Figure 3.89: Temperature distribution for individual $\text{temp}_{\text{max}} = p_{d_{\text{max}}}$.

Figure 3.90: Flow velocity distribution for individual $\text{temp}_{\text{max}} = p_{d_{\text{max}}}$.
3.4 Conclusions

First, we presented a heat configuration unit subject to optimization in aspects of heat performance, pressure drop and covered flow area. The recent multi-objective algorithm strength pareto evolutionary algorithm SPEA2 was applied to maximize heat performance, and minimize pressure drop and covered flow area. A Pareto front has been detected exhibiting a trade-off surface between all objectives. Thus, a compromise has to be made in aspects of these objectives.

In the second heat exchange configuration, the SPEA2, non-dominated sorting genetic algorithm NSGA-II, simple evolutionary multi-objective optimizer SEMO, and fair evolutionary multi-objective optimizer FEMO were applied to complex design optimization problems entailing four objective functions evaluated by a numerical solver. The present configurations represent a sufficiently complex problem from the optimization aspect. We compared the algorithms mainly regarding their ability to yield sufficiently diverse Pareto equivalent solutions according to defined objective functions. Thereby, we seek an optimal flow geometry in aspects of heat increase performance, pressure drop, flow velocity and covered area of the geometry. Although this might still be a somewhat artificial set up, the problem comes close to those which practicioners are confronted with.

In both configurations, the interpretation serves also for economic reasons so as to say that pressure drop indicates efficiency, surface area stands for ecological design and interior volume aids practical implementations. One may think of many reasons to seek designs that behave optimal to more than one objective.

In our numerical experiments, there is no analytical objective function available in closed form and, most distinguishing, computation time for each function evaluation is of major concern. So our experiments are representative for a class of multi-criteria optimization problems having a time-demanding objective function evaluation. The proposed methods show that efficiency can be exploited by using highly sophisticated evolutionary multi-objective optimization algorithms. The simple algorithms Semo and Femo quickly failed to provide sufficient insight into Pareto alternatives. One drawback with the EMO algorithms available today is that they are rather designed for given functions, such that an analysis about closeness and diversity to and on the Pareto front can easily be accomplished. In realistic cases, one only has “eye-sight” on the Pareto front which should actually be approved by a quantification analysis.

Our experiments allow an insight into design functionality which is unprecedented up to today. Clearly, there is almost no alternative of having such a clear view on flow geometry alternatives as is given by the Pareto set. We also noticed a non-convex Pareto front between surface area and temperature. As is well known, this observation can hardly be retraced by optimization algorithms not using evolutionary strategies. Up to now, there is a clear lack of investigations that concerns with the combination of multi-objective algorithms with real-world, or, if not feasible, near-world applications. One demand is to define metrics which indicate the closeness to and diversity on the Pareto front without optimizing given functions in analytical form. Moreover, strategies to save
function evaluations are necessary such that algorithms can find applications. Many algorithms were proposed in the last years in the multi-objective community. Now it is necessary to name and classify characteristics, advantages and disadvantages of algorithms for practical problems.

Future developments in the field of multi-objective optimization procedures are needed in aspects of ‘a posteriori’ error indicators and methods leading to as few objective function evaluations as possible. The last is crucial for the engineering practice: The practical value of a demanding multi-objective algorithm depends on the required number of function evaluations.
Chapter 4

Approximation Models

"Dem Anwenden muss das Erkennen vorausgehen."
Max Planck

4.1 Neural Network Approximation Models

4.1.1 Introduction

In applied shape optimization, a major aim is to keep function calls at a minimum where the only available problem dependent information are evaluations from a numerical solver. Here, a conventional optimization routine would also inquire gradient information which is not available in fluid dynamics without massive computations. Admitting acceptable tolerances of reaching the optimal state, an evolutionary strategy/neural network approach can find an optimum saving a number of function evaluations as compared to an evolutionary strategy working without the help of a network. In contrast to the preceding, the optimization problem (2.1) is considered with $n = 1$. Furthermore, the inequality constraints are put into the decision space,

$$\min_{x_l \leq x \leq x_u} \ f(x).$$

By doing this, the inequality constraints have the task to prevent the creation of invalid shapes.

