z(3) \end{bmatrix}^T = \begin{bmatrix} 1 & 1 & 2 & 2 \\ 1 & 1 & 2 & 2 \end{bmatrix}^T$$

is not orthogonal array. For more information see e.g. [Tang 1993].

Picking Sampling Technique

Above we have introduced a few sampling techniques, but in literature we can find numerous other techniques as well. Hence how to pick the best sampling technique for problem in hand. As we remember the training data must be as diverse as possible so that NN learns the whole problem as good as possible since it is not good for extrapolation. In our numerical experiment we are trying to answer this question as discovered sampling techniques are used to create inputs for simulator to work the outputs.

In [Simpson, Lin, and Chen 2001] it is shown that Hammersley sampling is the most appropriate choice for sampling technique when we are building a surrogate model. As it achieved accurate approximation error globally, although it did not perform so good in maximum error. The global error tent to lower when sample set gets larger when using Hammersley. The orthogonal array sampling got low maximum error, but author recommend that low global error is better than lower maximum error in single approximation.

Next we show simple visualization of Latin hypercube sampling, Hammersley sampling and Orthogonal array. In example the number of points is 100 and the number of variables is 2, hence orthogonal design is $OA(100,2,10,2)$ and they are bounded to [0,1], for visualization

see Figure 13. As we see in visualization Hammersley sampling is covering the sampling area more efficiently than Latin Hypercube and Orthogonal array. Although Latin Hypercube and Orthogonal array might be covering the minimum and maximum points more efficiently.

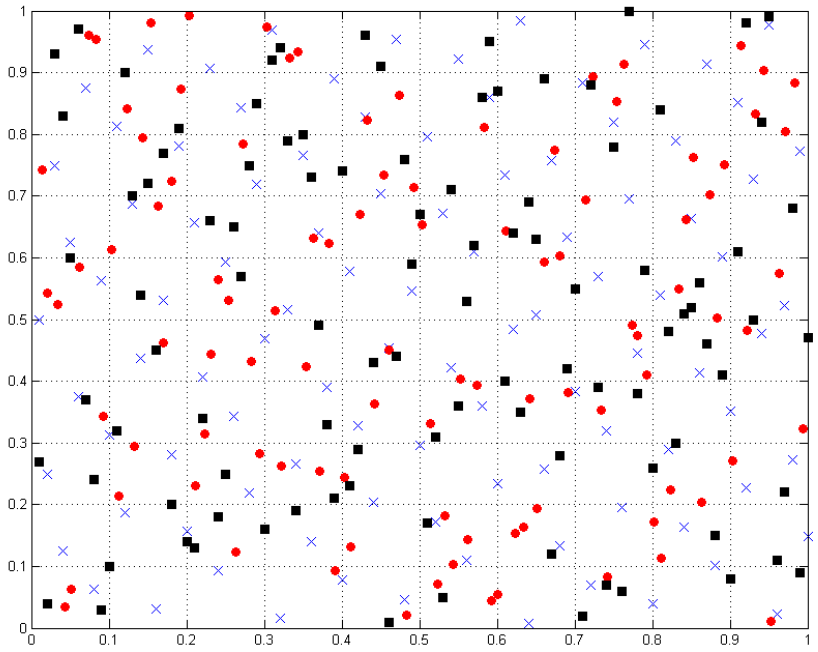

Figure 13: Visual comparisons of Latin Hypercube sampling, Hammersley sampling and Orthogonal array sampling. 'o' are points from Latin Hypercube sampling, 'x' are points from Hammersley sampling and ■ are points from Orthogonal array sampling.

2.3.2 Supervised training

Error correction method

One way to train a NN is by minimizing the error between the computed output y and the desired output d. Training the NN with this method is done by using $delta - rule$ [Widrow and Hoff 1960]. Where we need an error $e(ep)$ as

$$e_i(ep) = d_i(ep) - y_i(ep), i = 1, ..., k \tag{2.33}$$

where y is the computed output, d is the desired output and ep is the number of epochs and k is the number of neurons. A cost function $E(ep)$ to be minimized

$$E(ep) = \frac{1}{2}\sum_{i=1}^{k} e_i(ep)^2, \tag{2.34}$$

where ep is the number of epochs and k is the number of outputs.

$$\triangle w_{kj}(ep) = \eta e_k(ep) x_j(ep), \tag{2.35}$$

where $\triangle w(ep)$ is change in the weight matrix, η is learning-rate parameter, $e(ep)$ is error and x is input. This yields to weight update formulate

$$w_{kj}(ep+1) = w_{kj}(ep) + \triangle w_{kj}(ep), \tag{2.36}$$

where $w(ep+1)$ is a new weight matrix and $w(ep)$ is the current weight matrix. The learning-rate parameter η (see eq. (2.35)) determines how fast or accurately weights converge towards the optimum point. If η is too small the convergence will take longer time, but if it is too large convergence might start oscillating around the optimum point. Definition for too small or large depends on problem. We can consider the results from [Thimm and Fiesler 1997] as a guideline. Thereby the bounds for the learning rates are

Linear TF	[0.004, 0.7]
Log-sigmoid TF	[0.1, 20.0]
Hyperbolic tangent TF	[0.005, 2.5]

Table 4: Guidelines for the learning rate.

Training a Multilayer Perceptron

There are two ways for updating the synaptic weights.

1. *Sequential mode*: In sequential mode the weights are updated instantly after an error for training pattern, input/output pair, is calculated.
2. *Batch mode*: In batch mode the weights are updated after an error is calculated for every training pattern.

As an example of a training algorithm we consider backpropagation algorithm [Rumelhart, Hinton, and Williams 1986]. In [Rumelhart, Hinton, and Williams 1986; Haykin 1999, it is shown that the sequential mode is computationally faster and require less memory than the batch mode. Algorithm for sequential mode (see Algorithm 1) and for batch mode (see Algorithm 2). The weight updates in training algorithms involves calculation of local gradients. The formulas to calculate local gradients, in general, for hidden and output neurons (see Table 5) and local gradient derivations for log-sigmoid TF and hyperbolic tangent TF (see Table 6). For a complete derivation of formulas to calculate local gradient see e.g. [Haykin 1999]

Layer	Local Gradient
Output	$\delta_k = e f'_k(z^o_k)$
Hidden	$\delta_j = f'_j(z^l_j) \sum_{i=0}^{p_1} \delta_i^{l+1} w_{ij}^{l+1}$

Table 5: Local Gradients for neurons in output and hidden layers.

Layer	Function	Local Gradient
Output	$f(z_k) = \frac{1}{1+exp(-az_k)}$	$\delta_k = a y_k [d_k - y_k][1 - y_k]$
Hidden	–"–	$\delta_j = a y_j [1 - y_j] \sum_{i=0}^{k} \delta_i w_{ij}$
Output	$f(z_k) = atanh(bz_k)$	$\delta_k = \frac{b}{a}[d_k - y_k][a - y_k][a + y_k]$
Hidden	–"–	$\delta_j = \frac{b}{a}[a - y_j][a + y_j^c] \sum_{i=0}^{k} \delta_i w_{ij}$

Table 6: Local Gradient derivations for Log-Sigmoid TF and Hyperbolic Tangent TF

Algorithm 1 Sequential mode backpropagation algorithm

Step 1: *Initialization*. Build a NN, preprocess inputs/outputs and initialize the weights so that they are not zeros. Define maximum numbers of epochs.

Step 2: *Error calculation*. Error calculation for NN with current weights. Present training set $(x(tp), d(tp))$, x is the input vector, d is the desired output vector and tp is the number of training patterns, for NN. Approximate the output values y using initialized weights. Then calculate an input value z for every neuron

$$z_j(tp) = \sum_{i=0}^{p_1} w_{ji}^l(tp) y_i^{l-1}(tp), \tag{2.37}$$

where superscript $l = [I, II, ..., o]$ implies to the number of layers, p_1 is the number of neurons and y^{l-1} is the output from neurons in $l-1$ layer. For simplicity let's assume that in every layer has the same number of neurons. Then calculate the error

$$e(tp) = \frac{1}{2} \sum_{i=1}^{k} (d_i(tp) - y_i(tp))^2. \tag{2.38}$$

Step 3: *Local gradients*. Calculate the local gradients (δ) for every neuron. Firstly, calculate the local gradients for neurons in output layer. Secondly, calculate the local gradients for neurons in the previous hidden layer and continue going backwards until the input layer is met. For the local gradients see Table 5 and local gradient derivations for $Hyperbolic\ tangent\ TF$ and $Log-Sigmoid\ TF$ see Table 6.

Step 4: *Update weights*. After all local gradients are calculated we can update weights using

$$w_{kj}(tp+1) = w_{kj}(tp) + \eta \delta_k(tp) x_j(tp), \tag{2.39}$$

where $\delta(tp)$ is the local gradient for a neuron and η is the learning rate. It may take any value, for guideline see Table 4. An epoch (ep) is done when all the weights are changed according the error for every training pattern (tp).

Step 5: *Iterate*. Repeat steps 2, 3 and 4 as long as the maximum number of epochs is met or stopping criteria is met. For a stopping criteria see eq. (2.43).

Algorithm 2 Batch mode backpropagation algorithm

Step 1: *Initialization*. Build a NN, preprocess inputs/outputs and initialize the weights so that they are not zeros. Define maximum numbers of epochs.

Step 2: *Error calculation*. Error calculation for NN with current weights. Present training set $(x(tp), d(tp))$, x is the input vector, d is the desired output vector and tp is the number of training patterns, for NN. Approximate the output values y using initialized weights. Then calculate an input value z for every neuron

$$z_j(tp) = \sum_{i=0}^{p_1} w_{ji}^l(tp) y_i^{l-1}(tp), \tag{2.40}$$

where superscript $l = [I, II, ..., o]$ implies to the number of layers, p_1 is the number of neurons and y^{l-1} is the output from neurons in $l-1$ layer. For simplicity let's assume that in every layer has the same number of neurons. Then calculate the error of current epoch (ep)

$$e(ep) = \frac{1}{2} \sum_{i=1}^{tp} \sum_{j=1}^{k} (d_j(i) - y_j(i))^2. \tag{2.41}$$

Step 3: *Local gradients*. Then calculate the local gradients (δ) for every neuron. Firstly, calculate the local gradients for neurons in output layer. Secondly, calculate the local gradients for neurons in the previous hidden layer and continue going backwards until the input layer is met. For the local gradients see Table 5 and derivations from $Hyperbolic\ tangent\ TF$ and $Log-Sigmoid\ TF$ see Table 6.

Step 4: *Update weights*. After all local gradients are calculated we can update weights using

$$w_{kj}(ep+1) = w_{kj}(ep) + \eta \delta_k(ep) x_j(ep) + \alpha w_{kj}(ep-1), \tag{2.42}$$

where $\delta(ep)$ is the local gradient for a neuron, η is the learning rate and α is the momentum which determinate how much the weight change of previous epoch $ep-1$ effects on the new weight. It may take values between 0 and 1.

Step 5: *Iterate*. Repeat steps 3 and 4 as long as the maximum number of epochs is met or stopping criteria is met. For a stopping criteria see eq. (2.43).

One way to define a stopping criteria is to set a small positive scalar value e.g. $\varepsilon = 10^{-6}$ and when

$$d(E_{av}(ep), E_{av}(ep-1)) < \varepsilon \tag{2.43}$$

the algorithm will stop, where $d(...)$ is the Euclidean distance. Another way to do this is by cross-validation [Stone 1974]. We divide data to two separate sets, then another set is used for training and another for validating. After every epoch we test how the network generalize some input-output pair from the validation set and when generalization performance is good enough the training stops.

Training a Recurrent Multilayer Perceptron

For training a recurrent network we can use truncated back-propagation through time (BPTT(h)) algorithm [Williams and Peng 1990]. This is an extended version of standard sequential back-propagation algorithm and truncation means that we store and track the outputs to some time step h. Another version of BPTT is epochwise and it can be seen as an extended version of standard batch back-propagation algorithm [Williams and Peng 1990]. For optimization we can use same techniques than in MLP. The local gradient for neuron j in BPTT(h) is

$$\delta_j(t_c) = \begin{cases} f'(v_j(t_c))e_j(t_c) & \text{, when } t_c = t_e \\ f'(v_j(t_c))\sum_{k\in A} w_{kj}(t_c)\delta_k(t_c+1) & \text{, when } t_e - t_h < t_c < t_e, \end{cases} \tag{2.44}$$

where A indicates group of all synaptic weight, which include feedback loop weights and ordinary connection weights, t_c is the current time, t_h is the last time we remember and t_e is the ending time. When we get back to time step $t_e - t_h + 1$ the adjustment for the weights is

$$\Delta w_{ji}(t_c) = \eta \sum_{t_c = t_e - t_h + 1}^{t_e} \delta_j(t_c) x_i(t_c - 1). \tag{2.45}$$

When using gradient-based learning algorithms, like BPTT, recurrent networks may suffer from gradient vanishing problem. It means that during the training the inputs might not have any effect for training and training becomes impossible to finish. We can overcome this problem by using more complex training algorithms e.g. real-time recurrent learning [Williams and Peng 1990] and decoupled extend Kalman filter [Puskorius and Feldkamp 1994].

Training a Radial Basis Function network

Training a RBF network is about selecting the centers and calculating optimal weights for it. Centers can be chosen at least four different ways. First way is to set every input as to a center. This is not very efficiently and dimensionality will be the same as the number of inputs. Second approach is to select centers at random [Lowe 1989]. Let

$$f(\|x-t_i\|^2) = exp(-\frac{m}{d^2}\|x-t_i\|^2), \;\; i = 1,...,m, \tag{2.46}$$

where m is the number of centers and d is the distance between centers. So basically there is nothing random in this just that centers may not be in training data. Third method is self-organized selection of centers [Moody and Darken 1989]. This method contains two sections. First we estimate appropriate locations for the centers and secondly we train the weights between hidden layer and output layer. Supervised selection of centers is the fourth approach [Haykin 1999]. For this we need a cost function to be minimized

$$E = \frac{1}{2}\sum_{j=1}^{tp} e_j^2 \tag{2.47}$$

and

$$e_j = y_j - \sum_{i=1}^{m} w_i f(\|x_j - t_i\|_{C_i}), \tag{2.48}$$

where tp is the number of training patterns, m is the number of centers and y is the desired output. Parameters which we need obtain are weights w, centers t and spread C . The formulas for updating weights, locations of the centers and spread of the centers [Haykin 1999]. Formula for update weights is

$$w_i(ep+1) = w_i(ep) - \eta_1 \sum_{i=1}^{n} e_j f(\|x_j - t_i(ep)\|_{C_i}), \tag{2.49}$$

for locations of the centers

$$t_i(ep+1) = t_i(ep) - \eta_2 2 w_i(ep) \sum_{j=1}^{n} e_j(ep) f'(\|x_j - t_i(ep)\|_{C_i}) \Sigma_i^{-1}[x_j - t_i(ep)] \tag{2.50}$$

and for spread of the centers

$$\Sigma_i^{-1}(ep+1) = \Sigma_i^{-1}(ep) + \eta_3 w_i(ep) \sum_{j=1}^{n} e_j(ep) f'(\|x_j - t_i(ep)\|_{C_i})[x_j - t_i(ep)][x_j - t_i(ep)]^T, \tag{2.51}$$

where η_1, η_2 and η_3 are learning rates, ep is the number of epochs, n is the number of inputs and $f'(...)$ is the first derivative of the RBF with respect to its arguments. These updates are done until the wanted error is obtained or generalized cross-validating by [Craven and Wahba 1979], when it meets stopping criteria.

2.3.3 Unsupervised training

Unsupervised learning we have a training data, which contains only the inputs x. The objective is to categorize, discover features or regularities in the training data [Hassoun 1995]. We introduce briefly Hebbian learning and competitive learning for more comprehensive descriptions see e.g. [Haykin 1999; Hassoun 1995].

Hebbian learning

Hebb's postulate of learning is the oldest and the most famous of all the learning rules [Haykin 1999]. The original rule of Hebb's was that active synaptic weights, in brain, will grow and inactive synaptic weights will weaken [Hebb 1949]. That has motivated the learning rules in the field of artificial neural networks. The simplest form of Hebbian learning rule for weight update

$$w_{kj}(ep) = w_{kj} + \eta y_k(ep) x_j(ep), \tag{2.52}$$

where $\eta > 0$ is learning rate, y is the output and x is the input. In [Haykin 1999] is shown that repeatedly active weight will same point saturate the TF. Hence the information stored in weights and selectivity will be lost. Hence there is need for modified Hebb's rules see e.g. [Oja 1982].

Competitive learning

Like the name implies the neurons are competing among each other's to become active. Unlike in the Hebbian learning only one neuron can be active at the time. The simplest competitive network is a single layer network and it may contain feedback u among neurons. A possible choice for u is [Hassoun 1995]

$$u_{ij} = \begin{cases} 1 & \text{, if } i = j \\ -\varepsilon & \text{, } i \neq j, \end{cases} \tag{2.53}$$

where $0 < \varepsilon < 1/N$, N is the number of neurons. The neurons compete with the size of its input $z = \sum_{i=0}^{n} w_i x_i + u$ and the winner takes output signal to be 1 and others take 0. Hence we may write

$$y_k = \begin{cases} 1 & \text{, if } z_k > z_j,\ j \neq k \\ 0 & \text{, otherwise.} \end{cases} \tag{2.54}$$

Then the network learns be adding weight to the winner's weights. Hence we get weight update as

$$\Delta w_{kj} = \begin{cases} \eta(x_j - w_{kj}) & \text{, if the neuron } k \text{ wins} \\ 0 & \text{, otherwise.} \end{cases} \tag{2.55}$$

Overall effect of this rule is that weights of the winning neuron are moving towards inputs x [Hassoun 1995].

2.3.4 Supervised training as an optimization problem

Supervised learning can be seen as an optimization problem. The error is now seen as a surface and it is a function of the weights. The error surface can be formulated using Taylor series

$$E(w(ep) + \Delta w(ep)) = E(w(ep)) + \delta^T(ep)\Delta w(ep) + \frac{1}{2}\Delta w^T(ep)H(ep)\Delta w(ep), \tag{2.56}$$

where $\delta = \frac{\partial E(w)}{\partial w}$ is the local gradient and $H = \frac{\partial^2 E(w)}{\partial^2 w}$ is the Hessian matrix [Haykin 1999]. As we optimize, we are searching the bottom of the error surface "bowl" (see Figure 14).

Base optimization technique is the method of steepest descent, where the weight update take formula

$$\Delta w(ep)) = -\eta\,\delta(ep). \tag{2.57}$$

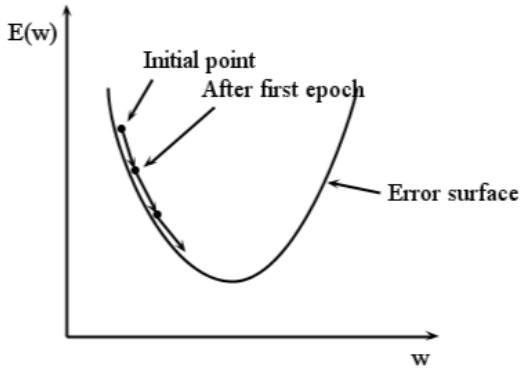

Figure 14: The fundamental of using steepest descent to find the optimal weight vector, which lie in the bottom of the error surface bowl.

