Tobias Peter

Using Deep Learning as a surrogate model in Multi-objective Evolutionary Algorithms

OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

INF FAKULTÄT FÜR INFORMATIK

Institute for Intelligent Cooperative Systems

Master Thesis

Using Deep Learning as a surrogate model in Multi-objective Evolutionary Algorithms

Author:	Tobias Peter
Supervisor:	Sanaz Mostaghim
Advisor:	Heiner Zille

Tobias Peter: *Using Deep Learning as a surrogate model in Multi-objective Evolutionary Algorithms*
Otto-von-Guericke-Universität
Magdeburg, 2018.

Abstract

Multi-objective Evolutionary Algorithms (MOEAs) are popular tools to solve optimization problems in the field of engineering, because of their solid performance on problems with large design spaces and difficult fitness landscapes. However, MOEAs still need many evaluations of the objective function to solve a typical real-world problem. This is further complicated by the fact that many such problems require multiple minutes or half an hour for even a single evaluation. Combined, this can make the use of MOEAs unfeasible. One way to alleviate this is the integration of surrogate functions which learn to approximate the fitness landscape from a training set of example evaluations. While other methods also exist, in this thesis we will focus on approaches using Artificial Neural Networks for the approximation task. We want to understand if the performance of such an approach can be improved by using deeper networks. Deep ANNs are the focus of the renewed interest in ANNs commonly named deep learning (DL). Deep learning concerns itself with the ability of ANNs with more than one hidden layer to decompose a problem into underlying factors which should give them the potential to approximate more complicated problems. We propose an extension of a previously published approach to surrogate-assisted MOEAs that allows us to compare ANNs with different numbers of hidden layers in a number of experiments. The different architectures are compared on problems of varying difficulty, from simple unimodal problems to problems with difficult multimodal or flat fitness landscapes. The results show that the proposed method outperforms a method with only one hidden layer, especially on higher-dimensional problems, on some common test problems, but the potential of deep learning is not fully unlocked on highly multimodal test problems.

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

CFD	computational fluid dynamics
FEM	finite element method
EA	evolutionary algorithm
ML	machine learning
ANN	artificial neural network
DL	deep learning
MOEA	multi-objective evolutionary algorithm
SBX	simulated binary crossover
ReLU	rectified linear unit
DL4J	Deep Learning for Java
SGD	Stochastic Gradient Descent
MSE	mean squared error
RMSE	root mean squared error
MAE	mean absolute error
tanh	hyperbolic tangent
MOP	multi-objective optimization problem
LSM	least squares method

RSM	response surface methodology
RBF	radial basis function
NEC	No Evolution Control
FEC	Fixed Evolution Control
AEC	Adaptive Evolution Control

1 Introduction

This chapter will start with an example to motivate why surrogate functions are used today and why we propose to use deep networks as a surrogate model. Then we will present the research questions this thesis will answer.

1.1 Motivation

For a minute, let us imagine we have just inherited a car factory and want to build a new car. There are two things we especially like about cars: They should be fast, therefore aerodynamically shaped, but still be as affordable as possible. Naturally, we cannot build a car that is exceptionally fast and inexpensive at the same time.

A very aerodynamic car would be a Formula One racing car. It is shaped like an arrow for minimum air resistance and made out of exotic materials, like carbon fiber reinforced polymers, to reduce the weight. All that to achieve a maximum speed of roughly 350 km/h, though, at a price tag of more than $6 million, it is anything but affordable. On the other end of the spectrum, the Tata Nano can be found. This Indian city car can reach speeds of 105km/h with its 38 horsepower engine, and a new one sells for only $2500. Those are two extreme examples, but the same observations are also valid for more reasonable examples. To make a car faster, we have to include a better (more expensive) engine and use lighter materials (also more expensive). We simply can not make a car that is both fast and affordable. We always have to decide between some alternatives that represent different trade-offs or compromises between high speed and low price.

Nonetheless, some cars that we could build in our factory are both faster and cheaper than some others. A small set of cars, however, belong to a set of car designs that can not be improved upon price without decreasing the speed of the vehicle or the other way around. These designs are called Pareto-optimal designs and belong to the Pareto-optimal set. As a car maker, we are interested in such designs because if we release a car with a sub-optimal design, i.e., not a Pareto-optimal design, another car maker could release a car that is both faster and cheaper, and a potential customer would have no reason to buy one of our automobiles. Therefore, we have to invest in some facilities that can be used to test potential car designs and guide us towards Pareto-optimal designs. Traditionally, we would use a wind tunnel to test the aerodynamics of our car, but today it would be more economical to use 3d models and computer simulations, like the finite element method (FEM) or computational fluid dynamics (CFD). What these computer programs have to provide are projected values for the material costs of the vehicle and the speed it can reach. Although, we do not want to do this search by hand and compare all these values manually. We are already using a computer, so we can just use a software product that was created for this kind of task. This type of software is called an optimizer. There exist many optimizers, but for our task, an evolutionary algorithm (EA) is a good choice. The circumstances that make an optimizer like an EA necessary are the fact that the only pieces of information we have about our car optimization problem are the functions that compute the maximum speed and the cost of the car. For example, we do not have a gradient, and some optimizers work only if gradient information is available. We could sample many designs and derive gradient information from that, but the number of possible car designs is too large for this approach. EAs can find the Pareto-optimal solutions with only the given pieces of information and without sampling an unreasonable number of car designs. They are inspired by biological evolution, primarily by the idea of "survival of the fittest." A typical EA can operate on an abstract description of our car and change parts of that description to find better and better vehicle designs, and ultimately the Pareto-optimal set.

If we apply an EA to our optimization problem, we will make two important observations. One is that an EA, because of its nature and the difficult circumstances, needs to test a lot of car designs. The other is that the evaluation of the speed objective takes a lot of time. The high-fidelity FEM or CFD simulations can take minutes or half an hour to process a single car design. If we add these factors up, the optimization can take an unfeasible amount of time to find the Pareto-optimal designs. Therefore, we search for a way to reduce the total amount of time that is needed to optimize the car shape. What we will most likely find are so-called surrogate functions. Surrogate functions can replace a computationally expensive objective function with a function that can be evaluated quickly. To use a surrogate function, machine learning (ML) algorithms and statistical methods are used to learn the relations between inputs and outputs of the real objective function. If the training is successful, the output of the surrogate function can replace the real objective function while being much cheaper to evaluate.

Different ML algorithms can be used to learn the objective function, for example, artificial neural networks (ANNs) which are trying to imitate mammalian brains. ANNs date back to the 1940's and fell out of favor until recently when they gained popularity again because of the success of deep learning (DL) methods. Traditionally, ANNs use only one hidden layer, but deep learning is concerned with the effectiveness of multiple hidden layers. This allows these deep ANNs to learn a problem as a hierarchy of representations, where higher levels in the hierarchy build on the representations given in the lower levels. The topic of this thesis will be about how the new DL methods can improve the performance of a surrogate method that used ANNs with only one hidden layer before. We will test its performance on multiple test problems, some easy and some more difficult. We are interested to see if deep networks can approximate difficult problems, which would be difficult to learn with only one hidden layer.

1.2 Research Goals

The main goal of this thesis is to investigate if deep neural networks can be used to improve an ANN-based surrogate method that has only used shallow networks before. ANNs have been used as surrogate models before, but as far as the author is aware, there is no published research where the effects of using multiple hidden layers on an ANNs-based surrogate methods have been studied. Deep networks have some properties that can potentially improve the efficiency of surrogate functions, but using deep networks as a surrogate model also introduces new hyperparameters that need to be tuned. To find good hyperparameters, our deep model is tested on common benchmark problems, as well as more difficult test problems. The goal is to determine if the increased expression power of deeper networks makes them better at handling more complex optimization problems. Deep networks also show promise for high-dimensional problems, for this reason, we will also look at how the performance of deep network surrogates scales with problems with different dimensions. Additionally, we want to relate the performance of our proposed method to network structures that have been used in previous research.

Summary of the research goals:

- Apply deep neural networks to surrogate-assisted multi-objective evolutionary algorithms.
- Up to now, most papers used the ZDT test suite to benchmark their surrogate models. Therefore, we want to see how our surrogate model performs on newer test suites with more complex problems.
- How our proposed method scales up when used on higher dimensional problems.
- Put the performance of the proposed method in relation to surrogate models using shallow networks.

1.3 Structure of the Thesis

In Chapter 2 we will review the background knowledge required to understand the terms and methods used in this thesis. The topics covered include multi-objective optimization problems (MOPs), which can be solved using MOEAs, how surrogate models can be integrated into said MOEAs and finally ANNs, which will be used as the surrogate model in the proposed surrogate method. Chapter 3 will introduce related research in the field of surrogate-assisted MOEAs. Chapter 4 will explain our proposed method in detail, and Chapter 5 contains various experiments to establish the quality of our surrogate model. Finally, Chapter 6 will draw a conclusion and suggest how further research can build on this thesis.

2 Background

This chapter reviews the necessary concepts. First, we look at the formal definition of multi-objective optimization problems (MOPs) and some of their properties. Then, we will introduce how multi-objective evolutionary algorithms (MOEAs) can be used to solve MOPs and take a detailed look at NSGA-II, a widely used MOEA. The next section will familiarize the reader with surrogate functions, which statistical models are used to build them and how they can be integrated into EAs. The last section explains artificial neural networks (ANNs), which is the surrogate model used in this thesis.

2.1 Multi-objective Optimization

Optimization is the process of finding the best solution from a set of feasible solutions or a feasible region. The quality of a solution is given by an objective function. In this thesis, we only look at minimization problems, i.e., the best solution is defined as the solution that receives the lowest possible value from the objective function. Furthermore, we are concerned with optimization problems where multiple objective functions have to be satisfied. These so-called multi-objective optimization problems can be formally stated as

$$\min_{\vec{x}} \vec{f}(\vec{x}) \qquad \text{where: } \vec{f}(\vec{x}) = [f_1(\vec{x}), ..., f_m(\vec{x})] \tag{2.1}$$

where $\vec{f}$ is a vector function, containing m objective functions, and $\vec{x}$ is one of the solutions.

A solution vector $\vec{x}$ consists of n real-valued variables $\vec{x} = [x_1, ..., x_n]$ and is part of the feasible solutions $\Omega \subseteq \mathbb{R}^n$. The feasible region Ω contains all solutions that satisfy some given constraints, for example

$$x_i^{\text{lower}} \leq x_i \leq x_i^{\text{upper}} \qquad i = 1, ..., n \tag{2.2}$$

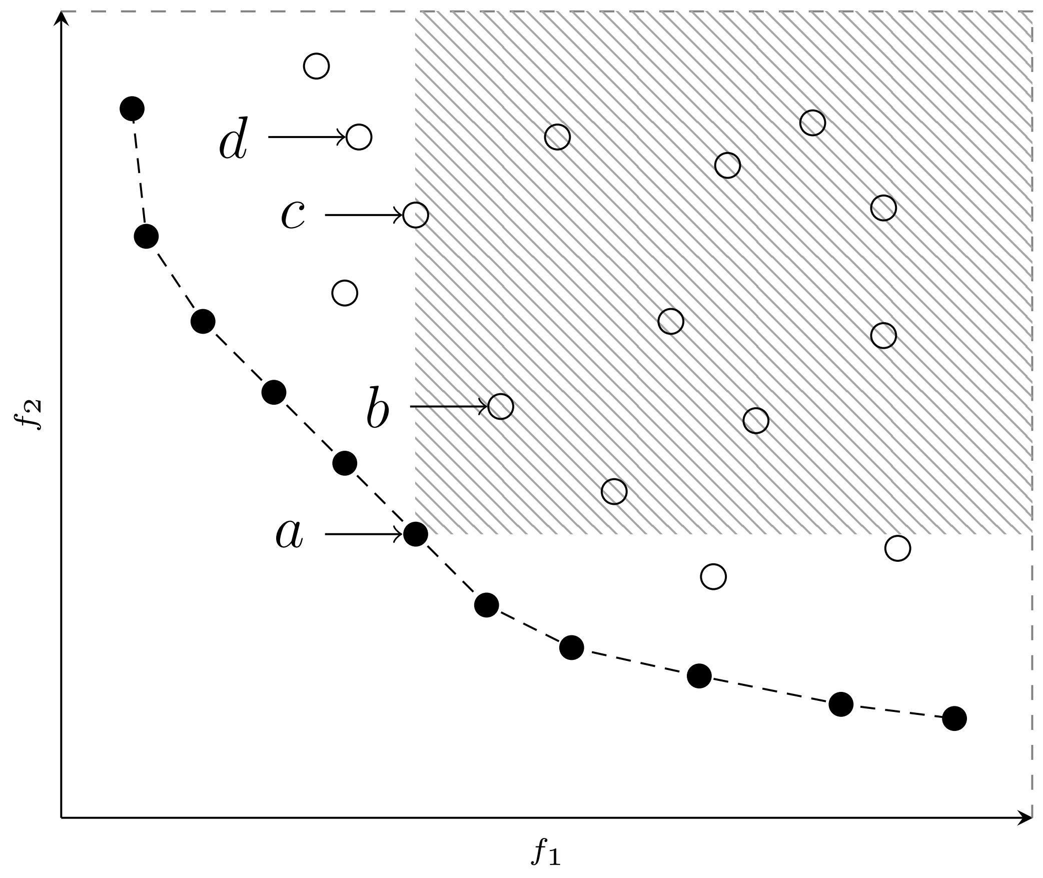


Figure 2.1: Example of an Pareto-front in the objective space O. The circles represent the objective vectors $\vec{f}(\vec{x})$ of some example solutions. The filled circles are the objective vectors of the Pareto-optimal solutions in $\mathcal{P}$. The dashed line is an approximation of the Pareto-optimal front. There exist no solutions with objective vectors beyond (closer to the origin) the Pareto-optimal front. All solutions in $\mathcal{P}$ lie somewhere on this front. The remaining unfilled circles show the objective values of sub-optimal solutions in Ω, i.e. solutions not in $\mathcal{P}$.
The shaded box shows the region of the objective space that is dominated by the Pareto-optimal solution a. Every point in the shaded region is dominated by a, for example b. The solution c is also dominated by a because it has the same objective value in f_1 but higher (worse) value in f_2. The solution d is not dominated by a because it has a better value in f_1.

which defines the minimum and maximum values any x_i can take.

The vector function $\vec{f}$ maps any solution $\vec{x} \in \Omega$ into the objective space (or fitness space) $O = \mathbb{R}^m$. How the feasible region is mapped into the fitness space shapes the fitness landscape of the optimization problem. The fitness landscape determines which difficulties are encountered during optimization. The next section details some commonly encountered types of fitness landscapes.

In other words, the task posed by Equation 2.1 is to determine one solution $\vec{x}^{utopian}$ from among the set of all feasible solutions $\vec{x} \in \Omega$, that yields the smallest values for all m objective functions. However, the objective functions in MOPs are usually conflicting, which means that there is no single solution $\vec{x}^{utopian} \in \Omega$ that minimizes all objective functions simultaneously. An objective function f_i conflicts with another objective function f_j if the solution $\vec{x}$ that generates the best possible value of f_i does not generate the optimal value of f_j, and vice versa. To optimize one objective function, one has to accept sub-optimal performance in the other objectives. Therefore, the solution to a MOP is not a single point $\vec{x}^{utopian}$ but a set of solutions. They are called the Pareto-optimal solutions and are contained in the Pareto-optimal set $\mathcal{P}$. All Pareto-optimal solutions are better than all solutions not in $\mathcal{P}$, but changing them to improve them in one objective will deteriorate their value in another objective function. Formally, the Pareto-optimal set is defined as

$$\mathcal{P} := \{\vec{x} \in \Omega | \nexists \vec{x}' \in \Omega : \vec{f}(\vec{x}') \preceq \vec{f}(\vec{x})\} \tag{2.3}$$

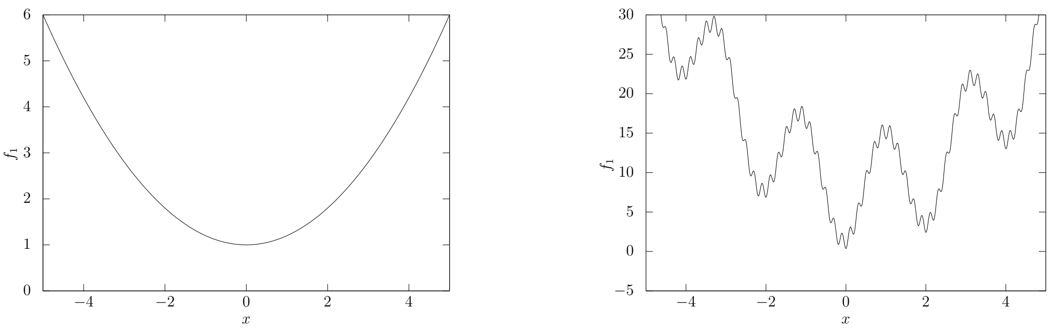
The symbol $\preceq$ is used to signify Pareto-dominance i.e. $\vec{x} \preceq \vec{x}'$ means $\vec{x}$ dominates $\vec{x}'$. A solution $\vec{x}$ is said to dominate another solution $\vec{x}'$ if and only if

$$\forall_i (f_i(\vec{x}) \leq f_i(\vec{x}')) \qquad i = 1, ..., m \tag{2.4}$$

and there is at least one i such that

$$f_i(\vec{x}) < f_i(\vec{x}') \tag{2.5}$$

In other words, a solution $\vec{x}$ is dominates another solution $\vec{x}'$ if and only if $\vec{x}$ is better or equal in all objectives and strictly better in at least one objective. Figure 2.1 shows some examples of Pareto-dominance. The solution a dominates any other solution that is inside the shaded area, like b. It also dominates c which has the same objective value in f_1 but a worse objective value in f_2.



(a) unimodal (b) multimodal

Figure 2.2: Examples of unimodal and multimodal fitness landscapes.

It does not dominate d, because d has a better value in f_1, or any of the other Pareto-optimal solution (represented as filled circles).

When each solution in the Pareto-optimal set is mapped into the objective space O using the objective functions $\vec{f}(\vec{x})$ they will create the Pareto-front $\mathcal{PF}$.

$$\mathcal{PF} := \{[f_1(\vec{x}), ..., f_m(\vec{x})] | \vec{x} \in \mathcal{P}\} \tag{2.6}$$

A visual example of a Pareto-front is shown in Figure 2.1. The filled circles represent the objective vectors $\vec{f}(\vec{x})$ of the Pareto-optimal solutions in $\mathcal{P}$. The Pareto-front shows the possible trade-offs between the two objectives. The solution in the top-left is a solution where much of the performance in objective f_2 is sacrificed to gain the best performance in f_1. Conversely, the bottom-right solution trades good performance in f_1 for the best possible performance in f_2. The solutions in between these two extremes represent different degrees of trade-off between f_1 and f_2.

In the context of this thesis we are interested in MOPs where the call to the function $\vec{f}(\vec{x})$, also called an evaluation, to calculate the objective values for a single solution is computationally expensive. More on that in Section 2.4.

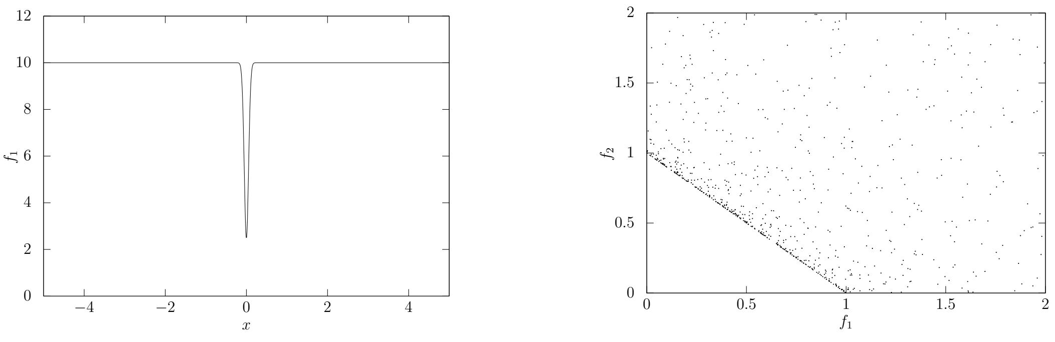
2.2 Fitness Landscapes

As mentioned earlier, the mapping from the search space to the fitness space defines the fitness landscape. Of interest here is the nature of the fitness landscape, which determines the difficulties that the optimization algorithm will encounter during optimization. Our examples will mostly show the fitness landscapes of one-dimensional objective functions, but they translate to multi-objective function because the fitness landscape of each objective can be analyzed in isolation.

The unimodal landscape is the most straightforward landscape an optimizer could encounter. This landscape is defined by having a single optimum which is also the global optimum, i. e. the best objective value that can be achieved. What makes it so straightforward, is the fact that any randomly generated solution can find the global optimum by just following the gradient of the objective function. This case is visualized in figure 2.2a. If an optimizer is at any position $x < 0$ he has only to increase x to find the global optimum or decrease x if he finds himself at any point $x > 0$.

A multimodal fitness landscape is more complicated, like the example shown in figure 2.2b. In the vast majority of cases, if the optimizer begins at a random position and just follows the gradient, like in the unimodal case, he will get stuck in a local optimum. Being stuck means that the optimizer thinks he has found the global optimum. For example, imagine we are at the point $x = 4$ after we followed the gradient. Irrespective of the direction we choose ($-x$ or $+x$ in this case) our objective value will get worse. Our gradient following optimizer will now stop and return $x = 4$ as the globally optimal solution. Unfortunately, there is no way of telling the optimizer that this is not the globally optimal solution because it is the task of the optimizer to find the optimal solution in the first place. To succeed with simple gradient following the initial random solution has to land in a very narrow region around $x = 0$. To succeed without luck, the optimizer has to employ a search strategy that negotiates the local optima and finds the small region containing the global optimum.

Another factor is whether or not unique values in the search space map to unique values in the objective space. The most fortunate case is a one-to-one mapping where each parameter vector $\vec{x}$ evaluates to a unique objective vector. A many-to-one fitness landscape is one where multiple parameter vectors evaluate to one objective vector. Why this is difficult is demonstrated in 2.3a. The figure shows an extreme case of many-to-one mapping, where entire regions of the parameter space Ω map to the same objective vector. If the optimizer finds itself in such a region, a so-called flat region, he cannot find information



(a) flat (b) biased

Figure 2.3: Examples of flat and biased fitness landscapes.

that guides him towards the global optimum because perturbations of the parameter vectors will most likely not generate a change in the objective vectors. Optimization problems containing flat regions are usually very hard to solve.

The bias of a fitness landscape relates to the distributions of the solutions in the search space and the distribution of their respective objective vectors in the objective space. A fitness landscape is biased if an equally distributed random sample of vectors in the search space maps to objective vectors that are more likely to be in a small region in the objective space. An example of bias can be seen in figure 2.3b where a sample of 3000 equally distributed vectors is shown after they were mapped into the objective space. We can see that the corresponding objective vectors are more likely to be close to the line through $(0, 1)$ and $(1, 0)$. Since this line would be the Pareto-optimal front $\mathcal{PF}$, this bias would actually make the search easier. If, on the other hand, the fitness landscape is biased towards regions which are far from the Pareto-optimal front the search would become more difficult.

2.3 Multi-objective Evolutionary Algorithms

Multi-objective evolutionary algorithms (MOEAs) belong the class of metaheuristic optimization techniques. These techniques are abstract computational techniques for solving numerical and combinatorial problems. Metaheuristics are usually applied to problems for which only the objective function is known and that have too many feasible solutions to be sampled thoroughly. In most cases where more information than the objective function is known there also exists a more performant method to solve it. If the feasible region, from here on also called search space, is small, the complete set can be evaluated to reveal the optimal solutions. Metaheuristics are applied when most other methods fail.

They use an abstract sequence of operations on abstract objects, and to implement a metaheuristic algorithm to solve a specific problem the abstract operations and objects have to be adapted to the problem. The abstract operators and objects used in an EA are inspired by the processes of biological evolution, i.e., survival of the fittest. They keep a population P of k individuals (the EA term for solution), and the fittest (best) individuals are allowed to reproduce (create new similar solutions) which improves the whole population over time. At the beginning of the search, the individuals of the population are dispersed over the whole search space. This is called exploration and is used to find regions in the search space that could contain the global optimum. If promising regions are found, the exploitation begins. Exploitation is the process of thoroughly searching only the promising regions of the search space, in the case of EAs by concentrating more individuals in that region. When a region is found to no longer be promising, the individuals move to other more promising regions in the search space. At the end of the search, the whole population is usually gathered around the global optimum, searching that region exhaustively.

In the multi-objective case, MOEAs also benefit from the population-based approach[13]. At the end of a successful search, the objective vectors of all individuals in the population are equally spread over the whole Pareto-front $\mathcal{PF}$. Other approaches that produce only one of the Pareto-optimal solution need to be applied multiple times. MOEAs have the potential to find an approximation of $\mathcal{P}$ in a single search.

The first EAs that could efficiently solve MOPs were introduced in the early '90s. Fonseca and Fleming's MOGA[22], Srinivas and Deb's NSGA[54] and Hoern et al.'s NPGA[29] are some examples. They all showed the necessary changes for conversion of a single-objective evolutionary algorithm to a multi-objective evolutionary algorithm. Common features were nondominated sorting and a strategy to preserve the diversity of the population, the equal spread along the $\mathcal{PF}$.

2.3.1 The General Evolutionary Algorithm

Algorithm 2.1: General Evolutionary Algorithm

```
1 $t \leftarrow 0$
2 initialize $P_0$ randomly
3 evaluate $P_0$
4 **while** *termination criterion not reached* **do**
5     $t \leftarrow t + 1$
6     select $P_t$ from $P_{t-1}$
7     apply genetic operators to $P_t$
8     evaluate $P_t$
9 **return** $P_t$
```

Before we can apply an EA to a problem we have to define what individual or a population is and how the fitness will be calculated. In our case, we can simply define that an individual is a feasible solution $\vec{x} = [x_1, ..., x_n]$ and the population is a list of solutions $P = \{\vec{x}^{(1)}, ..., \vec{x}^{(k)}\}$ with k being the population size. The objective function $\vec{f} = [f_1(\vec{x}), .., f_m(\vec{x})]$ is used as the fitness function.

Algorithm 2.1 shows how an EA would optimize a problem. We start in Line 2 where the random initial population P_0 is created, and the fitness of each individual in the initial population is determined using the objective function(Line 3).

In Line 4, we enter the main loop of the EA, every iteration of this loop is called a generation. The termination criterion can be a predefined number of generations (or evaluations) or one that measures the progress and stops the EA if said progress becomes too slow. The selection step is shown in Line 6, here a mating pool is created which contains the individuals who are allowed to reproduce and create new child individuals. The selection process should be designed in a way that fitter individuals are more likely to be selected for reproduction. This way information contained in the fitter individuals is disseminated to the other individuals in the population and will increase the average fitness of the population. In Line 7, we apply the genetic operators, this usually involves a crossover operator and a mutation operator. The crossover operator takes some individuals as input and outputs the same amount of individuals, but the new individuals represent a mix of the original individuals.

This way information is transferred among individuals and since the better individuals are more likely to be selected it should improve the overall fitness of the population. The mutation operator applies small random changes to an individual in the hope that these changes will increase the fitness of the individual. The mutation can be considered as a random search around the position around that individual in the search space. After all operators are applied, the new child generation is assessed (Line 8) and we reach the next iteration of the main loop, the next generation.

If the termination criterion is satisfied, the EA stops and returns the current population.

2.3.2 Selected Operators

This section presents selected evolutionary operators that will be used throughout this thesis. As the selection operator we will be using binary tournament selection, for the crossover operator we use simulated binary crossover and polynomial mutation as the mutation operator.

Binary Tournament Selection

Algorithm 2.2: Tournament Selection

```
Data: The population $P = \{\vec{x}^{(1)}, ..., \vec{x}^{(k)}\}$.
      The tournament size $s \in \{1, 2, ..., k\}$.
Result: The population after selection $P'$ with $|P'| = |P|$.
1 $P' \leftarrow \emptyset$
2 for $i \leftarrow$ to $k$ do
3     $G \leftarrow s$ randomly chosen individuals from $P$.
4     $g \leftarrow$ individual from $G$ with the best fitness.
5     $P' \leftarrow P' \cup \{g\}$
6 return $P'$
```

The selection process is used to control which individuals are allowed to reproduce. By selecting fit individuals, the information held by these individuals is allowed to scatter around in the population by means of reproduction and raise the average fitness of the population. Since EAs are a stochastic process, we do not want to give only the best individuals a shot at reproduction. We also want some of the lower quality individuals to reproduce, expecting that their information will ultimately lead to even better regions in the search space. This way selection can be used to balance exploitation and exploration. If the chance that low-quality individuals are allowed to reproduce is high, the population will spread around the whole search space and explore it. If only good individuals are allowed to reproduce newly generated individuals will mostly be generated around a smaller region near the better individuals and search for the optimum.