In this context, the network is assigned to provide computationally inexpensive estimates of analysis outputs required during the optimization process. This is achieved by providing input/output samples from the present optimization process and training the network to learn the functional coherence between design parameters and objective function output. After training, the network model is then used as a solver substitute to produce the objective function outputs on unseen input parameter combinations. Strategically spoken,
the information contained in each solver run is used to train a simulation tool which then produces ‘target’ evaluations on unseen parameter combinations. Conventional approximation tools like polynomial approximation, e.g. least squares or Newton methods require a huge computational overhead by interpolating in hundreds or even thousands of data points. Neural networks, in contrast, are able to learn a functional coherence, thereby needing less data samples than conventional techniques. Specifically, it is the nonlinear sigmoid function at the hidden layer that gives the neural network greater computational flexibility than a standard linear regression model.

Using a network as an approximation model leads to optimization results that can be produced in orders of magnitude less computational effort than the conventional computing process. However, the drawback by using approximation models means a compromise between precision and computing time. Also, the selection of training data is crucial in this process: The parameters may be chosen by an evolutionary algorithm or by a certain application of efficient Monte Carlo generated parameter samples. In this dissertation, we not only ensure a global and efficient search by the application of a latin hypercube sample generator. We also save a recognizable number of function evaluations when using a progressive network with advanced generalization capabilities.

This chapter investigates two specific networks as substitute models for the approximation of the functional coherence between decision variables and objective function. These are the Bayesian regularization network and the adaptive neuro-fuzzy inference system (ANFIS). But first, we will review the principles of artificial neural networks (ANN) also detailed in, e.g., MacKay [101] or Haykin [64]. We also note the IEEE transactions on neural networks society’ publications as an exhaustive resource for neural network literature. Different authors used neural networks as approximation models in engineering applications [51,58,66,77,101,157]. Halgamuge [62] and Wilke [157] describe a number of network variants in which they find applications. In particular, this section outlines methods which summarize detailed investigations done by Hurtado [75, 76], Wilke [157], Papadrakakis [113] and coworkers, respectively. Our approach, in all generality, may also be compared to the inexact pre-evaluation (IPE) method established by Giannakoglou [52].

Again, the objective is to investigate the ability of the ANN to predict accurate objective function outputs that are necessary for the evolutionary strategy optimizer. This is achieved by a proper training of the ANN comprising the following steps:

1. Selection of the proper training set
2. Finding a suitable network architecture and
3. determining the appropriate values of characteristic parameters, e.g. the learning rate.

The most important factor for the success of the learning procedure of an ANN architecture is to select a proper training set. A sufficient number of input data distributed in the design space together with the output data resulting from a numerical solver meets the requirement for successful training. On the other hand, overloading the ANN with
unnecessary similar information results in so called overtraining without increasing the accuracy of the predictions. It is possible to split the design space into subregions and try to combine randomly the values within each subregion in order to obtain a training set which is representative of the whole design space, describing the latin hypercube sampling (LHS). Since we aim to keep function evaluations at a minimum, the LHS sampling method is an important tool in our analysis.

### 4.1.2 Multi-Layer Perceptrons

An ANN is constituted by an architecture with activity and learning rules as the most important ingredients. The architecture is the functional form of the network and specifies the parameters in the network and their relationships. Variables are the network weights determining the activities of the neurons. The activity rule (see figure (4.1)) defines how the activities of the neurons change in response to each other. The learning rule specifies the way the neural network’s weights change.

![Sigmoid activation function](image)

**Figure 4.1:** Sigmoid activation function.

ANN are also classified by supervised or unsupervised networks. In our computations we only consider supervised networks to which data is given in the form of inputs and targets. The targets play the role of the ‘objective’ which the network is supposed to produce with a given input.