It is operating as a linear approximation in the local neighborhood of operating point w. Now the δ is the only information about the error surface and convergence towards the optimum weight vector will take a long time. To improve convergence speed we need higher-order information about the error surface. It can be done by creating a quadratic approximation of the error surface around the current point $w(ep)$. This yields to optimal value

$$w^*(ep) = H^{-1}(ep)\delta(ep), \tag{2.58}$$

where H^{-1} is inverse Hessian matrix, assuming that it exists [Haykin 1999]. This kind of formula can be solved with Newton's method using only one iteration. Although it is impractical for three reasons [Haykin 1999]:

- H^{-1} can be computationally expensive to solve.
- There is no guarantee that H^{-1} exists.
- When E has nonquadratic formula, there is no guarantee for convergence with Newton's method.

Quasi-Newton method can overcome those, when it only needs approximation of δ. However, quasi-Newton's computational complexity is $O(W^2)$, where W is the size of the weight

vector, hence it is impractical for large problems. Conjugate-gradient method has been developed to overcome the computational complexity of Newton's method and quasi-Newton methods and to be faster than steepest descent method. Conjugate-gradient methods computational complexity is $O(W)$, hence it is more suitable for large problems and therefore suitable for training the MLP [Haykin 1999].

2.4 Performance

Performance of neural network depends on problem and purpose of the network, thus it is not unambiguous. The generalization ability is typically the most important feature to achieve. In case we need to retrain NN multiple times during the problem the time spend on NNs training must be kept small, then generalization accuracy might not be that important. As the training time is machine dependent the best choice for measuring it is CPU time used in training [Prechelt 1994]. In some problems we might not be able produce the best NN for a problem, but it is illustrating the problem behavior so that it can be used as a surrogate (see Figure 15).

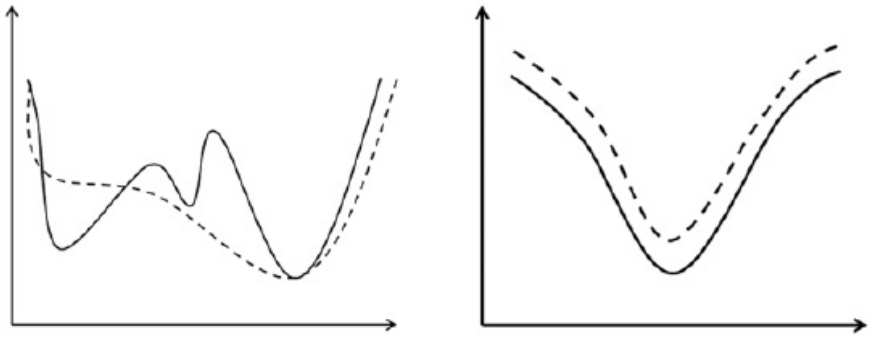

Figure 15: Examples of surrogates that have a large approximation error but are adequately good for evolutionary search. Solid curves detonate the original function and dashed curves are their approximation. [Jin 2011]

A feature which is not problem dependent is overtraining of the NN. Basically overtraining means that the NN has learned features of the training data, not the features of the problem. Overtraining might happen when the NN has too many hidden layers or hidden neurons. Illustration of the overtrained network, therefore the data is overfitted, is shown in Figure 16.

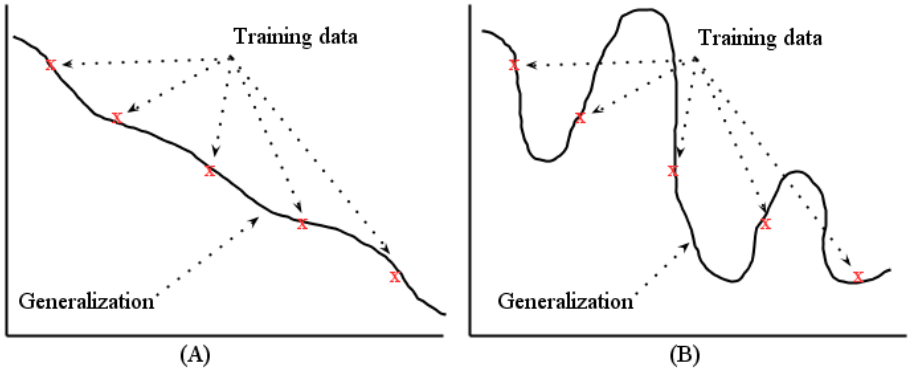

Figure 16: (A) Properly fitted data (good generalization). (B) Overfitted data (poor generalization). [Haykin 1999]

In next subsection we introduce some metrics to measure an error, some of them are already used in previous sections. In subsection 2.4.2, we introduce some definitions of how accurate NN can be and what is needed to achieve the wanted accuracy. In subsection 2.4.3, we introduce technique to compare two equally performing NNs, in errorwise, which can be compared via neural metrics [Leung and Simpson 2000] to determine their complexity. In this work we are interested only about function approximation so these metrics might not be applicable in pattern recognition and classification problems. For more information about pattern recognition and classification see e.g. [Bishop 1995].

2.4.1 Error metrics

Error is measured when the NN is trained and validated. A measured error determines how much the weights are altered and when to stop the training. The ones most commonly used are mean absolute error (MAE) (2.59), mean squared error (MSE) (2.60), rooted mean squared error (RMSE) (2.61) and sum of squared errors (SSE) (2.62). Any of these measures can be used as a cost function when training a NN. The data could be corrupt with noise, outliers, etc., which might lead to unwanted result on capturing the properties of the data, so the purpose of the metrics is to capture the "right" properties from the data and to abandon the unwanted properties. In [Bishop 1995] it is suggested to use SSE for training the NN and RMSE to test it. To use SSE for training, because it does not need a priori knowledge about

the outputs distribution. To use RMSE for testing, because its value does not grow with the number of testing patterns and from value of RMSE you can see if the approximation is perfect, when RMSE=0, and if it is "in the mean", when RMSE is output distributions variation. In [Twomey and Smith 1995] is recommend that practitioners should examine the minimum obtained from the different metrics, thus that study is for classification problems. It would be reasonable for function approximation problems as well, hence it was concluded that large approximation values might be misleading the measured error. So that we might abandon a NN, which might be good for most of the points, but it incorrectly approximates a few test patterns.

Mathematical formulation for error are for MAE

$$E = \frac{1}{N}\sum_{i=1}^{N}|y_i - d_i|, \tag{2.59}$$

for MSE

$$E = \frac{1}{N}\sum_{i=1}^{N}(y_i - d_i)^2, \tag{2.60}$$

for RMSE

$$E = \sqrt[2]{\frac{1}{N}\sum_{i=1}^{N}(y_i - d_i)^2}, \tag{2.61}$$

and for SSE

$$E = \frac{1}{2}\sum_{i=1}^{N}(y_i - d_i)^2, \tag{2.62}$$

where N is the number of data patterns, $y = [y_1, ..., y_N]$ is the approximated output and $d = [d_1, ..., d_N]$ is the desired output. A metric used in accuracy section is integrated squared error (ISE) and it takes form

$$E = \int (f(x) - f_n(x))^2 dx, \tag{2.63}$$

where $f(x)$ is unknown density and $f_n(x)$ is the estimation function, e.g. a neural network.

2.4.2 Theoretic Accuracy

A most theorems considering accuracy in NNs are derivate to use in a single hidden layer neural network. In 1993, Barron showed that for every choice of fixed basis functions

$h_1, ..., h_n$ there exists error bound measured via ISE

$$\inf_{h_1,...,h_n} \sup_{f \in \Gamma_C} d(f, span(h_1, ..., h_n)) \geq \frac{C}{8\pi\tau d} \left(\frac{1}{n}\right)^{\frac{1}{d}}, \tag{2.64}$$

where n is the number of hidden neurons, d is the dimension of input space, $\tau \geq exp(\pi - 1)$ is a universal constant and C needs more definitions as follows. In here the unknown density in ISE is a class of functions on $\mathbb{R}^d$ for which are Fourier presentation of the form

$$f(x) = \int_{\mathbb{R}^d} exp(i\omega x)\bar{f}(\omega)d\omega, \tag{2.65}$$

$$C_f = \int_{\mathbb{R}^d} \sqrt[2]{\omega * \omega}\bar{f}(\omega)d\omega. \tag{2.66}$$

For each $C > 0$, let Γ_C be the set of function f such that $C_f \leq C$. These results implies to RBF network as well. [Barron 1993] Another proof results that

$$\int_{\mathbb{R}^d} (f(x) - f_m(x))^2 dx \leq \frac{c^2 - \|f\|_\mu^2}{m}, \tag{2.67}$$

where $c > \|f\|_\mu$, $\|f\|_\mu^2 = \int_{\mathbb{R}^d} (f(x))^2 \mu dx$ and μ is a given positive measure on $\mathbb{R}^d$. Roughly, this says that $2m$ parameters can achieve an error of $O(1/\sqrt{m})$, where m is the number of weights. [Dingankar and Sandberg 1997]

It is proved that NN with specific TF has no lower bound for error, thus the author's claim that TF is highly complex, hence it is not good for implementations. The structure of NN was $3d$ neurons in the first hidden layer and $6d+3$ neurons in the second hidden layer, where d is the number of inputs. [Maiorov and Pinkus 1999]]

The size of training data has an effect to the accuracy as well. It is said in [Haykin 1999] that

$$N = O\left(\frac{w}{\varepsilon}\right), \tag{2.68}$$

where N is the number of training patterns, w is the number of weights and ε is the wanted error. In other words this says that if we have NN with 1000 weights and we want 10% accuracy then we need 10000 training patterns to achieve the demanded accuracy.

2.4.3 Neural metrics

A way to measure neural networks is via software metrics [Leung and Simpson 2000]. This can be used when we have two or more equally performing NNs and in entity they are measuring the complexity of the NN. These metrics are measuring neural networks understandability (UN), modifiability (MO), testability (TE), applicability (AP), consistency (CS), structuredness (ST), scalability (SC), efficiency (EF) and complexity (CO). Parameters, which are needed for measuring and what they correlate, are shown in Table 7.

Type	Metric	Definition	Quality Measurement
Primitive	n_1	Number of inputs	UN, ST, EF & CO
	n_k	Number of hidden neurons on the kth layer	UN, ST, EF & CO
	n_m	Number of outputs	UN, ST, EF & CO
	m	Number of layers	UN, ST, EF & CO
	M	Number of epochs	TE, AP, EF & CO
	P	Number of input patterns	MO, TE, CS, SC, EF & CO
Computed	N_k	Number of hidden neurons on the kth layer	UN, ST, EF & CO
	N_m	Number of output weights	UN, ST, EF & CO
	N	Number of weights in network	UN, ST, EF & CO
	S	Number of scaling operations	SC, EF & CO
	ACT	Number of transfer function invocations	EF & CO
	ADD	Number of additions and subtractions	EF & CO
	MUL	Number of multiplication and divisions	EF & CO
	TOT	Total number of operations	EF & CO

Table 7: Neural metrics for Backpropagation Neural Networks. [Leung and Simpson 2000]

Mathematical formulations for ADD, MUL, ACT and TOT are in sequential backpropagation

$$ADD(M,m,n_1,...,n_m) = M(\sum_{s=2}^{m-1} n_s(n_{s+1}+2n_{s-1}-1)+n_m(2n_{m-1}+1)), \tag{2.69}$$

$$MUL(M,m,n_1,...,n_m) = M(\sum_{s=2}^{m-1} n_s(n_{s+1}+3n_{s-1}+2)+n_m(3n_{m-1}+2)), \tag{2.70}$$

$$ACT(M,m,n_1,...,n_m) = M(\sum_{s=2}^{m} n_s), \tag{2.71}$$

$$TOT(M,m,n_1,...,n_m) = M(\sum_{s=2}^{m-1} n_s(2n_{s+1}+5n_{s-1}+2)+n_m(5n_{m-1}+4)), \tag{2.72}$$

And in batch backpropagation

$$ADD(P,M,m,n_1,...,n_m) = PM(\sum_{s=2}^{m-1} n_s(n_{s+1}+n_{s-1})+n_m(n_{m-1}+2))+M\sum_{s=2}^{m} n_s n_{s-1}, \tag{2.73}$$

$$MUL(P,M,m,n_1,...,n_m) = PM(\sum_{s=2}^{m-1} n_s(n_{s+1}+3n_{s-1}+2)+n_m(3n_{m-1}+1)), \tag{2.74}$$

$$ACT(P,M,m,n_1,...,n_m) = PM(\sum_{s=2}^{m} n_s), \tag{2.75}$$

$$TOT(P,M,m,n_1,...,n_m) = PM(\sum_{s=2}^{m-1} n_s(2n_{s+1}+4n_{s-1}+3)+n_m(4n_{m-1}+5))+M\sum_{s=2}^{m} n_s n_{s-1}). \tag{2.76}$$

As said earlier that exact training time can be compared when using same software and hardware, thus this gives metrics to measure the training time, namely the number of calculations, which is independent from software and hardware.

2.5 Neural network applicability

In this section we give two examples for each NN structures discovered in section 2.2. Examples demonstrated their applicability for classification problem, namely XOR-problem (see Table 8), and for function approximation (see eq. (2.79)). Examples are made with Matlab 2011 & Neural network toolbox and codes can be found in Appendix B.

In the XOR-problem, we have two inputs and an output, each of inputs may take value 0 or 1 and the output depends from those, if the inputs are equal the output is 0 and if the inputs are unequal the output is 1.

Inputs	Outputs
[0, 0]	0
[0, 1]	1
[1, 0]	1
[1, 1]	0

Table 8: XOR-problem values

The second example is about function approximation, which is append with some noise. Let the real value of problem be

$$f_r(x) = cos(x), \tag{2.77}$$

the noise, which may come for different sources, cables, machineries, etc., and makes an error to the measured data,

$$f_n(x) = \frac{1}{2} sin(x^2) \tag{2.78}$$

and the summary of these

$$f_s(x) = cos(x) + \frac{1}{2} sin(x^2). \tag{2.79}$$

In the next subsection the SLP is applied to solve these problems. In subsection the MLP is applied. In subsection 2.5.3 the RMLP is applied. And in subsection 2.5.4 the RBF network is applied.

2.5.1 Single Layer Neural Network

In this subsection SLP is applied to solve two examples shown in previous section. In Figure 17 it is shown that a SLP cannot classify XOR-problem correctly. This happens because it can only produce straight lines.

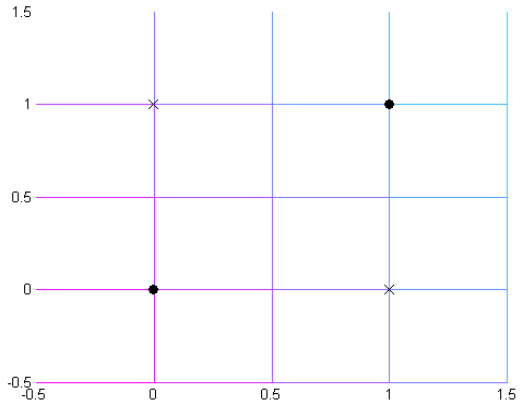

Figure 17: XOR-problem solved with single layer perceptron network. 'O's are valued as 0 and 'X's are valued as 1.

It is shown in Figure 18 that SLP cannot approximate nonlinear function. In addition effect of the bias is demonstrated that produces an affine transformation to approximation.

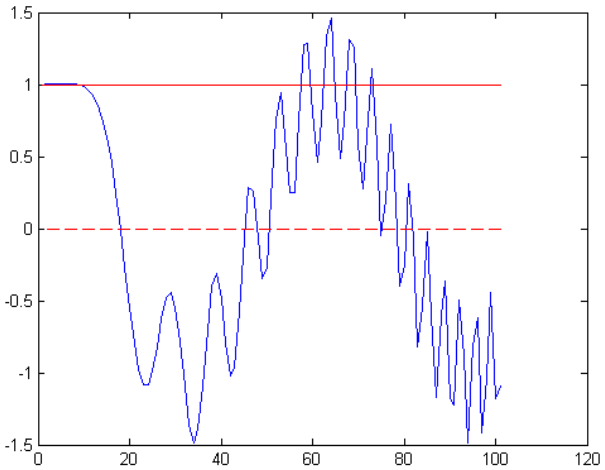

Figure 18: Function approximation using a SLP and effect of the bias. Solid (blue) line is the function, solid (red) line is approximation with bias and dashed (red) line is approximation without bias.

2.5.2 Multilayer Neural Network

In this subsection MLP is applied to solve examples shown in section 2.5. The MLP with one hidden layer and two hidden neurons can be used for solving the XOR-problem, which can be seen as a classification problem, whereas SLP could not do it. Result for XOR-problem is shown in Figure 19, where point 'x' is result 1 and 'o' is 0 and Pink (dark) area is classified as 1 and teal (light) area is classified as 0.

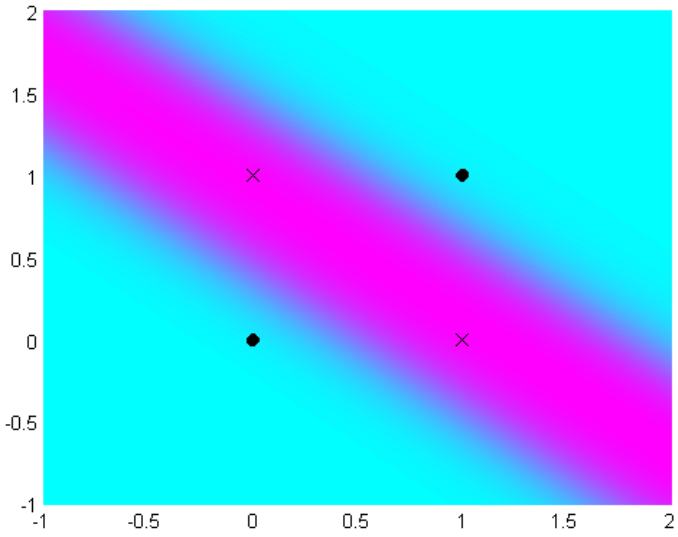

Figure 19: XOR problem solved with MLP. Pink (dark) area is classified as 1 and teal (light) area is classified as 0.

Another example is the function approximation (see (2.79)). Firstly a MLP with one hidden layer containing $tp-1=100$ neurons is used for approximation. The performance measured via mean squared error (MSE) is 9.20e-8 after 192 epochs and as shown in Figure 20 the approximation is exactly the function. Secondly a MLP with two hidden layers consisting $tp/2=50$ neurons in first hidden layer and 3 neurons in second hidden layer. After 1000 epochs the MSE is 0.000820 and as shown in Figure 21 the approximation is not as good as above. Although the number of epochs was multiplied from 192 to 1000. Thirdly a MLP with two hidden layers, where the number of hidden neurons is chosen randomly and let the number hidden neurons in first hidden layer be 10 and in second hidden layer 5. For this MLP a MSE is 0.0311 and it is obtained after 10000 epochs and as shown in Figure 22 the

approximation is not as good as in the first MLP and the second MLP.