Tournament selection is a popular strategy for selection[4, 26]. The broad idea is to create a mating population from the EA's current population. This mating pool has the same size as the original population, but better individuals are in that population more than once. This is done by randomly choosing s individuals from the original population and adding the best of those individuals to the mating population. This process is called a "tournament," and s is the tournament size. The participant individuals are "competing" against each other, by comparing their fitness values, and the winner is allowed into the mating population. This method is shown in Algorithm 2.2. The participant individuals are selected randomly with replacement. The winner of a tournament is always selected deterministically using the domination criterion (see Equations 2.4 and 2.5). A higher s will decrease the chance of selecting unfit individuals. More candidates in tournament group mean a higher chance that a good individual is also present, which will dominate the inferior individuals.

Binary tournament selection is a special case of tournament selection where $s = 2$, so only two individuals compete for a place in the mating pool. Binary tournament selection is used in NSGA-II which we introduce in Section 2.3.3.

Simulated Binary Crossover

Simulated binary crossover (SBX) is a crossover algorithm usually used in cases where the individual is defined as a vector of real numbers, as is the case in this thesis. It was created with the goal to recreate useful properties of single-point binary crossover, which is used when the individuals are encoded in binary form[14, 15, 6]. After the introduction of EAs, real values were customarily encoded as binary code, and the single-point crossover method was used on the binary code. This type of crossover had some useful properties, which

Agrawal et al.[14] wanted to keep when real variables were used directly, i.e., without binary encoding. They looked at the distribution of decoded real variables after binary single-point crossover and sought to create an operator that could simulate the same distribution after the crossover, hence the name simulated binary crossover. The authors also introduced a parameter named distribution index η_c, that controls the spread of child solutions around the parent solutions.

Let us assume we have randomly selected, from our mating pool, two parent solutions $\vec{x}^{(1)}$ and $\vec{x}^{(2)}$ of the form

$$\vec{x}^{(1)} = \{x_1^{(1)}, x_2^{(1)}, ..., x_n^{(1)}\} \qquad\qquad \vec{x}^{(2)} = \{x_1^{(2)}, x_2^{(2)}, ..., x_n^{(2)}\}$$

where n is the dimension of our search space and each $x_j^{(i)} \in \mathbb{R}$. Now each pair $\{(x_j^{(1)}, x_j^{(2)}) | j = 1, ..., n\}$ have a chance to be crossed over according to the crossover probability p_c. If such a pair is selected for crossover SBX is applied to $(x_j^{(1)}, x_j^{(2)})$ to create two new elements $(z_j^{(1)}, z_j^{(2)})$ which replace their respective parent elements in $\vec{x}^{(1)}$ and $\vec{x}^{(2)}$.

The two offspring elements $(z_j^{(1)}, z_j^{(2)})$ are created by first generating a random value u between 0 and 1 and then applying Equation 2.7

$$\begin{aligned} z_j^{(1)} &= 0.5[(x_j^{(1)} + x_j^{(2)}) - \bar{\beta}|x_j^{(2)} - x_j^{(1)}|] \\ z_j^{(2)} &= 0.5[(x_j^{(1)} + x_j^{(2)}) + \bar{\beta}|x_j^{(2)} - x_j^{(1)}|] \end{aligned} \tag{2.7}$$

$$\bar{\beta} = \begin{cases} (\alpha u)^{\frac{1}{\eta_c+1}}, & \text{if}\, u \leq \frac{1}{\alpha} \\ \left(\frac{1}{2-(\alpha u)}\right)^{\frac{1}{\eta_c+1}}, & \text{otherwise} \end{cases}$$

$$\begin{aligned} & u \in [0,1] \\ & \alpha = 2 - \beta^{-(\eta_c+1)} \\ & \beta = 1 + \frac{2}{x_j^{(2)} - x_j^{(1)}} \min \left\{ (x_j^{(1)} - x_j^{\text{lower}}), (x_j^{\text{upper}} - x_j^{(2)}) \right\} \end{aligned}$$

where we assume that $x_j^{(1)} < x_j^{(2)}$. x_j^{lower} and x_j^{upper} are j-th elements from lower and upper bounds of the search variables, see equation 2.2. $\bar{\beta}$ is the spread factor and depends on the distribution parameter η_c. Low values of η_c will produce offspring far away from their parents and high values offspring near to their parents.

Polynomial Mutation

Like SBX polynomial mutation[15] was created for real-coded parameter vectors and uses some of the same principles as SBX. It creates a new solution in the vicinity of a parent solution using a probability distribution.

Assume we have a solution of the form

$$\vec{x} = \{x_1, x_2, ..., x_n\}$$

where n is the number of design variables. Each $x_i,\ i = 1, ..., n$ has a chance to be mutated according to the mutation probability p_m. If a x_i is chosen for mutation, an $u \in [0, 1]$ is chosen from an uniform distribution and the mutation result z_i is generated according to equation 2.8 and replaces x_i.

$$z_i = x_i + \bar{\delta} \cdot \Delta_{\max} \qquad i = 1, ..., n \tag{2.8}$$

$$\bar{\delta} = \begin{cases} [2u + (1-2u)(1-\delta)^{\eta_m+1}]^{\frac{1}{\eta_m+1}}\,, & \text{if}\, u \leq 0.5 \\ 1 - [2(1-u) + 2(u-0.5)(1-\delta)^{\eta_m+1}]^{\frac{1}{\eta_m+1}}\,, & \text{otherwise} \end{cases}$$

$$\delta = \frac{\min[(x_i - x_i^{\text{lower}}), (x_i^{\text{upper}} - x_i)]}{x_i^{\text{upper}} - x_i^{\text{lower}}}$$

Where $\Delta_{\max}$ is maximum allowed perturbance of the parent. x_j^{lower} and x_j^{higher} are j-th elements from lower and upper bounds of the search variables, see equation 2.2. η_m is the distribution parameter for mutation and takes any nonnegative value like SBX a higher distribution index creates solutions closer to the parent.

2.3.3 NSGA-II

In this section, we will detail NSGA-II[16], a widely used multi-objective evolutionary algorithm (MOEA). The authors, Deb et al., improved upon earlier MOEAs which were held back because of their computational complexity

Algorithm 2.3: Fast Nondominated Sorting

Data: The population P.

Result: The population sorted into multiple nondominated fronts. The nondominated front $\mathcal{F}_1$ contains all solutions that are not dominated by any solution in P. The front $\mathcal{F}_2$ contains all solutions not dominated by any solution in $P \setminus \mathcal{F}_1$ and so on.

```
foreach $\vec{x}^{(p)} \in P$ do
    $S_p = \emptyset$
    $n_p = 0$
    foreach $\vec{x}^{(q)} \in P$ do
        if $\vec{x}^{(p)} \prec \vec{x}^{(q)}$ then
            $S_p = S_p \cup \{\vec{x}^{(q)}\}$
        else if $\vec{x}^{(q)} \prec \vec{x}^{(p)}$ then
            $n_p = n_p + 1$
    if $n_p = 0$ then
        $\vec{x}^{(p)}_{rank} = 1$
        $\mathcal{F}_1 = \mathcal{F}_1 \cup \{\vec{x}^{(p)}\}$
$i = 1$
while $\mathcal{F}_i \neq \emptyset$ do
    $Q = \emptyset$
    foreach $\vec{x}^{(p)} \in \mathcal{F}_i$ do
        foreach $\vec{x}^{(q)} \in S_p$ do
            $n_q = n_q - 1$
            if $n_q = 0$ then
                $\vec{x}^{(q)}_{rank} = i + 1$
                $Q = Q \cup \{\vec{x}^{(q)}\}$
    $i = i + 1$
    $\mathcal{F}_i = Q$
return $\{\mathcal{F}_1, ..., \mathcal{F}_i\}$
```

$O(mk^3)$ (m is the number of objectives and k is the population size), their non-elitist approach and the necessity of having to tune a sharing parameter.

The naive nondominated sorting compares each solution to all other solutions to see if any of the other solutions dominate the solution. That approach to sorting takes $O(mk^2)$ comparisons. All nondominated solutions are removed from the population and moved into the first nondominated front $\mathcal{F}_1$. To determine the second nondominated front $\mathcal{F}_2$ the comparisons have to be repeated which again takes $O(mk^2)$ comparisons even though the k is now smaller. The worst case, in which there are k fronts with one solution each, takes $O(mk^3)$ comparisons. To increase the performance, Deb et al.[16] introduced fast nondominated sorting (see Algorithm 2.3).

Fast non-dominated computes a domination count n_p and a set of solutions S_p for each solution $\vec{x}^{(p)}$. The domination count n_p is the number of solutions which dominate the solution $\vec{x}^{(p)}$. The set of solutions S_p contains the solutions that the solution $\vec{x}^{(p)}$ dominates. Calculating n_p and S_p requires $O(mk^2)$ comparisons. After completing these computations, some solutions will have a domination count of $n_p = 0$, meaning no other solutions dominate them. These solutions are in the first nondominated front $\mathcal{F}_1$. Then for each member in $\mathcal{F}_1$, we iterate through its set of solutions S_p and reduce the domination count for each solution in the set by one. Some solutions that were only dominated by the solutions in $\mathcal{F}_1$ will now have a domination count $n_p = 0$. These solutions are also removed and added to the second nondominated front $\mathcal{F}_2$. This procedure is now repeated until all solutions belong to a nondominated front $\mathcal{F}_i$. As we can see, only one round of comparisons is needed, after that, the remaining solutions can be efficiently sorted using information that was already computed. For reference, the pseudocode for fast nondominated sorting is shown in Algorithm 2.3.

The second improvement NSGA-II introduced is related to diversity preservation. This new *crowding distance* does away with the cumbersome sharing parameter used before, and is also computationally less complex.

The *crowding distance* is a rough indicator telling us how crowded the area (in the objective space) around the objective vector of the solution is. The crowdedness is approximated by sorting the objective vectors $\vec{f}(\vec{x})$ in the current nondominated front $\mathcal{F}$ according to its objective value in the first objective $f_1(\vec{x})$, and then computing the normalized distance to its neighboring solutions. Since the current nondominated front $\mathcal{F}$ is now sorted, the neighbors

Algorithm 2.4: Crowding Distance Assignment

```
Data: $F = \{\vec{f}(\vec{x}^{(1)}), ..., \vec{f}(\vec{x}^{(l)})\}$ the objective vectors for all $l$ individuals in a
      nondominated front $\mathcal{F}$.
      $\vec{f}^{\min}$ and $\vec{f}^{\max}$ the maximum and minimum objective values for each
      objective.
Result: Crowding Distance Measure
1  $l \leftarrow |F|$
2  $D \leftarrow \emptyset$
3  for 1 to $l$ do
4      $D \leftarrow D \cup \{0\}$
5  foreach objective $m$ do
6      $F = sort(F, m)$
7      $D[1] = D[l] \leftarrow \infty$
8      for $i = 2$ to $(l - 1)$ do
9          $D[i] \leftarrow D[i] + F[i+1]_m - F[i-1]_m)/(\vec{f}^{\max}_m - \vec{f}^{\min}_m)$
10 return $D$
```

are merely the predecessor and successor in $\mathcal{F}$. The distance is normalized according to the maximum and minimum objective values in the current population. The distance is stored in $D = [d_1, ..., d_l]$ that keeps the distance values for each of the l solutions in $\mathcal{F}$. This is repeated for the objectives f_2 through f_m, and the computed distance values are added to D. The *crowding distance* algorithm is shown in Algorithm 2.4. $F[i]_m$ refers to the m-th objective function value of the i-th individual in the current nondominated front $\mathcal{F}$ and the parameters $f^{\max}_m$ and $f^{\min}_m$ are the maximum and minimum values of the m-th objective function.

A higher *crowding distance* value means there is more space between the objective vectors of the solutions in the same nondominated front $\mathcal{F}$. The proposed measure is used as a secondary criterion to guide the selection process, as shown in the crowded comparison operator $\prec_n$, see Definition 2.1. Initially, it prefers a solution with a better rank, unless they have the same rank then the solution in a lesser crowded area (bigger distance measure) is preferred.

The final improvement proposed by Deb et al.[15] is elitism, which according to the authors prevents the loss of good solution and speeds up EAs significantly.

Algorithm 2.5: The NSGA-II Algorithm

```
1  $t \leftarrow 0$
2  initialize $P_0$ randomly
3  evaluate $P_0$
4  while termination criterion not reached do
5      $t \leftarrow t + 1$
6      select $Q_t$ from $P_{t-1}$ using binary tournament selection (see Page 15)
7      perform crossover on $Q_t$ using SBX (see Page 6)
8      mutate the resulting solutions in $Q_t$ using polynomial mutation (see Page
        18)
9      evaluate $Q_t$
10     $R_t \leftarrow Q_t \cup P_{t-1}$ ($R_t$ has the size $2k$)
11     $\{\mathcal{F}_1, ...\} \leftarrow$ sort $R_t$ according to nondomination (see Algorithm 2.3)
12     $P_t \leftarrow \emptyset$ and $i \leftarrow 1$
13     while $|P_t| + |\mathcal{F}_i| \leq k$ do
14         crowding distance to $\mathcal{F}_i$
15         $P_t \leftarrow P_t \cup \mathcal{F}_i$
16         $i = i + 1$
17     Sort $\mathcal{F}_i$ using $\preceq_n$
18     Select enough of the best elements of $\mathcal{F}_i$ to fill up $P_t$
19 return $P_t$
```

Definition 2.1 (Crowded Comparison Operator).
Assume that every individual $\vec{x}^{(i)} \in P$ has two attributes:

1. *nondomination rank: $\vec{x}^{(i)}_{rank}$*
2. *crowding distance: $\vec{x}^{(i)}_{distance}$*

Then the Crowded Comparison Operator $\prec_n$ is defined as:

$$\vec{x}^{(i)} \prec_n \vec{x}^{(j)} \quad \textbf{\textit{if}}\ (\vec{x}^{(i)}_{rank} < \vec{x}^{(j)}_{rank})$$
$$\textbf{\textit{or}}\ ((\vec{x}^{(i)}_{rank} = \vec{x}^{(j)}_{rank})\ \textbf{\textit{and}}\ (\vec{x}^{(i)}_{distance} > \vec{x}^{(j)}_{distance}))$$

Algorithm 2.5 shows the complete procedure and how NSGA-II implements elitism. NSGA-II starts, like the general EA (see Algorithm 2.1), by creating a random population which is then evaluated (Lines 2 and 3). However, the main loop differs from the general EA. The result of selection, crossover, and mutation is stored as the offspring population Q_t. The offspring population is evaluated and merged into R_t together with the population of the previous generation, which always contains the best individuals found so far. The new population R_t thus contains the best-known individuals and new individuals, some of which could be better than the best individuals found so far. However, R_t is of the size $2k$, to reduce it while keeping the best individuals from R_t it is sorted into nondominated fronts $\{\mathcal{F}_1, ...\}$. The best fronts are now added to P_t until adding another front would make P_t too big (lines 12 trough 16). The last front that could not be added, is now sorted using the crowding distance operator $\preceq_n$ (Line 17), and the individuals in the least crowded areas are added until P_t is filled up (Line 18).

2.4 Surrogate-assisted MOEAs

EAs are a direct search methods that do not need any information about optimization problem, besides the objective function. However, because of this a typical EA usually needs large populations and many generations requiring a large number of evaluations. This means that the overall computation time of the EA run scales with the time a single evaluation needs. To demonstrate this, let us assume a single evaluation needs 1 minute to complete and to solve the problem we need 30.000 evaluations. The overall time needed would be roughly 20 days and 20 hours. A lot of real-world problems that need complex

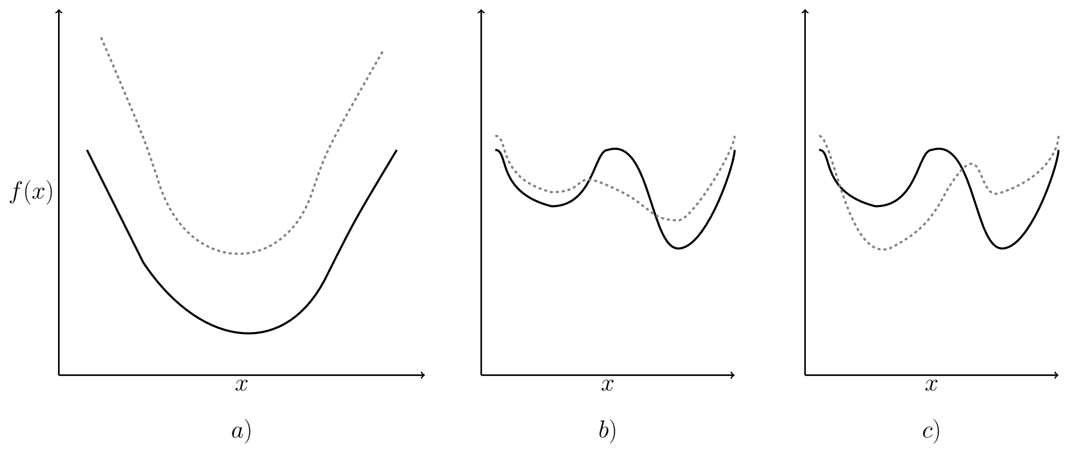


Figure 2.4: Examples for the quality of surrogate functions. The solid lines are the real objective function and the dashed line represent their approximations. Figures a) and b) show suitable approximations of their respective fitness landscapes. Figure c) shows an approximation that would mislead the optimizer.

FEM simulations to evaluate the solutions can need that much time and more. They are the so-called expensive MOPs. Expensive MOPs are a subclass of MOPs that are defined by the long time a single evaluation needs. In today's world, waiting that long for an EA to finish can be impracticable. Methods that can reduce the number of evaluations needed by an EA are needed. Surrogate functions are a class of methods that meet this need.

A surrogate function is a function that can be used instead of the real fitness functions. Such a function takes a solution $\vec{x} \in \Omega$ as input and returns an objective vector that approximates the real objective vector. For a surrogate function to be effective, the surrogate function should have the same global optimum and should not introduce false optima. For an example of a misleading approximation see 2.4c. If an optimizer would optimize the problem based on the surrogate (dashed line) he would find the global optimum in a place where the real problem has only a sub-optimal local optimum.

We said earlier that we use EAs because they work on problems for which no more information other than the objective function is provided. Therefore, we use statistical methods and ML to learn the fitness landscape based on a limited sample of N data points $(\vec{x}, f(\vec{x}))$ from the real fitness function. In our case, it is not possible to model the internal behavior of the problem,

The time taken to perform one evaluation is of the order of minutes or hours
Only one evaluation can be performed at one time (no parallelism is possible)
The total number of evaluations to be performed is limited by financial considerations
No realistic simulator or other method of approximating the full evaluation is readily available
Noise is low (repeated evaluations yield very similar results
The overall gains in quality (or reductions in cost) that can be achieved are high
The search landscape is multimodal but not highly rugged
The dimensionality of the search space is low-to-medium
The problem has multiple, possibly incompatible, objectives

Table 2.1: Common Features exhibited by problems suitable for surrogate-assisted MOEAs, according to Knowles and Hughes[36].

modeling only the relations between inputs and outputs has to suffice. Knowles and Hughes[36] recommended of surrogate-assisted MOEAs for problems that exhibit the features shown in Table 2.1.

We denoted the original fitness function as $f(\vec{x})$ and will denote the surrogate function as $f'(\vec{x})$. We omit the vector notation here, because some surrogate models only deal with single-objective functions. In those cases, multi-objective functions can be approximated by applying the surrogate construction for each objective separately.

A surrogate function is defined as

$$f'(\vec{x}) = f(\vec{x}) + e(\vec{x}) \tag{2.9}$$

with $e(\vec{x})$ being the approximation error. A high approximation error does not necessarily make a surrogate deficient, for example, see Figure 2.4a where the surrogate function (dashed line) has a high error. However, the error is almost constant over the whole objective space therefore not changing the location of the global optimum. Figure 2.4b shows an example where the error changes over the fitness landscape. Nonetheless, the global and local optimums do not change their location. Unlike the example that is shown in Figure 2.4c were both optima switch their positions. The overall error, however, is similar to Figure 2.4b and lower than Figure 2.4a, but any optimizer using the surrogate function shown in Figure 2.4c will not find the globally optimal solution.

2.4.1 Types of Surrogate Models

There are different kinds of surrogate models available.

Response Surface Methodology

Also known as polynomial models, response surface methodologys (RSMs) use methods from the field of statistics to approximate the objective function[5, 47]. Usually a second-order model of the following form is used:

$$f'(\vec{x}) = \beta^{(0)} + \sum_{1\leq i\leq N} \beta^{(i)}\vec{x}^{(i)} + \sum_{1\leq i\leq j\leq N} \beta^{(N-1+i+j)}\vec{x}^{(i)}\vec{x}^{(j)} \tag{2.10}$$

where N is the number of data points $(\vec{x}, f(\vec{x}))$. $\beta^{(0)}$ and $\beta^{(i)}$ are coefficients which can be estimated using the least squares method (LSM) or the gradient method. Either method requires the number of samples N to be equal or higher than the number of coefficients $N_t = (N+1)(N+2)/2$.

To compute the coefficients the following problem has to be solved using LSM[58]

$$f(\vec{x}) = Z\beta + e(\vec{x}) \tag{2.11}$$

where

$$f(x) = \begin{bmatrix} f^{(1)}(x) \\ f^{(2)}(x) \\ \vdots \\ f^{(N)}(x) \end{bmatrix}, \quad Z = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(2)} \\ x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)} \\ \vdots & \vdots & & \vdots \\ x_1^{(N)} & x_2^{(N)} & \cdots & x_n^{(N)} \end{bmatrix}, \quad \beta = \begin{bmatrix} \beta^{(1)} \\ \beta^{(2)} \\ \vdots \\ \beta^{(N)} \end{bmatrix}, \quad e(x) = \begin{bmatrix} e^{(1)}(x) \\ e^{(2)}(x) \\ \vdots \\ e^{(N)}(x) \end{bmatrix}$$

and where $f(x)$ are the examples of the real objective vector. $f(x)$ has the dimensions $n \times 1$, Z has the dimensions $N \times n$, β has the dimensions $n \times 1$, and e has the dimensions $N \times 1$.

Using polynomial expressions is equivalent to using a Taylor's series expansion of the true objective function $f(\vec{x})$. As such, a higher polynomial of higher degree can approximate $\vec{\vec{x}}$ better but also have more coefficients $\beta^{(i)}$ that need to be estimated. The second order polynomial in Equation 2.10 can approximate any kind of quadric surface and are found to be suitable to solve low-dimensional optimization problems [40, 25, 41, 33]. Arias-Montaño et al.[2] noted that the training cost is proportional to the size of the training set and that low order polynomials have only a limited capability to approximate highly multimodal fitness landscapes.

Kriging

The Kriging method was developed by a mining engineer named Krige who initially used it to predict the concentration of ores in South African mines[37, 38, 9]. The Kriging method has two components, a global model (usually a RSM) and a Gaussian random function that models the deviations from the global model.

Mathematically, it is described as

$$f'(\vec{x}) = g(\vec{x}) + Z(\vec{x}) \tag{2.12}$$

where $g(\vec{x})$ is the global model of the original fitness function and $Z(\vec{x})$ describes the localized deviations from $g(\vec{x})$. The global model $g(\vec{x})$ is usually a polynomial (see above) and $Z(\vec{x})$ is realized as a Gaussian random function with zero mean and non-zero covariance.

The covariance of $Z(\vec{x})$ is expressed as

$$\begin{aligned}\mathrm{Cov}[Z(\vec{x}^{(j)}), Z(\vec{x}^{(k)}] = \sigma^2\mathbf{R}[R(\vec{x}^{(j)}, \vec{x}^{(k)})],\\ j, k = 1, ..., N\end{aligned} \tag{2.13}$$

where R is the correlation function between any two of the N samples. The unknown correlation parameters are contained in $\mathbf{R}$, a symmetric correlation matrix of dimension $N \times N$. The maximum likelihood method is used to estimate the unknown correlation parameters. Choi et al.[8] note the ability of the Kriging method to approximate fitness landscapes with multiple local optima. Arias-Montaño et al.[2] count Kriging into the group of the most powerful interpolating methods. However, the construction of the model is very time intensive [35, 2, 19]

Radial Basis Function Networks

Radial basis function (RBF) networks are neural networks that use an radial basis function as the activation function and not more traditional choices like the sigmoid or hyperbolic tangent activation functions. As such, the output of the network is a linear combination of the inputs. The RBFN can be mathematically described as

$$f'(\vec{x}) = \sum_{j=1}^{N} w_j \phi(\|\vec{x} - \vec{x}^{(j)}\|) \tag{2.14}$$

where $\phi(\cdot)$ is the radial basis activation function, $\|\cdot\|$ is the Euclidean norm, the training samples $\vec{x}^{(j)}, j = 1, ..., N$ are used as the centers of the radial-basis activation functions and w_j are the unknown coefficients. This model is expensive to implement if the number of samples is large, but it can represent multimodal fitness landscapes well[2].

Artificial Neural Networks

Neural Networks can learn a smooth map that approximates the fitness landscape of a MOP and are therefore a suitable choice for a surrogate function. Feedforward Artificial Neural Networks are powerful learning machines that are discussed in more detail in section 2.5 on Page 30.

2.4.2 Integration of the Surrogate Model

There are different methods for the integration of surrogate models into a MOEA. Díaz-Manríquez et al.[18] distinguishes between methods with **direct fitness replacement** and **indirect fitness replacement**. Direct fitness replacement methods are further divided into three different Evolution control methods **no evolution control**, **fixed evolution control** and **adaptive evolution control**. The table 2.2 contains a short summary.

Direct Fitness Replacement

Direct fitness replacement is defined by the fact that the approximated fitness function takes over from the original fitness function. Only a small part of the population is evaluated using the original fitness function for control purposes. The evolution control methods are needed to check if the estimated fitness values are still on track or if the surrogate function needs to be updated. If the fitness function is off, the EA could move the population into false optima. Even with evolution control, there is no guaranteed that global optima are found.

- **No Evolution Control**
  No Evolution Control (NEC) is the case where the original fitness function is never used. NEC is used when it is known that the surrogate function is excellent and has no false optima or the original fitness function cannot be used.

- **Fixed Evolution Control**
  Fixed Evolution Control (FEC) means some generations or individuals are evaluated using the real function while the fitness of the rest is estimated. FEC allows for the surrogate function to be updated periodically with new data from real evaluations.

- **Adaptive Evolution Control**
  Adaptive Evolution Control (AEC) is similar in concept to FEC but the change between real or surrogate function is done automatically based on a heuristic.

Indirect Fitness Replacement

Indirect Fitness Replacement uses the original fitness function for the evaluation of the population. The estimated fitness values are used in other parts of the algorithm. Usually, this takes the form of some kind of preselection. If an operator, f.e., crossover, is used, more individuals than necessary are generated and then these individuals are compared with the help of the approximation function. The best individuals are then returned as the result of the operator. These operators are called **informed operators**.

- **Informed Mutation**
  Instead of just generating one random individual by mutation, multiple individuals are generated. Their fitness is estimated, and the best individual is returned.

- **Informed Crossover**
  Again instead of just generating one child by crossover, multiple children

Class	Advantages	Disadvantages
Direct Fitness Replacement		
No Evolution Control	(i),Computationally efficient (ii) Good behavior in low-dimensional problems	(i) requires an accurate surrogate model (ii) It can converge to a false optimum
Fixed Evolution Control	(i) It is capable of solving high-dimensional problems (ii) The surrogate model adapts during the optimization process	(i) It is necessary to define the parameter to alternate betweeen the surrogate modell and the real objective function
Adaptive Evolution Control	(i) Does not require defining the parameter to alternate between the surrogate model and the real objective function	(i) the automatic alternation is not easy to define
Indirect Fitness Replacement	(i) Usually uses a local search phase to optimize the surrogate model (ii) The metamodel is used for exploitation purposes (iii) avoid convergence to a false optimum	(i) it is the most computationally expensive

Table 2.2: Trade-offs of different integration methods according to [18].

are generated with different crossover methods and parameters. The children are then compared and the, according to its estimated fitness, is returned.

2.5 Artificial Neural Networks

Artificial neural networks (ANNs)[46, 39] are information processing systems commonly used as universal function approximators[11, 30, 10]. The way they process information is inspired by the operating principles of mammalian brains.