The proposed neuron type is the basic component used to form so-called multi-layer perceptrons (figure (4.2)), which is one of the most widely used ANN in many applications. It is composed by an input layer, in which no transformations are performed, one or more hidden layers, and an output layer. The operations in a single artificial neuron is illustrated in Figure (4.3). The model comprises the following operations:

1. A single neuron having a number of inputs each associated with a weight $w_{ij}$ and a numerical value $x_i$ representing the output from neuron $i$ from the previous layer.
The collection operation carried out by the neuron is the linear combination

\[ a_q = \sum_{l=1}^{N} w_{lq} x_l, \]

where \( N \) indicates the number of neurons in the preceding layer.

2. The neuron then produces an output, a single number which serves as the input to neurons in the following layer. The transformation performed is characterized by a linear or nonlinear transformation, i.e. the activity rule. The node output is set as a function

\[ y_q := h(a_q). \]

There are several possible activation functions. The most popular one is the sigmoid function (figure (4.1)) given by

\[ h(s) = \frac{1}{1 + e^{-\gamma s}}, \]

where \( \gamma \) is a parameter defining the slope of the function. Activation functions differ on the objective that is investigated. For linear controller problems, linear activation functions are most suitable. Non-linear activation functions, on the other hand, are more flexible and adequate for the kind of problems discussed in this work.
The central idea of supervised neural networks is to learn to model the relationship between an input vector $\mathbf{x}$ and a target $t$. This learning is realized by sample training vectors which make the network adjust its parameters such that it is able to simulate output values to yet unseen input vectors. The network architecture is known as feedforward backpropagation network model [101]. The term ‘feed-forward’ refers to the ‘forward’ direction of computing the output for a given input. Backpropagation means that the weights are adjusted by going ‘backwards’ from the output layer in direction to the input layer (figure (4.2)).

To this means, an error function is defined, usually the summed squared distance of the output training samples and the network output to a corresponding input vector. The error function is then a measure of how well the network with set weights approximates the target value. The training process is now a task of function minimization, i.e. to minimize the error function. This is captured by adjusting the network weights. For general feedforward-backpropagation networks the algorithm evaluates the gradient of the error function with respect to the network weights. Standard backpropagation involves a gradient descent algorithm, in which the network weights are adjusted according to the gradient of the error function. There are a number of variations on the basic algorithm that are based on other standard optimization techniques, such as variants of conjugate gradient and Newton methods. To elucidate how this process works in detail, we outline the feedforward-backpropagation algorithm explicitly on a simple network architecture.
4.1.3 The Network Learning Process

Backpropagation was created by generalizing the Delta rule (also known as Widrow-Hoff learning function) to multiple-layer networks and nonlinear differentiable transfer functions. The vector of the network weights is updated according to
\[ w_{ij,new} = w_{ij,old} + \eta \Delta w_{ij,old} \]
where \( \eta \) denotes the learning parameter. We have to approximate \( \Delta w_{ij,old} \) to obtain the last ingredient in our network. The different network learning functions do mostly differ the way in which they calculate the weight update \( \Delta w_{ij,old} \). Usually, the error function is the square error between the actual and estimated outputs. The weight updating can be performed pattern by pattern (incremental training) using
\[ E = \sum_{n=1}^{N} (y_n(p, w) - t_n(p))^2, \quad (4.1) \]
with \( p \) as one network input or, after all patterns have been shown to the network and all the estimates have been computed (batch training), by means of the accumulated error function:
\[ E = \sum_{p=1}^{P} \sum_{n=1}^{N} (y_n(p, w) - t_n(p))^2. \]

In these equations \( t_n(p) \) is the actual output associated with neuron \( n \) in the output layer, \( y_n(p, w) \) its network estimate, \( N \) the number of neurons in the output layer, and \( P \) the number of training patterns. In order to simplify notation, we here use (4.1) to detail the learning algorithm.