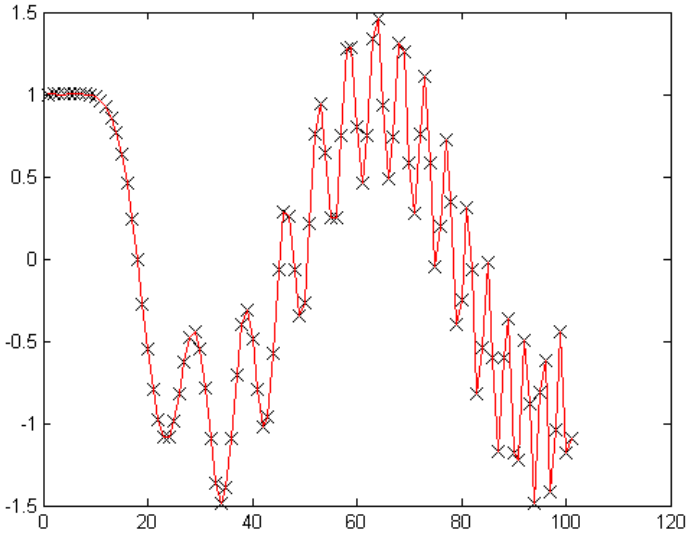

Figure 20: Function approximation using MLP one hidden layer. Line is the approximation and 'X's are the real values.

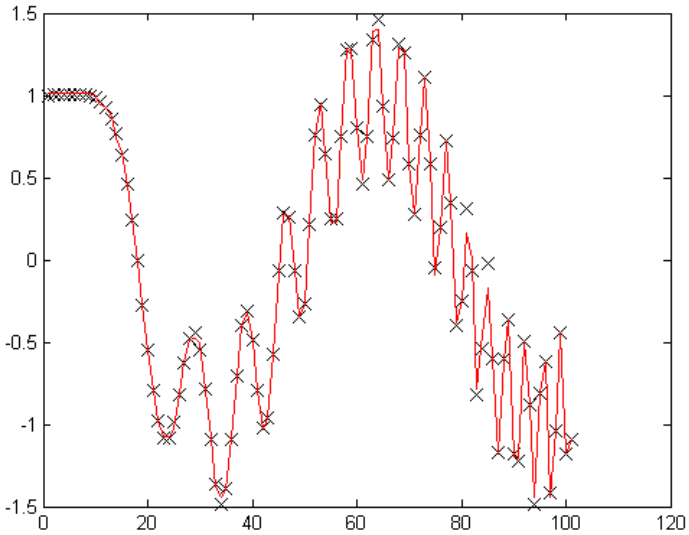

Figure 21: Function approximation using MLP with two hidden layers. Line is the approximation and 'X's are the real values.

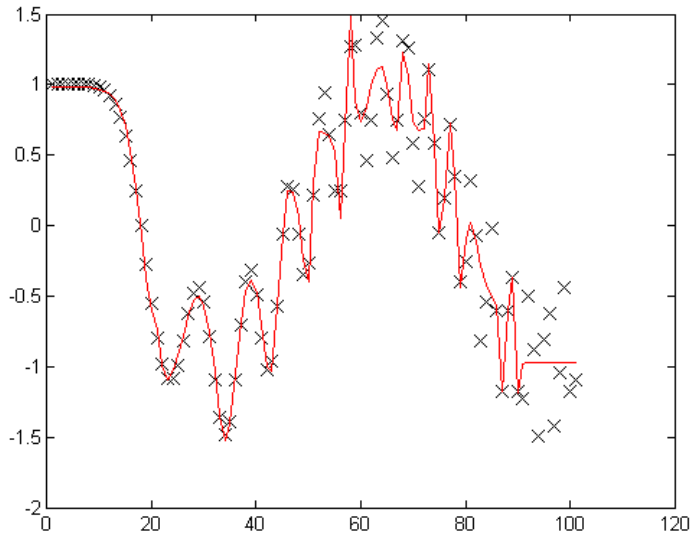

Figure 22: Function approximation using MLP with two hidden layers. Number of hidden neurons is random. Line is the approximation and 'X's are the real values.

2.5.3 Recurrent Multilayer Neural Network

In this subsection RMLP is applied only for function approximation example, because we are mainly interested about function approximation features of the NNs. The XOR-problem can be done e.g. by using Hopfield network [Brouwer 1997]. In Figure 23 (A) is the result of sequence identification and as we see the result follows nicely the original sequence and also removes the noise. The sequence prediction is illustrated in Figure 23 (B), where we see that the prediction is following the original sequence pass the last training time step but not very accurately. Although this result is not guaranteed and it took about 30 retraining to achieve. The real problem, where we do not know the future, it will be very hard to choose the right prediction.

2.5.4 Radial Basis Function Network

In this section we apply RBF network to solve examples shown in section 2.5. For solving the XOR-problem we need two RBFs and two centers. Let the centers be $t_1 = [0,0], t_2 = [1,1]$ and let the RBFs be Gaussian $f_1(x,t_1) = exp(-\|x-t_1\|^2)$ and $f_2(x,t_2) = exp(-\|x-t_2\|^2)$ when doing calculations we get results (see Table 9). Now the XOR-problem has returned to

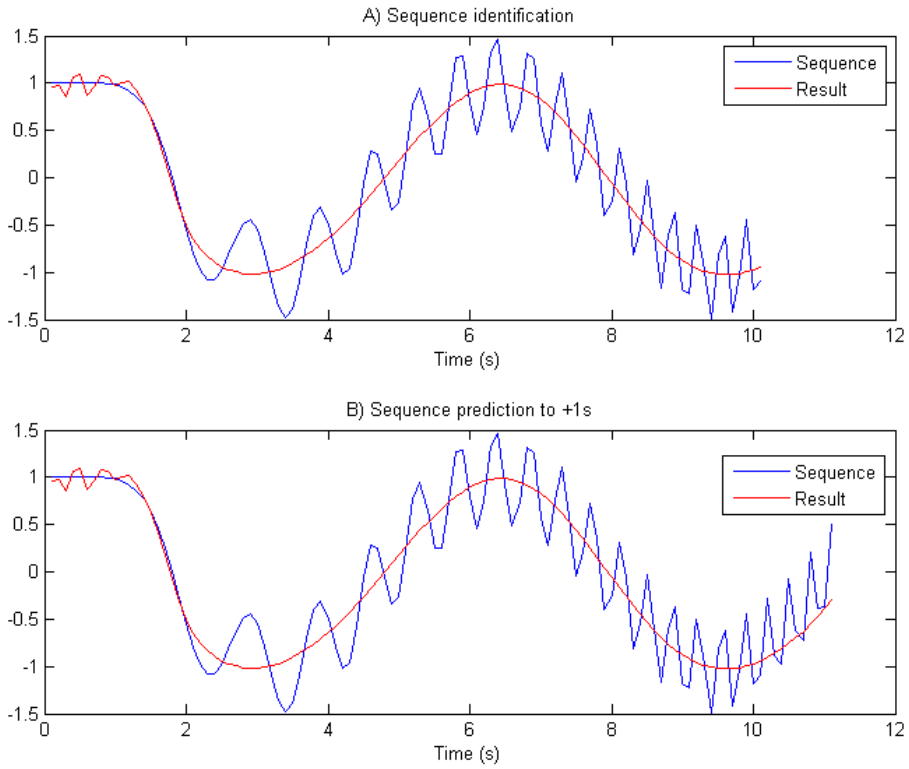

Figure 23: Recurrent Multilayer Perceptron example. A) Identification of the sequence B) Prediction of the sequence to +1 second.

linearly separable problem as shown in Figure 24 and points are easy to classify.

The function approximation is solved using two different RBF networks are employed and two different approximation error goals are set. Results are shown in Figure 25, where Xs are real values of the function, solid (blue) line is approximation using network (A) and dashed (red) line is approximation using network (B).

A RBF networks approximation error goal was set to 0,2 measured via MSE. Supervised selection of centers is performed and to reach the error goal only three centers was needed and therefore size of the network is pretty small. In Figure 25 these results can be seen as solid (blue) line and as we see the approximation is following the original function very smoothly.

B RBF networks approximation error goal was set to 0. Obtained error was 0.062375 and for that every input was needed to be as centers so the dimension of network is as

x	f_1	f_2
[0,0]	0.135	1
[1,0]	0.368	0.368
[0,1]	0.368	0.368
[1,1]	1	0.135

Table 9: XOR-problem results obtained using RBF network

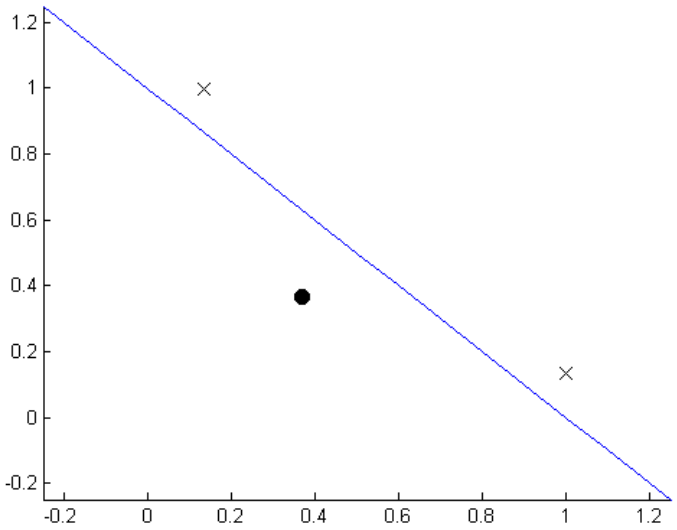

Figure 24: XOR-problem solved with RBF network. 'X's are valued as 1 and 'O' is valued as 0. Points are now linearly separable.

high as in similar example using MLP, but the error achieved is much bigger here. In Figure 25 dashed (red) line illustrates approximation result and we can clearly see that is not even nearly as good as in Figure 20.

In next chapter we present heuristics for improving the performance of NNs and backpropagation algorithm.

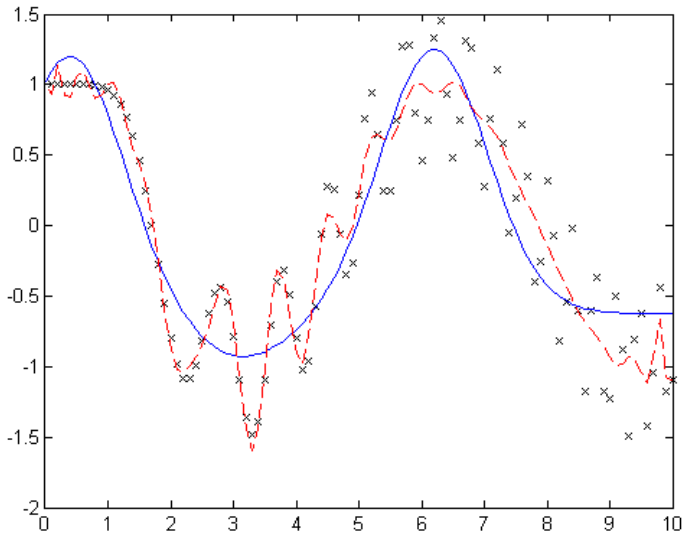

Figure 25: Function approximation using RBF network. 'X's are illustrating the real values of the function, solid (blue) line approximation when the goal was reach 0,2 error and dashed (red) line is the approximation when goal was to reach 0 error both measured via MSE.

3 Heuristics for improving the performance of Neural Networks

In this chapter we introduce some heuristics found in literature, which should help building a proper neural network. Heuristics might not give the best solution for each problem, but we are providing a starting point for neural network design. In the first section, we discuss about importance of inputs and their effect to NNs structure and learning. In the second section, we discuss about heuristics for structure of the NN and the number of neurons. In the last section, we discuss about how you can make the backpropagation algorithm to work efficiently.

3.1 Input dimension reduction

As NN are imitating brains some might think that they can somehow sort the training data to relevant and irrelevant data, then learn the relevant data only. This is not true, because when we are training the NN the main feature effecting to training is the error between the training data and the values calculated with NN. Hence, if we give irrelevant training data to NN, it will learn irrelevant features of the data and produce approximations according to those. In the next section, we present heuristics to determine structure of the NN and some of those are based on the number of inputs. Hence more inputs yields to more neurons and as the inputs are connected to the neurons via weights, this yields to increased number of weights. The less we have weights to train the less we need training patterns to make it more accurate or with same amount of training patterns we can make a smaller NN to be more accurate than a larger NN (see eq. (2.68)).

Reducing the number of inputs is not an easy task to complete. In [Fahmi and Cremaschi 2012], it could not be done because independent nature of the inputs. In [Jain and Nag 1995], reducing inputs resulted the performance to decrease rather than increase. The performance was found improving by 47% when the number of inputs was reduced from 51 to 5 [Schleiter et al. 1999]. In [Muknahallipatna and Chowdhury 1996] it is shown that dimension reduction maintain the same level of accuracy or improve it in MLP and RBF network. In [Zhu et

al. 1998] it is shown that decreasing inputs, in recurrent neural network, the performance of the recurrent neural network maintained in the same level. Hence, reducing the input dimension may increase the performance of the NN, but it can also decreased it, if it is applied incorrectly.

How can we determine the inputs, which we keep and which we reduce? Firstly, because NN designer cannot be an expert of all the domains, where NNs are used, designer has to discuss with domain experts. Their a priori knowledge of the domain might help to determine relevance of the inputs to the problem. Thereby the decision whether to keep or discard an input can be made. In [Walczak 1994], where the problem was to recognize students who apply to university and who do not, the number of inputs was reduced from 26 to 16 and the number of outputs was reduced from 3 to 2 with help of the domain experts.

Secondly, correlation of the inputs need to be calculated e.g. Pearson correlation matrix (see eq. (3.1) or [Pearson 1920]) or chi-square test (see [Chernoff and Lehmann 2012]).

$$\rho_{x,y} = \frac{cov(X,Y)}{\sigma_x \sigma_y} = \frac{E[(X-\bar{x})(Y-\bar{y})}{\sigma_x \sigma_y}, \tag{3.1}$$

where X and Y are variables, σ_x and σ_y are standard deviations of the X and Y, $\bar{x}$ and $\bar{y}$ are expected values and E is expected value operator. Pearson correlation takes values between -1 to 1, illustrations for different correlation coefficient see Figure 26. Hence, if there is high correlation between two inputs, then one of those two can be kept and other is reduced without adversely affecting NNs performance. High correlation is problem dependent and must be determined individually for every problem. In [Walczak and Cerpa 1999] it is suggested, that correlation $|\rho| > 0.2$ indicates a noise source, hence it can be taken as a guideline for a high correlation.

Thirdly, other statistical techniques can be applied e.g. regression [Sykes 1993] and factor analysis [Tryfos 1998]. Technique to be used depends on the distribution of the inputs. Multiple regression and factor analysis are working better for normally distributed data and logistic regression for a curvilinear data [Walczak and Cerpa 1999].

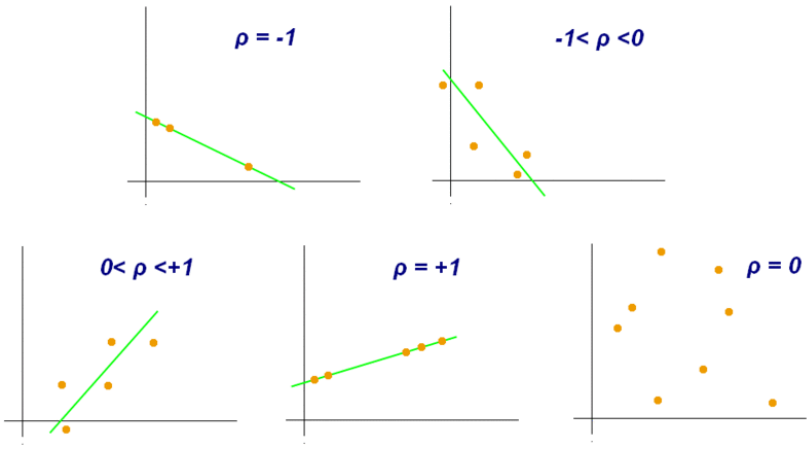

Figure 26: Correlation coefficient for Pearson correlation matrix. [http://en.wikipedia.org/wiki/Pearson_product-moment_correlation_coefficient]

3.2 Structure of the neural network

Neural networks with single hidden layer (MLP, RBF) are universal approximators and as shown in examples they can learn the data perfectly. In examples we were using heuristics, which will provide a perfect match for learning the training data, but those will not guarantee the accuracy for the data, which is not included in training data. On the other hand, if we have very large dataset ($> 1000 \ \ tp$), it makes no sense to have network with $tp - 1$ hidden neurons. The problem in hand determines some setups for neural network design e.g. the number of output neurons. In the first section, we will give some heuristics for the number of hidden layers. In the second section, we will provide some heuristics for the number of hidden neurons. The size of RBF network is determined with the number of radial basis functions, which are chosen with different method than neurons in MLP, hence this heuristic do not imply to it.

3.2.1 Number of hidden layers

A base model for NN is SLP, but it can be used to classify linearly only separable problems, hence for real world problems it has limited applications. A MLP with a hidden layer is a universal approximator, so it can approximate any arbitrary function, although this feature might not guarantee the generalization abilities and it gives no guidelines for structure. As

said earlier a MLP with two hidden layers can approximate function with lesser number of neurons than MLP with one hidden layer.

The number of hidden layers is the trade-off between smoothness and accuracy. As a small number of hidden layers increases the smoothness of approximation and a greater number of hidden layers increases the accuracy of approximation. The complexity of the problem corresponds to the number of hidden layers, hence as the complexity of the problem grows should the number of hidden layers grow. According to [Walczak and Cerpa 1999] the MLP with certain numbers of hidden layers can approximate different types of planes:

- MLP with 1 hidden layer: can create a hyperplane.
- MLP with 2 hidden layers: can combine hyperplanes to form convex decision areas.
- MLP with 3 hidden layers: can combine convex decision areas to form convex decision areas that contain concave regions.

The most functions can be approximated with MLP with one hidden layer [Walczak and Cerpa 1999; Basheer and Hajmeer 2000], hence we suggest that designing structure of the MLP should start from single hidden layer and, if it cannot provide proper behavior, the size of hidden layers should be increased. This implies to design of recurrent MLPs as well.

3.2.2 Number of hidden neurons

The number of hidden neurons is a trade-off between training time and accuracy [Walczak and Cerpa 1999]. A greater number of hidden neurons will take longer time to train, but it can be more accurate than having a smaller number of hidden neurons, which takes less time to train. On the other hand, too many hidden neurons may yield to overtraining and generalization accuracy will be poor. Too few hidden neurons cannot learn the training data and generalization accuracy will be poor also. Heuristics for the number of neurons is based on the number of inputs/outputs, the number of training patterns or some mixture of these (see Table 10).

If we compare structure used in MLP example 1, where we had 101 training patterns and 300 weights and 1 output, to heuristic provided by Widrow, we get $300 \leq 101 \leq 2469$ and from that we see the structure of MLP is too large. Although examples purpose was to show

Heuristic formula	Reference
$HN = 50\% * (n+k)$	Piramuthu, Shaw, and Gentry 1994
$HN = 75\% * n$	Jain and Nag 1995
$HN = 2 * n + 1$	Fletcher and Goss 1993
$HN \leq \frac{tp}{R*(n+k)}$	Jadid and Fairbairn 1996
$0.11 * tp \leq HN * (n+1) \leq 0.3 * tp$	Lachtermacher and Fuller 1995
$HN = \sqrt{n * k}$	Masters 1993
$w = tp * log_2(tp)$	Walczak and Cerpa 1999
$\frac{w}{k} \leq tp \leq \frac{w}{k} log_2(\frac{w}{k})$	Widrow and Lehr 1990

Table 10: Heuristics for the number of hidden neurons. Descriptions for variables: HN is the number of neurons, n is the number of inputs, k is the number of outputs, w is the number of weights, tp is the number of training patterns and R takes values from 5 to 10.

that MLP can approximate a nonlinear function perfectly. Hence our suggestion is to pick some of these heuristics to determine an upper bound and a lower bound for the number of hidden neurons. Then by using those bounds start to search the best network structure for the problem and when an upper is reached then add a hidden layer and append it with a lower bound of neurons.