In mammalian brains, there exists a type of cell called a neuron that collects and transmits electrical signals. A typical neuron consists of a cell body with a nucleus and is connected to other neurons by dendrites and axons. Specifically, the axons are the outputs of the neurons that lead to the dendrites (inputs) of other neurons. At the end of an axon, there are terminal buttons that are positioned very close to a dendrite of another neuron. Such a connection point, where the terminal button meets the dendrite, is called a synapse. Using chemicals called neurotransmitters, a terminal button can decrease or increase the potential difference of the target dendrite, these signals are called excitatory or inhibitory signals, respectively. As a metaphor, excitatory effects can be counted as positive and inhibitory effects as negative, and all incoming signals can be accumulated. If a neuron accumulates enough positive signals, a sudden reaction occurs and the neuron transmits an electric signal along its

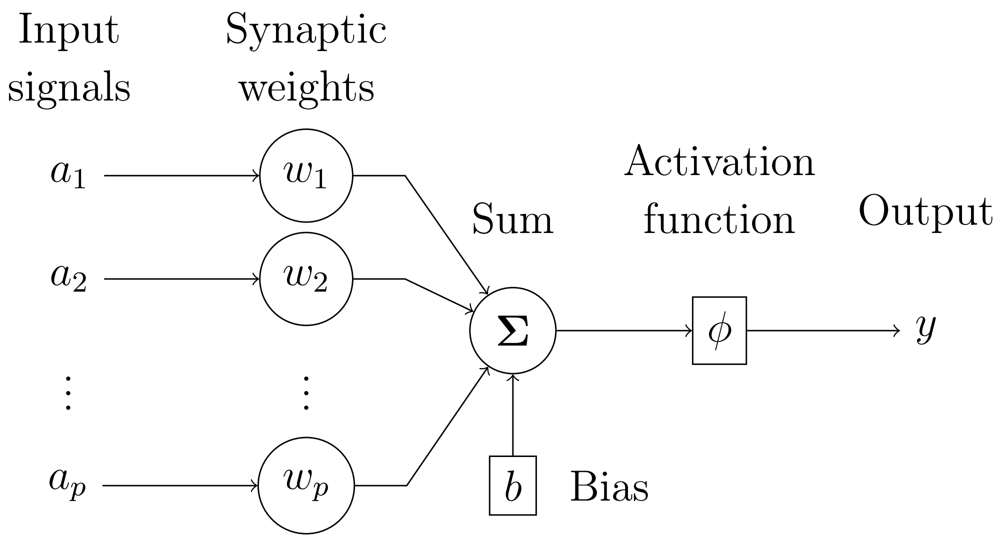


Figure 2.5: Model of an artificial neuron.

axons to the next neuron. For more in-depth information on this topic, the reader is referred to Anderson[1].

For use in computer science, these biological structures were mapped onto simplified mathematical models with similar functions. This way an ANN can simulate some of the workings of the brain. The first steps to recreate biological brains were taken in the 1940's where basic models like the McCulloch-Pitts Neuron[45] and fundamental theory[28] were researched. In 1958, the perceptron [51] was created, which was the first model (a physical machine actually) which was based on neurological models and was able to adapt its weights. They fell out of favor soon after, and interest renewed only after the backpropagation algorithm became popular in the 1980's[59, 52]. The backpropagation algorithm could efficiently train networks with more than one hidden layer. This caused a second wave of interest in ANNs until other ML algorithms overtook them in popularity. The third wave, which is still ongoing, started in the mid-2000's and with it, the term "deep learning" was introduced. This term emphasizes the focus of the related research on the benefits of using multiple hidden layers, see Section 2.6.

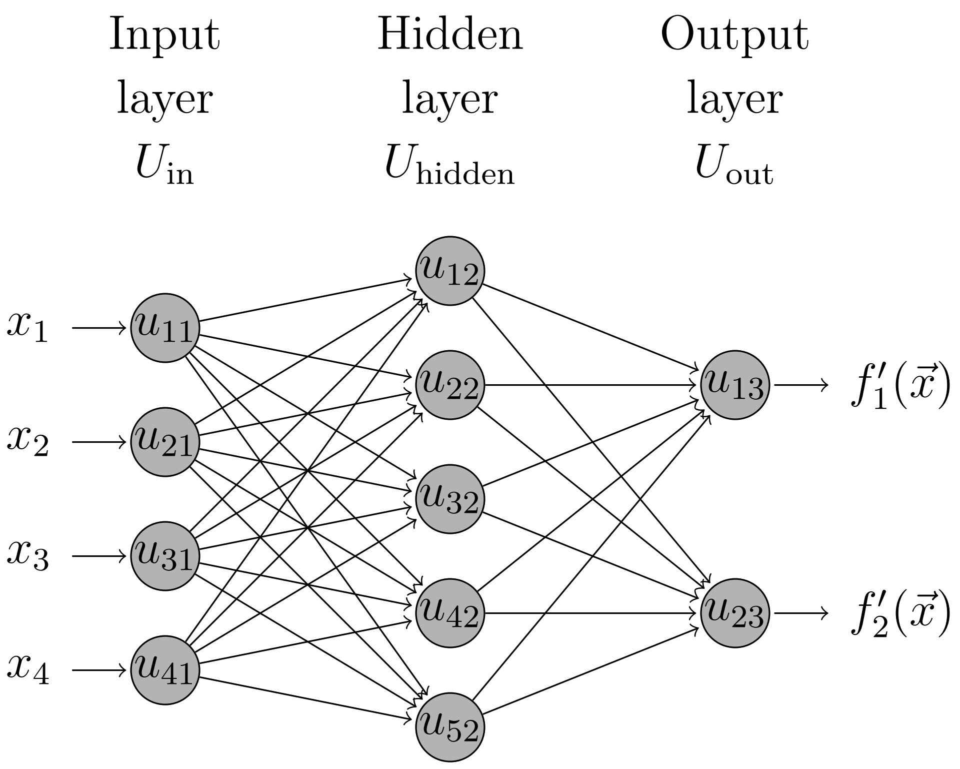


Figure 2.6: Example of a neural network. This network could learn to approximate any optimization problems which has $n = 4$ input variables and $m = 2$ objectives.

Definition 2.2 (Graph).
A directed graph is a pair $G = (V, E)$ consisting of a finite set V of vertices and a finite set $E \subseteq V \times V$ of edges. An edge $e = (u, v) \in E$ is called directed if the connection only goes from vertex u to the vertex v but not from vertex v to vertex u.

2.5.1 Neural Networks in Computer Science

Like in their biological counterparts, the smallest building block of an ANN is the so-called artificial neuron. A neuron processes received signals, producing an output, and then sends the output to other neurons connected to it. Figure 2.5 shows a schematic of the function of an artificial neuron u. The input signals $a_i^{(u)}$ are signals coming from other artificial neurons. Each of the incoming connections has a weight $w_i^{(u)}$ associated with them. The weight is multiplied with the incoming signal, i.e., $a_i^{(u)} \cdot w_i^{(u)}$, for all $i = 1, .., p^{(u)}$. The products $a_i^{(u)} \cdot w_i^{(u)}$ are then summed up and and a bias $b^{(u)}$ is added. The result is then subjected to an non-linear mapping by the activation function $\phi^{(u)}(\cdot)$, which results in the output $y^{(u)}$. Mathematically, the function of a neuron can be described as

$$y^{(u)} = \phi^{(u)}(\vec{a}^{(u)} \cdot \vec{w}^{(u)} + b^{(u)}) \tag{2.15}$$

A single neuron, however, is not very useful. Multiple neurons have to be connected to approximate our objective functions. Graphs are used to formally describe a collection of connected artificial neurons, see Definition 2.2. That makes the neurons vertices and the connections edges in a directed acyclic graph. An example for a neural network graph is shown in Figure 2.6. This ANN could be used to approximate an objective function with $n = 4$ search variables and $m = 2$ objective functions. What we can also see is that all neurons u_{ij} are grouped into three layers. The input layer U_{in} contains all u_{i1} neurons that receive the elements of the solution vector $\vec{x}$ as inputs. The hidden layer U_{hidden} contains all neurons u_{i2} which do some computations on the signals received from the input layers. These computations are defined during the learning process (see next section). The existence of at least one hidden layer allows the network to do non-linear transformations on the input values, as opposed to a network with no hidden layers which can do only linear transformations. The neurons u_{i3} in the output layer U_{out} also do learning-defined calculations and produce the final output values that approximate the objective vectors. Further, all neurons in U_{hidden} and U_{out} are connected, by the incoming edges, to all neurons in the previous layers, U_{in} and U_{hidden} respectively.

An approximated objective vector $\vec{f}'(\vec{x})$ for our optimization problem can be computed as follows:
Each artificial neuron $u_{i1} \in U_{\text{in}}$ in the input layer receives the corresponding element of the solution vector x_i and simply sets it as its output. Then all neurons in the hidden layer $u_{i2} \in U_{\text{hidden}}$ collect the outputs of the input neurons as input signals $\vec{a}^{(u_{i2})}$ and compute their output $y^{(u_{i2})}$ as shown in Equation 2.15. This process is repeated for the output neurons, all $\vec{a}^{(u_{i3})}$ are collected from the outputs of the hidden neurons, and the outputs $y^{(u_{i3})}$ are computed. These final outputs can now be used as objective vector $\vec{f}'(\vec{x}) = [f_1'(\vec{x}), ..., f_m'(\vec{x})] = [y^{(u_{13})}, ..., y^{(u_{m3})}]$. Mathematically, this can be described as:

$$f_o'(\vec{x}) = \phi^{(u_{o3})}\left(\sum_{h=1}^{|U_{\text{hidden}}|} w^{(u_{h2},u_{i3})}\phi^{(u_{h2})}\left(\sum_{i=1}^{|U_{\text{in}}|} w^{(u_{i1},u_{h2})}x_i + b^{(u_{h2})}\right) + b^{(u_{o3})}\right)$$
$$o = 1, ..., m \tag{2.16}$$

For the neural network to compute our objective function correctly, the weights and biases have to be set correctly. The process of computing the correct weights and biases is commonly known as learning.

2.5.2 Learning in Neural Networks

Learning the weights and biases might sound complicated at first, but is actually just an optimization problem similar to those described in Section 2.1. One property of those problems was that only the objective function is known and it was too expensive to compute the gradient. This was the reason why EAs are used to optimize these problems. The optimization problem posed by ANNs, however, allows us to compute a gradient. Therefore, in ANN learning gradient descent is the most commonly applied optimization algorithm.

The objective we are trying to minimize is now called the cost function $C(\vec{w})$, and the solution vector is now called the weight vector $\vec{w}$. For simplicity, and because biases can be modeled as weights, the weight vector contains the weights and biases. This leads us to the following optimization problem

$$\min_{\vec{w}} C(\vec{w}) \qquad \text{where: } C(\vec{w}) = \frac{1}{2N} \sum_{i=1}^{N} (\vec{f'}(\vec{x}^{(i)}) - \vec{f}(\vec{x}^{(i)}))^2 \tag{2.17}$$

where N is the number of samples in a training set. The training set contains pairs $(\vec{x}, \vec{f}(\vec{x}))$ of solutions and their respective objective vectors, which were evaluated with the real objective function. The cost function, in this example we use the mean squared error (MSE), computes the average of the distances (the errors) between the real objective vectors and the approximated. This average will be high if the weights are set poorly and the approximated objective vectors a far from the real ones. If the average is close to zero, the weights are set correctly. To use gradient descent, we have to compute the gradient of the cost function $\nabla C(\vec{w})$. In the case of neural networks, this is done using the backpropagation algorithm.

Backpropagation

The Backpropagation algorithm is a method for computing the partial derivatives of the cost function $C(\cdot)$ with respect to any of the adaptable parameters (weights and biases) of an ANN. First, the gradients for each example in the training set are computed and then averaged to recover gradient for the whole training set. To compute the gradient for a single example, the training example is presented to the neural network as input, which then computes an approximated output $\vec{f'}(\vec{x})$ (see Equation 2.16). For the purpose of backpropagation, we are interested in the difference between the approximated and real objective vectors $e_o = f'_o(\vec{x}) - f_o(\vec{x})$. This error tells us how we have to change the output value of the output neuron $u^{(o)}$ to achieve the desired output value. We also calculate how the outputs of the neurons in the hidden layer have to change to achieve the required change in $u^{(o)}$. This is called error backpropagation. The backpropagated error scales with influence the hidden neuron $u^{(h)}$ has on the output value of $u^{(o)}$. If the influence of $u^{(h)}$ is high, its contribution to the error e_o of the output neuron $u^{(o)}$ is also high. Therefore it needs to change more than a hidden neuron that has little influence on $u^{(o)}$. If there are more hidden layers, the error backpropagation step has to be repeated. After the backpropagation is completed, every neuron has an associated error value depending on the influence it has on the output value of $u^{(o)}$. From these errors, the required changes to the biases and weights can be derived. If there are multiple output neurons, this process has to be repeated for each output neuron, and the changes must be averaged. Now, we have the gradient for a single example, and we have to repeat this for all examples and each of their output neurons and average all changes to recover the gradient $\nabla C(\vec{w})$. This gradient is then used by an optimizer like Stochastic Gradient Descent (SGD) to optimize the weight values and decrease the cost function $C(\vec{w})$.

Stochastic Gradient Descent

Gradient descent is an optimizer that can quickly find (local) optima by using gradient information. To minimize the function $C(\vec{w})$, we compute the gradient $\nabla C(\vec{w})$ of $C(\vec{w})$. The gradient $\nabla C(\vec{w})$ shows us how we have to change $\vec{w}$ to get the largest increase in $C(\vec{w})$. If we compute a new $\vec{w}^{\text{updated}} = \vec{w} + \nabla C(\vec{w})$, the cost function will most likely be increased $C(\vec{w}^{\text{updated}}) \geq C(\vec{w})$. Since we want to minimize the cost function $C(\vec{w})$, we have to follow the negative of the gradient $-\nabla C(\vec{w})$

$$\vec{w}^{\text{updated}} = \vec{w} - \eta \nabla C(\vec{w}) \tag{2.18}$$

where η is the learning rate. Intuitively, the learning rate scale the size of the steps we the take into the direction of $-\nabla C(\vec{w})$. The learning rate should be small, i.e., we take only small steps because $-\nabla C(\vec{w})$ shows us the correct direction only in a very limited neighborhood of $\vec{w}$. If we repeatedly update $\vec{w}$, we will theoretically reach a point with a zero gradient, a minimum. The point which has the lowest possible value for $C(\vec{w})$ is the global minimum, but gradient descent does not guarantee that the global minimum is found, just that a minimum is found.

One important variant of gradient descent is Stochastic Gradient Descent (SGD). It is motivated by the observation that for a model with good generalization a large training set is needed but that has the drawback of large computational cost. To circumvent this drawback, SGD estimates the true gradient with fewer examples. Fewer examples can be used because the gradient of the cost function is an average over the gradients of all N training examples. An average, or in statistical terminology: an expectation, can be approximated by a smaller sample. Therefore the whole set of examples can be divided into so-called minibatches

$$\mathbb{B} = \{(\vec{x}, \vec{f}(\vec{x}))^{(1)}, ..., (\vec{x}, \vec{f}(\vec{x}))^{(N')}\} \qquad \text{,where } N' \ll N$$

drawn uniformly from the training set. The size of these mini batches usually ranges from one to a few hundred.

2.6 Deep Learning

Why deep learning (DL) has incited so much research interest in the last years, is related to the representation of data. For a computer, the best possible representation is to describe the world as logical symbols and all actions it can perform as formal rules operating on those symbols. Chess can be described this way, the chess pieces and their positions are symbols, and the moves are formally defined rules. If a computer is provided with the positions of the pieces, the movement rules, and enough computing power it can beat a human by searching all possible moves and selecting the best ones. A second example is the recommendation of a cesarean. The doctor can give a computer a set of features the patient exhibits and the computer can accurately predict the need for a cesarean. The limitation of this symbolic approach is its reliance on the fact that a human preprocesses the data and extracts the symbols that the computer can efficiently operate on. If we show a computer an image of a chessboard or an ultrasound image, the computer will not be able to compute its next move or if a cesarean is a good recommendation. A matrix of values between zero and one is an inadequate representation to base an algorithm on, that computes the next move in a chess game.

An algorithm that could compute the next move in a chess game from an arbitrary image would have to do the following steps. First, find all basic edges in the image. Second, use related edges to detect corners and contours. Then combine the corners and contours to detect basic object parts of the chess pieces. That means crowns, bishop hats, horse heads, battlements and so on. Then combine the basic object parts to detect what these shapes represent, i.e., kings, bishops, knights, rooks and so on. They also have to be detected robustly, meaning they have to be detected from different viewing angles, while being partially obstructed by other chess pieces, or under differing lighting conditions. Then compute the positions of the pieces relative to the chessboard, which has to be detected by the algorithm too. Only now do we have a representation that we can give to the symbolic algorithm to solve for the next chess move. What this algorithm does is to move through a hierarchy of representations. The algorithm starts at a representation with low symbolic meaning, the matrix of brightness values. Then based on the previous representation it computes one with more symbolic meaning. First edges, then contours, then object parts, then objects with labels that tell us which chess piece they are, and finally the position of each piece.

The basic idea of deep learning is to learn such a hierarchical algorithm with the help of deep networks. A deep network is an ANN with more than one hidden layer. Theoretically, an ANN with one hidden layer can learn any reasonable function if the hidden layer has enough neurons [11, 30, 10]. A deep ANN, however, can discover the hierarchy of representations during its training. This discovery causes the deep ANN to learn and generalize better than shallow networks. Eldan and Shamir[20] showed that more depth adds exponentially more expressive power than width to a neural network. Montufar et al.[48] showed that functions representable with a deep network using the rectified linear unit (ReLU) activation function could require an exponential number of hidden units with a shallow (one hidden layer) network. This expressive power makes deep ANN interesting for the approximation of difficult optimization problems.

The discovery of the representation of data and the mapping from the representation to the outputs is called representation learning. Computer-learned representations are usually created faster and are of higher quality. That is because the machine learning process is better at sifting through the vast amounts of available data to properly identify underlying factors that influence the observed data, even if those underlying factors are not observed themselves. Going back to the chess example, an unobserved factor that influences observable quantities is the light source (a lamp or the sun). The imaging device does not directly record the position of the light source and its light color but, they change the image by casting different shadows or changing the ambient light. In real-world problems many underlying factors influence the observed quantities, for example, most colors appear black at night, and the shape of most objects changes drastically with the angle it is viewed at. This makes it difficult to disentangle the different factors and concentrate on the most important ones. DL solves this by using the hierarchy approach. Each hidden layer is a representation in the hierarchy and uses previous layers (representations) to express a higher-level concept. The input layer is the image-level representation, and the next layers would compute the edges. The layers after that would use the output of the edge-representing layers to compute the corners and contours, and so on.

One related feature of deep learning which applies to surrogate functions is its performance on high-dimensional problems. The curse of dimensionality states that if the number of relevant dimensions increases the number of regions of interest grows exponentially. For each region of interest, a training example must be provided. At high dimensions, this curse leads to the existence of many more interesting regions than training examples. However, if the deep network can extract the correct underlying factors, it can better generalize to the unseen regions even without training examples. For example, if the fitness landscape of the optimization problem is periodic and the deep ANN learns this periodicity, it can correctly predict the objective vectors at each period without having seen a training example at most of the periods.

3 Related Work

This chapter presents other research in the field of surrogate-assisted MOEAs. It describes the which surrogate methods were used, how the surrogate models were integrated, and the results.

Choi et al.[8] used a procedure that only used a surrogate function (No Evolution Control (NEC)) based on Kriging approximations and used NSGA-II as an optimizer. The goal was to design a supersonic business jet. The conflicting objectives were the aerodynamic performance and the sonic boom loudness. Both of these objectives are computationally expensive to evaluate there the NEC approach was used.

Lian and Liou[40] wanted to redesign the NASA rotor67 compressor blade. They decided to maximize the stage pressure ratio p_{02}/p_{01} and minimize the blade weight W subject to an aerodynamic constraint related to the mass flow rate. They used 32 design variables that described some perturbation of the original compressor blade design. To optimize a response surface model was constructed and a genetic algorithm applied to it. The RS was not updated (NEC). After the convergence of GA slowed down a local search was used for final optimization. The representative solution where then verified against the CFD code (original fitness function).

Goel et al.[25] uses the response surface approximations for each objective separately and uses one surrogate for whole optimization (NEC) since the approximations turned out to be very accurate. He applied a modified NSGA-II to a liquid-rocket single element injector optimization. The conflicting objectives consisted of one performance related and two temperature(lifetime) related objectives. The problem was relatively low-dimensional with four design variables.

Liao et al.[41] presented an optimization procedure for vehicle design. The three objectives considered were the weight, acceleration characteristics and toe-board intrusion. Five design variables were used. The surrogate function used was the response surface method. The optimizer used was NSGA-II.

Husain and Kim[33] optimized a microchannel heat sink (MCHS). The authors considered three design variables and two objective functions (thermal resistance and pumping power). Polynomial regression, Kriging and Radial Basis Functions were compared as surrogate models and Kriging was found to be the most accurate (NEC). Global optimization was done with NSGA-II. The solutions found NSGA-II were improved with a local search.

Nain and Deb[49] used the technique of successive refinement of approximations. It uses Fixed Evolution Control (FEC) where the real function is used for a number of generation and then the surrogate function is used which was created from the examples of the previous real evaluations. The first iteration creates a coarse approximation to find the general location of optimum since the training data is dispersed over the whole problem domain. In the following iterations it always uses the newest training data and because the EA converges the to global optimum smaller and smaller regions are covered by the population. Therefore approximations also cover smaller and smaller areas and become more accurate. ANNs are used as the surrogate model.

D'Angelo and Minisci[12] optimized subsonic airfoils for minimizing drag and maximizing the lift. Five design variables were used that parametrized the bezier curves that described the shape of the airfoil. The optimizer used was MOPED that uses the Parzen method to build a probabilistic model of promising search space portions instead of crossover and mutation. The Kriging method was employed as the surrogate model. The authors used a controlled individuals approach (FEC) wherein every generation after ranking n_v random solutions from the best half of the population are chosen. If the distance a chosen solution to any solution already in the database is bigger than $dist_{min}$ the solution is added, and the surrogate model is updated.

Isaacs et al.[34] presented in their paper an evolutionary algorithm with spatially distributed surrogates (EASDS). This approach uses multiple spatially dispersed surrogates that rebuild periodically. The build surrogates are not used blindly but are assessed by a validation set. If a new point needs to be evaluated the surrogate which has the best accuracy in the neighborhood of the new point is used (FEC). Radial Basis Function Networks were used as the surrogate model. EASDS is based on NSGA-II. The authors report that

EASDS performs better than non-surrogate NSGA-II or single global surrogate models indicating a benefit from using multiple local surrogates.

Todoroki and Sekishiro[56] looked at the optimization of Hat-Stiffened Composite Panel with a Buckling Constraint. The two objective function where the weight reduction, which is easy to compute and therefore not approximated, and the uncertainty of satisfaction of the buckling constraint and had seven design variables. The second objective Kriging approximation was used to reduce the computational cost. A basic multi-objective genetic algorithm (MOGA) was used as the optimizer (FEC). With only 301 evaluations of the real function, this method achieved a result that had an error 3% from the true optimal structure.

Liu et al.[42] use a trust region approach. The trust region is defined in the design space which is moved according to rules explained in the paper. Every time the trust region is updated the surrogate function is relearned with examples sampled by LHS. Then a micro multi-objective genetic algorithm (μMOGA) is applied, which uses tiny populations, to find Pareto-optimal solutions (FEC). These solutions are compared to solutions in an external archive after which a new trust region is computed. According to the authors, this approach is less sensitive to the accuracy of the surrogate model. The authors reported savings of up to 90% of real function evaluations however the tested problems had only 2 or 3 design variables.

Fonseca et al.[23] use the nearest neighbors method as a lazy learner surrogate model. All real evaluations are saved in a database and if a surrogate evaluation is requested the method computes the nearest neighbors and determines the value of the solution as similarity-based interpolation between the nearest neighbors. They use the surrogate function for pre-selection after which the best solutions are evaluated with the real function, according to the parameter p_{sm} which represents the percentage of real function evaluations. The database is updated by adding all real evaluated solutions. If the maximum size of the database is reached the oldest solutions are discarded. Regarding Multi-objective problems, the authors noted that with fine-tuning of the parameter p_{sm} their approach had better results than a standard MOGA. Smaller values p_{sm} allow for the final solutions to be closer to the real Pareto-front, and the varying of the number of neighbors used in the surrogate evaluations does not significantly impact the performance. The used test problems had up to 30 design variables.

Martínez and Coello [43] used cooperating RBF networks as surrogate function. They use three RBF networks trained with different kernel types. To estimate one solution the three estimates from each RBF network are combined using weights that are computed based on how good the respective RBF network is. In every iteration, the three RBF networks are learned. Then the whole population is evaluated using the surrogate model (FEC), and MOEA/D is applied to that population to find an approximation of the Pareto-front. After that, some solutions are selected to be evaluated with the real function and are used to update the external archive and the training set. The authors showed that their approach compares favorably to the original MOEA/D on bi-objective ZDT test problems using 10 or 30 variables. The one exception was the multi-modal ZDT4 problem. On an airfoil optimization problem, MOEA/D-RBF only needed 25% of the evaluations of regular MOEA/D.

Stander [55] adapts the Sequential Response Surface Method to multiple objective problems(SRSM). In the single objective case, the SRSM method reduces a region of interest in the design space repeatedly towards the optimum. The direction towards the optimum is found by sampling the current ROI and building a surrogate model with the sampled and all known previous points. Then NSGA-II is applied on the surrogate model to find an approximate optimum (FEC). In the next iteration, the ROI is adapted (centered on the predicted optimum and shrunk), and the process is repeated. Since in the multi-objective case there is no single optimum to converge to, the author proposes to use multiple smaller ROIs that are equally distributed along the approximated optimal front. This results in the sampled points to be distributed along the approximated Pareto-optimal set, and the surrogate model is updated in the region of the Pareto-optimal front. For a 7-variable problem, the author reported that only 4% of the evaluations are needed, and for a 30-variable problem only 15% of the evaluations are needed.

Sreekanth and Datta[53] optimized pumping strategies for coastal aquifers. They consider two objectives the overall extracted fresh water and the reduction of pumping from so-called barrier wells which are used to reduce the salinity intrusion. The surrogate model is not used to approximate the objective functions directly instead it is used to predict the salinity levels in the specified monitoring locations, as a result of the groundwater extraction from the aquifer. The prediction of the salinity levels is an important part of the objective functions and still depends on the design variables which are seven pumping input values. Their optimization approach is to apply an NSGA-II on the current surrogate model which creates an approximation of the optimal solutions (FEC). Then the accuracy of the salinity prediction in this search region is validated using simulations. Logically if the salinity prediction is off the found optimal pumping strategies are also off. Therefore if the accuracy of the prediction is too low new solutions are sampled from the near optimal region and the surrogate model is retrained otherwise the optimization stops. The new training samples are generated from a hypercube around the approximate Pareto-optimal solutions and depend on the relative importance of the search variables in salinity prediction.

Rosalez-Pérez et al.[50] proposed an approach based on an ensemble of Support Vector Machines. Some features of their approach are the model selection process for finding suitable hyperparameters, incremental construction of the ensemble, that incorporates new knowledge gained during the optimization and preserves old knowledge, and a criterion for the fidelity of the surrogate. The general algorithm uses the surrogate function during all generations and only evaluates the non-dominated solutions with the real fitness function (FEC). These solutions are then used to update the surrogate model, and an external archive for the best solutions found so far. The ensemble of SVMs contains only one model, in the beginning, the one trained with initial data, whenever the suggestions of the surrogate model are worse than a random suggestion a new SVM model is trained and added to the ensemble. If the ensemble is full, the oldest model is removed. The training of a new model contains a model selection step where for each SVM Kernel type a grid search is performed to find the model with the least expected generalization error, which is determined by k-fold cross-validation, is chosen as the newest model. The proposed approach generated better results on the ZDT test suite than a standard NSGA-II. On ZDT1, ZDT2, ZDT3, and ZDT6 it approached the global Pareto-front with fewer than 300 real evaluations while standard NSGA-II did not reach the global Pareto-front with 3000 evaluations. However, it did not do well on the ZDT4 problem.

Chafekar et al.[7] based their approach on the genetic algorithm for design optimization (GADO) which was created for the engineering domain by using new operators and search strategies. The new method coined objective exchange genetic algorithm for design optimization (OEGADO) runs a single objective GA for each objective with independent populations. Every single objective GA builds a surrogate model of its own objective, but the models are being exchanged during the optimization, so the acpga get biased towards the optimal regions of the other objectives in addition to its own. The surrogate models are not used directly but as part of informed operators with the best result of the informed operator being evaluated with the real objective function. The authors compared OEGADO to NSGA-II ϵ-MOEA. For simple problems the results were comparable, but for complicated problems, OEGADO was better most of the time.