We derive the learning formulas for a very simple network with one input, one hidden and one output layer \( N = 1 \) giving an idea how the feedforward-backpropagation neural network algorithm works. We define
\[ a_q := \sum_{i=1}^{N} x_i w_{iq} \]
which is the weighted incoming signal from all preceding nodes to node \( q \).
\[ y_q := h(a_q) \]
are the signals transformed through the activation function \( f \) which represent the output of node \( q \) and serve as input for the next layer. We will denote the activation function for an output neuron with \( h_1 \) and for a hidden neuron with \( h_2 \). Since our network only has one output neuron, we omit the subindex and for clarity also the weights, so the output value is denoted by \( y(\mathbf{x}) \). Furthermore, \( w_{pq} \) denotes the weight from node \( p \) to node \( q \). Thus \( w_{pq} \) connects a preceding and an actual layer. We also define
\[ E^2 := E^2(\mathbf{x}, w) = (y(\mathbf{x}) - t(\mathbf{x}))^2, \]
which is the squared error between the predicted network output \( y(x) \) and the required training output \( t(x) \) according to a training sample \( x \). \( y(x) \) certainly depends on all weights and node values in the present network. We will later use this fact to formulate derivatives of the error with respect to the weights. To further simplify the notation, we omit the indication that the network output and so the network error \( E^2 \) depend on all weights \( w \) in the network.

The approximation used for the weight change is

\[
w_{pq;\text{new}} = w_{pq;\text{old}} - \eta \frac{\partial E^2}{\partial w_{pq}},
\]

where \( \eta \) is the learning rate parameter determining the rate of learning, and

\[
\frac{\partial E^2}{\partial w_{pq}}
\]

is the sensitivity of the error, \( E \), to the weight \( w_{pq} \) and determines the direction of search in the weight space for the new weight value. From the chain rule,

\[
\frac{\partial E^2}{\partial w_{pq}} = \frac{\partial E^2}{\partial a_q} \frac{\partial a_q}{\partial w_{pq}}
\]

and

\[
\frac{\partial a_q}{\partial w_{pq}} = \frac{\partial}{\partial w_{pq}} \left( \sum_{i=1}^{n} x_i w_{iq} \right) = \frac{\partial (x_p w_{pq})}{\partial w_{pq}} + \frac{\partial}{\partial w_{pq}} \left( \sum_{i=1, i \neq p}^{n} x_i w_{iq} \right) = x_p
\]

since the rest of the inputs to neuron \( q \) have no dependency on the weight \( w_{pq} \), we obtain

\[
w_{pq;\text{new}} = w_{pq;\text{old}} - \eta \frac{\partial E^2}{\partial a_q} x_p
\]

and the weight change of \( w_{pq} \) depends on the sensitivity of the squared error \( E^2 \) to the input \( a_q \) of unit \( q \) and on the input signal \( x_p \). There are two possible situations: the neuron \( q \) is either an output neuron or a hidden neuron. In the first case, the change in the squared error due to an adjustment of \( w_{pq} \) is simply the change in the squared error of the output \( q \):

\[
\frac{\partial E^2}{\partial a_q} = 2(y_1(p, w) - t_1(p)) \frac{\partial y_1(p, w)}{\partial a_q} = 2E \frac{\partial h_1(a_q)}{\partial a_q}
\]

Combining equations, we get

\[
w_{pq;\text{new}} = w_{pq;\text{old}} - 2 \eta x_p E h_1'(a_q)
\]

which is the rule for modifying the weights when neuron \( q \) is an output neuron. With, for instance, choosing the the tangens hyperbolicius as activation function, i.e.

\[
h_1(x) := \tanh(x) \quad \text{we have that} \quad h_1' = 1 - h_1^2
\]
we get
\[ w_{pq,\text{new}} = w_{pq,\text{old}} - 2 \eta x_p E (1 - y_q^2). \]

If \( h_1 \) is the standard linear rule, the derivative of \( h_1 \) is simply \( h_1' \equiv 1 \).

Now we consider the second case where \( q \) is a hidden neuron. Since the error \( E^2 \) depends on the sum of the weighted incoming signals \( a_r = \sum_j y_j w_{rj} \) and this again depends on \( y_q = h(a_q) \) for \( q \) as one of the \( j \) indices, we may write by the chain rule
\[
\frac{\partial E^2}{\partial a_q} = \frac{\partial E^2}{\partial a_r} \frac{\partial a_r}{\partial y_q} \frac{\partial y_q}{\partial a_q},
\]
where the subscript \( r \) represents the output neuron. Now,
\[
\frac{\partial y_q}{\partial a_q} = \frac{\partial h(a_q)}{\partial a_q} = h'_2(a_q) \quad \text{and}
\]
\[
\frac{\partial a_r}{\partial y_q} = \frac{\partial \left( \sum_{j=1}^m y_j w_{rj} \right)}{\partial y_q}, \tag{4.2}
\]
where \( j \) is an index that ranges over all the neurons including neuron \( q \) that provide input signals to the output neuron.