3.3 Backpropagation to work efficiently

In this section we introduce some heuristics for backpropagation algorithm to perform better. In first subsection we discuss about choice of neuron model. In second subsection we discuss about importance of initializing weights and how to do it. And in third subsection we introduce heuristics for determining learning rate and momentum.

3.3.1 Choice of transfer function

To build a NN we have basically two transfer functions, discovered in section 2.1, which we may choose: log-sigmoid TF and hyperbolic tangent TF. In [LeCun, Kanter, and Solla 1990] it is shown that neurons with an antisymmetric TF yield to faster convergence than neurons with nonsymmetric TF. TF is antisymmetric when $f(-z) = -f(z)$ and hyperbolic tangent TF meets this requirement. In [LeCun et al. 1998], a good choice for TF (see Figure 27) is founded to be

$$f(z) = 1.7159 * tanh(\frac{2}{3}z) \tag{3.2}$$

and it has following features:

- $f(1) = 1$ and $f(-1) = -1$.
- At the origin the slope of the TF is close to unity.
- The second derivative of $f(z)$ attains its maximum value at $z = 1$.

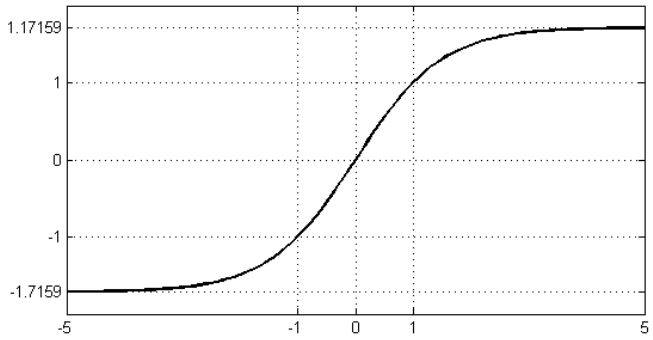

Figure 27: Specified hyperbolic tangent transfer function.

The learning time of backpropagation algorithm is sensitive to condition number $\lambda_{max}/\lambda_{min}$, where λ_{max} is the largest eigenvalue of the Hessian matrix and λ_{min} is the smallest eigenvalue. The inputs with nonzero mean have a larger ratio of $\lambda_{max}/\lambda_{min}$ than zero mean inputs, hence the smaller the ratio the better the result. The TF, which takes values in the interval [-1, 1], is more likely to get zero mean than TF, which takes values in the interval [0, 1]. [LeCun, Kanter, and Solla 1990].

Hence it can be suggested that TF to be used should be hyperbolic tangent TF and in particular TF eq. (3.2).

3.3.2 Initialization of the weights

Some might think that initialization of the weights is just a common practice in field of NN. In [Li et al. 1993] it is shown, that properly set initial weights have an effect to speed up backpropagation algorithm. In [Lee, Oh, and Kim 1991] it is shown that too large initial weight values cause premature saturation to neurons, which leads to slow learning even if the error is large and it is suggested that initial weights should be small. Although in [Hassoun 1995] it is shown that small initial weights produce a flat error surface and training will be slow. What is proper initial weight then? In [LeCun et al. 1998] formula for initial weights formula is derived by

$$\sigma_z^2 = w\sigma_w^2, \tag{3.3}$$

where σ_z is standard deviation of neurons inputs, w is number of weights and σ_w is standard deviation of initialized weights. If using TF as specified in eq. (3.2), then $\sigma_z = 1$ and we get

$$\sigma_w = \frac{1}{\sqrt{w}}. \tag{3.4}$$

In [Walczak and Cerpa 1999] it is shown that a good interval for initial weights is [-0.3, 0.3].

3.3.3 Learning rate and Momentum

The learning rate affects on how much the weights are altered due to the error. A large learning rate will make changes to be big and convergence will be fast, but this may result in oscillation near the optimum point and it may not be obtained. A small learning rate will make changes to be small and convergence will be slow towards the optimal points, but obtaining it might take a long time.

The learning rate can be a constant for all the time. Some heuristics for constant learning rates are shown in Table 11,

which are very similar to results in [Thimm and Fiesler 1997]. The learning rate may also vary over time (epochs) and in [Jacobs 1988] it is shown that adaptive learning rate makes backpropagation algorithm perform better than constant learning rate. The methods for adap-

Heuristic values for learning rate	Reference
0.1 - 10	Walczak and Cerpa 1999
0.1 - 0.9	Gasteiger and Zupan 1993
0 - 1	Walczak and Cerpa 1999

Table 11: Heuristics for constant learning rate.

tive learning rates are:

- Every parameter should have individual learning rate.
- Every learning rate should be allowed to vary over epochs.
- If parameters derivative is constant over several epochs then its learning rate should be increased.
- If parameters derivative alternates over several epochs then its learning rate should be decreased.

Here parameter mean weights and biases. Another term affecting the weight change is the momentum term. It has same kind of effect to learning process than learning rate. It can accelerate the weight updates when high learning rate might lead to oscillating, but too high momentum might lead to oscillating as well [Walczak and Cerpa 1999]. Heuristics for the momentum are shown in Table 12.

Heuristics for momentum	Reference
0.4 - 0.9	Walczak and Cerpa 1999
0 - 1	Hassoun 1995
1	Walczak and Cerpa 1999

Table 12: Heuristics for momentum.

As learning rate (η) and momentum (α) are related to each other, it is suggested in [Gasteiger and Zupan 1993] that $\eta + \alpha \approx 1$. Using $\eta = 0.9$ and $\alpha = 0.25$ has been found to be useful if no better solution is in hand [Walczak and Cerpa 1999].

4 Numerical experiments of Neural Network design.

Our numerical experiment is about creating a surrogate model for a control model as it is in [Sindhya et al. 2013], where APROS, a dynamic process simulator, is used. APROS takes about 15 seconds for single evaluation.

4.1 The Control Model

The control model consists of a mixing tank, a feed inlet, an outlet and a concentration control loop. The flow sheet of the control model is shown in Figure 28.

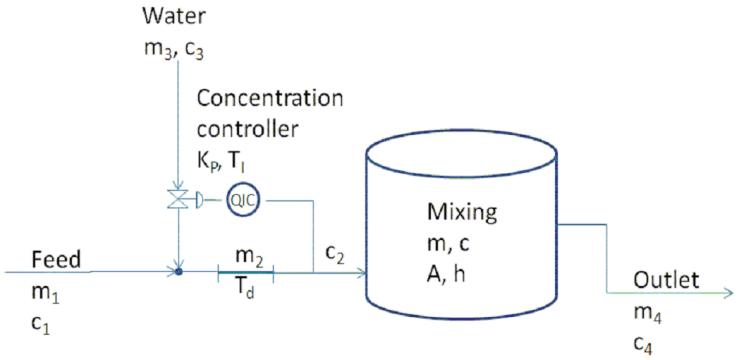

Figure 28: Flow sheet of the control model. [Sindhya et al. 2013]

The concentration dynamic in the mixing tank is

$$m\frac{dc}{dt} = m_2c_2(t) - m_4c_4(t), \tag{4.1}$$

where m is the total mass in the tank, m_2 and c_2 are mass and concentration of the inlet flow, m_4 and c_4 are mass and concentration of the outlet flow and t is time. The total mass m is

$$m = \rho AL, \tag{4.2}$$

where ρ is fluid density, A is the cross-sectional area of the tank and L is liquid level. The mass flow is fixed to 100 kg/s and liquid level L is adjusted to 0.75 h (h is height of the tank), the concentration is allowed to vary over time. The inlet feed flow is diluted with pure water,

before it enters to mixing tank. The water addition m_3 is determined by the concentration controller QIC using the PI control law

$$m_3 = K_p \left[(c_2^{ref} - c_2(t)) + \frac{1}{T_I} \int_0^{\infty} (c_2^{ref} - c_2(t) dt \right], \tag{4.3}$$

where K_p is the controller gain parameter, T_I is the controller integration time parameter and c_2^{ref} is the desired value for c_2.

Multiobjective optimization problem

The problem is formulated such that we have three objectives, which are minimized. The first objective is the ratio of inlet concentration c_1 variations and outlet concentration c_4. The second objective is the investment cost of the mixing tank. The third objective is Integrated Absolute Error of the outlet concentration c_4 and the desired concentration c_2^{ref}, which determines how well the system achieves the target dilution. The variables are cross-sectional area of the mixing tank A, height of the mixing tank h, controller gain parameter K_p and a coefficient k_I, which is used to determine the integration time T_1 of the controller as

$$T_1 = k_I \frac{Ah}{Q}, \tag{4.4}$$

where $Q = 0.1 \;\; m^3/s$ is the volumetric flow rate through the tank. Hence multiobjective optimization problem is described as

$$\begin{aligned}
\text{minimize} \quad & f_1 = \sum_{\kappa=1}^{4} \frac{A_{\kappa,4}}{A_{\kappa,1}}, \\
& f_2 = 10000(Ah)^{0,7}, \\
& f_3 = \frac{1}{D} \int_0^D \mid c_4(t) - c_2^{ref} \mid dt, \\
\text{subject to} \quad & 0 \leq A \leq 10m^2, \\
& 1 \leq h \leq 5m, \\
& 0 \leq K_p \leq 10, \\
& 0 \leq k_I \leq 5, \\
& \mid h - 1.5d \mid < 0.5,
\end{aligned} \tag{4.5}$$

where κ is the number of sinusoids of different cycle times, D is the simulation time (here $D = 6$ hours) and d is diameter of the mixing tank's cross-section. In f_1, $A_{\kappa,4}$ is the amplitude of the concentration of flow 4 at cycle time T_κ and $A_{\kappa,1}$ is the amplitude of the concentration of flow 1. $T_\kappa = [400, 100, 10, 0.2]^T$ are the sinusoid cycle times.

4.2 Experimental setting

Our experiment is about building a surrogate for multiobjective optimization problem, which is described in the previous section. Our goal is to replace computationally expensive APROS with NN, which is approximating the three objectives and one constraint. NN structures for experiment are MLP and RBF network as we are not trying to model the dynamics of the problem, but just single objective values. Another goal is to compare how different training data set affects on accuracy of the NN. Experiment is done with Matlab 2011 using NN toolbox, computer is running with Windows 8 and it has 4-core i7-processor running at 2,2 GHz and 8 GB memory.

The problem determines some NN parameters, which are four inputs, namely A, h, K_p and k_I, and four outputs, namely f_1, f_2, f_3 and constraint $\mid h - 1.5d \mid < 0.5$. For NN structure we have basically two choices:

- *Common*, all of the objectives share a NN, hence it consists four outputs.
- *Individual*, each of the objectives has its own NN, hence it consists one output. In total we have four NNs.

These terms implies to MLP and RBF network and let's call Common MLP as CMLP, Individual MLP as IMLP and Common RBF network as CRBF and Individual RBF network as IRBF.

As advised in Chapter 3, the structure for MLP is one hidden layer (see Figure 29A). In addition, to see if there are differences in accuracy, MLP with two hidden layers (see Figure 29B) is also build and tested. The transfer function in hidden neurons is chosen to be hyperbolic tangent according the justification in Chapter 3 and output layer transfer functions are linear. To determine the number of hidden neurons one heuristic is chosen, randomly,

where the number of hidden neurons in the first hidden layer is $2x\ \ inputs + 1$. The number of hidden neurons in the second layer is chosen to be six. Training method for MLP is conjugate-gradient method, maximum epochs is set to 30000 and accuracy to achieve is 0.00001 measured via SSE.

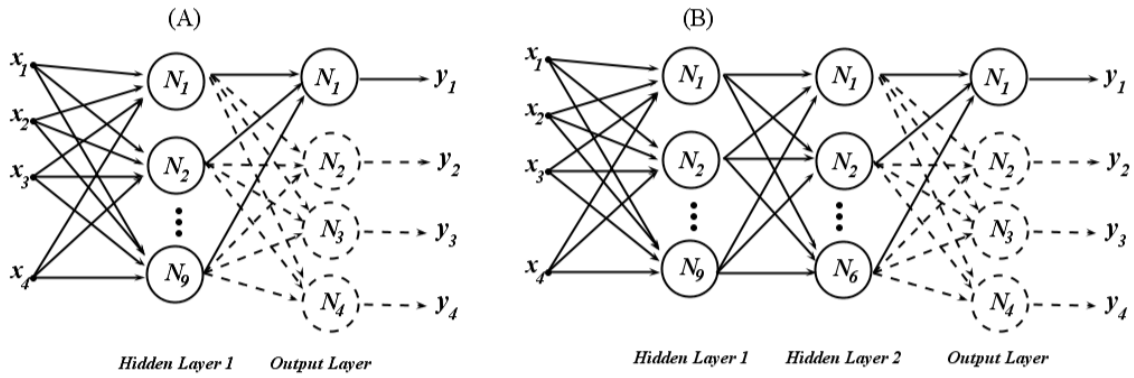

Figure 29: The MLPs for the numerical experiment. A.) The MLP with one hidden layer. B.) The MLP with two hidden layer. Dotted parts are illustrating the Common MLPs.

The structures for RBF network are a CRBF with all training points as centers (CRBF a), a CRBF with supervised selected training points as centers (CRBF as) and a IRBF with supervised selected training points as centers (IRBF as). Different accuracies to achieve are 0.1 and 0.5 measured via MSE, since Matlab RBF network uses it and it cannot be easily changed. In the Matlab implementation we can set "spread", which determines the diversity of radial basis function, and after some test-runs the spread 10 is found to be good choice for this problem. Different data sampling techniques are discussed in Chapter 2. Techniques are used to create input space for APROS, which then calculates corresponding outputs. Inputs are bounded according to optimization problem (4.5). Data set sizes for Latin Hypercube sampling and Hammersley sampling are 100, 500, 1000 and 1500 points and because of Orthogonal arrays optimal design it gives data sets of 100, 529, 1024 and 1521 points. Hence Orthogonal array has little advance in sense of data points, as we remember for Chapter 2, the accuracy of the NN depends, partly, on the number of training points. Then the data sets are divided with ratio of 15:85, where 15% of the data points are used to validation and 85% of data points is used for training. Then the set of 85% is divided with ratio of 30:70, where 30% is used for testing and 70% is used for training. Same sets are used to train and

validate each NN. Validation error is measured via RMSE, as it was suggested previously to training the network with different metric than validate. Then we pick the best networks for the final validations, where validation set consists of 50 points from each of the biggest validation sets. The training data, usually, needs some preprocessing e.g. normalization, outlier detection and removal to achieve flawless data. We are assuming that APROS generates flawless data, which does not contain measure errors, noise or outliers. Hence we do not perform any noise/outlier removal techniques to the training data. Although the input values are normalized to [0,1] and output values are matched to range of transfer functions [-1,1], as it is suggested in [LeCun et al. 1998], in test-runs this has found to have positive effect on accuracy and training time. Output values of constraint are altered so that feasible results take value of [-1] and not feasible results take [1], hence constraint approximation derives to a classification problem. Hence we see if the function approximation methods implies to the classification problem as well.

4.3 Results

In this section we introduce results of our experiment. In the next section we give some analysis of the results. All of the results are shown in Appendix A. Figures 30 - 38 show approximation values marked as blue 'x' and desired values marked as red 'o'. Measurements are done to values, which are given by the NNs and they are not scaled back to the real values, because the real values consists of very large values (>70000) and very small values (<0.1), thereby the comparison with scaled values would not be easy.

We have chosen two best designs given by CMLP and CRBF and also the best IMLP and the best IRBF for each function. All of the training results can be found in Appendix A and the best results for our experiment are shown in Table 13. The first column gives the objective, where *overall* is average error of all of the objectives and individual error for each objective is following the overall. The second column gives sampling technique (Lhs is Latin hypercube, Oa is Orthogonal array and Ham is Hammersley sampling), sample size, the number of layers (hl1 stands for one hidden layer and hl2 stands for two hidden layers) and for RBF network it gives if all training points are selected for center (*a*) or if centers are supervised selected (*as*) and the goal for RBF network to achieve e.g. *g 0.1* stands for

goal *0.1*. Some design consists two different designs because their accuracy was the same. The third column gives either RMSE for function approximation or classification percent for correctly classified constraint. Constraint is the only one, which takes classification % as a result. Results for final validation are shown in Table 14. Descriptions are the same as in Table 13. The final validation results consists overall RMSE for each chosen candidate and the individual results are shown for the most accurate NN designs. The individual constraint is shown only once, since we have chosen only one NN design for final validation.

Table 13: The best NN design in numerical experiment

The best Common MLP	Design	RMSE/Classification%
Overall	Lhs 100 hl1	0,0349
Objective 1	Lhs 100 hl1	0,0393
Objective 2	Lhs 100 hl1	0,0396
Objective 3	Lhs 100 hl1	0,0234
Constraint (%)	Lhs 100 hl1	66,6667
2. best Common MLP		
Overall	Ham 1500 hl2	0,0378
Objective 1	Ham 1500 hl2	0,0396
Objective 2	Ham 1500 hl2	0,0291
Objective 3	Ham 1500 hl2	0,0432
Constraint (%)	Ham 1500 hl2	97,3333
The best Individual MLP		
Overall	Oa 1024 hl1/hl2	0,0788
Objective 1	Oa 1024 hl1/hl2	0,0743
Objective 2	Oa 1024 hl1/hl2	0,1304
Objective 3	Oa 1024 hl1/hl2	0,0317
Constraint (%)	Oa 1024 hl1/hl2	26,6667
The best Common RBF		
Overall	Oa1024 as g 0.1	0,0393
Objective 1	Oa 1024 as g 0.1	0,0373
Objective 2	Oa 1024 as g 0.1	0,0072
Objective 3	Oa 1024 as g 0.1	0,0564
Constraint (%)	Oa 1024 as g 0.1	15,5844
2. best Common RBF		
Overall	Oa1024 a g 0.1	0,0408
Objective 1	Oa 1024 a g 0.1	0,0368
Objective 2	Oa 1024 a g 0.1	0,0066
Objective 3	Oa 1024 a g 0.1	0,0600
Constraint (%)	Oa 1024 a g 0.1	12,3377
The best Individual RBF		
Overall	Oa 1024 as g 0.1/0.5	0,0788
Objective 1	Oa 1024 as g 0.1/0.5	0,0743
Objective 2	Oa 1024 as g 0.1/0.5	0,1304
Objective 3	Oa 1024 as g 0.1/0.5	0,0317
Constraint (%)	Oa 100 as g 0.1	26,6667

Table 14: Final validation results

The best Common MLPs	Design	RMSE	Classification %
Overall	Lhs 100 hl1	0,2739	67,3333
Overall	Ham 1500 hl2	0,2995	96,0000
Objective 1	Ham 1500 hl2	0,4160	0,0000
Objective 2	Ham 1500 hl2	0,0719	0,0000
Objective 3	Ham 1500 hl2	0,2162	0,0000
Constraint	Ham 1500 hl2	0,0000	96,0000
The best Individual MLPs			
Overall		0,1515	
Objective 1	Oa 1024 hl1	0,3129	0,0000
Objective 2	Oa 1024 hl1	0,0184	0,0000
Objective 3	Oa 1024 hl1	0,1231	0,0000
Constraint	Oa 100 hl1	0,0000	46,0000
Overall		0,2071	
Objective 1	Oa 1024 hl2	0,4477	0,0000
Objective 2	Oa 1024 hl2	0,0150	0,0000
Objective 3	Oa 1024 hl2	0,1587	0,0000
The best Common RBFs	Design	RMSE Overall	Classification %
Overall	Oa 1024 a g 0.1	0,1631	16,6667
Overall	Oa 1024 as g 0.1	0,1631	16,6667
Objective 1	Oa 1024 as g 0.1	0,1898	0,0000
Objective 2	Oa 1024 as g 0.1	0,0183	0
Objective 3	Oa 1024 as g 0.1	0,2084	0,0000
Constraint	Oa 1024 as g 0.1	0,0000	16,6667
The best Individual RBFs			
Overall		0,1918	
Objective 1	Oa 1024 g 0.1	0,2102	0,0000
Objective 2	Oa 1024 g 0.1	0,1450	0,0000
Objective 3	Oa 1024 g 0.1	0,2201	0,0000
Constraint	Oa 100 g 0.1	0,0000	4,6667
Overall		0,1918	
Objective 1	Oa 1024 g 0.5	0,2102	0,0000
Objective 2	Oa 1024 g 0.5	0,1450	0,0000
Objective 3	Oa 1024 g 0.5	0,2201	0,0000

Even though the best CMLP design (Lhs 100 with one hidden layer) performs nicely when validated with its own validation set as it is shown in Table 13. When we use larger final validation set, its accuracy decreases, although it is performing better than the second best CMLP (Ham 1500 with two hidden layers). In Figure 30 it is shown the approximations of the final validation set by the best CMLP, where upper figure gives as the approximation of objective functions (RMSE 0,2739) and lower gives the classifications for the constraint. As we can see from Figure 30 that for real it is not that good as there seems to be a systematic error on every approximation, although the approximation value front is close to the real objective function front. Hence the RMSE does not tell the whole truth. Also the best CMLP is classifying the constraint function very poorly, only 67% correct classifications.