Emmerich et al.[21] proposed a search method based on evolutionary algorithms (EA) assisted by local Gaussian random field metamodels (GRFM). The authors based their approach on an evolutionary strategy that uses a pre-screening procedure. New solutions are first evaluated in the meta-model. A promising subset of these solutions is chosen to be evaluated in the real function. From the subset and the parent solutions, the next generation of parents is selection via ranking. For the pre-screening procedure, the authors used measures derived from the kriging model, like the expected improvement and the probability of improvement. This approach outperformed NSGA-II on bi-objective and three-objective test problems with ten variables.

Knowles[35] presented the ParEGO algorithm which extends the EGO algorithm of Jones et al. for the multiobjective case. It is based on Kriging DACE. The EGO algorithm generates its initial samples with the latin hypercube approach to build the maximum-likelihood DACE model. Then it searches this model for the point with the best function value and regions where the function value is uncertain. If a point in an uncertain region is found that could potentially improve the already found best function value this point is evaluated with the real function and added the Kriging DACE model. To extend EGO to the multiobjective case the author used parameterized scalarizing weight vector to convert the all the objective values into one scalar. Since using only one weight vector would only find one point on the Pareto-optimal front the weight vector is chosen randomly at each iteration, ensuring that multiple points on the Pareto-optimal front are found. ParEGO outperformed NSGA-II on nine low-dimensional but difficult test functions.

To our knowledge, there exists no research that investigates how deep networks affect the performance of surrogate-assisted multi-objective evolutionary algorithms. If neural networks were used, the approaches usually adopted only one hidden layer. Additionally, we can also see that there exists a multitude of ways a surrogate model can be integrated into an evolutionary algorithm. After looking at different methods, we decided that the proposed algorithm of

Nain and Deb[49] is the best-suited approach to answer the research questions, since it is easy to implement and extend.

4 Concept

There are multiple algorithms which use shallow neural networks. We use the approach of [49] and extend it to use deep networks. Therefore this approach is described in in more detail.

4.1 Basic Coarse to Fine Approximation

The idea for the approach stemmed from the observation that in the first generations of the EA run a fine-grained approximation of the search space is not needed. The EA only needs to find the search direction towards the optimum, the exact position of the optimum or details of the search space do not need to be modeled. Only later in the optimization do the models need to be more accurate.

To exploit that Nain and Deb[49] use some inherent properties of the EA. The EA is initialized with solutions that are randomly distributed over the whole search space. So if the initial solutions and those of the following generations are used as the training set for a surrogate model, a rough model of the fitness landscape will be created. However, if we wait for the EA to reach the neighborhood of the global optimum the population will not be dispersed across the whole search space but highly concentrated around the optimum and if used as training data will produce a surrogate model that captures most details around the optimum. This is visualized in Figure 4.1 where we can see an example of a fitness function and two examples of how a surrogate model could have learned the fitness landscape with different training sets. **Model 1** is created after the first few generations and is good enough to guide the EA towards the global optimum. **Model N**, however, captures the details around the global optimum because the sample points are focused around that region.

The same logic applies to intermediary generations. The approximations become better but cover an increasingly smaller area around the optimum. If

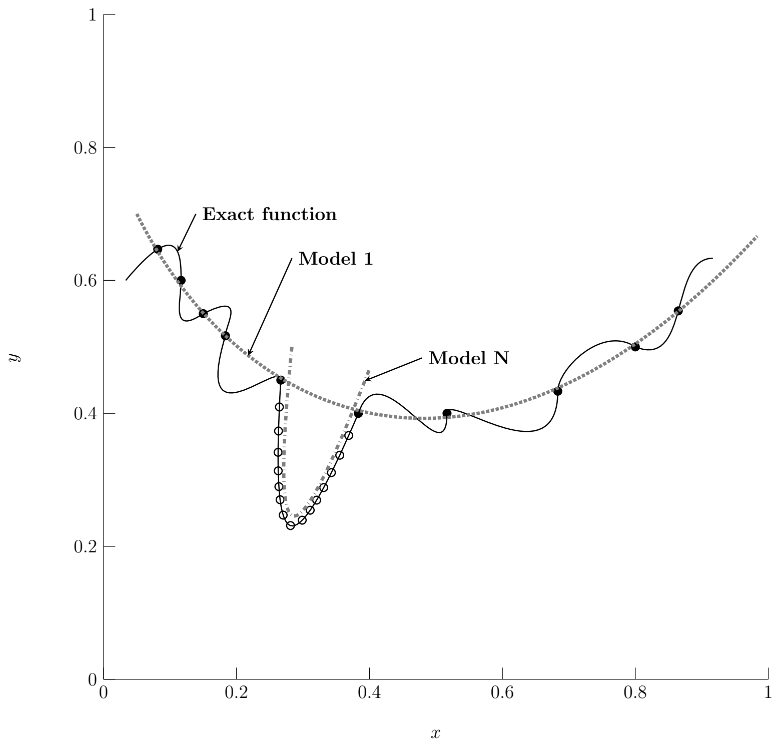


Figure 4.1: The concept of successive approximation. The first model uses the ● sample points to create **Model 1**, which approximates the rough shape of the fitness landscape. The last model uses the ○ sample points to create **Model N**, which accurately approximates the global optimum. According to [49].

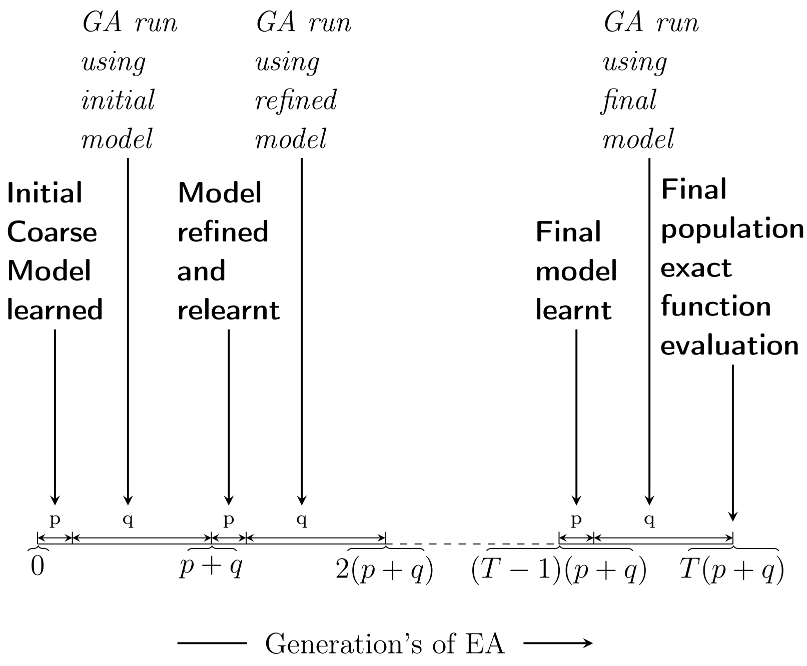

GA run using initial model
GA run using refined model
GA run using final model
Initial Coarse Model learned
Model refined and relearnt
Final model learnt
Final population exact function evaluation
p
q
p
q
p
q
0
p + q
2(p + q)
(T − 1)(p + q)
T(p + q)
Generation's of EA

Figure 4.2: Diagram of the successive approximation procedure proposed by Nain and Deb. T is the maximum number of generations. p and q are the number of generations the exact and approximate evaluations are used, respectively. According to [49].

the surrogate model is regularly renewed over the duration of the optimization with solutions of the last few generations as training data, the approximations will be accurate and appropriate for the next few generations.

The formalized procedure begins like any other EA. The initial population is randomly created and improved in successive generations. But in contrast to the normal EA all evaluated solutions are stored in a training database, and the EA is stopped after p generations. The database with the evaluations contains now $N' = pk$ exact solutions, where k is the population size. Using the N' data points as the training set, the first coarse surrogate model is created, it is subsequently used to for q generations for all evaluations. After q generations the exact objective function is used again for p generations to refill the training database with new samples. The surrogate model is then retrained, and this cycle repeats until the termination criterion is met (see Figure 4.2). In the best case, only a fraction $\frac{p}{p+q}$ of the generations need to be evaluated using the real objective function to achieve the same performance. If that theoretical gain can be attained depends on the quality of the surrogate models.

Nain and Deb proposed to use ANNs as the surrogate model, pointing out its proven capabilities as a function approximation tool[27] and its adaptability. ANNs can produce a rough or detailed approximation depending on how training samples are distributed. The number of design variables and objective values can are modeled as the number of input neurons and output neurons, respectively.

4.2 Proposed Method

The complete proposed procedure is visualized as a flowchart in Figure 4.3. We will now demonstrate, in an exemplary manner, how this flowchart is traversed. As in any MOEA, we begin by generating a random initial population and evaluate it using the exact objective function. Since we use the numbers of exact evaluations as the stopping criterion, we have to increase the number of evaluations accordingly. Next, we initialize our surrogate model. We initialize two variables, the GenerationCounter, which tells us when we have to switch between the real and the surrogate function, and a flag named UseSurrogate, that tells which of the two we are currently using. After that, we generate an empty training database which is then immediately filled with the individuals we just evaluated.

Then we check the stopping condition which is not fulfilled the first time we visit this node. So we go on to generate the offspring population. We create the mating population using the binary tournament selection algorithm. This algorithm draws two random solutions from the current population and adds the dominant individual to the mating pool. If no solutions dominate the other one of the two is chosen randomly. This is repeated until the mating pool is full. Then an offspring population is created using SBX[14], and polynomial mutation is applied to the children.

The next step in an EA(see Algorithm 2.1) would be to evaluate the population. We can use the real function or the surrogate function to do that. In this example, we are currently still in the phase where we collect more training data, so we use the real function for evaluation and increase the number of evaluations accordingly. These individuals are also added to the training

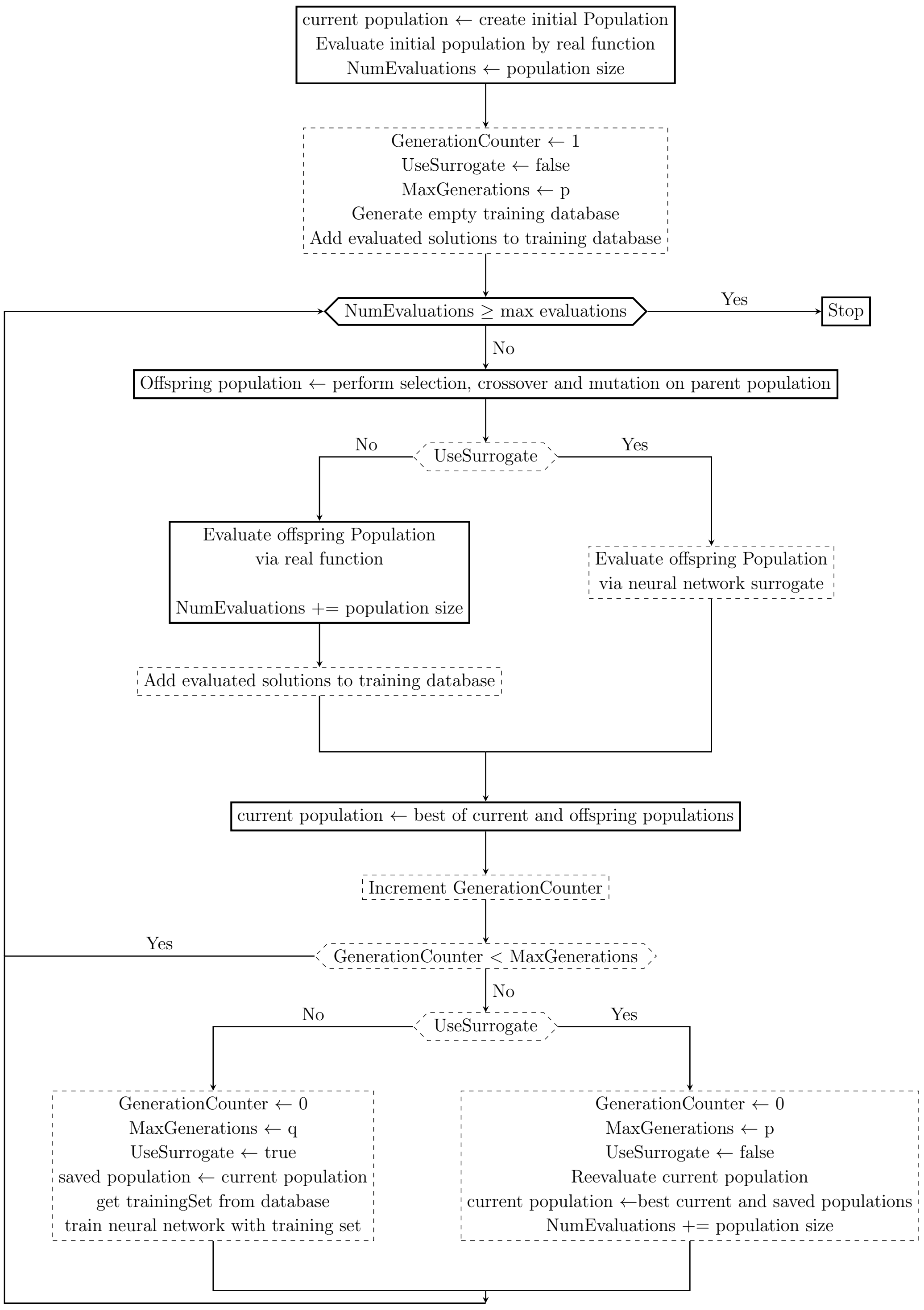

current population ← create initial Population
Evaluate initial population by real function
NumEvaluations ← population size
GenerationCounter ← 1
UseSurrogate ← false
MaxGenerations ← p
Generate empty training database
Add evaluated solutions to training database
NumEvaluations ≥ max evaluations
Yes
Stop
No
Offspring population ← perform selection, crossover and mutation on parent population
No
UseSurrogate
Yes
Evaluate offspring Population via real function
NumEvaluations += population size
Evaluate offspring Population via neural network surrogate
Add evaluated solutions to training database
current population ← best of current and offspring populations
Increment GenerationCounter
Yes
GenerationCounter < MaxGenerations
No
No
UseSurrogate
Yes
GenerationCounter ← 0
MaxGenerations ← q
UseSurrogate ← true
saved population ← current population
get trainingSet from database
train neural network with training set
GenerationCounter ← 0
MaxGenerations ← p
UseSurrogate ← false
Reevaluate current population
current population ←best current and saved populations
NumEvaluations += population size

Figure 4.3: Flowchart of the procedure. Dashed boxes are added logic to facilitate the use of the surrogate model. The other boxes contain logic from the regular NSGA-II process.

database. Now, the best individuals from both the current population and new offspring population are selected using the non-dominated sorting algorithm from Algorithm 2.3 using the comparison operator from Definition 2.1. This is the new current population, and normally NSGA-II would now go on to repeat the process from the point where the stopping condition is checked.

In our algorithm, we first have to check if we need to switch between the real and the surrogate function. To do that the generation counter is incremented and compared to the maximum number of generations for the current section. If we are in a section where the surrogate function is not used, MaxGenerations equals p. If we are currently using the surrogate function, it equals q. If the current generation counter is smaller than MaxGenerations, we will continue without changing the mode. Since we are still at the beginning of the run, in this example, this will most likely be the case, and we would continue until the generation counter reaches p. Then we have accumulated enough training examples and can change from the real to the approximated function. The last $p \cdot k$ individuals from the training database are selected.

Using this training data, a neural network is trained using the hyperparameters described in Section 5.1. Some of the most critical hyperparameters concerning the ANN are the number of hidden layers, number of neurons in each hidden layer, the learning rate and the number of training epochs. The number of input and output neurons is set to match the number of problem variables and objectives, respectively. Furthermore, some of the flow variables are updated: the GenerationCounter is reset to 0, MaxGenerations is set to the number of surrogate generations q and UseSurrogate is now true. Furthermore, a copy of the current population is stored.

When entering the main loop the next time we enter the branch where we evaluate the newly created offspring generation with our surrogate model instead of the real function. Since evaluations with surrogate function should take a negligible amount of time compared to an evaluation of an expensive real function, we do not increase the number of real function evaluations.

We continue this way until the generation counter reaches q and we have to switch back to real function because the current approximation is no longer exact enough. The current population should now contain a mix of individuals that were evaluated with the real function and the surrogate function or only the latter. A problem that was identified in the early tests that were conducted was that it could happen, that errors in the approximated fitness landscape can evaluate some individuals to have unfeasible objective values. This can manifest by objective values that are better than the global Pareto-optimal front, collapsed into zero (either in both or only one objective) or are negative. While some optimization problems can have negative objective values, which they should not since the test problems in this thesis do not have negative objective values. The resulting problem is that these unfeasible objective values are never corrected the exact objective function will never find a better individual than the wrong ones. The erroneous individuals are then staying in the population until the optimization is completed. Since these these individuals are not correct but can not be dominated by correctly evaluated individuals they waste their slots in the population P. They effectively decrease the poulation size, and therefore, slow the EA down or if all individuals fall into this trap completely stall the EA. The severity of this problem depends on the fitness landscape of the test function. An optimization problem with a simple landscape is less likely to produce approximations with this flaw. For example, in ZDT1 this can sometimes be ignored, while in DTLZ1 the whole population quickly degenerates. To prevent this degeneration, the current (eventually flawed) population is reevaluated using the exact function and then the best individuals from the reevaluated population and the saved population from earlier are selected to make up the new current population. The saved population is a copy of the last population that was evaluated with the real function, as described above. This way degenerated individuals are kept out but evaluations of our evaluation budget are needed, so the number of evaluations is increased.

Now we are using the real function again and collect new training data. This cycle is repeated until the maximum number of evaluations is reached, after which the algorithm stops. There is no last step to reevaluate the final population with the real function because the process makes sure that the algorithm only stops if all individuals have been just evaluated with the real function.

5 Evaluation

This chapter presents how the proposed method was implemented, as well as the experiments that compare our approach on test problems with varying difficulty and with different number of search variables and a discussion of the results.

5.1 Implementation

The proposed method was implemented in Java based on the jMetal[1] Framework. jMetal is a Java-based framework with the goal to support researchers in the field of metaheuristics, with a focus on multi-objective optimization. As such, it contains classes for performing experiments, as well as, implementations of many standard algorithms. To facilitate deep learning, jMetal was integrated with Deep Learning for Java (DL4J)[2]. The implementation of NSGA-II contained in jMetal was extended to record all solutions, a neural network surrogate model, and the switching logic. DL4J is a Java-based toolkit that contains implementations for many deep neural network architectures, we used their implementation of the deep feedforward neural network which can be configured in various ways. Initial tests were conducted, and the rest of this section will describe the hyperparameters we found relevant.

These hyperparameters come from three sources. The first source is the standard NSGA-II itself, the second the integration of the surrogate model into the NSGA-II and finally the hyperparameters concerning the ANN

The hyperparameters of the NSGA-II are the following:

The **population size**, abbreviated with k, sets the number of individuals in the population used by the EA and therefore also determines how many

[1] `jmetal.github.io/jMetal/`

[2] `deeplearning4j.org`

evaluations have to be done each generation. Values such as 100 or 200 are usually used for this hyperparameter.

The **maximum number of evaluations** controls after how many real evaluations the EA stops. More difficult problems need more evaluations. We use 3.000, 5.000, 20.000 or 30.000 evaluations according to the difficulty of the problem.

The **selection** hyperparameter sets the procedure which generates the mating pool from the current population. NSGA-II usually uses binary tournament selection, but other algorithms can be used to for example fitness proportional selection.

The **crossover** hyperparameter sets the procedure that creates child individuals from parent individuals, selected from the mating pool. NSGA-II usually uses simulated binary crossover (SBX)[14].

The **mutation** hyperparameter sets the procedure which introduces small random changes to the child individuals after crossover. NSGA-II usually uses polynomial mutation[14].

These hyperparameters are related to the integration of the surrogate model:

The **surrogate generations**, we abbreviate it with q, hyperparameter sets the number of generations for which the surrogate is used after it was (re-)learned. In papers that use a similar algorithm[49, 24] this number moves between 3 and 20.

The **learning generations** hyperparameter p, on the other hand, determines for how many generations the real fitness functions are used and how long the training data is collected. In the related papers, this number is usually set between 3 and 10.

The remaining hyperparameters and the most numerous belong to the artificial neural network:

The **number of hidden layers**, we will abbreviate it with d, hyperparameter tells us how many hidden layers the neural net uses. Research indicates that more hidden layers give network more expressive power i.e. it can approximate more difficult problems. Most of the related research uses shallow neural networks, but in the experiments, we will vary the number of hidden layers to see if deeper networks improve the quality of the surrogate function.

The **network width**, we will abbreviate it with w, hyperparameters sets how many neurons are in each hidden layer.

The **activation function** is needed to apply non-linear transformations to the weighted sum computed by a neuron from its inputs and wights, which is a linear operation. The non-linear transformations are needed so the network can learn complicated functions, which is not possible when the network only does linear transforms. This hyperparameter allows to sets the activation functions used by the hidden layers of the network. Theoretically, any activation function supported by DL4J can be used here, but the most common options are ReLU, random ReLU, leaky ReLU, sigmoid or hyperbolic tangent (tanh).

The **minibatch size** allows the user to set the size of the minibatches for SGD as described in Section 2.5.2. According to Bengio[3], the minibatch should be set between 1 and a few hundred, with 32 being a reasonable default.

The **learning rate** of the neural network controls how fast the weights change with every minibatch. For every minibatch, the derivative of the cost function with respect to the weights and biases is computed and subtracted from the weights. The learning rate scales the impact of the derivative. A small learning rate will make the learning converge very slowly, while a large learning rate will cause gradient descent overshoot the optimum again and again. Usually, a learning rate between 0.001 and 0.0001 is recommended.

The **loss function** is used by the neural network to compute the error between the training set and the output of the network. The choices are: mean squared error (MSE), root mean squared error (RMSE) and mean absolute error (MAE).

normalization is recommended for neural network learning. This means linearly scaling the input and output values of training dataset between 0 and 1. If this is on, we also need to scale the inputs during evaluations and denormalize the results.

The **number of epochs** hyperparameter tells how often the whole training set is presented to the neural network during training. A small learning rate or a difficult learning problem will need a higher number of epochs.

The **l2** hyperparameter determines which value is used for the l2 regularization. This type of regularization penalizes large network weights. Large weights are discouraged because they are associated with overfitting, i.e., the network only memorizes the data and does not generalize. Usually, a value of 0.001 is recommended.

The **early stopping** hyperparameter turns the use of early stopping on or off. This kind of regularization tries to find the optimal stopping point between under- and overfitting. To find the optimal stopping point, an additional test set is created from the original training set. The test set is used to measure the generalization error. At every epoch, the training and generalization errors are measured, and at the beginning, both will decrease. At some point, the test error will still decrease, but the generalization error will start to increase at this point the network begins to overfit the data, and the learning process is stopped.

The **target score** sets an error value which if reached during the learning procedure will abort the learning. Can be seen as a way achieve the effects of early stopping manually. Nain and Deb[49] also used this approach.

The **optimization algorithm** hyperparameter determines which algorithm is used during the learning process. We will use Stochastic Gradient Descent (SGD) but DL4J also provides other optimization algorithms.

The **updater** hyperparameter sets the algorithm that dynamically updates the learning rate during the learning process. We will usually use RMSprop but other updaters provided by DL4J can be used.

The **weight initialization** hyperparameter sets which procedure is used to initialize the weights of a new neural network. We usually use the Xavier method but other initializers provided by DL4J can be used.

5.2 Experiment Settings

As we can see, our implementation of the proposed method has many hyperparameters. Investigating the influence of all these hyperparameters is not feasible because the number of possible combinations is too high. We have to select some of the hyperparameters for our experiments. To answer our research questions, we decided to investigate the following parameters in detail:

- **width w** and **depth d**
  In Section 2.6, we talked about how, according to some researchers

[20, 48], adding more layers to a network makes the network more powerful than adding more hidden neurons in the existing layers. In the first experiment, we will investigate ANN surrogates that have a differing number of network hidden layers and widths. We expect to see that deeper networks with more layers are performing better than networks with fewer layers.

- **learning generations p** and **surrogate generations q**
  In our initial tests, these two parameters, which are part of the integration of the surrogate into the MOEA, were parameters with the most influence on how far the surrogate-assisted MOEA progressed towards the Pareto-optimal solutions. We will in a second experiment compare the best surrogate using a deep network to a surrogate with an architecture (width and depth) similar to Nain and Deb[49]. They used a shallow network with one hidden layer and 40 neurons.

The remaining hyperparameters were searched for a suitable combination based on recommendations in the literature and initial results of our test runs. These parameters, which are constant for all experiments in this chapter, can be seen in table 5.1.

Parameter	Value
crossover	SBX($p_c = 0.9$, $\eta_c = 10$)
population size	$k = 200$
loss function	MSE
target score	0.008
training epochs	150
learning rate	$\eta = 0.0009$
mutation	polynomial mutation ($p_m = \frac{1}{\#variables}$, $\eta_m = 50$)
network optimizer	SGD
activation function	ReLU
weight init	XAVIER
l2 regularization	0.001
updater	RMS PROP
normalization	yes, between [0,1]
early stopping	no

Table 5.1: Constant parameters for the experiments in Section 5.3

The other research questions are related to the test problems, specifically the performance of deep networks on high dimensional problems and also on difficult problems. Therefore, we will present a selection of test problems different test problems in the following section. Each of those test problems can be configured to use different numbers of search variables and presents different fitness landscapes to the surrogate-assisted MOEA.

5.2.1 Multi-objective Test Problems

It is the goal of this thesis to test the performance of the proposed surrogate-assisted MOEA. The testing entails many optimization runs since multiple hyperparameter combinations need to be tested and to achieve statistical soundness the runs have to be repeated. If we would use real-world expensive MOPs, it would quickly exceed the reasonable time frame for this thesis. To avoid this, we use well-established test suites for multi-objective problems and measure the effects and performance of the proposed approach. Using test suites has the other advantage of making our results comparable to other papers. Therefore, three common test suites found in other EA literature are presented in this section. Each test suite will be described briefly as are the chosen test problems. The ZDT test suite is the oldest and contains the easiest problems but is also often used as a benchmark in the related research. Therefore testing our approach on some ZDT problems will make sure we can compare it to other papers. The DTLZ and WFG test suites are often used in other research and contain more difficult problems. Using these suites allows us to test if deep learning can be used to approximate more difficult problems.

The ZDT test suite was introduced in 2000 by Zitzler et al. [60] and contained six bi-objective test problems. We chose the ZDT1 and ZDT4 problems. ZDT1 is a relatively simple problem with a convex Pareto-optimal front. The reason for including ZDT1 is that many papers in the field have used it to benchmark their surrogate models[57, 44, 55, 50, 43], and as a simple entrance test. ZDT4 is a more complicated problem that tests if the EA can deal with a multimodal fitness landscape. It has 21^9 local Pareto-optimal fronts where an EA can get stuck. The ZDT4 was chosen because it was the most difficult of the ZDT test suite, and some proposed surrogate models that have done well on the other ZDT test functions struggled with ZDT4[44, 50, 43].

Name	Objective	Modality	Landscape	Geometry
ZDT1	f_1	unimodal	-	convex
	f_2	unimodal		
ZDT4	f_1	unimodal	-	convex
	f_2	multimodal		
DTLZ1	$f_{1,2}$	multimodal	Pareto-many-to-One	linear
DTLZ2	$f_{1,2}$	unimodal	Pareto-many-to-One	concave
WFG1	$f_{1,2}$	unimodal	polynomial, flat	convex, mixed
WFG2	f_1	unimodal	-	convex, disconnected
	f_2	multimodal		

Table 5.2: Chosen multi-objective test problems. Properties according to Huband et al.[32]

The DTLZ suite was introduced in 2002[17], mainly to remedy the low number of test problems with a configurable number of objectives. DTLZ1 is a multimodal problem with $(11^k - 1)$ local Pareto-optimal fronts. The authors reported that NSGA-II needed 30.000 evaluations to reach the Pareto-optimal front[17]. DTLZ2 has a spherical Pareto-optimal front, it is still a rather easy problem but will present an optimizer with more challenges than ZDT1.

The WFG test suite was created by Huband et al.[32, 31]. Their goal was to create a toolkit where features can be added to test problems in a modular fashion. The selected problems from the WFG test suite are the WFG1 and WFG2 problem. We divide the parameters into equal parts position-related and distance-related parameters and use two objectives. WFG1 has a linear Pareto-front. Both objectives are unimodal, it still difficult because it has flat regions and a biased fitness landscape. The Pareto-optimal front consists of convex and concave segments. WFG2 has one multimodal and one multi-modal objective, and a convex and disconnected Pareto-optimal front.