Expanding the right hand side of equation (4.2)
\[
\frac{\partial \left( \sum_{j=1}^m y_j w_{rj} \right)}{\partial y_q} = \frac{\partial y_q w_{rq}}{\partial y_q} + \frac{\partial \left( \sum_{j=1,j\neq q}^m y_j w_{rj} \right)}{\partial y_q} = w_{rq},
\]
since the weights of the other neurons, \( w_{rj}, \ j \neq q \) have no dependency on \( y_q \). Inserting gives
\[
\frac{\partial E^2}{\partial a_q} = \frac{\partial E^2}{\partial a_r} w_{rq} h'_2(a_q).
\]

Thus \( \frac{\partial E^2}{\partial a_q} \) is now expressed as a function of \( \frac{\partial E^2}{\partial a_r} \), which can easily be calculated as in case one. The complete rule for modifying the weights between neuron \( p \) sending a signal to a neuron \( q \) is,
\[
w_{pq,\text{new}} = w_{pq,\text{old}} - \eta x_p \frac{\partial E}{\partial a_q},
\]
with
\[
\frac{\partial E^2}{\partial a_q} = \frac{\partial E^2}{\partial a_r} h'_2(a_q) w_{rq}.
\]

The algorithm above is a simplified version in that there is only one output neuron. Certainly, more than one output is allowed and the gradient descent minimizes the total squared error of all the outputs.
A theorem by Hornik, Stinchcombe, and White [71] is often quoted as the basis of the ANN methodology for functional approximation. It states that ANN with as few as one hidden layer and arbitrary activation functions can approximate virtually any function (those with a finite number of discontinuities), provided that sufficiently many hidden neurons are included in the perceptron. Properly trained backpropagation networks tend to give reasonable answers on inputs that they have never seen. Typically, a new input leads to an output similar to the correct output for input vectors used in training. This generalization property makes it possible to train a network on a representative set of input/target pairs and get good results without training the network on all possible input/output pairs.

### 4.1.4 Levenberg-Marquardt Algorithm

Since the employed network has to process an enormous amount of data, we seek to optimize the training procedure. The Gauss-Newton approximation used in the Levenberg-Marquardt training can converge orders of magnitude faster than the gradient descent approach [50, 59]. This approach is designed to achieve second-order training speed without having to compute the Hessian matrix which is estimated using the gradients. The Levenberg-Marquardt algorithm is a virtual standard in nonlinear optimization and outperforms gradient descent methods for medium sized problems (cf. Nocedal and Wright [109]).

The Levenberg-Marquardt approach relies on estimating some curvature information from only function evaluations and first order derivatives. The Jacobian matrix is calculated by a standard backpropagation technique which involves considerably less complex computation than the Hessian matrix. The Hessian is then approximated by the Jacobian resulting in a second order method. In general, the Levenberg-Marquardt algorithm can be thought of as a trust-region modification of the Gauss-Newton algorithm (a line search algorithm in nonlinear least squares problems [109]). Since the network performance function has the form of a sum of squares, the Hessian can here be approximated as

$$H = J^T J$$

and the gradient may be computed as

$$g = J^T e,$$

where $J$ is the Jacobian matrix that contains first derivatives of the network errors with respect to the weights and biases and $e$ is a vector of network errors.

The approximation to the Hessian matrix is then used in the Newton-like update:

$$w_{ij,\text{new}} = w_{ij,\text{old}} - [J^T J + \mu I]^{-1} J^T e$$  \hspace{1cm} (4.3)

When $\mu$ is zero, (4.3) is identical to the Gauss-Newton method. As $\mu$ tends to infinity, the more we realize gradient descent with a small step size. Since Newton’s method is more accurate near a minimum, we try to achieve Newton’s method as quickly as possible. So
the parameter $\mu$ will be decreased when we observe a reduction in the performance function and will only be increased if a further step would increase the performance function. After a sufficient number of iterations, the performance function will be reduced at each iteration of the algorithm.