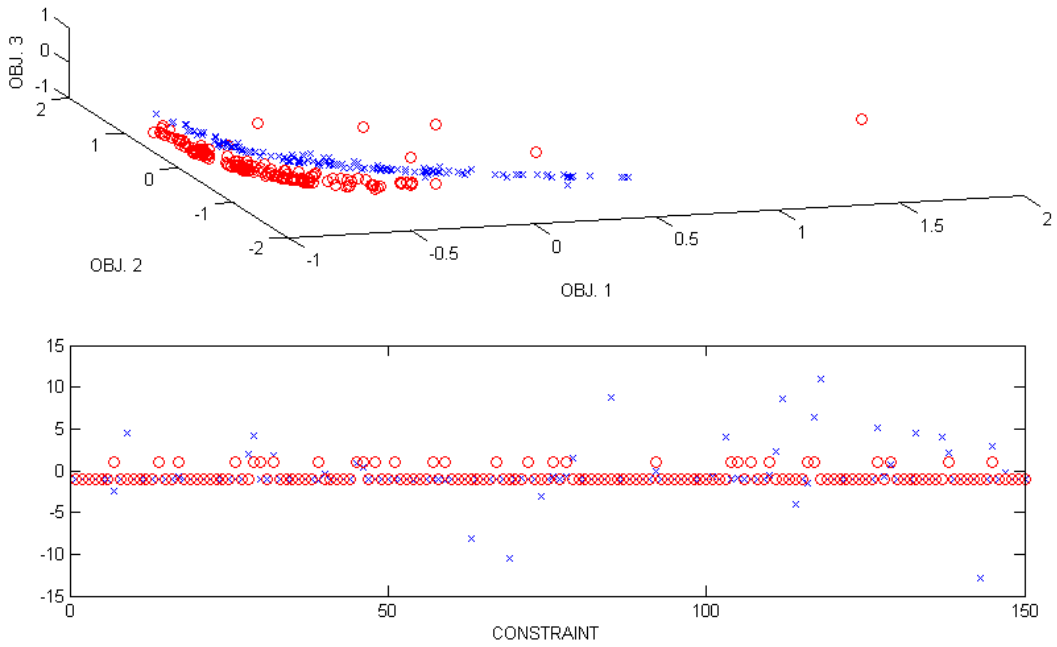

Figure 30: The best CMLP final validations results. This was obtained by training single hidden layer Multilayer Perceptron with 100 training points generated by Latin hypercube sampling.

In Figure 31 it is shown the approximations of the final validation set by the second best CMLP. The second best CMLP is approximating final validations solutions more accurately (RMSE 0,2995), but there are a few approximations, which have a high error. Also classify-

ing the constraint is done very accurately (96%). Individual objective function approximations by the second best IMLP are shown in Figure 32, where the top figure is approximations of the first objective function (RMSE 0,416), the middle figure is for the second objective function (RMSE 0,0719) and the lowest figure is for the third objective function (RMSE 0,2162).

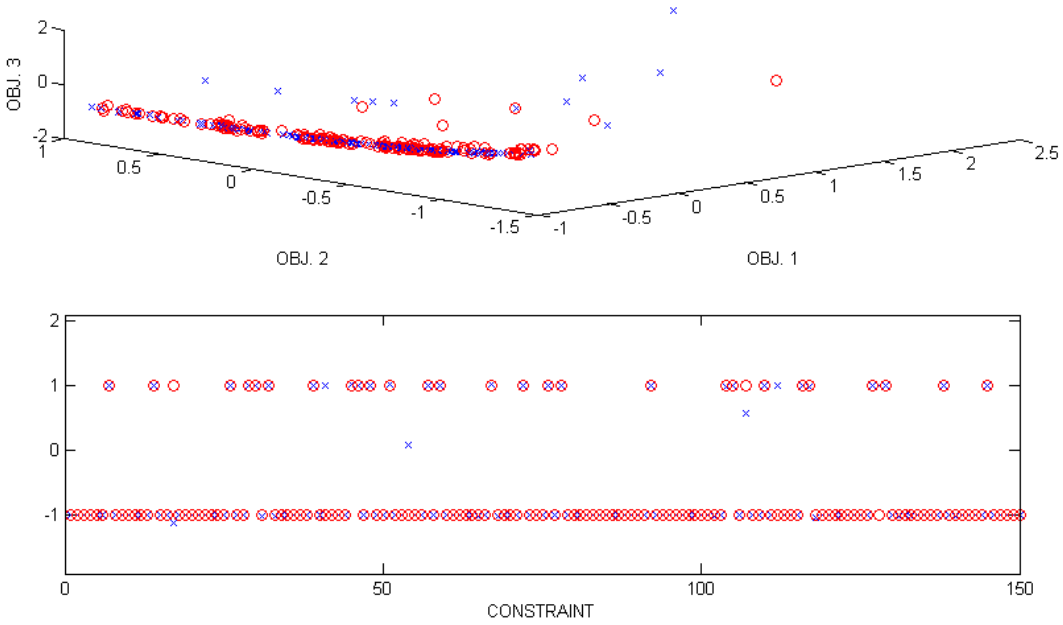

Figure 31: The second best CMLP final validations results. This was obtained by training two hidden layer MLP with 1500 training points generated by Hammersley sampling.

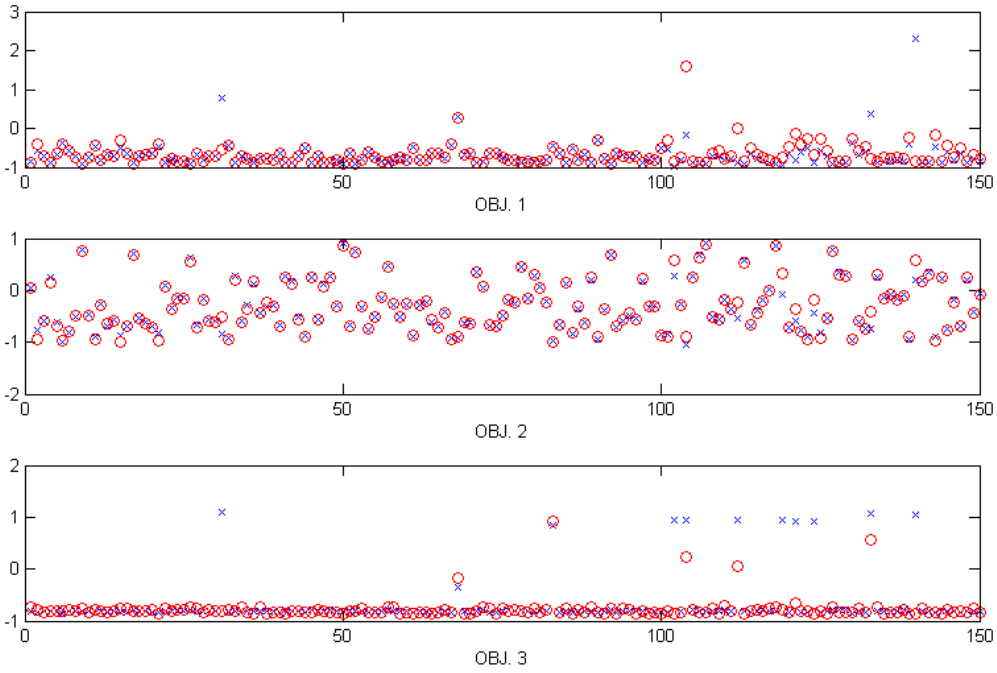

Figure 32: The second best CMLP final validations results, where objectives are shown individually. This was obtained by training two hidden layer MLP with 1500 training points generated by Hammersley sampling.

When we are using IMLP (see Figure 33), we obtain better approximation error (0,1515) than the CMLP (0,2995). Although IMLP, which is classifying the constraint, is not doing it very correctly (46%). Individual approximations for each of the objectives from IMLPs are shown in Figure 34, where the first objective surrogate takes error of 0,3129, the second objective surrogate is approximating with accuracy of 0,0184 and third objective surrogate takes error of 0,1231.

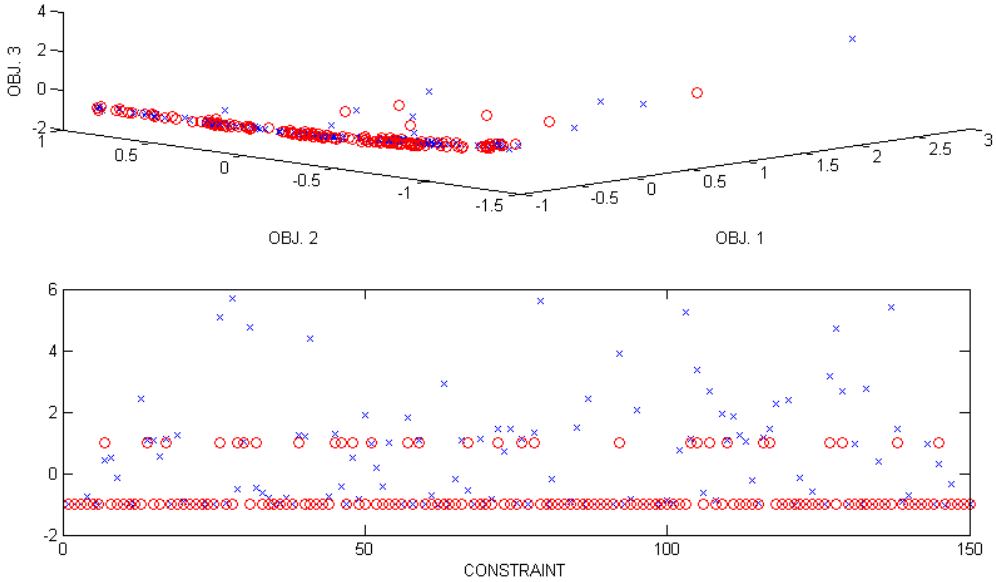

Figure 33: The best IMLPs final validations results. All of these were obtained by training single hidden layer MLP with 1000 training points generated by Orthogonal array sampling.

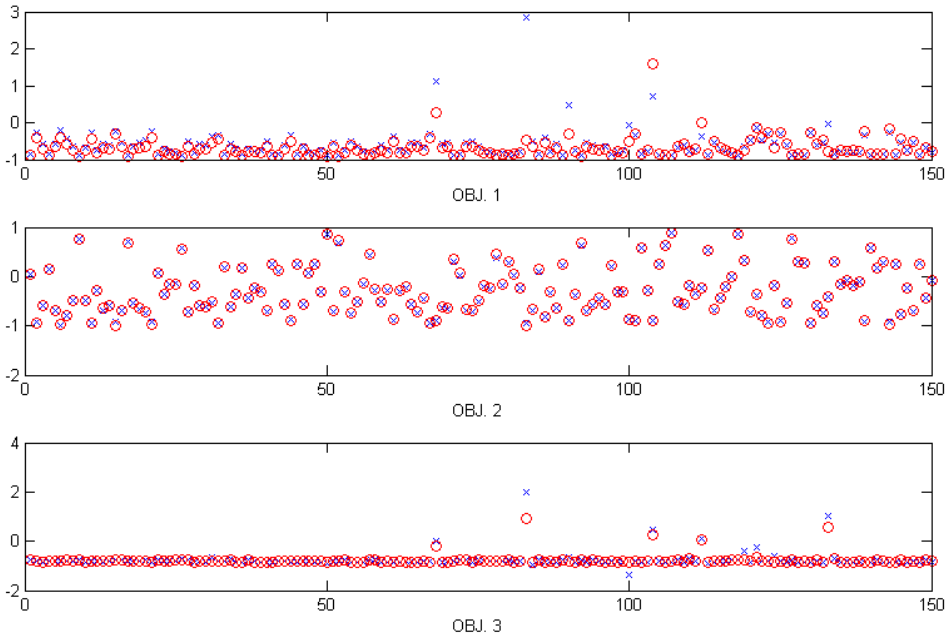

Figure 34: The best IMLPs final validations results, where objectives are shown individually. All of these were obtained by training single hidden layer MLP with 1024 training points generated by Orthogonal array sampling.

The best CRBF network (RMSE 0,1631) is performing almost as good as IMLP (RMSE 0,1515), although it is classifying constraint poorly 16,66%. Approximations of CRBF is shown in Figure 35. And the individual objective results from the best CRBF are shown in Figure 36, where first objective function approximation achieves RMSE 0,1898, the second objective function approximation takes error of 0,0183 and the third objective function approximation achieves RMSE 0,2084.

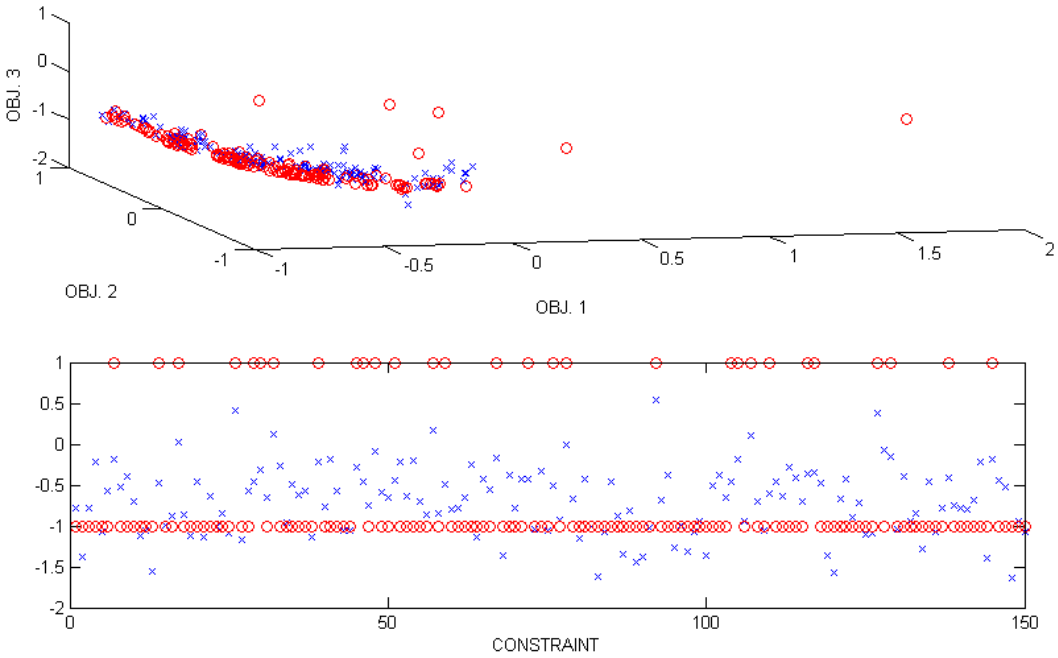

Figure 35: The best CRBF final validations results. This was obtained by supervised selection centers when goal was 0.1 via MSE and 1024 training points generated by Orthogonal array sampling.

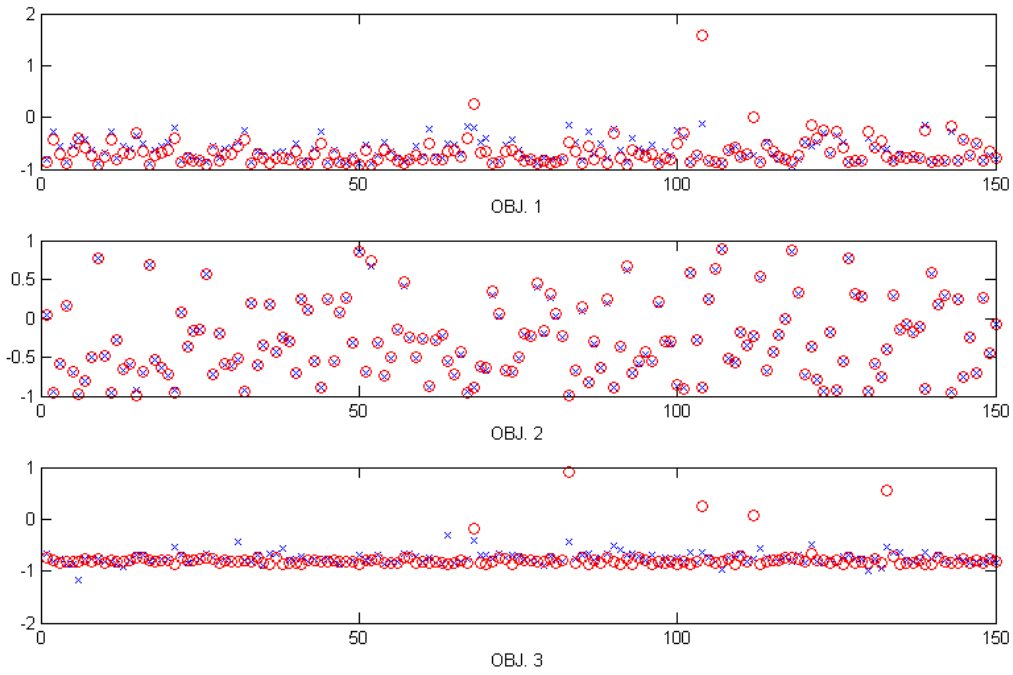

Figure 36: The best CRBF final validations results, where objectives are shown individually. This was obtained by supervised selection of centers when goal was 0.1 via MSE and 1024 training points generated by Orthogonal array sampling.