5.2.2 Structure of the Results

For every experiment, there will be a table showing the performance of different surrogate models on different test problems. The columns show the performance of each surrogate model on one problem. For example, the leftmost column in Table 5.3 shows the performance of each surrogate model on the test problem DTLZ1 with 30 variables. The number of variables will be abbreviated with a v. We test each problem with 30, 50 and 70 variables.

The rows contain the performance of each algorithm on each test problem. The standard NSGA-II algorithm, shown in the row "NSGAII", is present in every experiment, for comparison purposes. The remaining rows show our proposed algorithms with different parameter settings. These parameters can be distinguished by abbreviations, to not overload the tables. For every experiment, we will explain what these abbreviations mean.

The values in each cell are the hypervolumes of the median performance over 31 test runs. The subscripted values show the respective interquartile range. The reference point used to compute the hypervolume is the same for all algorithms in a column and across tables. The hypervolumes in "DTLZ1 v30" column in Table 5.3 can be compared to the "DTLZ1 v30" column in Table 5.6. The reference point is the worst objective value of any algorithm on that particular problem.

The most performant algorithm in each column (on each problem) is highlighted in bold font. All other algorithms that are not significantly worse than the best algorithm are highlighted with colored cells. To achieve this, the Mann-Whitney U test was used to compute if an algorithm has the same distribution as the best algorithm in its column. The two-tailed test was used (sample size for each algorithm is 31) and the null hypothesis was rejected if $p < 0.05$. In this context, a rejected hypothesis means that the algorithm is significantly worse than the best algorithm in its column.

Furthermore, for each problem, there is a plot showing the median performance of NSGA-II and the best performing surrogate algorithm, so the reader can better visualize what the numbers in the table mean. For example, see Figures 5.3, 5.4 or 5.8. Each plot also shows the true Pareto-optimal front, it is labeled as "Global". The true Pareto-optimal front of DTLZ1 can not be seen in the plots of DTLZ1 because the optimal objective values are so small compared to optimal values found by the MOEAs

5.3 Results and Analysis

This section goes over the results in detail. Multiple experiments were conducted to evaluate the proposed algorithm testing different combinations of hyperparameters.

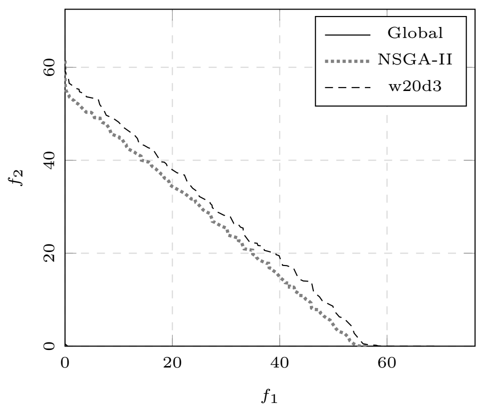
5.3.1 Influence of the Neural Network Architecture

The goal of the first experiment was to find out if the depth or width of the neural network surrogate model influences the quality of the approximations. Therefore, the parameters for neurons per layer and the number of hidden layers were varied. The tested parameters for the width were 10, 20, 50 and 100, the number hidden layers 1, 3, 6 and 9. In the tables w was used to abbreviate the width of the network and d to abbreviate the number of hidden layers, for example, the algorithm described by "w20 d3" has three hidden layers each with 20 neurons. Both the number of learning generations p and the number of surrogate generations q were set to five. The results of the first experiment can be seen in the Tables 5.3 and 5.4 and 5.5.

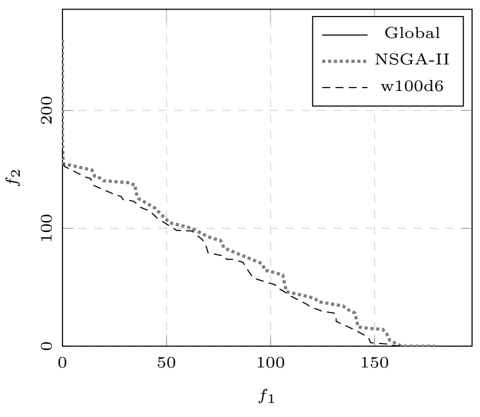
First, let us go over what can be seen in each table. Table 5.3 shows us the performance of each algorithm on the chosen test problems of the DTLZ test suite. The DTLZ1 problem is one of the more difficult test problems.

The standard NSGA-II has the best performance on the low-dimensional variant (30 variables) of DTLZ1. The surrogate algorithms with the best performance are "w20 d3" and "w20 d9". This goes against our expectations that deeper networks should perform better than shallower networks. The surrogate-assisted MOEA "w20 d9" is better than "w20 d1" and "w20 d6," but we would also expect that "w20 d6" is better than "w20 d3" but an algorithm with only half the number of hidden layers outperforms this algorithm with six hidden layers. Figure 5.1a also shows the slight advantage standard NSGA-II has over "w20 d3".

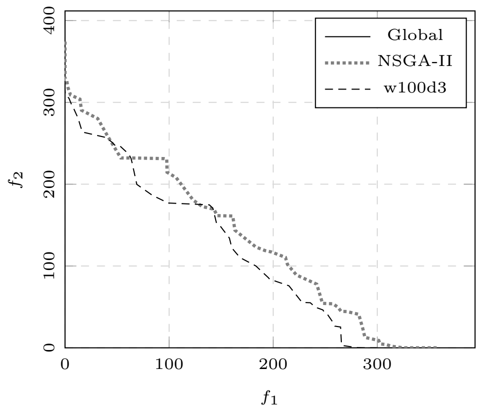
Both of the higher dimensional variants of DTLZ1 are cases where many surrogate-algorithms performed similarly to NSGA-II. We can also see that more surrogate algorithms in the lower parts of both columns have that similar performance. This indicates that the width of the ANN is more important than the depth in these two higher-dimensional variants of DTLZ1. The best



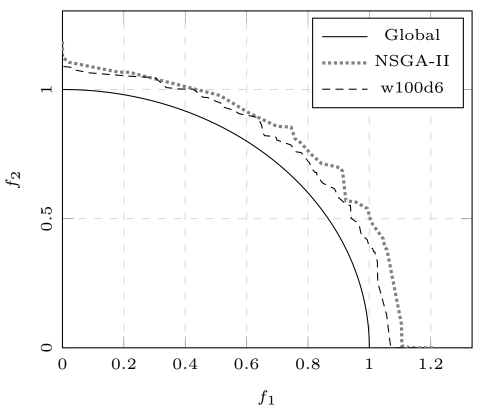
(a) DTLZ1 v30



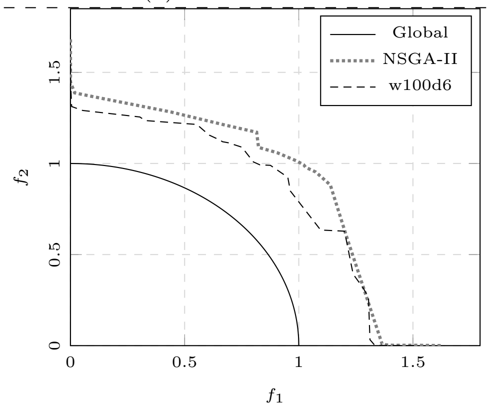
(b) DTLZ1 v50



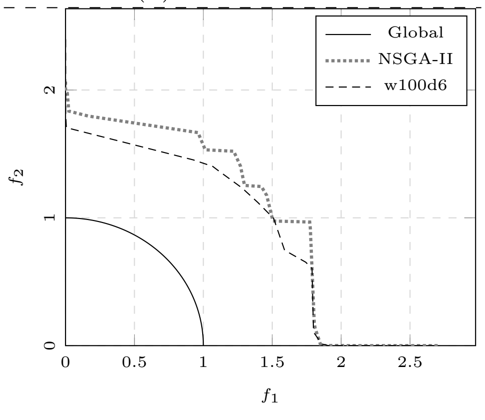
(c) DTLZ2 v70



(d) DTLZ2 v30

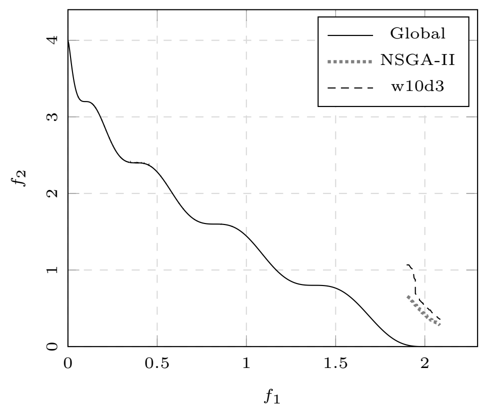


(e) DTLZ2 v50

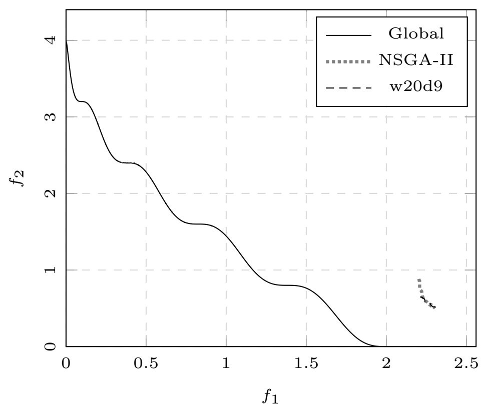


(f) DTLZ2 v70

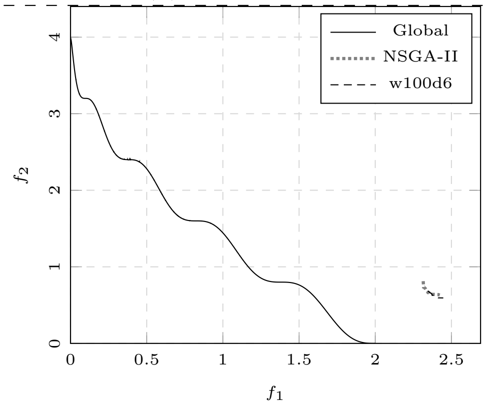
Figure 5.1: Adapting the network architecture. Plots for the DTLZ1 and DTLZ2 problems.



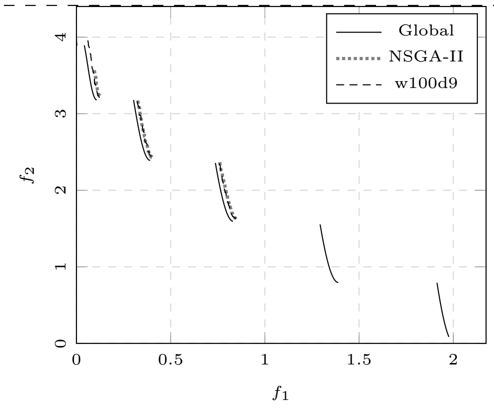
(a) WFG1 v30



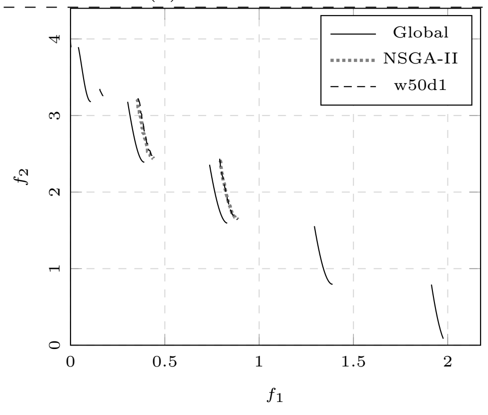
(b) WFG1 v50



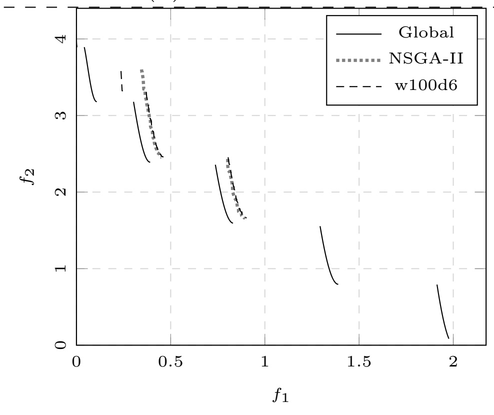
(c) WFG1 v70



(d) WFG2 v30



(e) WFG2 v50



(f) WFG2 v70

Figure 5.2: Adapting the network architecture. Plots for the WFG1 and WFG2 problems.

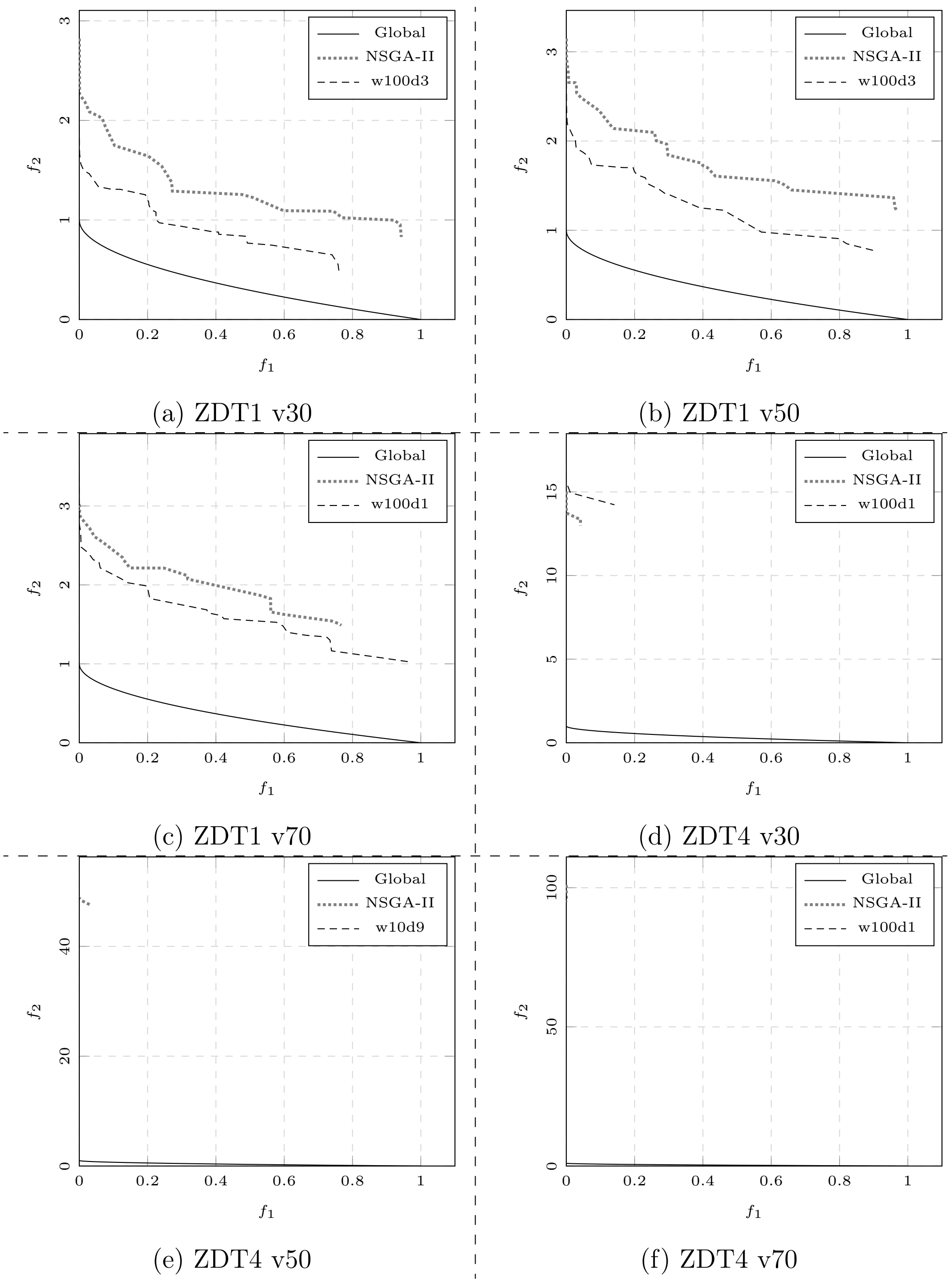


Figure 5.3: Adapting the network architecture. Plots for the ZDT1 and ZDT4 problems.

	DTLZ1 v30	DTLZ1 v50	DTLZ1 v70	DTLZ2 v30	DTLZ2 v50	DTLZ2 v70
NSGAII	$\mathbf{0.9971_{1.73E-03}}$	$0.9818_{9.68E-03}$	$0.9557_{2.15E-02}$	$0.9447_{8.32E-03}$	$0.9065_{1.48E-02}$	$0.8633_{1.49E-02}$
w10 d1	$0.9955_{2.47E-03}$	$0.9787_{5.42E-03}$	$0.9504_{1.68E-02}$	$0.9278_{1.93E-02}$	$0.8756_{1.28E-02}$	$0.8172_{2.40E-02}$
w10 d3	$0.9957_{2.89E-03}$	$0.9820_{1.18E-02}$	$0.9469_{2.85E-02}$	$0.9262_{1.21E-02}$	$0.8792_{2.14E-02}$	$0.8279_{2.47E-02}$
w10 d6	$0.9959_{1.66E-03}$	$0.9750_{1.39E-02}$	$0.9582_{1.27E-02}$	$0.9251_{2.12E-02}$	$0.8862_{1.47E-02}$	$0.8389_{2.21E-02}$
w10 d9	$0.9961_{1.40E-03}$	$0.9764_{1.21E-02}$	$0.9560_{2.14E-02}$	$0.9251_{1.97E-02}$	$0.8947_{2.15E-02}$	$0.8238_{2.70E-02}$
w20 d1	$0.9963_{1.88E-03}$	$0.9813_{9.22E-03}$	$0.9492_{3.57E-02}$	$0.9266_{1.34E-02}$	$0.8779_{2.10E-02}$	$0.8349_{2.74E-02}$
w20 d3	$0.9967_{2.04E-03}$	$0.9794_{9.62E-03}$	$0.9549_{1.11E-02}$	$0.9289_{1.46E-02}$	$0.8771_{2.44E-02}$	$0.8377_{2.28E-02}$
w20 d6	$0.9964_{1.98E-03}$	$0.9805_{5.50E-03}$	$0.9518_{2.40E-02}$	$0.9278_{1.51E-02}$	$0.8844_{2.38E-02}$	$0.8605_{2.22E-02}$
w20 d9	$0.9966_{1.34E-03}$	$0.9809_{6.46E-03}$	$0.9520_{2.03E-02}$	$0.9305_{1.57E-02}$	$0.8660_{1.77E-02}$	$0.8420_{2.00E-02}$
w50 d1	$0.9956_{3.12E-03}$	$0.9767_{9.64E-03}$	$0.9495_{2.48E-02}$	$0.9357_{2.07E-02}$	$0.8831_{2.96E-02}$	$0.8374_{2.31E-02}$
w50 d3	$0.9959_{2.35E-03}$	$0.9830_{4.81E-03}$	$0.9615_{8.16E-03}$	$0.9403_{1.31E-02}$	$0.9020_{1.30E-02}$	$0.8611_{1.29E-02}$
w50 d6	$0.9965_{1.91E-03}$	$0.9830_{5.61E-03}$	$0.9558_{1.64E-02}$	$0.9414_{1.27E-02}$	$0.8892_{1.68E-02}$	$0.8343_{1.94E-02}$
w50 d9	$0.9962_{1.89E-03}$	$0.9827_{6.80E-03}$	$0.9534_{2.62E-02}$	$0.9421_{1.32E-02}$	$0.8851_{1.15E-02}$	$0.8356_{2.89E-02}$
w100 d1	$0.9961_{2.11E-03}$	$0.9835_{3.59E-03}$	$0.9609_{1.24E-02}$	$0.9515_{8.21E-03}$	$0.9140_{1.63E-02}$	$0.8728_{2.61E-02}$
w100 d3	$0.9965_{1.99E-03}$	$0.9828_{1.00E-02}$	$\mathbf{0.9619_{1.57E-02}}$	$0.9522_{1.25E-02}$	$0.8932_{2.63E-02}$	$0.8768_{1.07E-02}$
w100 d6	$0.9953_{2.56E-03}$	$\mathbf{0.9836_{5.38E-03}}$	$0.9472_{2.69E-02}$	$\mathbf{0.9657_{1.19E-02}}$	$\mathbf{0.9294_{1.69E-02}}$	$\mathbf{0.8782_{2.42E-02}}$
w100 d9	$0.9955_{1.76E-03}$	$0.9823_{6.13E-03}$	$0.9581_{1.92E-02}$	$0.9409_{1.85E-02}$	$0.9008_{1.40E-02}$	$0.8657_{2.13E-02}$

Table 5.3: Adapting the network architecture. Median for DTLZ1 and DTLZ2. 30.000 and 5.000 evaluations were used, respectively.

algorithm for "DTLZ1 v50" is "w100d6" and for "DTLZ1 v70" it is "w100d3". Both of them have multiple hidden layers, their slight advantage over standard NSGA-II is shown in Figure 5.1b and Figure5.1c.

The Figures related to the DTLZ1 problem also show the degrading performance when the number of search variables are increased. This is revealed by looking at the objective values on the axis, and what values are achieved for both objectives. The degradation is similar for both standard NSGA-II and the top-performing surrogate algorithms, this indicates that these algorithms can deal with higher-dimensional problems.

The best surrogate algorithm for "DTLZ1v50", i.e., "w100d6", is also the algorithm that clearly outperforms standard NSGA-II on the DTLZ2 irrespective of the chosen number of variables. When looking at the figures 5.1d, 5.1e and 5.1f we also see how "w100d6" covers more hypervolume than standard NSGA-II in all three cases. DTLZ2 is one of easier problems we use here. It has a unimodal landscape that is easier to approximate than the highly multimodal landscape of DTLZ1.

All variants of the WFG1 problem are dominated by the standard NSGA-II. In the highest-dimensional case "WFG1 v70", the standard NSGA-II is significantly better than any of our surrogate algorithms. The surrogate algorithm "w10 d3" has a similar performance to standard NSGA-II. The "w100 d6" al-

	WFG1 v30	WFG1 v50	WFG1 v70	WFG2 v30	WFG2 v50	WFG2 v70
NSGAII	$\mathbf{0.3576_{3.88E-02}}$	$\mathbf{0.1714_{2.50E-02}}$	$\mathbf{0.1332_{2.05E-02}}$	$0.6988_{5.09E-03}$	$0.6830_{8.56E-03}$	$0.6721_{6.61E-03}$
w10 d1	$0.3334_{5.84E-02}$	$0.1464_{2.88E-02}$	$0.1078_{1.68E-02}$	$0.6985_{7.73E-03}$	$0.6807_{5.88E-03}$	$0.6721_{8.62E-03}$
w10 d3	$0.3392_{5.12E-02}$	$0.1646_{3.30E-02}$	$0.1078_{1.09E-02}$	$0.6989_{1.27E-02}$	$0.6813_{1.02E-02}$	$0.6690_{8.57E-03}$
w10 d6	$0.2890_{3.70E-02}$	$0.1473_{1.96E-02}$	$0.0955_{1.13E-02}$	$0.6953_{5.54E-03}$	$0.6793_{7.91E-03}$	$0.6671_{6.88E-03}$
w10 d9	$0.3000_{4.41E-02}$	$0.1639_{2.73E-02}$	$0.1067_{2.80E-02}$	$0.6952_{7.15E-03}$	$0.6808_{8.56E-03}$	$0.6652_{1.33E-02}$
w20 d1	$0.3170_{4.44E-02}$	$0.1390_{1.59E-02}$	$0.1034_{1.80E-02}$	$0.6982_{7.44E-03}$	$0.6807_{1.02E-02}$	$0.6699_{1.12E-02}$
w20 d3	$0.2998_{2.73E-02}$	$0.1514_{1.64E-02}$	$0.1071_{2.55E-02}$	$0.6984_{4.55E-03}$	$0.6771_{1.11E-02}$	$0.6700_{1.01E-02}$
w20 d6	$0.3058_{4.98E-02}$	$0.1386_{2.54E-02}$	$0.1085_{1.41E-02}$	$0.6963_{8.81E-03}$	$0.6818_{7.52E-03}$	$0.6665_{8.70E-03}$
w20 d9	$0.2945_{4.73E-02}$	$0.1683_{2.21E-02}$	$0.1010_{1.51E-02}$	$0.6986_{6.11E-03}$	$0.6768_{6.99E-03}$	$0.6656_{1.45E-02}$
w50 d1	$0.2864_{3.68E-02}$	$0.1497_{2.09E-02}$	$0.1153_{2.14E-02}$	$0.6989_{1.10E-02}$	$\mathbf{0.6840_{1.08E-02}}$	$0.6700_{7.58E-03}$
w50 d3	$0.2819_{5.53E-02}$	$0.1551_{1.48E-02}$	$0.1017_{1.80E-02}$	$0.6993_{6.22E-03}$	$0.6812_{1.05E-02}$	$0.6715_{5.91E-03}$
w50 d6	$0.2847_{4.55E-02}$	$0.1466_{2.69E-02}$	$0.1197_{1.92E-02}$	$0.7002_{8.31E-03}$	$0.6825_{8.17E-03}$	$0.6708_{9.06E-03}$
w50 d9	$0.3154_{4.79E-02}$	$0.1378_{1.79E-02}$	$0.1084_{1.49E-02}$	$0.6966_{9.39E-03}$	$0.6791_{1.08E-02}$	$0.6713_{1.35E-02}$
w100 d1	$0.3069_{2.39E-02}$	$0.1510_{1.31E-02}$	$0.1097_{2.03E-02}$	$0.6985_{6.48E-03}$	$0.6834_{8.34E-03}$	$0.6689_{8.71E-03}$
w100 d3	$0.2896_{3.70E-02}$	$0.1547_{2.87E-02}$	$0.1123_{1.33E-02}$	$0.7012_{5.90E-03}$	$0.6829_{1.17E-02}$	$0.6665_{8.70E-03}$
w100 d6	$0.3389_{5.09E-02}$	$0.1537_{3.09E-02}$	$0.1201_{8.97E-03}$	$0.6978_{8.94E-03}$	$0.6811_{8.57E-03}$	$\mathbf{0.6730_{7.49E-03}}$
w100 d9	$0.3103_{4.41E-02}$	$0.1621_{1.95E-02}$	$0.1057_{2.03E-02}$	$\mathbf{0.7016_{1.49E-01}}$	$0.6835_{1.09E-02}$	$0.6667_{1.47E-02}$

Table 5.4: Adapting the network architecture. Median for WFG1 and WFG2. With WFG1 30.000 and 20.000 evaluations were used, respectively.

gorithm also has a similar performance to NSGA-II on "WFG1 v30", as does "w20 d9" on "WFG1 v50". There is no pattern visible in the performance of the surrogate algorithms on WFG1, neither the width nor the number of hidden layers hints at better performing surrogate variants. If we look at Figures 5.2a through 5.2c, we can see the related search results. We see that in all three cases the search algorithm moves only towards the lower part of the Pareto-optimal front, the biased fitness landscape of WFG1 may cause this.

The WFG2 problem has many surrogate algorithms that perform similarly to the standard NSGA-II, like the higher-dimensional cases of DTLZ1. There seems to also be a very slight bias towards surrogate algorithms with wider ANNs. In all three cases, surrogate algorithms are the top performers: "w100d9" for "WFG2 v30", "w50 d1" with a shallow ANN for "WFG2 v50", and "w100 d6" for "WFG2 v70". This surrogate variant with six hidden layers is also among the top-performing surrogate algorithms for both lower-dimensional WFG2 cases. When we look at the figures 5.2d, 5.2e and 5.2f we can see which parts of the disconnected Pareto-front of WFG2 are found by the MOEAs. In the WFG2 problem with 30 variables the first three disconnected parts are found, but also "100d9" finds more of the first part. This observation is also true for the higher-dimensional cases.