4.2 Bayesian Regularization

Neural networks have yet performed poorly on complex function approximation tasks. A great advancement by using networks can be made by the application of progressive network configurations from which we expect considerably improved function approximation capabilities. This section examines a progressive network model to which we also give numerical examples.

A novel aspect in our networks is the Bayesian regularization feature. In this framework, the weights of the network are assumed to be random variables with specified distributions. Regularization parameters are related to the unknown variances associated with these distributions. Bayesian regularization has first been presented by David MacKay [100] and has been applied to Levenberg Marquardt network training by Hagan and Foresee [50]. To our best knowledge, the power of this method has not yet been explored by applications in context of engineering implementations.

4.2.1 Improving Generalization

We consider regularization as the main ingredient to the successful application of neural networks in our optimization context. Due to our aim to reduce computation time, we seek to train a network with as few training samples as possible. The drawback is that such a network is quickly overfitted which means that the network error is driven to a small value for the training samples but will become large when new input is presented. This indicates that the network has memorized the training samples but is not able to generalize to give reasonable answers on unseen input parameter combinations.

One way to tackle this problem is to set up a network architecture that is small enough not to be able to overfit the data - a small network will not have the computational power and flexibility to overfit. The crucial parameter is the number of network weights which determines the functional complexity a network can model. Thus, the larger a network is, the more complex are the functions the network can create - and so will “interpolate” the given data. We do not need to worry about overfitting effects, if we are able to train the network with arbitrary amount of data - a large network would then depict the correct functional coherence. Regularization only applies to those situations in which one wants to make the most out of a limited pool of data. Furthermore, it is difficult to know beforehand how large a network should be for a specific application.

One regularization approach is to use early stopping, where the algorithm which minimizes the error function prevent it from doing so by stopping the algorithm at some point. In
early stopping the available data is divided into a training, a validation and a test subset. The training set is used for training the network, computing the gradient, and updating the network weights. The validation subset is not used for training, yet the performance function indicates how the trained network responds to these samples. The validation error will normally decrease during the initial phase of training, as does the training set error. When the network begins to overfit the data, the error on the validation set will typically begin to increase. The test set is not used during the training, but utilized to compare different networks. If the response on the test set is too weak we may decide to restart the network training with a different division of data sets.

A more sophisticated approach to improve generalization is to modify the performance function by adding a term that consists of the mean of the sum of squares of the network weights. This modifies the performance function such that it incorporates a bias against the adjustment of weights which we do not like to obtain. What we don’t like is a ‘sharp decision boundary’ which is again associated with large network weights, and so we use a penalty term in the error function, defined by

\[ F(w) := \alpha E_D(w) + \beta E_W(w) \]

where the most obvious choice for \( E_W(w) \) is a weight decay regularizer

\[ E_W(w) := \frac{1}{n} \sum_{i,j=1}^{n} w_{ij}^2, \]

with \( n \) the total number of network weights and \( E_D(w) \) is defined as in (4.1). The parameters \( \alpha \) and \( \beta \) determine how much the weight decay regularizer is involved in the error function \( F(w) \). These parameters now become objectives of the network optimization process and need to be determined.

### 4.2.2 Bayesian Parameters

First suggested by David McKay \[100, 101\], the neural network learning process can be given a probabilistic approach. We here give a rough idea how statistical concepts can be used to find suitable network structures. For a detailed insight we refer to [50,59,100,101].