The last results are from surrogate model IRBF (see Figure 37). RMSE (0,1918) of IRBF is slightly worse than CRBF (0,1631), but a little bit better than CMLP (0,2995). The last results, where objectives are individually, are shown in Figure 38, where the first objective function surrogate takes RMSE 0,2102, the second objective function surrogate achieves error of 0,1450 and the third objective function surrogate takes error of 0,2201. Classification results are very poor 4,66% .

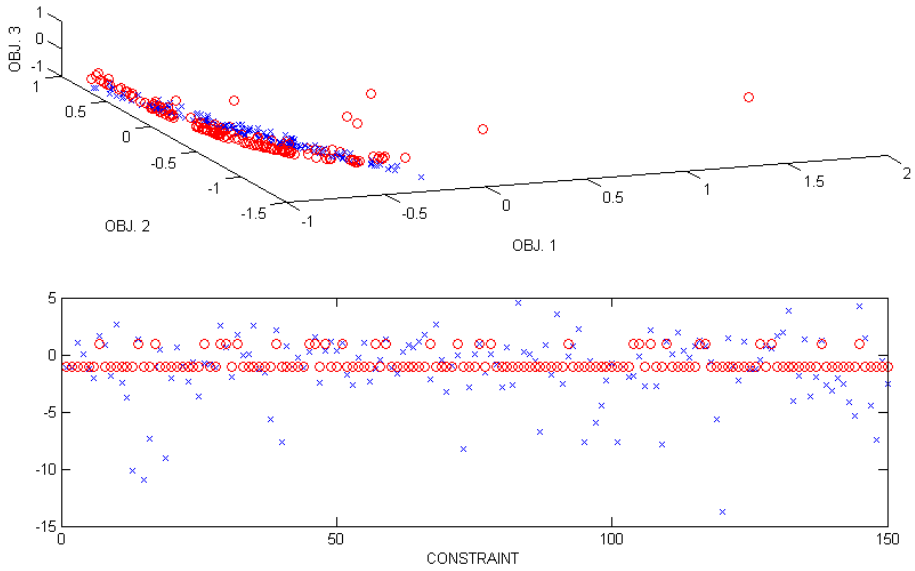

Figure 37: The best IRBF network final validations results. All of these were obtained by supervised selections of centers when goal was 0.1 via MSE and 1024 training points generated by Orthogonal array sampling.

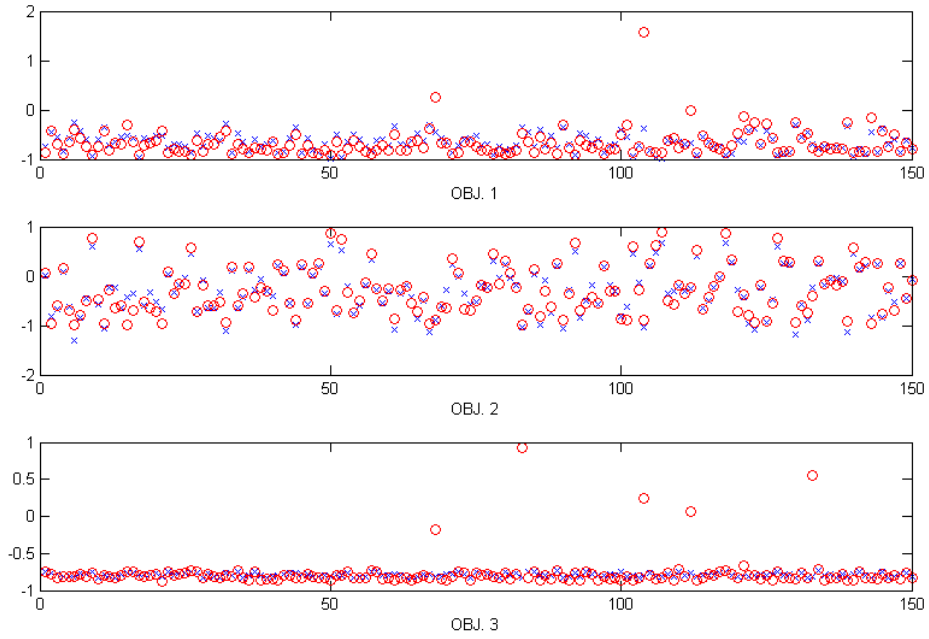

Figure 38: The best IRBF final validations, where results objectives are shown individually. All of these were obtained by supervised selection of centers when goal was 0.1 via MSE and 1024 training points generated by Orthogonal array sampling.

4.4 Result analysis

In this section we analyze the results. We start by analyzing the figures, the sampling techniques and sample sizes, then we continue to analyze the NN designs. Since we have not discussed about classification we do not analyze those results, but clearly function approximation approach does not work very well in a classification problem.

As shown in Figures 30 - 38 the output values of the training data would have need some preprocessing, since objectives 1 and 3 values are concentrated on $0,5$ and $0,9$. The output values were normalized to interval of [-1,1], hence the output values of the training data have consisted a few outliers, which have deformed the normalized data. Those output values should be analyzed to see, if they consist some crucial information about the control model and remove them, if not. If those can be removed the models should be retrained and revalidate. In Table 14 is shown that the second objective surrogates are the most accurate and in corresponding figures we see that output values are the most diverse in the second objective. Hence outlier detection for output values is crucial to the generalization accuracy of the NN.

In this thesis one of our goals was to compare different sampling techniques and their effect to generalization performance. We provided three different sampling techniques, namely Latin hypercube, Hammersley sampling and Orthogonal array. Each of those generated four input sets and them were used to calculate the corresponding outputs using APROS. Hence we got 12 different training sets, which were used to train neural networks. Data sets where divided into training, testing and validation sets. Training and testing sets, where used to train the network and validation set to validate it. Thus the built NNs are not comparable to each other more than within the same sample technique and size, therefore three best networks where taken and them where compared to final validation set, which was created by taking 50 points from each of the biggest validation sets randomly. Mean errors and their standard deviations for different data sampling techniques are shown in Table 15. Accordingly the Latin hypercube sampling gives the best mean error and is the most consistent.

Mean errors and standard deviations for different sample sizes are shown in Table 16, errors are picked from each sampling technique sets. As seen from there sample size of 1000 training point seems to be enough for the NNs in this problem, as the mean error starts to

Table 15: Mean error and standard deviation for each sampling technique

Technique	Mean error (MLP)	STD	Mean error (RBF)	STD
Latin Hypercube	0,0942	0,0188	0,1242	0,0481
Orthogonal Array	0,1936	0,0614	0,1363	0,01601
Hammersley	0,2227	0,0769	0,1790	0,0281

climb after it. Although sample sizes from 500 to 1500 are in the same range and in our MLP designs the number of free parameters takes values 55, 85, 111 and 133. From this we can conclude that the number of training patterns needed for MLP, should be at least four times the number of free parameters and roughly 10 times the number of free parameters is enough.

Table 16: Mean error and standard deviation for each sample size

Sample size	Mean error (MLP)	STD	Mean error (RBF)	STD
100	0,2867	0,3802	0,2277	0,1775
500	0,1216	0,0722	0,1288	0,0733
1000	0,0946	0,0546	0,1002	0,0690
1500	0,1097	0,0648	0,1110	0,0595

Mean errors and standard deviations for different MLP and RBF network designs are shown in Table 17, where constraint classification results are excluded. As seen from the results the mean error of NN designs is pretty much the same, but consistency of the results is lower when we are using NN to approximate only one objective function. Another conclusions, which can be made from this result, is that MLP and RBF network are performing equally and there are no differences on MLPs performance, whether it consists of one or two hidden layers.

Table 17: Mean error and standard deviation for each neural network design.

Design	Mean error	STD
Common MLP hl1	0,1678	0,3068
Common MLP hl2	0,1802	0,2717
Individual MLP hl1	0,1679	0,0841
Individual MLP hl2	0,1679	0,0841
Common RBF a g 0.1	0,1607	0,1158
Common RBF as g 0.1	0,1216	0,0827
Common RBF a g 0.5	0,1607	0,1158
Common RBF as g 0.5	0,1732	0,0816
Individual RBF as g 0.1	0,1679	0,0841
Individual RBF as g 0.5	0,1679	0,0841

Results of final validation are quite interesting as they are reverse when they are compared to results obtained when error was measured to their own validation set. Obviously there are two reasons for this

1. The neural network has not learned the features of the training data. Namely the MLP with one hidden layer and 100 training points generated by Latin hypercube sampling. Hence we had an insufficient number of training points.
2. The neural network has overlearned the features of the training data. Namely the MLP with two hidden layers and 1500 training points generated by Hammersley sampling. Hence had too many training points.

Even if those were overtrained and not well enough trained, our opinion is that they could be used as a surrogate by using certain cautions. Since (1) is approximation the landscape of function, but there seems to be a systematic error in every approximation. Although we would not know if the solution is feasible or not since it is classifying constraint function very badly, so that would need some other handling. The (2) would be good enough, as seen from Figure 32, that most of solutions are correctly approximated, but there are a few solutions, which have big error to desired value. Although (2) is the only one that classifies

the constraint well enough. Hence (2) would be our choice for a surrogate model to be used in optimization. Single evaluation using (2) takes approximately 0,006 seconds with this hardware, hence computationally costs is decreased significantly compared to the APROS.

A more accurate solution would be a combination of single function approximation NNs, but it would still need some development to ensure that the constraint is classified correctly. For example, for the first objective function surrogate model, RBF network with selected centers and accuracy target of 0.1, which was trained using Orthogonal array sampled 1024 training points. For the second and the third objective functions, surrogate models MLPs with one hidden layer and they were trained by using Orthogonal array sampled 1000 training points. Although in this setup we would need to study bit more about NNs for classification problem or handle the constraint with some other way. Justification for choosing surrogate models, which each of them approximates single function, is that this would made the surrogate handling more versatile as each model could be modified as a unit and because of this would be the most accurate model.

When summing up the results of this experiment, as it is a bit conflicting. Latin hypercube sampling technique proved to be the best according to training, but the NN, which was trained with it, was not that good after all. Then the other sampling techniques proved to be better at the end. Hence after all the sampling techniques did not have a big effect on the performance of the NNs. The sample size validated to be roughly about 10 times the free parameters. Although one can make a surrogate model with a less training points, but it would not be so accurate. When comparing the results from different structure, we saw that MLP and RBF network are basically equal. And due to the fact that RBF network is much easier to design than MLP, it might be more preferable model to practitioners to start working with. Although dimension of RBF network climbed very high, in our experiment, and it might yield to memory issues in some applications as we need to keep almost the complete training set with us.

5 Conclusion

In this thesis we have studied NNs for computationally expensive problems. Our main focus in this thesis was about function approximation features of the NNs, hence we can use them as surrogate models for computationally expensive simulators, in future. We have introduced four different NNs for function approximation and discussed their features. Theory of NNs, heuristics for the NNs structure design and training were studied. In addition, an example of NN design using heuristics was presented and the effect of different sampling techniques and sample sizes to NNs generalization accuracy was compared.

NNs seem to be quite simple and their implementations can be found in different programming language and applications, but it is good for practitioners to study theory of NNs to understand their features and applicability. This thesis is partly written so that other practitioners could study theory of NNs easily and practically. Designing a NN might be time consuming, because the optimal structure is different for each problem. Heuristics will help in designing of a NN, but using heuristics one might not obtain the optimal NN design for the problem in hand. Therefore it would be reasonable to implement some optimization method for NN structure, thus automate the structure design. Although due to the experience practitioners should learn the good structures for different problems. In our numerical experiment we have verified the number of training patterns should be 4-10 times the number of free parameters. The sample technique does not have a big effect in training. MLP and RBF network seem to be equally good for function approximation.

In future we need to do outlier detection for training data and retrain and revalidate the NNs. Then we need to study neural networks classification features and other surrogate models as well. Then study more about multiobjective optimization and model management, hence we can do the optimization with surrogate models and ensure that solutions evaluated by surrogate model are leading us to right optimum. After the optimization we can continue our research towards implementation of surrogate models to other ways than function approximation.

It is our interest to use NNs for optimization. The following steps need to be accomplished

before actual optimization process can begin.

- *Surrogate model*: Final choice of the surrogate model.
- *Constraint handling*: At the end if we choose a surrogate model, which cannot handle constraint. How can we make sure that we are using only the feasible solutions?
- *Optimization method*: We have to choose our optimization method, which in high probability would be a posteriori method and some evolutionary algorithm would be employed.
- *Model management*: We have to ensure that optimization algorithm convergence to right optimum, since the surrogate models are not as accurate as the original simulator. For example, after some number of iterations we could evaluate solutions with the simulator and retrain the surrogate model.
- *Optimization setup*: For evolutionary algorithm we need to setup crossover, mutation, population size, etc. Hence we need study, which are a good choices for these.

Hence further knowledge of at least these topics is needed before optimization can begin.

Bibliography

Ackley, D.H., G.E. Hinton, and T.J. Sejnowski. 1985. "A Learning Algorithm for Boltzmann Machines". *Cognitive Science* 9:147–169.

Barron, A.R. 1993. "Universal approximation bounds for superpositions of a sigmoidal function". *IEEE Transactions on Information Theory* 39:930–945.

Basheer, I.A., and M. Hajmeer. 2000. "Artificial neural networks: fundamentals, computing, design, and application". *Journal of Microbiological Methods* 43:3–31.

Benedetti, A., M. Farina, and M. Gobbi. 2006. "Evolutionary multiobjective industrial design: the case of a racing car tire-suspension system". *IEEE Transactions on Evolutionary Computation* 10:230–244.

Bishop, C.M. 1995. *Neural Networks for Pattern Recognition.* Oxford University Press.

Branke, J., K. Deb, K. Miettinen, and R. Slowinski, editors. 2008. *Multiobjective Optimization: Interactive and Evolutionary Approaches.* Springer.

Broomhead, D. S., and D. Lowe. 1988. "Multivariable Functional Interpolation and Adaptive Networks". *Complex Systems* 2:321–355.

Brouwer, R. 1997. "Implementation of the Exclusive-Or Function in a Hopfield Style Recurrent Network". *Neural Processing Letters* 5:1–7.

Chernoff, H., and E.L. Lehmann. 2012. "The Use of Maximum Likelihood Estimates in χ^2 Tests for Goodness of Fit". In *Selected Works of E. L. Lehmann,* edited by Javier Rojo. Springer.

Cover, T. M. 1965. "Geometrical and Statistical Properties of Systems of Linear Inequalities with Applications in Pattern Recognition". *IEEE Transactions on Electronic Computers* 14:326–334.

Craven, P., and G. Wahba. 1979. "Smoothing noisy data with spline functions: Estimating the correct degree of smoothing by the method of generalized cross - validation". *Numerische Mathematik* 31:377–403.

Cybenko, G. 1989. "Approximation by superpositions of a sigmoidal function". *Mathematics of Control, Signals and Systems* 2:303–314.

Deb, K. 2001. *Multi-Objective Optimization Using Evolutionary Algorithms.* Wiley.

Dingankar, A.T., and I.W. Sandberg. 1997. "A note on error bounds for function approximation using nonlinear networks". In *Proceedings of the 40th Midwest Symposium on Circuits and Systems,* 1248–1251. Volume 2.

Duch, W., and N. Jankowski. 1999. "Survey of Neural Transfer Functions". *Neural Computing Surveys* 2:163–213.

Fahmi, I., and S. Cremaschi. 2012. "Process synthesis of biodiesel production plant using artificial neural networks as the surrogate models". *Computers and Chemical Engineering* 46:105–123.

Fletcher, D., and E. Goss. 1993. "Forecasting with neural networks: An application using bankruptcy data". *Information & Management* 24:159–167.

Gasteiger, J., and J. Zupan. 1993. "Neural Networks in Chemistry". *Angewandte Chemie International Edition in English* 32:503–527.

Haley, P.J., and D. Soloway. 1992. "Extrapolation limitations of multilayer feedforward neural networks". In *International Joint Conference on Neural Networks,* 25–30. Volume 4.

Hassibi, B., D.G. Stork, and G.J. Wolff. 1993. "Optimal Brain Surgeon and general network pruning". In *IEEE International Conference on Neural Networks,* 293–299. Volume 1.

Hassoun, M.H. 1995. *Fundamentals of Artificial Neural Networks.* 1st. MIT Press.

Haykin, S. 1999. *Neural Networks: A Comprehensive Foundation.* 2nd. Prentice Hall PTR.

Hebb, D.O. 1949. *The organization of behavior : a neuropsychological theory.* Wiley.

Hooke, R., and T. A. Jeeves. 1961. "Direct Search Solution of Numerical and Statistical Problems". *Journal of the Association for Computing Machinery* 8:212–229.

Hopfield, J.J. 1982. "Neural networks and physical systems with emergent collective computational abilities". *Proceedings of the National Academy of Sciences* 79:2554–2558.

Hüsken, M., Y. Jin, and B. Sendhoff. 2005. ”Structure optimization of neural networks for evolutionary design optimization”. *Soft Computing* 9:21–28.

Jacobs, R.A. 1988. ”Increased rates of convergence through learning rate adaptation”. *Neural Networks* 1:295–307.

Jadid, M.N., and D.R. Fairbairn. 1996. ”Neural-network applications in predicting moment-curvature parameters from experimental data”. *Engineering Applications of Artificial Intelligence* 9:309–319.

Jain, B.A., and B.N. Nag. 1995. ”Artificial Neural Network Models for Pricing Initial Public Offerings”. *Decision Sciences* 26:283–302.

Jin, Y. 2005. ”A comprehensive survey of fitness approximation in evolutionary computation”. *Soft Computing* 9:3–12.

———. 2011. ”Surrogate-assisted evolutionary computation: Recent advances and future challenges”. *Swarm and Evolutionary Computation* 1:61–70.

Kusiak, A., Z. Zhang, and M. Li. 2010. ”Optimization of Wind Turbine Performance With Data-Driven Models”. *IEEE Transactions on Sustainable Energy* 1:66–76.

Lachtermacher, G., and J.D. Fuller. 1995. ”Backpropagation in time-series forecasting”. *Journal of Forecasting* 14:381–393.

LeCun, Y., L. Bottou, G. Orr, and K. Muller. 1998. ”Efficient BackProp”. In *Neural Networks: Tricks of the trade,* edited by G. Orr and Muller K. Springer.

LeCun, Y., I. Kanter, and S.A. Solla. 1990. ”Second order properties of error surfaces: learning time and generalization”. *Advances in neural information processing systems* 3:918–924.

Lee, Y., S. Oh, and M.W. Kim. 1991. ”The effect of initial weights on premature saturation in back-propagation learning”. *Neural Networks* 1:765–770.

Leung, W.K., and R. Simpson. 2000. ”Neural metrics-software metrics in artificial neural networks”. In *Fourth International Conference on Knowledge-Based Intelligent Engineering Systems and Allied Technologies,* 209–212. Volume 1.

Li, G., H. Alnuweiri, Y. Wu, and H. Li. 1993. "Acceleration of back propagation through initial weight pre-training with delta rule". In *IEEE International Conference on Neural Networks,* 580–585. Volume 1.

Lowe, D. 1989. "Adaptive radial basis function nonlinearities, and the problem of generalisation". In *First IEEE International Conference on Artificial Neural Networks,* 171–175.