	ZDT1 v30	ZDT1 v50	ZDT1 v70	ZDT4 v30	ZDT4 v50	ZDT4 v70
NSGAII	$0.8345_{2.44E-02}$	$0.7365_{2.67E-02}$	$0.7020_{2.58E-02}$	$\mathbf{0.7352_{7.91E-02}}$	$0.6008_{7.24E-02}$	$0.5206_{6.62E-02}$
w10 d1	$0.8618_{1.76E-02}$	$0.7394_{2.09E-02}$	$0.7239_{2.87E-02}$	$0.6298_{1.26E-01}$	$0.5517_{9.71E-02}$	$0.4258_{9.38E-02}$
w10 d3	$0.8229_{1.80E-02}$	$0.7603_{3.01E-02}$	$0.7326_{1.89E-02}$	$0.6556_{8.11E-02}$	$0.5337_{8.23E-02}$	$0.4725_{8.16E-02}$
w10 d6	$0.8383_{2.10E-02}$	$0.7492_{3.08E-02}$	$0.7019_{2.53E-02}$	$0.6873_{1.21E-01}$	$0.5790_{8.57E-02}$	$0.4753_{5.87E-02}$
w10 d9	$0.8623_{2.76E-02}$	$0.7310_{4.71E-02}$	$0.7041_{4.39E-02}$	$0.6490_{1.13E-01}$	$\mathbf{0.6078_{1.14E-01}}$	$0.4763_{5.73E-02}$
w20 d1	$0.8732_{1.47E-02}$	$0.7759_{2.34E-02}$	$0.7280_{4.78E-02}$	$0.6530_{7.90E-02}$	$0.5754_{1.28E-01}$	$0.4340_{7.11E-02}$
w20 d3	$0.8626_{3.17E-02}$	$0.7209_{3.01E-02}$	$0.7456_{1.95E-02}$	$0.6836_{8.05E-02}$	$0.4859_{1.56E-01}$	$0.4126_{7.29E-02}$
w20 d6	$0.8505_{1.72E-02}$	$0.7713_{3.36E-02}$	$0.7431_{2.60E-02}$	$0.6869_{1.19E-01}$	$0.5673_{8.46E-02}$	$0.4793_{1.00E-01}$
w20 d9	$0.8539_{2.47E-02}$	$0.7628_{2.34E-02}$	$0.7479_{1.98E-02}$	$0.6150_{1.01E-01}$	$0.5219_{1.19E-01}$	$0.4649_{9.72E-02}$
w50 d1	$0.8762_{2.07E-02}$	$0.7921_{3.40E-02}$	$0.7509_{1.58E-02}$	$0.6865_{5.56E-02}$	$0.5790_{1.19E-01}$	$0.4304_{1.07E-01}$
w50 d3	$0.9022_{1.89E-02}$	$0.8082_{2.80E-02}$	$0.7074_{8.80E-02}$	$0.7056_{9.04E-02}$	$0.5656_{1.20E-01}$	$0.4375_{9.32E-02}$
w50 d6	$0.8838_{1.22E-02}$	$0.8105_{1.52E-02}$	$0.7175_{5.85E-02}$	$0.6798_{1.04E-01}$	$0.5770_{1.16E-01}$	$0.4462_{1.01E-01}$
w50 d9	$0.8920_{1.86E-02}$	$0.8001_{2.05E-02}$	$0.7001_{5.67E-02}$	$0.7040_{8.13E-02}$	$0.5330_{1.04E-01}$	$0.4587_{1.08E-01}$
w100 d1	$0.8994_{2.06E-02}$	$0.8171_{4.00E-02}$	$\mathbf{0.7881_{1.45E-02}}$	$0.7098_{1.02E-01}$	$0.5118_{1.07E-01}$	$\mathbf{0.5218_{1.08E-01}}$
w100 d3	$\mathbf{0.9064_{1.67E-02}}$	$\mathbf{0.8329_{1.35E-02}}$	$0.7537_{6.40E-02}$	$0.6879_{1.17E-01}$	$0.5345_{1.06E-01}$	$0.4263_{1.52E-01}$
w100 d6	$0.8961_{1.53E-02}$	$0.8316_{2.10E-02}$	$0.7784_{2.32E-02}$	$0.6774_{8.96E-02}$	$0.5683_{7.39E-02}$	$0.4441_{9.67E-02}$
w100 d9	$0.8309_{1.81E-02}$	$0.8142_{2.60E-02}$	$0.7849_{1.83E-02}$	$0.6902_{1.13E-01}$	$0.5829_{1.01E-01}$	$0.4100_{7.62E-02}$

Table 5.5: Adapting the network architecture. Median for ZDT1 and ZDT4. 3.000 and 30.000 evaluations were used, respectively.

ZDT1 is a very easy problem with a simple unimodal fitness landscape that should be easy to approximate and search. These expectations are met if we look at Table 5.5. The surrogate algorithms clearly outperform standard NSGA-II, again, with wider network architectures being more indicative of good performance than the depth of the network alone. The related figures 5.3a through 5.3c visualize this advantage.

In ZDT4 the first objective is unimodal and the second is multimodal. This is the reason for why sometimes the plots for the algorithms (figures 5.3e through 5.3f) can be hard to see. The MOEAs quickly find the optimal values for f_1 but take longer for the second objective. Therefore, the plotted fronts are squashed into the upper left corner of the plots. A look at Table 5.5 tells us that for "ZDT4 v30" standard NSGA-II is the best performing MOEA followed only by "w100 d1" and "w100 d9" and no other surrogate algorithm. They both use wider networks, but "w100 d3" and "w100 d6" also have 100 neurons in their hidden layers and more depth, still they are significantly worse than the standard NSGA-II. Both higher-dimensional cases of ZDT4 have a surrogate algorithm as the top performer. These are "w10 d9" for ZDT4 v50" and "w100 d1" for "ZDT v70". But NSGA-II is not significantly outperformed by them. There are also many more surrogate algorithms that perform similarly good on these problems. A clear pattern, however, does not present itself. Neither width nor depth seems to predict better performing surrogates, in these two cases.

One question that arises here is: Why are any of the surrogate variants worse than the basic NSGA-II if the mergeback (see Section 4.2) procedure makes sure that the found Pareto-optimal solutions never deteriorate?
To understand this phenomenon one has to look at the current setup concerning rotation between real and surrogate evaluations. Currently $p \cdot k$ real evaluations are needed, in this experiment that are exactly $5 \cdot 200 = 1000$ real evaluations, to build a new surrogate model. Then the surrogate model is exploited for q generations. After the surrogate phase, we use the mergeback procedure. The mergeback procedure needs another k evaluations to reevaluate all k individuals in the population. The point here is that those k evaluations are not contributing towards finding the Pareto-optimal front, they only reevaluate solutions we already have. The mergeback procedure verifies that the solutions are correct, but "wastes" 200 extra real evaluations to do so.
Therefore every algorithm using our surrogate model "loses" up to 20% of its evaluation budget in the hope that the exploitation of the surrogate makes up for this "loss" and even improve the result on top of that. With the current hyperparameter settings, especially the current number of learning and surrogate generations, most variants are not or just barely achieving that goal, except for the simpler problems.

As such, we can class the results into three groups: group A where the standard NSGA-II is (most often) the best algorithm, and there are only some surrogate algorithms that perform similarly (DTLZ1 v30, all WFG1, and ZDT4 v30). Group A is the group where only some surrogate algorithms have made up for the "lost" evaluations. Group B, where some of the surrogate algorithms outperform standard NSGA-II (all DTLZ2 and ZDT1), the group where the exploitation of the surrogate made up for the "wasted" evaluations and more. Finally, a group C where most algorithms performed similarly to the best algorithms (DTLZ1 v50, DTLZ1 v70, all WFG2, ZDT4 v50 and ZDT4 v70).

In Group A, there is not a clear pattern emerging where one group of similar hyperparameter settings is consistently better than other configurations. If we assume that there are optimal hyperparameter settings for each problem we see examples like "w20 d9" on DTLZ1 which is among the best algorithms for each number of variables but "w20 d3" is only among the best for only "DTLZ1 v30" but not the higher-dimensional cases of "DTLZ1 v30".

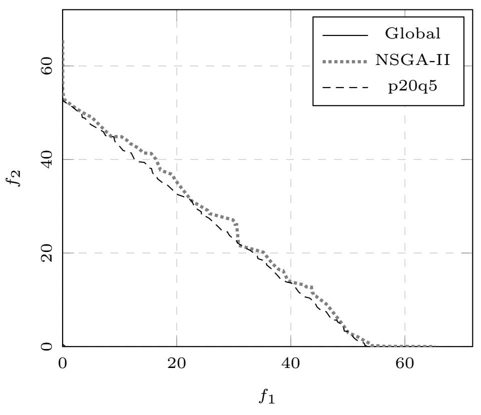
WFG1 is a bit of an outlier since the WFG1s with all dimensions are dominated by the standard NSGA-II, unlike DTLZ1 and ZDT4 where NSGA-II is only clearly better in the low-dimensional variants. Among these problems, WFG1 is the most difficult, because it contains flat regions. In this type of fitness landscape, a surrogate model cannot help an EA to find a non-flat region. If the training set contains only samples from the flat regions, the resulting surrogate will most likely also be completely flat. Even in regions where there is an optimum, simply because the training algorithm has not seen examples that tells him that there is one. If this surrogate is then subsequently used, there is no chance to find the optimum even if by chance one solution lies in an optimal region. If a solution which is from a non-flat region is evaluated with a surrogate that was not trained with examples from its region the surrogate would give it an objective value from the flat region. This way, the surrogate would mislead the EA to think there is no optimum. That would mean there is a lower chance for the surrogate algorithms to recover the lost 20% of evaluations. The underlying NSGA-II has to first find a non-flat region by himself, only then can the surrogate model be expected to be helpful.

Group B, where surrogate algorithm clearly outperforms the standard NSGA-II, is made up of all DTLZ2 and ZDT1 problems, the easiest of the chosen problems. They are both unimodal, and other difficulties like deceptiveness or flat regions are not present. This indicates that the current hyperparameter settings are not suitable for the multimodal problems. The problem here seems to be similar to that of problems with flat fitness landscapes but less pronounced. With the samples the EA has at any one time the EA can only approximate the best optimum currently visible in the training data and help the optimizer to get closer to that local optimum. But unless the neural network training data does contain samples of better regions beyond this local optimum the exploitation of the surrogate model cannot help to guide the optimizer into better regions. That means that the standard NSGA-II has to find the better regions itself.

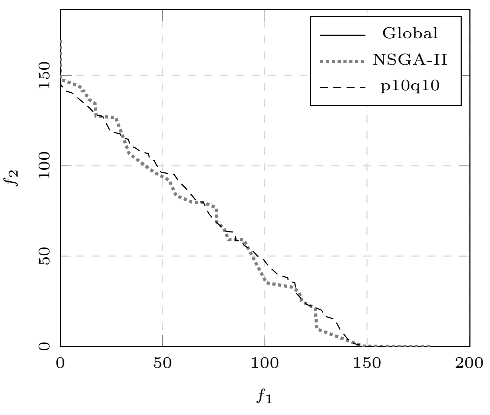
Group C contains the DTLZ1, WFG2 and ZDT4 problems. In most of these cases, one surrogate algorithm is the best but is not significantly better than the standard NSGA-II. So the lost 20% were recovered, but the surrogate could not be exploited to improve the result and outperform standard NSGA-II. To achieve that goal the hyperparameters have to be tuned further. According to our initial test runs, the number of learning and surrogate generations (p and q) are the most likely hyperparameters to do that. Increasing the learning generations will give the ANN more training examples to learn from and improve the quality of the surrogates. Increasing the surrogate generations will allow for more prolonged exploitation of the surrogate. Therefore, in the next experiment, we vary the learning and surrogate generation parameters. For the number of hidden layers, we chose six, and each hidden layer has 100 neurons. This configuration was among the top-performing architectures across all problems in this experiment.

5.3.2 Influence of Training and Exploitation of the Surrogate on a Deep Network

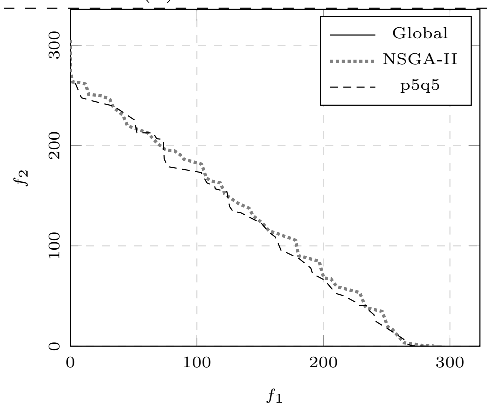
This experiment is designed to determine if the performance of most promising network structure of the previous experiment can be improved by varying the parameters for the learning generations p and surrogate generations q. As a reminder, the number of learning generations sets for how many generations the training data is collected. Each learning generation increases the number of training examples by the individuals in the population k. In our case $k = 200$, so going from five learning generations to ten increases the training set from 1000 to 2000 examples. More training examples lead to better approximations. However, generating too many training examples with real function evaluations can negatively impact the overall evaluation budget. The surrogate generations determine for how many generations the surrogate function is used. Increasing the surrogate generations could enhance the performance, but could also lead the MOEA into regions that are not adequately approximated. Searching in those regions will produce objective vectors with too many errors and could ultimately decrease performance. Both parameters were tested for the values of 5, 10, 15 and 20. In the Tables, p is used to abbreviate the learn generations and q to abbreviate the number of surrogate generations. For example "p5 q20" stands for a variant that collects training data for five generations and then uses the surrogate function for 20 generations. Since we used $p = 5$ and $q = 5$ in the previous experiment, the algorithm with the abbreviation "p5 q5" is the same as "w100 d6" in the previous experiment. The number of hidden layers was set to six layers, and each layer has 100 neurons. The results of this



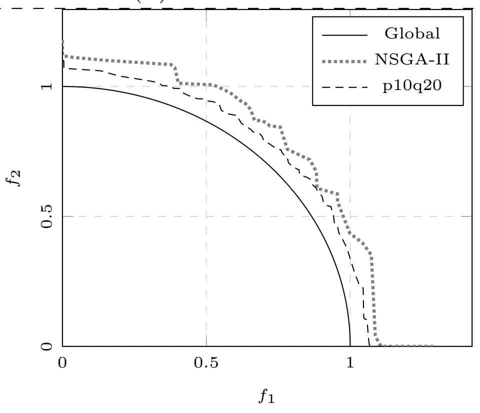
(a) DTLZ1 v30



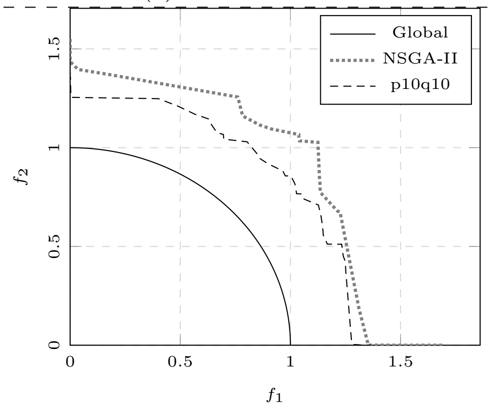
(b) DTLZ1 v50



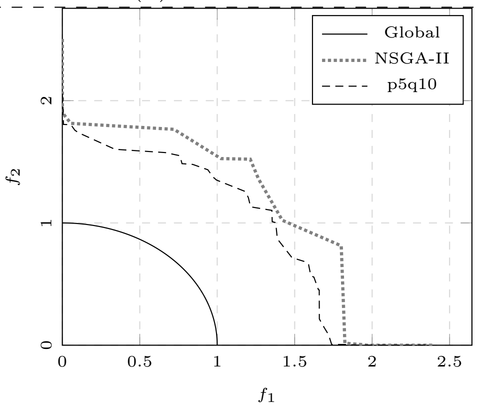
(c) DTLZ2 v70



(d) DTLZ2 v30

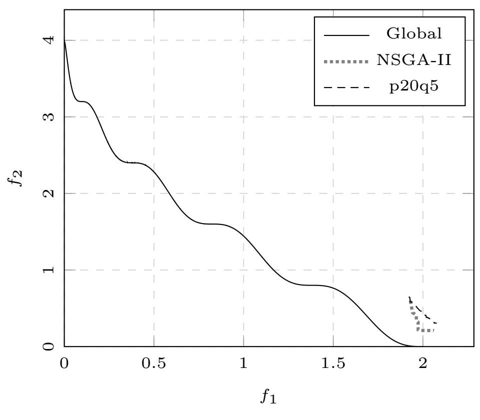


(e) DTLZ2 v50

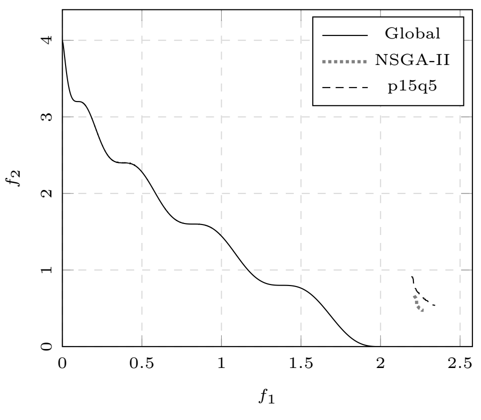


(f) DTLZ2 v70

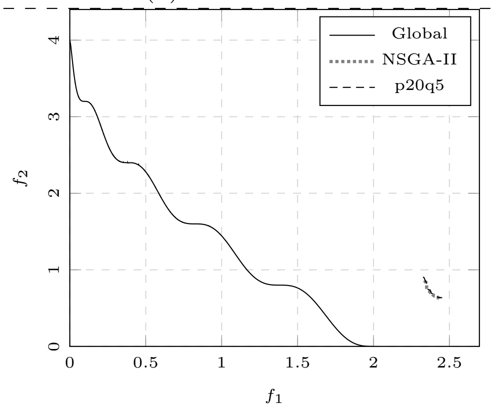
Figure 5.4: Adapting p and q for deep networks. Plots for the DTLZ1 and DTLZ2 problems.



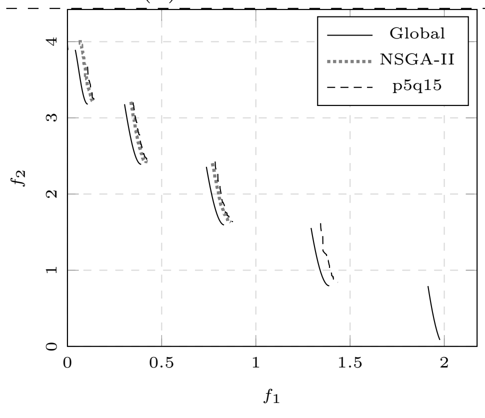
(a) WFG1 v30



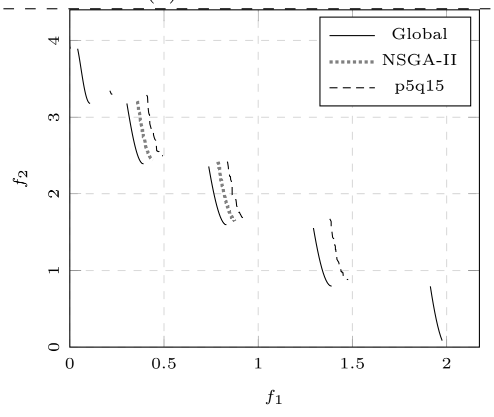
(b) WFG1 v50



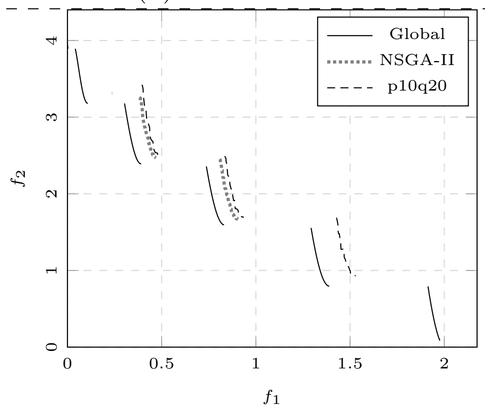
(c) WFG1 v70



(d) WFG2 v30

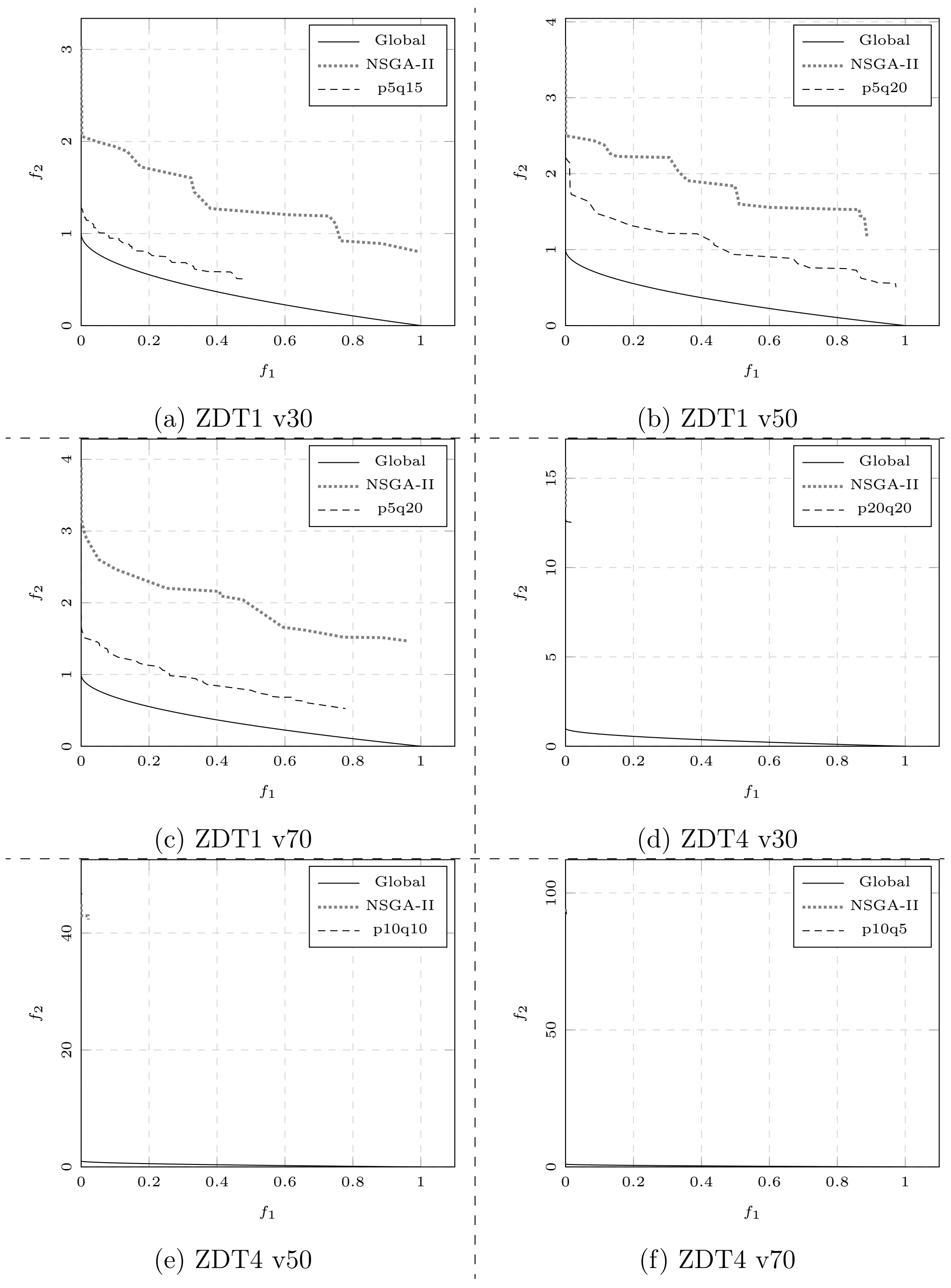


(e) WFG2 v50



(f) WFG2 v70

Figure 5.5: Adapting p and q for deep networks. Plots for the WFG1 and WFG2 problems.



(a) ZDT1 v30
(b) ZDT1 v50
(c) ZDT1 v70
(d) ZDT4 v30
(e) ZDT4 v50
(f) ZDT4 v70

Figure 5.6: Adapting p and q for deep networks. Plots for the ZDT1 and ZDT4 problems.

	DTLZ1 v30	DTLZ1 v50	DTLZ1 v70	DTLZ2 v30	DTLZ2 v50	DTLZ2 v70
NSGAII	$0.9971_{1.64E-03}$	$\mathbf{0.9854_{6.08E-03}}$	$0.9656_{1.74E-02}$	$0.9463_{8.33E-03}$	$0.9029_{2.09E-02}$	$0.8614_{1.28E-02}$
p5 q5	$0.9966_{1.83E-03}$	$0.9825_{6.86E-03}$	$\mathbf{0.9669_{5.96E-03}}$	$0.9516_{1.23E-02}$	$0.9010_{1.78E-02}$	$0.8866_{1.93E-02}$
p5 q10	$0.9962_{2.26E-03}$	$0.9806_{9.25E-03}$	$0.9553_{1.73E-02}$	$0.9649_{1.06E-02}$	$0.8818_{1.95E-02}$	$\mathbf{0.8969_{2.71E-02}}$
p5 q15	$0.9963_{1.89E-03}$	$0.9788_{1.01E-02}$	$0.9630_{1.03E-02}$	$0.9588_{1.96E-02}$	$0.9018_{2.24E-02}$	$0.8397_{2.15E-02}$
p5 q20	$0.9957_{2.93E-03}$	$0.9833_{6.25E-03}$	$0.9538_{2.01E-02}$	$0.9422_{1.84E-02}$	$0.8844_{2.01E-02}$	$0.8374_{2.03E-02}$
p10 q5	$0.9969_{1.38E-03}$	$0.9828_{1.15E-02}$	$0.9630_{1.27E-02}$	$0.9631_{7.78E-03}$	$0.9229_{1.91E-02}$	$0.8594_{2.99E-02}$
p10 q10	$0.9967_{1.30E-03}$	$0.9853_{5.96E-03}$	$0.9656_{5.55E-03}$	$0.9559_{2.12E-02}$	$\mathbf{0.9352_{1.36E-02}}$	$0.8790_{1.97E-02}$
p10 q15	$0.9969_{2.56E-03}$	$0.9830_{6.85E-03}$	$0.9645_{1.16E-02}$	$0.9500_{1.75E-02}$	$0.8984_{9.54E-03}$	$0.8946_{3.22E-02}$
p10 q20	$0.9965_{2.10E-03}$	$0.9830_{8.85E-03}$	$0.9619_{1.23E-02}$	$\mathbf{0.9703_{1.08E-02}}$	$0.9007_{1.94E-02}$	$0.8625_{3.06E-02}$
p15 q5	$0.9969_{2.59E-03}$	$0.9836_{1.09E-02}$	$0.9557_{1.66E-02}$	$0.9566_{6.52E-03}$	$0.9089_{1.81E-02}$	$0.8736_{1.96E-02}$
p15 q10	$0.9969_{1.89E-03}$	$0.9822_{6.50E-03}$	$0.9606_{1.47E-02}$	$0.9370_{1.44E-02}$	$0.9253_{1.60E-02}$	$0.8812_{1.76E-02}$
p15 q15	$0.9964_{2.29E-03}$	$0.9825_{1.03E-02}$	$0.9626_{1.51E-02}$	$0.9608_{2.25E-02}$	$0.9267_{1.36E-02}$	$0.8854_{2.24E-02}$
p15 q20	$0.9968_{1.65E-03}$	$0.9837_{9.53E-03}$	$0.9611_{1.23E-02}$	$0.9430_{7.83E-03}$	$0.9010_{1.18E-02}$	$0.8591_{1.85E-02}$
p20 q5	$\mathbf{0.9972_{1.67E-03}}$	$0.9838_{5.97E-03}$	$0.9576_{1.44E-02}$	$0.9553_{6.80E-03}$	$0.9163_{1.43E-02}$	$0.8761_{2.06E-02}$
p20 q10	$0.9969_{1.80E-03}$	$0.9825_{7.25E-03}$	$0.9536_{2.43E-02}$	$0.9584_{6.72E-03}$	$0.9195_{1.49E-02}$	$0.8531_{1.45E-02}$
p20 q15	$0.9969_{1.26E-03}$	$0.9824_{1.05E-02}$	$0.9616_{1.88E-02}$	$0.9547_{1.21E-02}$	$0.9203_{1.12E-02}$	$0.8654_{2.25E-02}$
p20 q20	$0.9969_{1.74E-03}$	$0.9831_{3.40E-03}$	$0.9627_{1.75E-02}$	$0.9554_{8.89E-03}$	$0.9280_{1.61E-02}$	$0.8573_{2.46E-02}$

Table 5.6: Adapting p and q for deep networks. Median for DTLZ1 and DTLZ2. 30.000 and 5.000 evaluations were used, respectively.

experiment can be seen int tables 5.6 and 5.7 and 5.8 and figures 5.4, 5.5 and 5.6.