As we have noted beforehand, the performance function first decreases and, after overfitting the data, increases again. The model overfits the data and generalizes poorly. This problem can be addressed by using the Bayesian approach to control model complexity. In the Bayesian framework, the weights of the network are considered to be random variables. One can then apply statistical techniques to estimate distribution parameters, e.g. variances. According to Bayes’ rule the probability distribution can be written as

\[ P(w|D, \alpha, \beta, M) = \frac{P(D|w, \beta, M) P(w|\alpha, M)}{P(D|\alpha, \beta, M)}, \tag{4.4} \]

where \( D \) corresponds to the input-output data samples, \( M \) denotes the network model and architecture and \( w \) are the network weights. The conditional probability \( P(w|\beta, M) \)
is the prior distribution, which corresponds to our knowledge of the weights before any data is collected. \( P(D|w, \alpha, M) \) is the likelihood of the data occurring, given the weights \( w \). Since the network training is an optimization problem by itself, the objective function is now \( F(w) \). The error function is interpreted as defining the probability distribution of a noise model:

\[
P(D|w, \alpha, M) = \frac{1}{Z_D(\alpha)} e^{-\alpha E_D},
\]

where \( Z_D(\alpha) = (\pi/\alpha)^{N/2} \). So the use of the sum squared error \( E_D \) corresponds to an assumption of Gaussian noise on the target variables. Similarly, the regularizer \( E_W(w) \) is interpreted in terms of a prior distribution over the parameters:

\[
P(w|\beta, M) = \frac{1}{Z_W(\beta)} e^{-\beta E_W},
\]

with \( Z_W(\beta) = (\pi/\beta)^{n/2} \). If \( E_W \) is quadratic as above, then the corresponding prior distribution is a Gaussian one. The optimal weights should maximize the posterior probability \( P(w|D, \alpha, \beta, M) \), this is equivalent to minimize the regularized function \( F(w) \).

Substituting the expressions for the prior probability and the likelihood function into (4.4) gives

\[
P(w|D, \alpha, \beta, M) = \frac{1}{Z_F(\alpha, \beta)} e^{-F(w)}.
\]

---

**Figure 4.4:** Adjustment of \( \alpha, \beta \).
with $Z_F(\alpha, \beta) = Z_D(\alpha) Z_W(\beta)$. Next we need to determine the objective function parameters $\alpha$ and $\beta$. To these ends, we again apply Bayes’ rule as

$$P(\alpha, \beta|D, M) = \frac{P(D|\alpha, \beta, M) P(\alpha, \beta|M)}{P(D|M)}.$$

The optimal parameters as well as the effective number of weights used in the network can be inferred by maximizing the posterior probability $P(D|\alpha, \beta, M)$. Again, the discussion is quite extensive and so we refer to the cited literature for a detailed analysis.

The optimal regularization parameters can so be obtained in an automated fashion (figure 4.4). The Bayesian approach has the considerable benefits that no ‘test set’ or ‘validation set’ is needed, all available training data can be used for both model fitting and model comparison. The salient advantage of this algorithm is that it implicitly provides a measure of how many network parameters (weights) are effectively used by the network. This indicates whether a sufficient number of training samples are presented to the network and automatically determines the optimum network size.

**Validation**

In all our numerical examples, a test set is used to see if the trained network has properly adapted to the functional coherence between input parameters and objective function. This test set is chosen from the data the network will not be trained with. On the one hand, it needs to represent the search space though important data samples, on the other hand, needs to be incorporated in the training set. We decided in our numerical examples to randomly choose about 10% of the available data to enter the validation set. When the difference between the flow solver evaluations and the network suggestion exceeds a certain limit, the training is considered as not successful and is restarted.

**Optimization based on Evolutionary Strategies and Artificial Neural Networks**

After the selection of a suitable network architecture the training procedure is performed using a number of data sets in order to obtain the input/output pairs needed for the network training. We decided for network feeding and evolutionary network feeding for selecting the training set (see also [76,77,112,114]). The methods were detailed in section 2.3.3.

In the evolutionary strategies, we decided to apply the two-membered evolutionary strategy, proposed by Rechenberg [123]. This is realized by the $(\lambda, \mu + \lambda)$ evolutionary strategy which applies the identity for recombination and variation. The population is initialized in each generation by one member which then produces a population by the mutation operator only. The mutation operator is a Gaussian distribution applied to each component of the parameter vector. The selection operator picks the one best fit and viable element from the population (also confer Papadrakakis et al. [114]).
4.2.3 Numerical Results

The problem we consider is a staggered channel junction for which a numerical solver computes the model objective, the pressure drop. Flow model defining parameters are fluid properties, inflow velocity, and geometric attributes. The configuration is illustrated in figure (4.5) where the Reynolds number is $Re = 100$ and the flow is defined to be water. The optimization process aims to find the parameter combination for which the corresponding geometry best fit the optimization objective. This is a single-objective problem having six design variables as indicated in figure (2.11) (chapter 2, section 2.5).