Maiorov, V., and A. Pinkus. 1999. "Lower Bounds for Approximation by MLP Neural Networks". *Neurocomputing* 25:81–91.

Marianik, G., G. Palermo, C. Silvano, and V. Zaccaria. 2009. "Meta-model Assisted Optimization for Design Space Exploration of Multi-Processor Systems-on-Chip". In *12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools,* 383–389.

Masters, T. 1993. *Practical neural network recipes in C++.* Academic Press Professional, Inc.

McCulloch, W.S., and W. Pitts. 1943. "A logical calculus of the ideas immanent in nervous activity". *The bulletin of mathematical biophysics* 5:115–133.

Micchelli, C.A. 1986. "Interpolation of scattered data: Distance matrices and conditionally positive definite functions". *Constructive Approximation* 2:11–22.

Mitchell, M. 1999. *An Introduction to Genetic Algorithms.* MIT Press.

Moody, J., and C.J. Darken. 1989. "Fast learning in networks of locally-tuned processing units". *Neural Computation* 1:281–294.

Muknahallipatna, S., and B.H. Chowdhury. 1996. "Input dimension reduction in neural network training-case study in transient stability assessment of large systems". In *International Conference on Intelligent Systems Applications to Power Systems,* 50–54.

Narendra, K.S., and K. Parthasarathy. 1990. "Identification and control of dynamical systems using neural networks". *IEEE Transactions on Neural Networks* 1:4–27.

Oja, E. 1982. "Simplified neuron model as a principal component analyzer". *Journal of Mathematical Biology* 15:267–273.

Park, Ky., B.S. Kim, J. Lee, and K.S. Kim. 2009. "Aerodynamics and optimization of airfoil under ground effect". *International Journal of Mechanical Systems Science and Engineering* 1:332–338.

Pearson, K. 1920. "Notes on the history of correlation". *Biometrika* 13:25–45.

Piramuthu, S., M.J. Shaw, and J.A. Gentry. 1994. "A classification approach using multi-layered neural networks". *Decision Support Systems* 11:509–525.

Poggio, T., and F. Girosi. 1989. *A Theory of Networks for Approximation and Learning.* Technical report. Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

Pollack, J.B. 1991. "The Induction of Dynamical Recognizers". *Machine Learning* 7:227–252.

Powell, M. J. D. 1964. "An efficient method for finding the minimum of a function of several variables without calculating derivatives". *The Computer Journal* 7:155–162.

Prechelt, L. 1994. *PROBEN1 - a set of neural network benchmark problems and benchmarking rules.* Technical report. Faculty of Computer Science, University of Karlsruhe.

Price, K., R.M. Storn, and J.A. Lampinen. 2005. *Differential Evolution: A Practical Approach to Global Optimization.* Springer.

Puskorius, G.V., and L.A. Feldkamp. 1994. "Neurocontrol of nonlinear dynamical systems with Kalman filter trained recurrent networks". *IEEE Transactions on Neural Networks* 5:279–297.

Puskorius, G.V., L.A. Feldkamp, and Jr. Davis L.I. 1996. "Dynamic neural network methods applied to on-vehicle idle speed control". *Proceedings of the IEEE* 84:1407–1420.

Rosenblatt, F. 1958. "The perceptron: A probabilistic model for information storage and organization in the brain." *Psychological Review - PSYCHOL REV* 65:386–408.

Rumelhart, D. E., G.E. Hinton, and R.J. Williams. 1986. "Learning representations by back-propagating errors". *Nature* 323:533–536.

Schleiter, I.M., D. Borchardt, R. Wagner, T. Dapper, K. Schmidt, H. Schmidt, and H. Werner. 1999. ”Modelling water quality, bioindication and population dynamics in lotic ecosystems using neural networks”. *Ecological Modelling* 120:271–286.

Simpson, T.W., J.J. Korte, T.M. Mauery, and F. Mistree. 2001. ”Kriging Models for Global Approximation in Simulation-Based Multidisciplinary Design Optimization”. *AIAA Journal* 39:2233–2241.

Simpson, T.W, D.KJ. Lin, and W. Chen. 2001. ”Sampling strategies for computer experiments: design and analysis”. *International Journal of Reliability and Applications* 2:209–240.

Sindhya, K., V. Ojalehto, J. Savolainen, H. Niemistö, J. Hakanen, and K. Miettinen. 2013. ”APROS-NIMBUS: Dynamic Process Simulator and Interactive Multiobjective Optimization in Plant Automation.” In *Proceedings of the ESCAPE 23, the 23rd European Symposium on Computer Aided Process Engineering,* edited by A. Kraslawski and I. Turunen. Elsevier, to appear.

Stein, M. 1987. ”Large sample properties of simulations using latin hypercube sampling”. *Technometrics* 29:143–151.

Stone, M. 1974. ”Cross-Validatory Choice and Assessment of Statistical Predictions”. *Journal of the Royal Statistical Society. Series B (Methodological)* 36:111–147.

Sun, M., A. Stam, and R.E. Steuer. 1996. ”Solving multiple objective programming problems using feed-forward artificial neural networks: the interactive FFANN procedure”. *Management Science* 42:835–849.

Sykes, A.O. 1993. *An Introduction to Regression Analysis.* Law School, University of Chicago.

Tamura, S., and M. Tateishi. 1997. ”Capabilities of a four-layered feedforward neural network: four layers versus three”. *IEEE Transactions on Neural Networks* 8:251–255.

Tang, B. 1993. ”Orthogonal Array-Based Latin Hypercubes”. *Journal of the American Statistical Association* 88:1392–1397.

Thimm, G., and E. Fiesler. 1997. *Optimal Setting of Weights, Learning Rate, and Gain.* Technical report. IDIAP Research Institute.

Tikhonov, A.N., and V.Y. Arsenin. 1977. *Solutions of ill-posed problems.* Translated by John Fritz. V.H. Winston & SONS.

Tryfos, P. 1998. *Methods for Business Analysis and Forecasting: Text and Cases.* Wiley.

Twomey, J. M., and A. E. Smith. 1995. "Performance measures, consistency, and power for artificial neural network models". *Mathematical and Computer Modelling: An International Journal* 21:243–258.

Walczak, S. 1994. "Categorizing university student applicants with neural networks". In *IEEE International Conference on Neural Networks,* 3680–3685. Volume 6.

Walczak, S., and N. Cerpa. 1999. "Heuristic principles for the design of artificial neural networks". *Information and Software Technology* 41:107–117.

Wettschereck, D., and T. Dietterich. 1992. "Improving the Performance of Radial Basis Function Networks by Learning Center Locations". *Advances in neural information processing systems* 4:1133–1140.

Widrow, B., and M. E. Hoff. 1960. "Adaptive switching circuits."

Widrow, B., and M.A. Lehr. 1990. "30 years of adaptive neural networks: perceptron, Madaline, and backpropagation". *Proceedings of the IEEE* 78:1415–1442.

Williams, R.J., and J. Peng. 1990. "An Efficient Gradient-Based Algorithm for On-Line Training of Recurrent Network Trajectories". *Neural Computation* 2:490–501.

Wilson, B., D. Cappelleri, T.W. Simpson, and M. Frecker. 2001. "Efficient Pareto Frontier Exploration using Surrogate Approximations". *Optimization and Engineering* 2:31–50.

Wong, T., W. Luk, and P. Heng. 1997. "Sampling with Hammersley and Halton points". *Journal of Graphics Tools* 2:9–24.

Xie, H., H. Tang, and Y. Liao. 2009. "Time series prediction based on NARX neural networks: An advanced approach". In *International Conference on Machine Learning and Cybernetics,* 1275–1279. Volume 3.

Zamarreño, J.M., and P. Vega. 1998. "State space neural network. Properties and application". *Neural Networks* 11:1099–1112.

Zhu, J., J. Zurcher, M. Rao, and M.Q-H. Meng. 1998. ”An on-line wastewater quality predication system based on a time-delay neural network”. *Engineering Applications of Artificial Intelligence* 11:747–758.

Zitzler, E., and L. Thiele. 1998. *An Evolutionary Approach for Multiobjective Optimization: The Strength Pareto Approach.* Technical report. Computer Engineering and Networks Laboratory, Swiss Federal Institute of Technology Zurich.

Appendices

A Neural Network design experiment results

Result tables for MLP and RBF designs.

Table 18: Training results for MLP designs (Latin hypercube sampling)

	Sampling technique	and no. of points	Error in RMSE	Constraint is classification %	
	Lhs 100	Lhs 500	Lhs 1000	Lhs 1500	Average
Common MLP hl1			Lhs= Latin Hypercube		
Overall	0,0349314562	0,0703056764	0,0731929434	0,0651410122	0,060892772
Objective 1	0,0393241767	0,1014733697	0,0911352807	0,0694005346	0,0753333404
Objective 2	0,0395951259	0,0358319769	0,0326564235	0,0316598135	0,0349358349
Objective 3	0,0233763778	0,0569902541	0,0818507147	0,0831340869	0,0613378584
Constraint	66,6666666667	86,6666666667	65,3333333333	76	73,6666666667
Individual MLP hl1					
Overall	0,1393474408	0,1178265105	0,1669271849	0,1196175784	0,1359296786
Objective 1	0,1513514774	0,1386897487	0,0900722994	0,0957305225	0,118961012
Objective 2	0,1770874769	0,1366569949	0,1624520111	0,1518282608	0,1570061859
Objective 3	0,0631338542	0,0611496447	0,2215634901	0,1034840157	0,1123327512
Constraint	6,6666666667	16	11,3333333333	20	13,5
Common MLP hl2					
Overall	0,0608078959	0,0678191844	0,0489241389	0,0588279054	0,0590947812
Objective 1	0,0750938575	0,0932302449	0,0626336759	0,0660588304	0,0742541522
Objective 2	0,0501655516	0,0181423815	0,0234597451	0,0303218493	0,0305223819
Objective 3	0,0541953006	0,0691180203	0,0520324622	0,0714071687	0,0616882379
Constraint	60	92	97,3333333333	96,4444444444	86,4444444444
Individual MLP hl2					
Overall	0,1393474408	0,1178265105	0,1669271849	0,1196175784	0,1359296786
Objective 1	0,1513514774	0,1386897487	0,0900722994	0,0957305225	0,118961012
Objective 2	0,1770874769	0,1366569949	0,1624520111	0,1518282608	0,1570061859
Objective 3	0,0631338542	0,0611496447	0,2215634901	0,1034840157	0,1123327512
Constraint	13,3333333333	13,3333333333	6	17,3333333333	12,5

Table 19: Training results for MLP designs (Orthogonal array)

	Sampling technique	and no. of points	Error in RMSE	Constraint is classification %	
	Oa 100	Oa 529	Oa 1024	Oa 1521	Average
Common MLP hl1			Oa=Orthogonal array		
Overall	0,0938797782	0,0672325933	0,04574397	0,1421469947	0,0872508341
Objective 1	0,1037506908	0,0854320262	0,0460756929	0,1478677066	0,0957815291
Objective 2	0,1126094949	0,0340318806	0,0298263373	0,0199735723	0,0491103213
Objective 3	0,0547278183	0,0714413382	0,057139763	0,1958405015	0,0947873552
Constraint	60	87,3417721519	75,3246753247	77,6315789474	75,074506606
Individual MLP hl1					
Overall	0,1329645801	0,1636289836	0,0885410678	0,1726609903	0,1394489055
Objective 1	0,133314516	0,1520505712	0,0742859119	0,1764853102	0,1340340773
Objective 2	0,1713446628	0,1748761237	0,1303759654	0,1440826947	0,1551698617
Objective 3	0,0768569129	0,1631634087	0,0316586956	0,193722908	0,1163504813
Constraint	26,6666666667	15,1898734177	12,3376623377	14,0350877193	17,0573225353
Common MLP hl2					
Overall	0,9841008157	0,0837132408	0,0503858286	0,2344942442	0,3381735323
Objective 1	1,4537105325	0,0709294855	0,0674336387	0,319245428	0,4778297712
Objective 2	0,3404943421	0,0331311991	0,0385704376	0,0222085612	0,108601135
Objective 3	0,8222849488	0,1220452859	0,0397645681	0,2501035549	0,3085495894
Constraint	66,6666666667	88,6075949367	91,5584415584	96,4912280702	85,830982808
Individual MLP hl2					
Overall	0,1329645801	0,1636289836	0,0885410678	0,1726609903	0,1394489055
Objective 1	0,133314516	0,1520505712	0,0742859119	0,1764853102	0,1340340773
Objective 2	0,1713446628	0,1748761237	0,1303759654	0,1440826947	0,1551698617
Objective 3	0,0768569129	0,1631634087	0,0316586956	0,193722908	0,1163504813
Constraint	0	22,7848101266	12,987012987	9,649122807	11,3552364802

Table 20: Training results for MLP designs (Hammersley sampling)

	Sampling technique	and no. of points	Error in RMSE		Constraint is classification %	
	Ham 100	Ham 500	Ham 1000	Ham 1500	Average	
Common MLP hl1			Ham=Hammersley sampling			
Overall	1,1357580005	0,1508314728	0,0719311712	0,0620252615	0,3551364765	
Objective 1	0,9446038059	0,240448001	0,0940409012	0,0612517979	0,3350861265	
Objective 2	0,2693220465	0,0440756835	0,0369419348	0,072994848	0,1058336282	
Objective 3	1,7044142667	0,0921547202	0,072896382	0,0496121823	0,4797693878	
Constraint	40	76	80,6666666667	87,5555555556	71,0555555556	
Individual MLP hl1						
Overall	0,4098209101	0,2302168798	0,1452905798	0,1274201494	0,2281871298	
Objective 1	0,5554313702	0,2208873734	0,1637732285	0,1020128996	0,2605262179	
Objective 2	0,1879167945	0,1822680391	0,1430175153	0,1477674367	0,1652424464	
Objective 3	0,4000535046	0,27746453	0,1266979781	0,1283192753	0,233133822	
Constraint	13,3333333333	17,3333333333	24,6666666667	20,8888888889	19,0555555556	
Common MLP hl2						
Overall	0,374318366	0,0748724329	0,0865704264	0,0377663975	0,1433819057	
Objective 1	0,4608443749	0,1079195766	0,1135959951	0,0396004938	0,1804901101	
Objective 2	0,0181121646	0,0195720357	0,0288417553	0,0290640406	0,0238974991	
Objective 3	0,4556721728	0,0691949711	0,0935276389	0,0431970458	0,1653979572	
Constraint	73,3333333333	94,6666666667	96	97,3333333333	90,3333333333	
Individual MLP hl2						
Overall	0,4098209101	0,2302168798	0,1452905798	0,1274201494	0,2281871298	
Objective 1	0,5554313702	0,2208873734	0,1637732285	0,1020128996	0,2605262179	
Objective 2	0,1879167945	0,1822680391	0,1430175153	0,1477674367	0,1652424464	
Objective 3	0,4000535046	0,27746453	0,1266979781	0,1283192753	0,233133822	
Constraint	13,3333333333	8	5,3333333333	9,3333333333	9	

Table 21: Statistics for MLP designs

	Statistics (MLP)							
	Average RMSE for hl1	STD	min value	max value	Average RMSE for hl2	STD	min value	max value
Common								
Overall	0,167760	0,306801	0,034931	1,135758	0,180217	0,271702	0,037766	0,984101
Objective 1	0,168734	0,228312	0,039324	0,944604	0,244191	0,364285	0,039600	1,453711
Objective 2	0,063293	0,063223	0,019974	0,269322	0,054340	0,082255	0,018112	0,340494
Objective 3	0,211965	0,427401	0,023376	1,704414	0,178545	0,214131	0,039765	0,822285
Constraint	73,265576	12,166469	40,000000	87,555556	87,536254	11,706753	60,000000	97,333333
Individual								
Overall	0,167855	0,084085	0,088541	0,409821	0,167855	0,084085	0,088541	0,409821
Objective 1	0,171174	0,127883	0,074286	0,555431	0,171174	0,127883	0,074286	0,555431
Objective 2	0,159139	0,019294	0,130376	0,187917	0,159139	0,019294	0,130376	0,187917
Objective 3	0,153939	0,106073	0,031659	0,400054	0,153939	0,106073	0,031659	0,400054
Constraint	16,537626	5,728948	6,666667	26,666667	10,951745	5,961698	0,000000	22,784810

Table 22: Training results for RBF network designs (Latin hypercube sampling (1/2))

		Sampling technique	and no. of points	Error in RMSE	Constraint is classification %	
	goal	Lhs 100	Lhs 500	Lhs 1000	Lhs 1500	Average
Common RBF a				Lhs= Latin Hypercube		
Overall	0,1	0,3175252213	0,0765107886	0,1333116268	0,0796538526	0,1517503723
Objective 1		0,3849147799	0,0967737015	0,0450679551	0,0734453737	0,1500504526
Objective 2		0,0182299851	0,0055310429	0,0085039513	0,0090551251	0,0103300261
Objective 3		0,392396583	0,0903657045	0,2263018597	0,1164387843	0,2063757329
Constraint		0	16	11,3333333333	20	11,8333333333
Common RBF as						
Overall	0,1	0,1000588437	0,0748890264	0,1333109871	0,0805342056	0,0971982657
Objective 1		0,088597526	0,0968423046	0,0450721196	0,073195145	0,0759267738
Objective 2		0,0381214553	0,0102931937	0,0085038629	0,0093391731	0,0165644213
Objective 3		0,143988019	0,0856779844	0,2262999032	0,1183745133	0,143585105
Constraint		0	17,3333333333	11,3333333333	19,1111111111	11,9444444444
Individual RBF as						
Overall	0,1	0,1393474408	0,1178265105	0,1669271849	0,1196175784	0,1359296786
Objective 1		0,1513514774	0,1386897487	0,0900722994	0,0957305225	0,118961012
Objective 2		0,1770874769	0,1366569949	0,1624520111	0,1518282608	0,1570061859
Objective 3		0,0631338542	0,0611496447	0,2215634901	0,1034840157	0,1123327512
Constraint		6,6666666667	16	11,3333333333	20	13,5

Table 23: Training results for RBF network designs (Latin hypercube sampling (2/2))

		Sampling technique	and no. of points	Error in RMSE	Constraint is classification %	
	goal	Lhs 100	Lhs 500	Lhs 1000	Lhs 1500	Average
Common RBF a				Lhs= Latin Hypercube		
Overall	0,5	0,3175075998	0,0765107886	0,1333116268	0,0796538526	0,1517459669
Objective 1		0,3848626705	0,0967737015	0,0450679551	0,0734453737	0,1500374252
Objective 2		0,0182307429	0,0055310429	0,0085039513	0,0090551251	0,0103302155
Objective 3		0,3924048835	0,0903657045	0,2263018597	0,1164387843	0,206377808
Constraint		0	16	11,3333333333	20	11,8333333333
Common RBF as						
Overall	0,5	0,1343590268	0,1219023643	0,1690410911	0,1612444124	0,1466367237
Objective 1		0,13130988	0,1459298358	0,1033330025	0,1026006157	0,1207933335
Objective 2		0,1770874769	0,1367682744	0,1613135115	0,2383989972	0,178392065
Objective 3		0,0745304311	0,0676718659	0,2214157019	0,1031421997	0,1166900497
Constraint		6,6666666667	0	0	15,1111111111	5,4444444444
Individual RBF as						
Overall	0,5	0,1393474408	0,1178265105	0,1669271849	0,1196175784	0,1359296786
Objective 1		0,1513514774	0,1386897487	0,0900722994	0,0957305225	0,118961012
Objective 2		0,1770874769	0,1366569949	0,1624520111	0,1518282608	0,1570061859
Objective 3		0,0631338542	0,0611496447	0,2215634901	0,1034840157	0,1123327512
Constraint		13,3333333333	13,3333333333	6	17,3333333333	12,5