Let us start again with DTLZ1 in Table 5.6. Many surrogate algorithms can adequately approximate "DTLZ1 v30" and "DTLZ1 v50". None of the surrogate algorithms did outperform the standard NSGA-II. In "DTLZ v30" it seems that using more than ten learning generations improves the quality so much that these surrogate variants are consistently achieving performance similar to NSGA-II. The same trend seems to also show in "DTLZ1 v50", with some exceptions ("p15 q10" and "p20 q20"). The plot for "DTLZ1 v30" in Figure 5.4a shows the slight advantage of the surrogate over standard NSGA-II. Figure 5.4b shows that the performance of the best shows the slight edge of the standard NSGA-II.

In "DTLZ1 v70" (Table 5.6) we can see that adjusting p and q seems to not improve the performance on higher-dimensional problems. The surrogate algorithm "p5 q5" ("w100 d6" in the previous experiment) is the best-performing MOEA, even though "100 d6" was not among the best-performing algorithms in the first experiment (see Table 5.3) for "DTLZ1 v70".

DTLZ2 was dominated by "w100 d6" in the first experiment (see Table 5.3) which meant it was already easy to approximate, and adapting the learning and surrogate generations increased the performance of this network configuration

	WFG1 v30	WFG1 v50	WFG1 v70	WFG2 v30	WFG2 v50	WFG2 v70
NSGAII	$\mathbf{0.3554_{3.61E-02}}$	$0.1707_{2.47E-02}$	$0.1212_{1.66E-02}$	$0.6948_{9.21E-03}$	$0.6839_{1.12E-02}$	$0.6712_{1.14E-02}$
p5 q5	$0.3014_{4.80E-02}$	$0.1414_{2.08E-02}$	$0.1145_{1.65E-02}$	$0.6991_{1.05E-02}$	$0.6842_{5.83E-03}$	$0.6712_{8.65E-03}$
p5 q10	$0.2603_{2.72E-02}$	$0.1288_{1.93E-02}$	$0.0900_{2.10E-02}$	$0.7006_{7.45E-03}$	$0.6841_{1.04E-02}$	$0.6706_{1.01E-02}$
p5 q15	$0.2570_{2.80E-02}$	$0.1226_{2.29E-02}$	$0.0830_{1.48E-02}$	$\mathbf{0.8562_{5.22E-03}}$	$\mathbf{0.8279_{1.54E-01}}$	$0.6724_{7.01E-03}$
p5 q20	$0.2532_{2.40E-02}$	$0.1204_{1.64E-02}$	$0.0969_{1.77E-02}$	$0.6990_{4.13E-03}$	$0.6839_{6.38E-03}$	$0.6695_{1.71E-02}$
p10 q5	$0.3270_{4.73E-02}$	$0.1601_{2.88E-02}$	$0.1208_{1.26E-02}$	$0.6958_{1.07E-02}$	$0.6836_{9.31E-03}$	$0.6729_{5.91E-03}$
p10 q10	$0.2912_{2.37E-02}$	$0.1640_{2.74E-02}$	$0.0992_{1.43E-02}$	$0.6992_{5.61E-03}$	$0.6827_{7.37E-03}$	$0.6703_{1.10E-02}$
p10 q15	$0.2950_{2.22E-02}$	$0.1586_{3.04E-02}$	$0.1127_{1.57E-02}$	$0.6999_{5.86E-03}$	$0.6821_{1.03E-02}$	$0.6746_{7.64E-03}$
p10 q20	$0.2956_{3.34E-02}$	$0.1462_{1.42E-02}$	$0.1026_{2.12E-02}$	$0.6998_{8.07E-03}$	$0.6839_{6.53E-03}$	$\mathbf{0.8081_{1.36E-01}}$
p15 q5	$0.3386_{4.75E-02}$	$\mathbf{0.1716_{2.21E-02}}$	$0.1239_{1.00E-02}$	$0.6995_{1.05E-02}$	$0.6855_{6.75E-03}$	$0.6700_{1.07E-02}$
p15 q10	$0.3214_{3.77E-02}$	$0.1673_{2.08E-02}$	$0.1149_{1.40E-02}$	$0.7004_{5.35E-03}$	$0.6847_{1.30E-02}$	$0.6719_{6.75E-03}$
p15 q15	$0.3315_{3.81E-02}$	$0.1593_{2.36E-02}$	$0.1100_{1.22E-02}$	$0.7016_{8.37E-03}$	$0.6827_{6.92E-03}$	$0.6717_{9.62E-03}$
p15 q20	$0.3088_{4.79E-02}$	$0.1537_{2.49E-02}$	$0.1175_{1.63E-02}$	$0.6992_{9.31E-03}$	$0.6839_{7.39E-03}$	$0.6716_{5.49E-03}$
p20 q5	$0.3424_{3.30E-02}$	$0.1625_{2.71E-02}$	$\mathbf{0.1248_{1.49E-02}}$	$0.6999_{5.80E-03}$	$0.6798_{8.55E-03}$	$0.6695_{9.00E-03}$
p20 q10	$0.3126_{3.80E-02}$	$0.1696_{1.25E-02}$	$0.1191_{1.38E-02}$	$0.6973_{9.25E-03}$	$0.6832_{7.54E-03}$	$0.6725_{1.03E-02}$
p20 q15	$0.3272_{4.66E-02}$	$0.1669_{1.99E-02}$	$0.1163_{1.78E-02}$	$0.6997_{6.53E-03}$	$0.6839_{6.06E-03}$	$0.7994_{1.44E-01}$
p20 q20	$0.3347_{4.15E-02}$	$0.1626_{1.92E-02}$	$0.1152_{1.69E-02}$	$0.7005_{7.35E-03}$	$0.6814_{6.19E-03}$	$0.6716_{8.99E-03}$

Table 5.7: Adapting p and q for deep networks. Median for WFG1 and WFG2. With WFG1 30.000 and 20.000 evaluations were used, respectively.

further. The figures 5.4e through 5.4f also show how more hypervolume is covered if compared to figures 5.1e through 5.1a.

Overall, adapting the learning and surrogate generations seems to improve their performance on WFG1 (see Table 5.7). Still, the standard NSGA-II is not clearly outperformed, irrespective of the chosen number of search variables.

An interesting observation can be made in the WFG2 problem, see Table 5.7. The hypervolume of the best surrogate variant is increased by 13% compared to the standard NSGA-II. The reason for this increase can be found by comparing the plots of WFG2 fronts (for example figure 5.5d and figure 5.2d). The best surrogate variant is better at approximating the disconnected Pareto-front of the WFG2 function. There are five disconnected sections of the Pareto-front, both NSGA-II and our method find the second and third section and small parts of the first section, depending on the number of search variables. Some surrogate-assisted algorithms which exploit the surrogate function for more generations (either for 15 generations or 20 generations) appear to be better at finding the fourth section of the Pareto-front. For "WFG2 v30" and "WFG2 v50" it seems that using five learning generations is enough and using more (see "p10 q15") actually degrades the performance. The same appears to be the case for the surrogate generations, the algorithm "p5 q20" appar-

	ZDT1 v30	ZDT1 v50	ZDT1 v70	ZDT4 v30	ZDT4 v50	ZDT4 v70
NSGAII	$0.8368_{2.20E-02}$	$0.7440_{2.70E-02}$	$0.7022_{2.11E-02}$	$0.7279_{1.12E-01}$	$\mathbf{0.6450_{8.31E-02}}$	$\mathbf{0.5354_{1.13E-01}}$
p5 q5	$0.9038_{1.43E-02}$	$0.8284_{2.11E-02}$	$0.6834_{2.30E-02}$	$0.6961_{7.47E-02}$	$0.5890_{1.39E-01}$	$0.4240_{1.07E-01}$
p5 q10	$0.8210_{3.06E-02}$	$0.8665_{2.71E-02}$	$0.7948_{2.26E-02}$	$0.6851_{1.35E-01}$	$0.4966_{1.00E-01}$	$0.4606_{1.36E-01}$
p5 q15	$\mathbf{0.9105_{2.61E-02}}$	$0.8797_{2.59E-02}$	$0.8687_{1.91E-02}$	$0.6626_{1.27E-01}$	$0.4779_{1.26E-01}$	$0.4810_{8.71E-02}$
p5 q20	$0.8154_{1.52E-02}$	$\mathbf{0.8844_{1.40E-02}}$	$\mathbf{0.8968_{3.46E-02}}$	$0.6226_{1.93E-01}$	$0.5953_{7.80E-02}$	$0.4581_{1.08E-01}$
p10 q5	$0.8648_{2.25E-02}$	$0.7916_{1.89E-02}$	$0.7570_{1.83E-02}$	$0.7333_{9.56E-02}$	$0.5674_{1.03E-01}$	$0.5343_{8.87E-02}$
p10 q10	$0.8345_{3.15E-02}$	$0.7244_{3.02E-02}$	$0.6973_{2.74E-02}$	$0.6882_{6.32E-02}$	$0.6092_{1.05E-01}$	$0.4992_{9.39E-02}$
p10 q15	$0.8937_{1.07E-02}$	$0.8528_{2.89E-02}$	$0.8178_{2.56E-02}$	$0.6929_{1.29E-01}$	$0.5707_{1.27E-01}$	$0.4854_{1.27E-01}$
p10 q20	$0.8929_{1.61E-02}$	$0.8467_{1.61E-02}$	$0.8576_{2.13E-02}$	$0.7080_{6.20E-02}$	$0.5782_{1.03E-01}$	$0.4944_{1.45E-01}$
p15 q5	$0.8351_{1.68E-02}$	$0.7340_{1.68E-02}$	$0.7040_{2.09E-02}$	$0.7272_{7.97E-02}$	$0.5914_{1.08E-01}$	$0.5093_{7.82E-02}$
p15 q10	$0.8297_{1.67E-02}$	$0.7372_{2.97E-02}$	$0.7105_{2.59E-02}$	$0.6988_{7.80E-02}$	$0.6009_{8.00E-02}$	$0.4910_{8.80E-02}$
p15 q15	$0.8291_{2.34E-02}$	$0.7330_{2.48E-02}$	$0.7056_{1.85E-02}$	$0.7170_{9.31E-02}$	$0.5718_{8.19E-02}$	$0.4998_{7.39E-02}$
p15 q20	$0.8342_{2.16E-02}$	$0.7393_{2.63E-02}$	$0.7020_{2.25E-02}$	$0.7209_{6.37E-02}$	$0.5680_{1.02E-01}$	$0.5176_{1.05E-01}$
p20 q5	$0.8325_{1.55E-02}$	$0.7394_{2.36E-02}$	$0.7120_{2.32E-02}$	$0.7200_{6.44E-02}$	$0.5886_{9.74E-02}$	$0.5297_{6.58E-02}$
p20 q10	$0.8260_{2.00E-02}$	$0.7388_{2.17E-02}$	$0.7038_{2.37E-02}$	$0.7416_{7.90E-02}$	$0.6060_{9.41E-02}$	$0.4937_{8.92E-02}$
p20 q15	$0.8399_{2.02E-02}$	$0.7333_{2.13E-02}$	$0.7060_{2.04E-02}$	$0.6984_{1.04E-01}$	$0.5952_{5.69E-02}$	$0.4960_{1.20E-01}$
p20 q20	$0.8386_{2.00E-02}$	$0.7235_{2.92E-02}$	$0.7006_{1.90E-02}$	$\mathbf{0.7451_{1.02E-01}}$	$0.6029_{7.20E-02}$	$0.5337_{1.21E-01}$

Table 5.8: Adapting p and q for deep networks. Median for ZDT1 and ZDT4. 3.000 and 30.000 evaluations were used, respectively.

ently moves into regions were the approximation is poor. WFG2 seems to be a problem where there is a high sensitivity to the p and q hyperparameters.

The performance on ZDT1 could be improved upon as seen in Table 5.8. It seems that five learning generations are enough for a good approximation of ZDT1 and increasing the surrogate generations is the primary predictor for top-performing surrogates.

Like in DTLZ1 adjusting p and q seems to increase the performance of the "w100 d6" network architecture in the ZDT4 problem (see Table 5.8), but is also not enough to clearly outperform the standard NSGA-II.

Overall it seems the learning generations and surrogate generations can be used to improve the performance, as expected. The most eye-catching examples are WFG2, where using the adjusted surrogate allowed the MOEA to find more parts of the disconnected Pareto-front. In the ZDT1 problem, which was easy to approximate, the fine-tuning of p and q increases performance predictably. However, we still do not see significant advantage of the surrogate-assisted variants on the highly multimodal problems like DTLZ4 or DTLZ1.

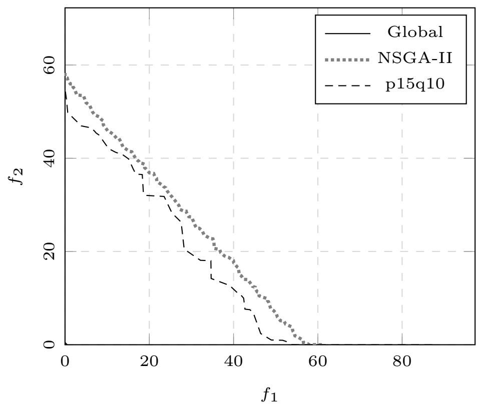
	DTLZ1 v30	DTLZ1 v50	DTLZ1 v70	DTLZ2 v30	DTLZ2 v50	DTLZ2 v70
NSGAII	$0.9968_{1.99E-03}$	$0.9833_{6.88E-03}$	$0.9601_{1.90E-02}$	$0.9418_{1.42E-02}$	$0.9003_{1.73E-02}$	$\mathbf{0.8658_{2.42E-02}}$
p5 q5	$0.9961_{3.20E-03}$	$0.9774_{1.07E-02}$	$0.9522_{2.18E-02}$	$0.9349_{1.10E-02}$	$0.8773_{1.37E-02}$	$0.8496_{2.15E-02}$
p5 q10	$0.9952_{1.97E-03}$	$0.9781_{1.79E-02}$	$0.9434_{2.18E-02}$	$0.9241_{1.88E-02}$	$0.8724_{2.11E-02}$	$0.8247_{2.03E-02}$
p5 q15	$0.9960_{2.62E-03}$	$0.9760_{1.06E-02}$	$0.9397_{2.38E-02}$	$0.9316_{1.19E-02}$	$0.8609_{1.43E-02}$	$0.8036_{1.79E-02}$
p5 q20	$0.9957_{2.87E-03}$	$0.9780_{1.37E-02}$	$0.9440_{2.30E-02}$	$0.9154_{1.34E-02}$	$0.8760_{1.96E-02}$	$0.8034_{3.22E-02}$
p10 q5	$0.9965_{2.13E-03}$	$0.9823_{1.07E-02}$	$0.9525_{2.71E-02}$	$0.9388_{1.39E-02}$	$0.8939_{1.82E-02}$	$0.8499_{1.84E-02}$
p10 q10	$0.9956_{2.39E-03}$	$0.9810_{8.77E-03}$	$0.9528_{2.15E-02}$	$0.9387_{1.43E-02}$	$0.8884_{1.19E-02}$	$0.8552_{2.43E-02}$
p10 q15	$0.9965_{1.53E-03}$	$0.9798_{8.76E-03}$	$0.9558_{2.00E-02}$	$0.9385_{1.25E-02}$	$0.8862_{2.53E-02}$	$0.8471_{2.53E-02}$
p10 q20	$0.9966_{1.91E-03}$	$0.9804_{5.85E-03}$	$0.9517_{2.42E-02}$	$0.9378_{1.30E-02}$	$0.8827_{2.31E-02}$	$0.8450_{2.25E-02}$
p15 q5	$0.9969_{2.05E-03}$	$0.9818_{7.50E-03}$	$0.9559_{2.24E-02}$	$\mathbf{0.9441_{9.90E-03}}$	$0.9001_{1.25E-02}$	$0.8566_{1.72E-02}$
p15 q10	$\mathbf{0.9973_{1.96E-03}}$	$0.9806_{1.03E-02}$	$0.9558_{1.71E-02}$	$0.9408_{1.12E-02}$	$\mathbf{0.9019_{2.34E-02}}$	$0.8578_{1.93E-02}$
p15 q15	$0.9969_{1.65E-03}$	$0.9822_{5.24E-03}$	$0.9557_{1.64E-02}$	$0.9388_{1.14E-02}$	$0.8979_{1.47E-02}$	$0.8526_{2.01E-02}$
p15 q20	$0.9970_{1.56E-03}$	$0.9819_{8.47E-03}$	$0.9564_{2.36E-02}$	$0.9394_{8.17E-03}$	$0.8961_{1.60E-02}$	$0.8535_{1.44E-02}$
p20 q5	$0.9973_{1.80E-03}$	$\mathbf{0.9843_{7.01E-03}}$	$\mathbf{0.9604_{1.30E-02}}$	$0.9407_{1.10E-02}$	$0.8983_{9.44E-03}$	$0.8638_{1.59E-02}$
p20 q10	$0.9969_{1.86E-03}$	$0.9816_{6.98E-03}$	$0.9585_{1.82E-02}$	$0.9401_{1.06E-02}$	$0.9002_{1.35E-02}$	$0.8570_{2.57E-02}$
p20 q15	$0.9965_{1.80E-03}$	$0.9814_{8.48E-03}$	$0.9603_{2.61E-02}$	$0.9427_{1.28E-02}$	$0.8986_{1.65E-02}$	$0.8528_{1.91E-02}$
p20 q20	$0.9968_{1.75E-03}$	$0.9822_{9.65E-03}$	$0.9490_{2.27E-02}$	$0.9372_{1.20E-02}$	$0.8989_{1.21E-02}$	$0.8581_{1.99E-02}$

Table 5.9: Adapting p and q for shallow networks. Median for DTLZ1 and DTLZ2. 30.000 and 5.000 evaluations were used, respectively.

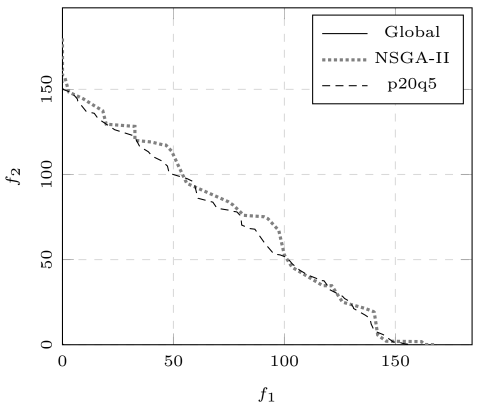
5.3.3 Influence of Training and Exploitation of the Surrogate on a Shallow Network

Here we also want to adjust the learning and surrogate generations for a network with only one hidden layer. The reason is to see if the adjustment of these parameters allows a shallow network to perform similarly to a deep network. The architecture of the network uses one hidden layer with 50 neurons, see "w50 d1" in the first experiment. This architecture is similar to the one described in Nain and Deb[49], who used one hidden layer with 40 neurons.

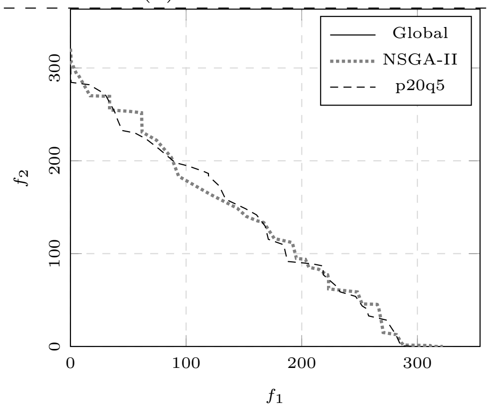
We take a look at the columns for DTLZ1 in Table 5.9. All three cases can be well enough approximated to keep up with the standard NSGA-II. Like the deep network, we see that the shallow network needs at least ten learning generations to achieve a good quality surrogate. However, using the improved approximation for more than five generations does not raise its performance. For example, "p10 q5", which uses ten learning generations and five surrogate generations, is equal or better than "p10 q10" and "p10 q15". Only "p10 q20" is better, but not significantly so. This pattern repeats for all DTLZ1 cases. If we compare the median performance of the best surrogate algorithms between Table 5.6 and Table 5.9, it appears that in higher dimensions deep networks achieve better performance, we will see if that first impression is also true for the other problems.



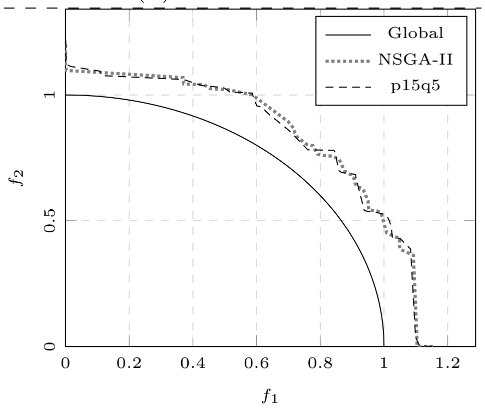
(a) DTLZ1 v30



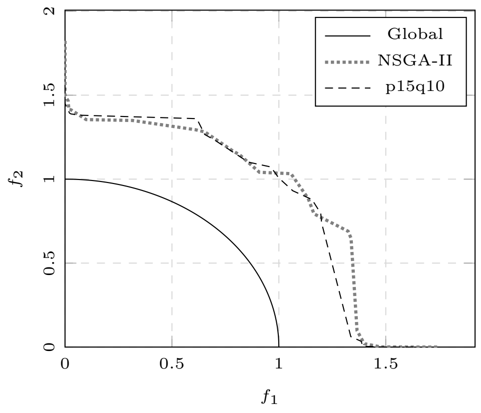
(b) DTLZ1 v50



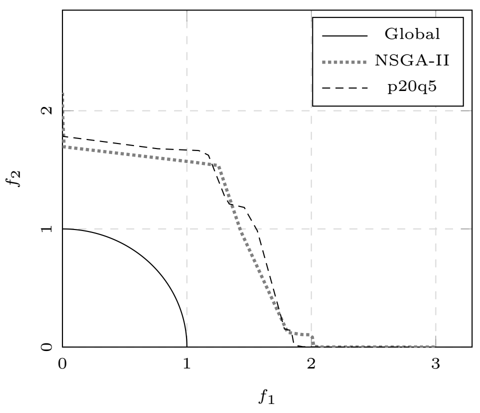
(c) DTLZ2 v70



(d) DTLZ2 v30

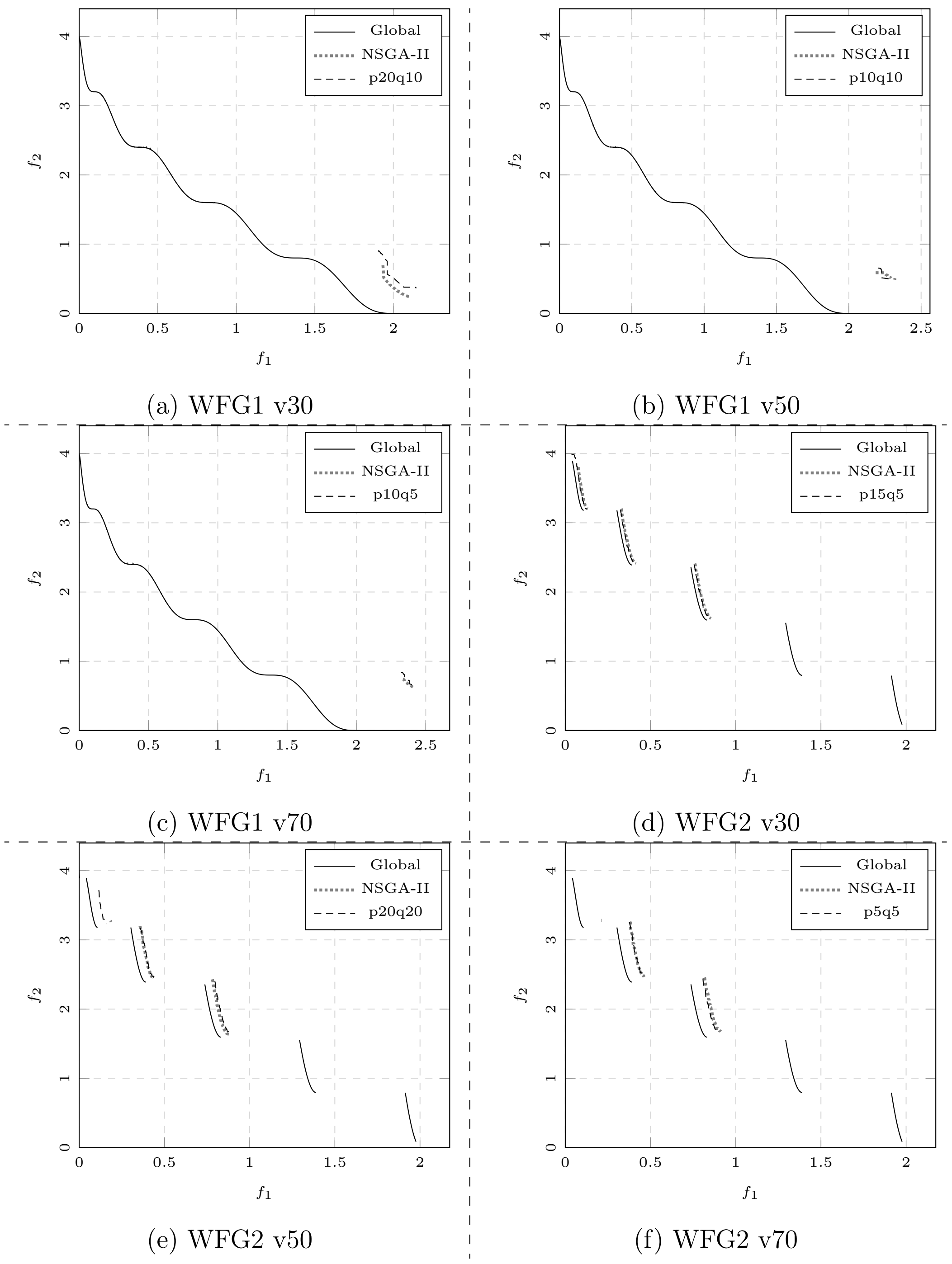


(e) DTLZ2 v50



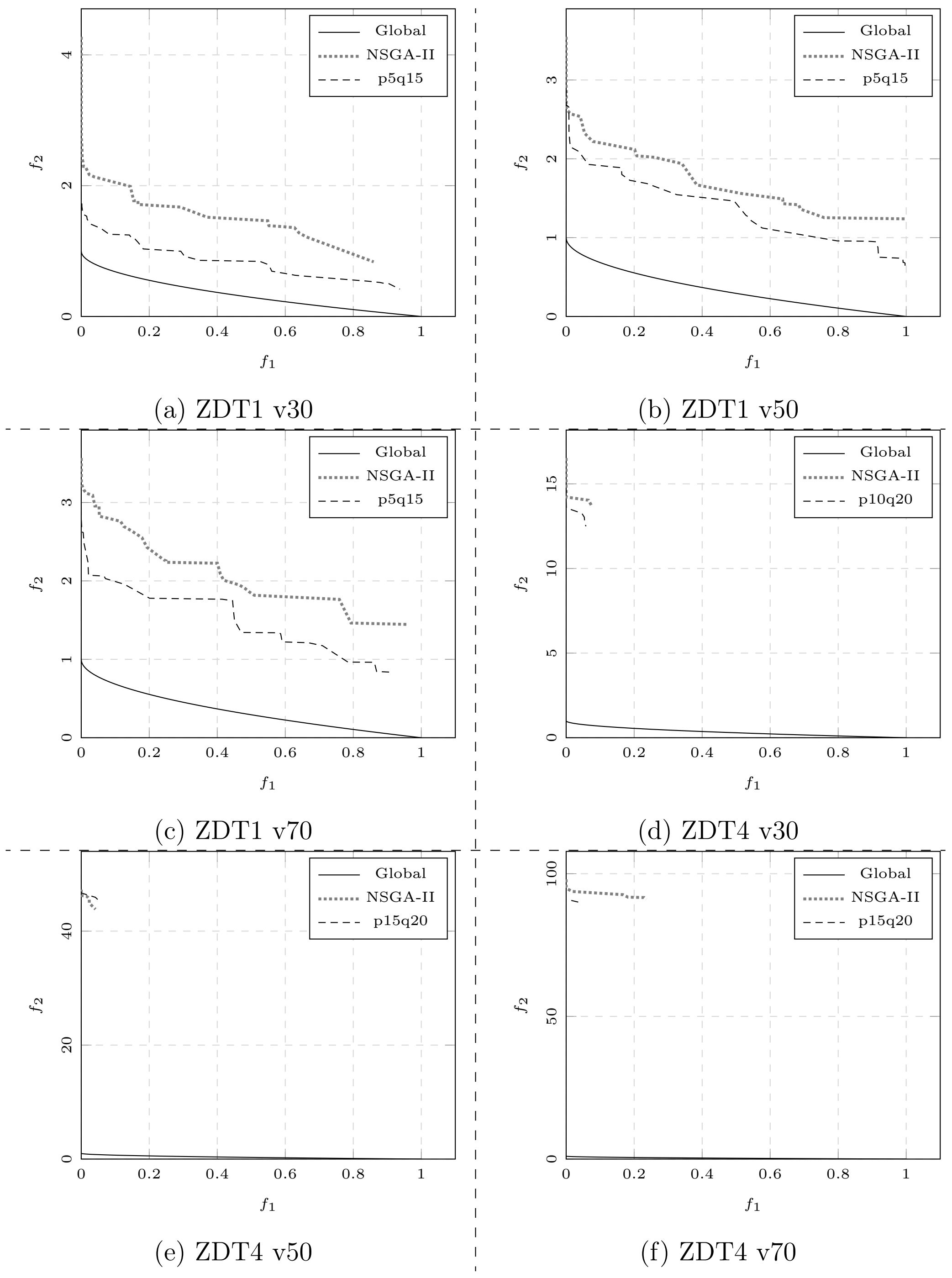
(f) DTLZ2 v70

Figure 5.7: Adapting p and q for shallow networks. Plots for the DTLZ1 and DTLZ2 problems.