Table 24: Training results for RBF network designs (Orthogonal array (1/2))

		Sampling technique	and no. of points	Error in RMSE	Constraint is classification %	
	goal	Oa 100	Oa 529	Oa 1024	Oa 1521	Average
Common RBF a				Oa=Orthogonal array		
Overall	0,1	0,3750795496	0,1053321308	0,0408272779	0,1356574633	0,1642241054
Objective 1		0,3170646942	0,0915426365	0,0368098899	0,1449865513	0,147600943
Objective 2		0,0122195129	0,0065020529	0,0065945293	0,005188467	0,0076261405
Objective 3		0,5668991698	0,1576776532	0,0600178644	0,1848264641	0,2423552879
Constraint		20	15,1898734177	12,3376623377	14,0350877193	15,3906558687
Common RBF as						
Overall	0,1	0,0917194243	0,1151111317	0,0392519141	0,136709264	0,0956979335
Objective 1		0,0800663155	0,1168288897	0,0373089805	0,1480914407	0,0955739066
Objective 2		0,0372641878	0,0247589681	0,0072231263	0,0099750741	0,0198053391
Objective 3		0,1320534885	0,1596550092	0,0563737944	0,1844930668	0,1331438397
Constraint		0	20,253164557	15,5844155844	14,4736842105	12,577816088
Individual RBF as						
Overall	0,1	0,1329645801	0,1636289836	0,0885410678	0,1726609903	0,1394489055
Objective 1		0,133314516	0,1520505712	0,0742859119	0,1764853102	0,1340340773
Objective 2		0,1713446628	0,1748761237	0,1303759654	0,1440826947	0,1551698617
Objective 3		0,0768569129	0,1631634087	0,0316586956	0,193722908	0,1163504813
Constraint		26,6666666667	15,1898734177	12,3376623377	14,0350877193	17,0573225353

Table 25: Training results for RBF network designs (Orthogonal array (2/2))

		Sampling technique	and no. of points	Error in RMSE	Constraint is classification %	
		Oa 100	Oa 529	Oa 1024	Oa 1521	Average
Common RBF a	goal			Oa=Orthogonal array		
Overall	0,5	0,3750942911	0,1053303383	0,0408272779	0,1356572445	0,1642272879
Objective 1		0,3171563064	0,0915402835	0,0368098899	0,1449868737	0,1476233384
Objective 2		0,0122179256	0,0065020509	0,0065945293	0,0051888116	0,0076258293
Objective 3		0,5668772188	0,1576754271	0,0600178644	0,1848257197	0,2423490575
Constraint		20	15,1898734177	12,3376623377	14,0350877193	15,3906558687
Common RBF as						
Overall	0,5	0,1463520728	0,1663228312	0,0926790734	0,1734204976	0,1446936188
Objective 1		0,1301300839	0,1530262879	0,0845827662	0,177753165	0,1363730758
Objective 2		0,1995947621	0,1783254952	0,1325215237	0,1444520213	0,1637234506
Objective 3		0,0865152003	0,1666518091	0,0324350636	0,1943230118	0,1199812712
Constraint		0	11,3924050633	1,9480519481	0	3,3351142528
Individual RBF as						
Overall	0,5	0,1329645801	0,1636289836	0,0885410678	0,1726609903	0,1394489055
Objective 1		0,133314516	0,1520505712	0,0742859119	0,1764853102	0,1340340773
Objective 2		0,1713446628	0,1748761237	0,1303759654	0,1440826947	0,1551698617
Objective 3		0,0768569129	0,1631634087	0,0316586956	0,193722908	0,1163504813
Constraint		0	22,7848101266	12,987012987	9,649122807	11,3552364802

Table 26: Training results for RBF network designs (Hammersley sampling (1/2))

	Sampling technique	and no. of points	Error in RMSE	Constraint is classification %	
	Ham 100	Ham 500	Ham 1000	Ham 1500	Average
Common RBF a	goal		Ham=Hammersley sampling		
Overall	0,1	0,3442746328	0,1393939332	0,1061417109	0,0747793913	0,166147417
Objective 1		0,3832488454	0,1301859388	0,125781488	0,0687709537	0,1769968065
Objective 2		0,0127481299	0,005966579	0,0061672987	0,0080378297	0,0082299593
Objective 3		0,4566540003	0,2032437628	0,1339371871	0,1094615061	0,2258241141
Constraint		13,3333333333	17,3333333333	24,6666666667	20,8888888889	19,0555555556
Common RBF as						
Overall	0,1	0,3666754345	0,1394964438	0,1061414183	0,0747840431	0,1717743349
Objective 1		0,4822602358	0,1316164215	0,1257865921	0,0687752026	0,202109613
Objective 2		0,0270188272	0,0100943388	0,0061682348	0,008037232	0,0128296705
Objective 3		0,4123683677	0,202368462	0,1339316549	0,1094684143	0,2145342247
Constraint		0	17,3333333333	24,6666666667	20,8888888889	15,7222222222
Individual RBF as						
Overall	0,1	0,4098209101	0,2302168798	0,1452905798	0,1274201494	0,2281871298
Objective 1		0,5554313702	0,2208873734	0,1637732285	0,1020128996	0,2605262179
Objective 2		0,1879167945	0,1822680391	0,1430175153	0,1477674367	0,1652424464
Objective 3		0,4000535046	0,27746453	0,1266979781	0,1283192753	0,233133822
Constraint		13,3333333333	17,3333333333	24,6666666667	20,8888888889	19,0555555556

Table 27: Training results for RBF network designs (Hammersley sampling (2/2))

		Sampling technique	and no. of points	Error in RMSE	Constraint is classification %	
		Ham 100	Ham 500	Ham 1000	Ham 1500	Average
Common RBF a	goal			Ham=Hammersley sampling		
Overall	0,1	0,3442746328	0,1393939332	0,1061417109	0,0747793913	0,166147417
Objective 1		0,3832488454	0,1301859388	0,125781488	0,0687709537	0,1769968065
Objective 2		0,0127481299	0,005966579	0,0061672987	0,0080378297	0,0082299593
Objective 3		0,4566540003	0,2032437628	0,1339371871	0,1094615061	0,2258241141
Constraint		13,3333333333	17,3333333333	24,6666666667	20,8888888889	19,0555555556
Common RBF as						
Overall	0,1	0,3666754345	0,1394964438	0,1061414183	0,0747840431	0,1717743349
Objective 1		0,4822602358	0,1316164215	0,1257865921	0,0687752026	0,202109613
Objective 2		0,0270188272	0,0100943388	0,0061682348	0,008037232	0,0128296705
Objective 3		0,4123683677	0,202368462	0,1339316549	0,1094684143	0,2145342247
Constraint		0	17,3333333333	24,6666666667	20,8888888889	15,7222222222
Individual RBF as						
Overall	0,1	0,4098209101	0,2302168798	0,1452905798	0,1274201494	0,2281871298
Objective 1		0,5554313702	0,2208873734	0,1637732285	0,1020128996	0,2605262179
Objective 2		0,1879167945	0,1822680391	0,1430175153	0,1477674367	0,1652424464
Objective 3		0,4000535046	0,27746453	0,1266979781	0,1283192753	0,233133822
Constraint		13,3333333333	17,3333333333	24,6666666667	20,8888888889	19,0555555556

Table 28: Statistics for RBF designs

	Statistics (RBF)							
	Average RMSE of 0,1	STD	min value	max value	Average RMSE of 0,5	STD	min value	max value
Common RBF a								
Overall	0,160707	0,115811	0,040827	0,375080	0,160695	0,115791	0,040827	0,375094
Objective 1	0,158216	0,128001	0,036810	0,384915	0,158207	0,127979	0,036810	0,384863
Objective 2	0,008729	0,003881	0,005188	0,018230	0,008730	0,003882	0,005189	0,018231
Objective 3	0,224852	0,160673	0,060018	0,566899	0,224834	0,160644	0,060018	0,566877
Constraint	15,426515	6,264346	0,000000	24,666667	15,426515	6,264346	0,000000	24,666667
Common RBF as								
Overall	0,121557	0,082697	0,039252	0,366675	0,173221	0,081558	0,092679	0,409296
Objective 1	0,124537	0,117681	0,037309	0,482260	0,174274	0,129254	0,084583	0,564477
Objective 2	0,016400	0,011982	0,006168	0,038121	0,168714	0,030623	0,132522	0,238399
Objective 3	0,163754	0,091739	0,056374	0,412368	0,154841	0,100852	0,032435	0,387696
Constraint	13,414828	8,751203	0,000000	24,666667	4,519112	5,640321	0,000000	15,111111
Individual RBF as								
Overall	0,167855	0,084085	0,088541	0,409821	0,167855	0,084085	0,088541	0,409821
Objective 1	0,171174	0,127883	0,074286	0,555431	0,171174	0,127883	0,074286	0,555431
Objective 2	0,159139	0,019294	0,130376	0,187917	0,159139	0,019294	0,130376	0,187917
Objective 3	0,153939	0,106073	0,031659	0,400054	0,153939	0,106073	0,031659	0,400054
Constraint	16,537626	5,728948	6,666667	26,666667	10,951745	5,961698	0,000000	22,784810

B Matlab codes for Single layer network examples

Training algorithms might stuck on a local minimum point, hence achieving the same picture than in examples it might take several runs. In some examples the plot might need some 'twisting' as well.

```
% Curve fitting
x=[0 0 0 0.5 0.5 0.5 1 1 1 1.5 1.5 1.5 2 2 2 2.5 2.5 2.5 3 3 3];
y=[0 1 2 1 2 3 2 3 4 3 4 5 4 5 6 5 6 7 6 7 8];
net = perceptron;
net.layers{1}.transferfcn='purelin';
net.layers{1}.dimensions=1;
net.performfcn='mse';
net.trainfcn='trainbfg';
net=configure(net,X,Y);
net=train(net,X,Y);
plot(x,y,'o');
hold on
xt=[0 1 2 3 ];
yt=net(xt);
plot(xt,yt,'r');
```

```
% XOR-problem
x=[0 0 1 1;0 1 0 1];
y=[0 1 1 0];
net=perceptron;
net.layers{1}.transferfcn='purelin';
X=repmat(con2seq(x),1,3);
Y=repmat(con2seq(y),1,3);
net=adapt(net,X,Y);
span = -0.5:.5:1.5;
[P1,P2] = meshgrid(span,span);
mt = [P1(:) P2(:)]';
mr=net(mt);
mesh(P1,P2,reshape(mr,length(span),length(span))-5);
hold on;
plot3(0,0,0,'ko','MarkerEdgeColor','k','MarkerFaceColor','k','MarkerSize',7)
plot3(1,1,0,'ko','MarkerEdgeColor','k','MarkerFaceColor','k','MarkerSize',7)
plot3(0,1,0,'kx','MarkerEdgeColor','k','MarkerFaceColor','k','MarkerSize',10)
plot3(1,0,0,'kx','MarkerEdgeColor','k','MarkerFaceColor','k','MarkerSize',10)
colormap cool;
```

```
% Function approximation
x=[0:.1:10];
y=cos(x)+0.5*sin(x.^ 2);
X=con2seq(x);
Y=con2seq(y);
net=perceptron;
net.layers{1}.transferfcn='purelin';
net.trainParam.epochs=10;
net=train(net,X,Y);
yt=net(X);
plot(cell2mat(Y));
hold on
plot(cell2mat(yt),'r');
net.biasconnect=0;
yt2=net(X);
plot(cell2mat(yt2),'–r');
```

C Matlab codes for Multilayer network examples

```
% XOR-problem
x=[0 0 1 1;0 1 0 1];
y=[0 1 1 0];
net=feedforwardnet;
net.numlayers=2;
net.biasConnect= [1; 1];
net.inputConnect= [1; 0];
net.layerConnect= [0 0;1 0];
net.outputConnect= [0 1];
net.layers{1}.transferfcn='tansig';
net.layers{2}.transferfcn='purelin';
net.trainfcn='trainbfg';
net.layers{1}.dimensions=2;
net.divideparam.trainratio=1;
net.divideparam.valratio=0;
net.divideparam.testratio=0;
net=train(net,x,y);
span = -1:.005:2;
[P1,P2] = meshgrid(span,span);
mt = [P1(:) P2(:)]';
mr=net(mt);
mesh(P1,P2,reshape(mr,length(span),length(span))-5);
hold on;
plot3(0,0,0,'ko','MarkerEdgeColor','k','MarkerFaceColor','k','MarkerSize',7)
plot3(1,1,0,'ko','MarkerEdgeColor','k','MarkerFaceColor','k','MarkerSize',7)
plot3(0,1,0,'kx','MarkerEdgeColor','k','MarkerFaceColor','k','MarkerSize',10)
plot3(1,0,0,'kx','MarkerEdgeColor','k','MarkerFaceColor','k','MarkerSize',10)
colormap cool;
```

```
% Function approximation with one hidden layer
x=[0:.1:10];
y=cos(x)+0.5*sin(x.^ 2);
X=con2seq(x);
Y=con2seq(y);
net = feedforwardnet;
net.numlayers=2;
net.biasConnect= [1; 1];
net.inputConnect= [1; 0];
net.layerConnect= [0 0; 1 0];
net.outputConnect= [0 1];
net.layers{1}.transferfcn='tansig';
net.layers{2}.transferfcn='purelin';
net.trainfcn='trainbfg';
net.trainParam.epochs=1000;
net.trainParam.min_grad=1e-010;
net.trainparam.goal=0.0000001;
net.layers{1}.dimensions=100;
net.divideparam.trainratio=1;
net.divideparam.valratio=0;
net.divideparam.testratio=0;
net=configure(net,X,Y);
net=train(net,X,Y);
yt=net(X);
plot(cell2mat(Y),'kx','Markersize',10);
hold on
plot(cell2mat(yt),'r');
```

```
% Function approximation with two hidden layer
x=[0:.1:10];
y=cos(x)+0.5*sin(x.^ 2);
X=con2seq(x);
Y=con2seq(y);
net = feedforwardnet;
net.numlayers=3;
net.biasConnect= [1; 1; 1];
net.inputConnect= [1; 0; 0];
net.layerConnect= [0 0 0; 1 0 0; 0 1 0 ];
net.outputConnect= [0 0 1];
net.layers{1}.transferfcn='tansig';
net.layers{2}.transferfcn='tansig';
net.layers{3}.transferfcn='purelin';
net.trainfcn='trainbfg';
net.trainParam.epochs=1000;
net.trainParam.min_grad=1e-010;
net.trainparam.goal=0.0000001;
net.layers{1}.dimensions=50;
net.layers{2}.dimensions=3;
net.divideparam.trainratio=1;
net.divideparam.valratio=0;
net.divideparam.testratio=0;
net=configure(net,X,Y);
net=train(net,X,Y);
yt=net(X);
plot(cell2mat(Y),'kx','Markersize',10);
hold on
plot(cell2mat(yt),'r');
```

```
% Function approximation with random number of hidden neurons
x=[0:.1:10];
y=cos(x)+0.5*sin(x.^ 2);
X=con2seq(x);
Y=con2seq(y);
net = feedforwardnet;
net.numlayers=3;
net.biasConnect= [1; 1; 1];
net.inputConnect= [1; 0; 0];
net.layerConnect= [0 0 0; 1 0 0; 0 1 0 ];
net.outputConnect= [0 0 1];
net.layers{1}.transferfcn='tansig';
net.layers{2}.transferfcn='tansig';
net.layers{3}.transferfcn='purelin';
net.trainfcn='trainbfg';
net.trainParam.epochs=10000;
net.trainParam.min_grad=1e-010;
net.trainparam.goal=0.0000001;
net.layers{1}.dimensions=10;
net.layers{2}.dimensions=5;
net.divideparam.trainratio=1;
net.divideparam.valratio=0;
net.divideparam.testratio=0;
net=configure(net,X,Y);
net=train(net,X,Y);
yt=net(X);
plot(cell2mat(Y),'kx','Markersize',10);
hold on
plot(cell2mat(yt),'r');
```

D Matlab code for Recurrent multilayer network example

Training algorithms might stuck on a local minimum point, hence achieving the same picture than in examples it might take several runs. In some examples the plot might need some 'twisting' as well.

```
% Function approximation
x=[0:.1:10];
y=cos(x)+0.5*sin(x.^ 2);
xp=[0:.1:11];
yv=cos(xp)+0.5*sin(xp.^ 2);
X=con2seq(x);
Y=con2seq(y);
XP=con2seq(xp);
YV=con2seq(yv);
net = feedforwardnet;
net.numlayers=3;
net.biasConnect= [1; 1; 1];
net.inputConnect= [1; 0; 0];
net.layerConnect= [1 0 0; 1 1 0; 0 1 1];
net.outputConnect= [0 0 1];
net.layers{1}.transferfcn='tansig';
net.layers{2}.transferfcn='tansig';
net.layers{3}.transferfcn='purelin';
net.trainfcn='trainbfg';
net.trainparam.goal=0.1;
net.layers{1}.dimensions=8;
net.layers{2}.dimensions=6;
% Continues in next page
```

```
% Continue from previous page
net.layerWeights{1,1}.delays=1;
net.layerWeights{2,2}.delays=1;
net.layerWeights{3,3}.delays=1;
net.divideparam.trainratio=1;
net.divideparam.valratio=0;
net.divideparam.testratio=0;
net=configure(net,X,Y);
net=train(net,X,Y);
yt=net(X);
yp=net(XP);
subplot(2,1,1);
plot(cell2mat(Y));
hold on
plot(cell2mat(yt),'r');
title('A) Sequence recognition');
subplot(2,1,2);
plot(cell2mat(YV));
hold on
plot(cell2mat(yp),'r');
title('B) Sequence prediction to +1s');
```

E Matlab code for RBF network examples

```
% XOR-problem
x=[0 0 1 1;0 1 0 1];
y=[0 1 1 0];
t=[1 0;1 0];
for i=1:4
f1(i)=exp(-distance(x(:,i)',t(:,1)')^ 2);
f2(i)=exp(-distance(x(:,i)',t(:,2)')^ 2);
hold on
if (f2(i) - f1(i))^ 2 <= 0.1 % For some reason Matlab calculates distance of [1,0]-[1,1] bit differently so
plot(f1(i),f2(i),'ko','MarkerEdgeColor','k','MarkerFaceColor','k','MarkerSize',8)
end
if (f2(i) - f1(i))^ 2 >= 0.5
plot(f1(i),f2(i),'kx','MarkerEdgeColor','k','MarkerFaceColor','k','MarkerSize',10)
end
end
axis([-0.25 1.25 -0.25 1.25]);
span = -0.5:.1:1.5;
f3=-span+1; % linear function separating classes
plot(f3,span)
```

```
% Function approximation
x=[0:.1:10];
y=cos(x)+0.5*sin(x.^ 2); net=newrb(x,y,0.2);
p=net(x);
net2=newrb(x,y);
z=net2(x);
plot(x,y,'kx')
hold on
plot(x,p)
plot(x,z,'–r')
```