(a) WFG1 v30 (b) WFG1 v50 (c) WFG1 v70 (d) WFG2 v30 (e) WFG2 v50 (f) WFG2 v70

Figure 5.8: Adapting p and q for shallow networks. Plots for the WFG1 and WFG2 problems.



(a) ZDT1 v30

(b) ZDT1 v50

(c) ZDT1 v70

(d) ZDT4 v30

(e) ZDT4 v50

(f) ZDT4 v70

Figure 5.9: Adapting p and q for shallow networks. Plots for the ZDT1 and ZDT4 problems.

	WFG1 v30	WFG1 v50	WFG1 v70	WFG2 v30	WFG2 v50	WFG2 v70
NSGAII	$\mathbf{0.3440_{2.32E-02}}$	$\mathbf{0.1828_{2.88E-02}}$	$0.1232_{1.50E-02}$	$0.6991_{7.19E-03}$	$0.6846_{8.18E-03}$	$0.6705_{8.09E-03}$
p5 q5	$0.2995_{3.53E-02}$	$0.1426_{1.62E-02}$	$0.1264_{1.39E-02}$	$0.6964_{7.74E-03}$	$0.6847_{9.77E-03}$	$\mathbf{0.6732_{9.13E-03}}$
p5 q10	$0.2666_{3.06E-02}$	$0.1320_{2.41E-02}$	$0.0873_{1.19E-02}$	$0.6964_{6.77E-03}$	$0.6815_{1.01E-02}$	$0.6682_{6.80E-03}$
p5 q15	$0.2491_{2.67E-02}$	$0.1231_{1.97E-02}$	$0.0861_{1.39E-02}$	$0.6968_{7.89E-03}$	$0.6805_{6.42E-03}$	$0.6637_{1.00E-02}$
p5 q20	$0.2501_{3.51E-02}$	$0.1213_{1.91E-02}$	$0.0806_{1.57E-02}$	$0.6926_{7.24E-03}$	$0.6820_{7.62E-03}$	$0.6653_{8.09E-03}$
p10 q5	$0.3184_{3.68E-02}$	$0.1700_{2.70E-02}$	$\mathbf{0.1268_{1.69E-02}}$	$0.6981_{7.95E-03}$	$0.6843_{9.35E-03}$	$0.6683_{9.44E-03}$
p10 q10	$0.3036_{3.50E-02}$	$0.1738_{2.19E-02}$	$0.1089_{1.30E-02}$	$0.7006_{9.38E-03}$	$0.6832_{7.63E-03}$	$0.6701_{1.07E-02}$
p10 q15	$0.3138_{5.27E-02}$	$0.1505_{1.70E-02}$	$0.1035_{1.63E-02}$	$0.6971_{6.17E-03}$	$0.6830_{8.88E-03}$	$0.6703_{8.58E-03}$
p10 q20	$0.3041_{3.33E-02}$	$0.1516_{1.90E-02}$	$0.1027_{1.72E-02}$	$0.6961_{1.15E-02}$	$0.6820_{1.11E-02}$	$0.6694_{9.42E-03}$
p15 q5	$0.3225_{3.04E-02}$	$0.1671_{2.61E-02}$	$0.1258_{1.45E-02}$	$\mathbf{0.7011_{1.22E-02}}$	$0.6843_{9.95E-03}$	$0.6674_{8.90E-03}$
p15 q10	$0.3240_{4.76E-02}$	$0.1560_{1.86E-02}$	$0.1115_{1.47E-02}$	$0.7007_{4.90E-03}$	$0.6836_{9.70E-03}$	$0.6696_{1.38E-02}$
p15 q15	$0.3149_{6.61E-02}$	$0.1530_{3.20E-02}$	$0.1078_{2.07E-02}$	$0.6991_{7.86E-03}$	$0.6814_{6.11E-03}$	$0.6703_{9.09E-03}$
p15 q20	$0.3182_{3.73E-02}$	$0.1610_{2.30E-02}$	$0.1069_{1.42E-02}$	$0.6989_{4.55E-03}$	$0.6839_{7.17E-03}$	$0.6686_{8.59E-03}$
p20 q5	$0.3290_{3.60E-02}$	$0.1711_{1.88E-02}$	$0.1180_{1.05E-02}$	$0.6994_{6.05E-03}$	$0.6803_{1.19E-02}$	$0.6686_{1.08E-02}$
p20 q10	$0.3356_{3.84E-02}$	$0.1664_{2.58E-02}$	$0.1137_{1.29E-02}$	$0.6979_{8.32E-03}$	$0.6846_{1.13E-02}$	$0.6724_{6.94E-03}$
p20 q15	$0.3269_{4.35E-02}$	$0.1564_{3.45E-02}$	$0.1142_{1.71E-02}$	$0.7006_{6.16E-03}$	$0.6850_{7.31E-03}$	$0.6701_{1.19E-02}$
p20 q20	$0.3150_{3.13E-02}$	$0.1573_{2.20E-02}$	$0.1136_{2.12E-02}$	$0.6991_{1.05E-02}$	$\mathbf{0.6850_{1.22E-02}}$	$0.6656_{9.73E-03}$

Table 5.10: Adapting p and q for shallow networks. Median for WFG1 and WFG2. With WFG1 30.000 and 20.000 evaluations were used, respectively.

DTLZ2 (see Table 5.9) is interesting because with the shallow network it can only achieve a good enough approximation of the fitness landscape if the training data is collected for at least fifteen generations (in the case of "DTLZ1 v70" twenty are needed) and perform similarly to the standard NSGA-II. The deep network (see Table 5.6), however, significantly outperforms NSGA-II with five or ten learning generations and most of the surrogate variants that not outperform NSGA-II have a similar performance to it. It seems like the shallow network only memorizes the DTLZ2 problem and fails if not enough examples are provided, while the deep network appears to have built a simple model of fitness landscape that allow it generalize correctly to unseen regions. The deep network is 3% better than the shallow one on all three DTLZ2 cases.

In the WFG1 problem (see Table 5.10) there are no big differences between the shallow and deep network. This similarity is to be expected, WFG1 has flat landscape, i.e, not a lot of local features, so we do not need an approximator with great expression power.

Unlike, the deep network the shallow network did not enable the MOEA to find the fourth part of WFG2 Pareto-optimal front. This finding indicates that a deep network is needed to adequately approximate the multimodal second objective of WFG2.

	ZDT1 v30	ZDT1 v50	ZDT1 v70	ZDT4 v30	ZDT4 v50	ZDT4 v70
NSGAII	$0.8277_{2.05E-02}$	$0.7352_{2.37E-02}$	$0.7046_{2.09E-02}$	$0.7215_{4.29E-02}$	$\mathbf{0.6321_{8.85E-02}}$	$0.5404_{7.04E-02}$
p5 q5	$0.8815_{1.46E-02}$	$0.8051_{2.11E-02}$	$0.7320_{2.53E-02}$	$0.6079_{1.16E-01}$	$0.4749_{1.19E-01}$	$0.3892_{1.24E-01}$
p5 q10	$0.8768_{2.95E-02}$	$0.8141_{2.19E-02}$	$0.8119_{2.40E-02}$	$0.5981_{1.29E-01}$	$0.5615_{1.01E-01}$	$0.4538_{1.01E-01}$
p5 q15	$\mathbf{0.9162_{2.66E-02}}$	$\mathbf{0.8472_{2.16E-02}}$	$\mathbf{0.8234_{2.49E-02}}$	$0.6500_{1.42E-01}$	$0.5291_{1.16E-01}$	$0.4766_{8.09E-02}$
p5 q20	$0.8886_{2.57E-02}$	$0.7550_{3.18E-02}$	$0.7982_{3.21E-02}$	$0.6186_{9.01E-02}$	$0.5753_{9.24E-02}$	$0.4760_{1.11E-01}$
p10 q5	$0.8702_{2.29E-02}$	$0.7633_{2.29E-02}$	$0.7439_{1.20E-02}$	$0.6930_{1.06E-01}$	$0.6098_{5.79E-02}$	$0.5074_{1.10E-01}$
p10 q10	$0.8926_{1.75E-02}$	$0.7600_{4.02E-02}$	$0.7616_{2.64E-02}$	$0.7032_{7.68E-02}$	$0.5871_{8.56E-02}$	$0.5089_{7.49E-02}$
p10 q15	$0.8383_{2.03E-02}$	$0.7755_{3.75E-02}$	$0.6963_{3.37E-02}$	$0.7268_{5.83E-02}$	$0.6134_{9.87E-02}$	$0.5172_{6.67E-02}$
p10 q20	$0.8452_{3.61E-02}$	$0.7851_{3.52E-02}$	$0.7360_{2.79E-02}$	$\mathbf{0.7452_{7.51E-02}}$	$0.5753_{1.41E-01}$	$0.5138_{1.08E-01}$
p15 q5	$0.8335_{2.31E-02}$	$0.7407_{2.20E-02}$	$0.7066_{2.01E-02}$	$0.7005_{6.32E-02}$	$0.5930_{8.71E-02}$	$0.5264_{8.78E-02}$
p15 q10	$0.8280_{1.97E-02}$	$0.7341_{3.90E-02}$	$0.6998_{2.66E-02}$	$0.6967_{7.28E-02}$	$0.5990_{9.98E-02}$	$0.5030_{9.48E-02}$
p15 q15	$0.8262_{2.68E-02}$	$0.7360_{2.47E-02}$	$0.7080_{1.92E-02}$	$0.6804_{7.17E-02}$	$0.6078_{8.54E-02}$	$0.5107_{1.29E-01}$
p15 q20	$0.8308_{2.32E-02}$	$0.7413_{1.70E-02}$	$0.7004_{2.12E-02}$	$0.7335_{1.12E-01}$	$0.6208_{9.37E-02}$	$\mathbf{0.5454_{8.41E-02}}$
p20 q5	$0.8224_{1.89E-02}$	$0.7356_{1.71E-02}$	$0.7004_{1.54E-02}$	$0.7038_{7.75E-02}$	$0.5840_{9.67E-02}$	$0.5419_{8.55E-02}$
p20 q10	$0.8379_{2.31E-02}$	$0.7333_{2.69E-02}$	$0.6987_{2.13E-02}$	$0.7281_{1.12E-01}$	$0.5787_{1.03E-01}$	$0.5088_{7.39E-02}$
p20 q15	$0.8339_{2.21E-02}$	$0.7359_{1.93E-02}$	$0.7055_{1.77E-02}$	$0.7033_{7.48E-02}$	$0.5862_{9.84E-02}$	$0.5101_{4.95E-02}$
p20 q20	$0.8343_{1.66E-02}$	$0.7384_{2.96E-02}$	$0.6991_{1.55E-02}$	$0.6921_{1.29E-01}$	$0.6113_{1.05E-01}$	$0.4939_{9.25E-02}$

Table 5.11: Adapting p and q for shallow networks. Median for ZDT1 and ZDT4. 3.000 and 30.000 evaluations were used, respectively.

In ZDT1 (see Table 5.11) we also observe increased performance, like with the deeper network. A surrogate algorithm that uses five learning generations but exploits the surrogate for more generations clearly outperforms the standard NSGA-II. Five learning generations are also enough for the shallow network and exploiting that surrogate for the right amount of generations (20 generations is too much) increases performance the most. When looking at the higher-dimensional problems, we see that the deep network has a performance advantage (4% for 50 variables and 7% for 70 variables).

In ZDT4 (Table5.11) we see no difference between deep and shallow networks. Similar to DTLZ1, at least ten learning generations are needed to train adequate surrogates.

Overall, we observe that shallow networks are performing worse than their deeper counterparts at higher-dimensional problems. They also do not allow for the MOEA to find the fourth part of the Pareto-optimal front in WFG2 and they are significantly outperformed by a deeper network in DTLZ2.

6 Conclusion and Future Research

In this thesis, we have proposed an extension to a surrogate-assisted MOEA which allows it to use multiple hidden layers. Then we investigated if more hidden layers allow for the approximation of more difficult multi-objective optimization problems (MOPs).

Our research goals were:

- Apply deep neural networks to surrogate-assisted multi-objective evolutionary algorithms.
- Up to now, most papers used the ZDT test suite to benchmark their surrogate models. Therefore, we want to see how our surrogate model performs on newer test suites with more complex problems.
- How our proposed method scales up when used on higher dimensional problems.
- Put the performance of the proposed method in relation to surrogate models using shallow networks.

Section 4.2 gives an overview of the proposed method and Section 5.1 describes its implementation in Java using jMetal and DL4J. In chapter 5 we have tested different combinations of hyperparameters to establish the effects of deeper networks on the performance of the surrogate method. The first experiment compared multiple ANN architectures to get an initial idea of how the number of hidden layers and the number of neurons in them impact the performance of the surrogate-assisted MOEA, and to find a promising deep ANN architecture that can be studied further. We decided to examine a network which has six hidden layers with 100 neurons each more thoroughly. The next two experiments were designed to find the optimal values for the learning generations and the surrogate generations. The first determines how much training data is collected and more training data will improve the quality of the approximation. The second determines for how many generations the surrogate is used, which allows us to gauge its quality. These experiments were done for a deep and a shallow ANN, and both were compared to establish in which cases the deeper network has better performance than the shallow network.

Our main observations were that deeper networks perform better on higher-dimensional problems, significantly outperform the shallow network on some problems (DTLZ2 and WFG2). However, they still struggled with highly multimodal problems like ZDT4 and DTLZ1.

Furthermore, most of the time there is no clear indication what combination of the tested hyperparameters is a predictor of better performance. Only in ZDT1, the easiest test problem, can a clear trend be established, and WFG2 is an example of a problem which is extremely sensitive to the hyperparameter combination. This means that to use our implementation in a real world scenario a optimization of hyperparameters has to be done first. Another possibility is to build hyperparameter optimization into the surrogate method. An example of this is proposed in the paper of Rosalez-Pérez et al.[50], the authors used a grid-searching approach to find suitable hyperparameters for each model that is created during the optimization. Using hyperparameter optimization is always a time-consuming process, and it has to be determined for each individual problem if the potential gains from using a surrogate are not outweighed by time spent to find good hyperparameters.

The width of the networks played a more important role then we initially expected. We hypothesize that the network architectures used were so small that not enough adaptable (weights and biases) parameters were available to approximate all features of the more difficult problems. Therefore, we recommend repeating the first experiment with even wider networks, with up to 1000 neurons. However, that many adaptable parameters also increase the need for training data, which needs to be evaluated using the real objective function, running counter to the basic idea of surrogate-assisted MOEAs. Needing more training data is also valid regarding the depth of the network. It may be useful to also test even deeper networks, but in the first experiment we did also test an architecture with nine hidden layers which did not perform as well as the ANN with six hidden layers. Increasing the width and depth will lead to large models with a high number of adaptable parameters that can easily overfit the data. Models like this need robust regularization and much more training examples. Given that the more difficult test problems seem to profit from more training data already, it seems like an idea worth studying.

Our thesis demonstrates the feasibility of deep feedforward neural networks as a surrogate model, but we have not yet unlocked the full potential of deep learning with regard to the highly multimodal problems like DTLZ1 or ZDT4. However, the representation learning aspect is, at first, only theoretical, it does not automatically manifest itself in practice. Therefore, we recommend to research what kind of representations are learned in our case. The field of deep learning also contains other deep neural network types like autoencoders or recurrent neural networks which could be used to support the feed-forward networks or integrated directly into the MOEA.

Bibliography

[1] John. R. (Carnegie Mellon University) Anderson. *Cognitive Psychology and its Implications*. Worth, New York, 8th edition, 2014.

[2] Alfredo Arias-Montaño, Carlos A. Coello Coello, and Efrén Mezura-Montes. Multi-objective airfoil shape optimization using a multiple-surrogate approach. In *2012 IEEE Congress on Evolutionary Computation, CEC 2012*, pages 1–8, jun 2012.

[3] Yoshua Bengio. Practical recommendations for gradient-based training of deep architectures. In *Neural Networks: Tricks of the Trade*, Lecture Notes in Computer Science, pages 437–478. Springer, Berlin, Heidelberg, 2012.

[4] Tobias Blickle and Lothar Thiele. A Comparison of Selection Schemes Used in Evolutionary Algorithms. *Evolutionary Computation*, 4(4):361–394, dec 1996.

[5] George E. P. Box and Norman R. Draper. *Response Surfaces, Mixtures, and Ridge Analyses*. Wiley-Interscience, Hoboken, N.J, 2 edition, apr 2007.

[6] Adriana Cervantes-Castillo, Efren Mezura-Montes, and Carlos A Coello Coello. An empirical comparison of two crossover operators in real-coded genetic algorithms for constrained numerical optimization problems. In *2014 IEEE International Autumn Meeting on Power, Electronics and Computing, ROPEC 2014*, pages 1–5, nov 2014.

[7] Deepti Chafekar, Liang Shi, Khaled Rasheed, and Jiang Xuan. Multiobjective GA optimization using reduced models. *IEEE Transactions on Systems, Man and Cybernetics Part C: Applications and Reviews*, 35(2):261–265, may 2005.

[8] Seongim Choi, Juan J Alonso, and Hyoung S Chung. Design of a Low-Boom Supersonic Business Jet Using Evolutionary Algorithms and an

Adaptive Unstructured Mesh Method. In *45th AIAA/ASME/ASCE/AHS/ASC Structures, Structural Dynamics & Materials Conference*, 2004.

[9] Noel Cressie. The origins of kriging. *Mathematical Geology*, 22(3):239–252, apr 1990.

[10] Balázs Csanád Csáji. *Approximation with Artificial Neural Networks.* PhD thesis, Eötvös Loránd University, 2001.

[11] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4):303–314, dec 1989.

[12] S. D'Angelo and E.A. Minisci. Multi-objective evolutionary optimization of subsonic airfoils by kriging approximation and evolution control. *2005 IEEE Congress on Evolutionary Computation, IEEE CEC 2005. Proceedings*, 2:1262–1267, sep 2005.

[13] Kalayanmoy Deb. *Multi-Objective Optimization Using Evolutionary Algorithms.* John Wiley & Sons, jul 2001.

[14] Kalyanmoy Deb and Ram Bhushan Agrawal. Simulated Binary Crossover for Continuous Search Space. *Complex Systems*, 9(2):115–148, 1995.

[15] Kalyanmoy Deb and Mayank Goyal. A combined genetic adaptive search (GeneAS) for engineering design. *Computer Science and Informatics*, 26(1):30–45, 1996.

[16] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.

[17] Kalyanmoy Deb, Lothar Thiele, Marco Laumanns, and Eckart Zitzler. Scalable multi-objective optimization test problems. In *Proceedings of the 2002 Congress on Evolutionary Computation, CEC 2002*, volume 1, pages 825–830, 2002.

[18] Alan Díaz-Manríquez, Gregorio Toscano, Jose Hugo Barron-Zambrano, and Edgar Tello-Leal. A review of surrogate assisted multiobjective evolutionary algorithms, 2016.

[19] Alan Díaz-Manríquez, Gregorio Toscano, and Carlos A. Coello Coello. Comparison of metamodeling techniques in evolutionary algorithms. *Soft Computing*, 21(19):5647–5663, 2017.

[20] Ronen Eldan and Ohad Shamir. The Power of Depth for Feedforward Neural Networks. *Conference on Learning Theory*, pages 907–940, 2015.

[21] Michael T.M. Emmerich, Kyriakos C Giannakoglou, and Boris Naujoks. Single- and multiobjective evolutionary optimization assisted by Gaussian random field metamodels. *IEEE Transactions on Evolutionary Computation*, 10(4):421–439, 2006.

[22] Carlos M Fonseca and Peter J Fleming. Genetic Algorithms for Multiobjective Optimization: FormulationDiscussion and Generalization. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 416–423. Morgan Kaufmann Publishers Inc., 1993.

[23] L G Fonseca, H J C Barbosa, and A C C Lemonge. On Similarity-Based Surrogate Models for Expensive Single- and Multi-objective Evolutionary Optimization. In *Computational Intelligence in Expensive Optimization Problems*, Adaptation Learning and Optimization, pages 219–248. Springer, Berlin, Heidelberg, 2010.

[24] António Gaspar-Cunha and Armando Vieira. A Multi-Objective Evolutionary Algorithm Using Neural Networks to Approximate Fitness Evaluations. *Int. J. Comput. Syst. Signal*, 6(1):18–36, 2005.

[25] Tushar Goel, Rajkumar Vaidyanathan, Raphael T. Haftka, Wei Shyy, Nestor V. Queipo, and Kevin Tucker. Response surface approximation of Pareto optimal front in multi-objective optimization. *Computer Methods in Applied Mechanics and Engineering*, 196(4-6):879–893, jan 2007.

[26] D. E. Goldberg and K. Deb. A Comparative Analysis of Selection Schemes Used in Genetic Algorithms. In *Foundations of genetic algorithms, Morgan Kaufmann Publishers*, volume 1, pages 69–93. Elsevier, jan 1991.

[27] Simon Haykin. *Neural Networks: A Comprehensive Foundation.* Prentice Hall, Upper Saddle River, N.J, 2 edition edition, jul 1998.

[28] D. O. Hebb. *The Organization of Behavior: A Neuropsychological Theory.* Psychology Press, Mahwah, N.J, 1 edition edition, may 2002.

[29] J Horn, N Nafpliotis, and D.E. Goldberg. A niched Pareto genetic algorithm for multiobjective optimization. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pages 82–87, 1994.

[30] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural Networks*, 4(2):251–257, jan 1991.

[31] Simon Huband, Luigi Barone, Lyndon While, and Phil Hingston. A Scalable Multi-objective Test Problem Toolkit. In *Evolutionary Multi-Criterion Optimization*, Lecture Notes in Computer Science, pages 280–295. Springer, Berlin, Heidelberg, mar 2005.

[32] Simon Huband, Phil Hingston, Luigi Barone, and Lyndon While. A review of multiobjective test problems and a scalable test problem toolkit, 2006.

[33] Afzal Husain and Kwang Yong Kim. Enhanced multi-objective optimization of a microchannel heat sink through evolutionary algorithm coupled with multiple surrogate models. *Applied Thermal Engineering*, 30(13):1683–1691, sep 2010.

[34] Amitay Isaacs, Tapabrata Ray, and Warren Smith. An evolutionary algorithm with spatially distributed surrogates for multiobjective optimization. In *Proceedings of the 3rd Australian conference on Progress in artificial life*, Lecture Notes in Computer Science, pages 257–268. Springer, Berlin, Heidelberg, dec 2007.

[35] Joshua Knowles. ParEGO: A hybrid algorithm with on-line landscape approximation for expensive multiobjective optimization problems. *IEEE Transactions on Evolutionary Computation*, 10(1):50–66, 2006.

[36] Joshua Knowles and Evan J Hughes. Multiobjective Optimization on a Budget of 250 Evaluations. In *Evolutionary Multi-Criterion Optimization*, pages 176–190. Springer, Berlin, Heidelberg, 2005.

[37] D G Krige. A Statistical Approach to Some Basic Mine Valuation Problems on the Witwatersrand. *Journal of the Chemical, Metallurgical and Mining Society of South Africa*, 52(6):201–215, 1952.

[38] D. G. Krige. A study of gold and uranium distribution patterns in the Klerksdorp gold field. *Geoexploration*, 4(1):43–53, 1966.

[39] Rudolf Kruse, Christian Borgelt, Christian Braune, Sanaz Mostaghim, and Matthias Steinbrecher. *Computational Intelligence: A Methodological Introduction.* Texts in Computer Science. Springer-Verlag, London, 2 edition, 2016.

[40] Yongsheng Lian and Meng-Sing Liou. Multi-Objective Optimization of Transonic Compressor Blade Using Evolutionary Algorithm. *Journal of Propulsion and Power*, 21(6):979–987, 2005.

[41] Xingtao Liao, Qing Li, Xujing Yang, Weigang Zhang, and Wei Li. Multiobjective optimization for crash safety design of vehicles using stepwise regression model. *Structural and Multidisciplinary Optimization*, 35(6):561–569, jun 2008.

[42] G. P. Liu, X. Han, and C. Jiang. A novel multi-objective optimization method based on an approximation model management technique. *Computer Methods in Applied Mechanics and Engineering*, 197(33-40):2719–2731, jun 2008.

[43] S Z Martinez and C A C Coello. MOEA/D assisted by RBF Networks for Expensive Multi-Objective Optimization Problems. *Gecco'13: Proceedings of the 2013 Genetic and Evolutionary Computation Conference*, pages 1405–1412, 2013.

[44] Saúl Zapotecas Martínez and Carlos A. Coello Coello. A memetic algorithm with non gradient-based local search assisted by a meta-model. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 6238 LNCS of *Lecture Notes in Computer Science*, pages 576–585. Springer, Berlin, Heidelberg, sep 2010.

[45] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, dec 1943.

[46] Tom M Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[47] Douglas C Montgomery, Anderson-Cook Christine M., and Raymond H Myers. *Response Surface Methodology: Process and Product Optimization Using Designed Experiments*. Wiley, Hoboken, N.J, 3 edition edition, jan 2009.

[48] Guido Montúfar, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio. On the Number of Linear Regions of Deep Neural Networks. In *Proceedings of the 27th International Conference on Neural Information Processing Systems*, NIPS'14, pages 2924–2932, Cambridge, MA, USA, 2014. MIT Press.

[49] Pawan K S Nain and Kalyanmoy Deb. Computationally effective search and optimization procedure using coarse to fine approximation. *Congress on Evolutionary Computation*, pages 2081–2088, 2003.

[50] Alejandro Rosales-Perez, Carlos A.Coello Coello, Jesus A. Gonzalez, Carlos A. Reyes-Garcia, and Hugo Jair Escalante. A hybrid surrogate-based approach for evolutionary multi-objective optimization. In *2013 IEEE Congress on Evolutionary Computation, CEC 2013*, pages 2548–2555. IEEE, 2013.

[51] F. Rosenblatt. The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):386–408, 1958.

[52] D.E. Rumelhardt, G.E. Hinton, and R.J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1*, chapter 8, pages 318–362. MIT Press, Cambridge, MA, USA, 1986.

[53] J. Sreekanth and Bithin Datta. Multi-objective management of saltwater intrusion in coastal aquifers using genetic programming and modular neural network based surrogate models. *Journal of Hydrology*, 393(3-4):245–256, nov 2010.

[54] N Srinivas and Kalyanmoy Deb. Muiltiobjective Optimization Using Nondominated Sorting in Genetic Algorithms. *Evolutionary Computation*, 2(3):221–248, 1994.

[55] Nielen Stander. An adaptive surrogate-assisted strategy for multi-objective optimization. In *Proceedings of the World Congress on Structural and Multidisciplinary Optimization*, Orlando, Florida, USA, 2013.

[56] A. Todoroki and M. Sekishiro. Modified Efficient Global Optimization for a Hat-Stiffened Composite Panel with Buckling Constraint. *AIAA Journal*, 46(9):2257–2264, 2008.

[57] A Vieira and R S Tome. a Multi-Objective Evolutionary Algorithm Using Neural Networks To Approximate Fitness. *International Journal of Computers , Systems and Signals*, 6(1):18–36, 2005.

[58] Geoffrey S Watson. Linear Least Squares Regression. *The Annals of Mathematical Statistics*, 38(6):1679–1699, dec 1967.

[59] P. Werbos. *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* PhD thesis, Harvard University, jan 1974.

[60] Eckart Zitzler, Kalyanmoy Deb, and Lothar Thiele. Comparison of Multiobjective Evolutionary Algorithms: Empirical Results. *Evolutionary Computation*, 8(2):173–195, jun 2000.

Declaration of Independence

I hereby declare that this thesis was created by me and me alone using only the stated sources and tools.

Tobias Peter   Magdeburg, March 15, 2018