Reference Solution

<table>
<thead>
<tr>
<th>Method</th>
<th>No. Design variables</th>
<th>No. Evaluation</th>
<th>Obj. value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFO</td>
<td>12</td>
<td>760</td>
<td>5.5419</td>
</tr>
<tr>
<td>DFO</td>
<td>6</td>
<td>191</td>
<td>5.5767</td>
</tr>
</tbody>
</table>

**Table 4.1:** DFO Algorithm Performance

Table (4.1) indicates the performance by a derivative-free optimization (DFO) algorithm. This DFO algorithm employs a substitute to the gradient by (here: 3) function evaluations. It then employs a quadratic interpolation through these points and performs a steepest descent algorithm on this quadratic (the sequentiell quadratic programming - SQP). For 6 design variables, the DFO needs 191 function evaluations to converge at its global
minimum. Figure (4.6) indicates the channel with 6 deflection vectors for which a pressure drop of 5.57 was recognized to be the global minimum (see table 4.1). In case of 12 design variables, the DFO algorithm converges after 790 function evaluations. This sets the benchmark to compare with our method. The DFO arrives and finally converges at the global minimum but, obviously needs a noticeable increasing number of evaluations with increasing complexity.

Generally, the presented ‘channel junction’ example is a very complex one. First, by extending the boundaries the pressure drop remains on a quite low level. By squeezing the channel to only a small amount, the pressure drop explodes to values in the region of 100 to 200. It was also observed that the functional coherence between shape parameters and objective function appears to be convex although this convexity behaves as ‘shallow’, i.e. the gradient is positive, but in each design variable approximately zero. Second, there obviously exist many different combinations of the design variables which all yield a similar (low) pressure drop of about 6 to 8. In a loose sense, the mapping is not ‘injective’. Third, the dimension in the output space is one. So, if we imagine that there were more objective functions (uncorrelated), the network would be less complex since it was able to classify data by the dimensions. We note that steep gradients (squeezed channel) avert the network to be precise in the region of low pressure drop values. Thus the presented example is sufficiently complex to demonstrate a high-performance network.

Numerical Results

We divide our numerical examples by the number of design variables and one- respectively two layer network types. As detailed in the neural network section, we employed a Bayesian regularization network. We also conducted networks working with the plain Levenberg-Marquardt algorithm which converged more quickly but failed to give reasonable optimization results. We thus restrict our documentation to Bayesian regularization networks. The network calculations are performed with the Matlab neural network toolbox [102]. An overview of all methods employed are given in table 4.2.

<table>
<thead>
<tr>
<th>Sampling Method</th>
<th>Uniform-/ Gaussian Latin Hypercube Samples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network Type</td>
<td>One- and two layer networks of type feedforward-backpropagation</td>
</tr>
<tr>
<td>Learning Algorithm</td>
<td>Gauss-Newton Approximation in Levenberg-Marquardt training with Bayesian regularization</td>
</tr>
<tr>
<td>Flow Solver</td>
<td>Fastest 2-D Finite Volume Solver Multigrid Algorithm, 6 grid levels, 140,000 control volumes</td>
</tr>
<tr>
<td>Grid</td>
<td>Block-structured Grid</td>
</tr>
<tr>
<td>Optimization Strategies</td>
<td>- Network Feeding (NF)</td>
</tr>
<tr>
<td></td>
<td>- Simulated Annealing (SA)</td>
</tr>
<tr>
<td></td>
<td>- Evolutionary Network Feeding (ENF)</td>
</tr>
</tbody>
</table>

**Table 4.2:** Methods used in this numerical example

The evolutionary strategy is used with 30 individuals in 30 generations. This is a huge
number of flow evaluations, which are processed by the network in a few seconds. All simulations which involved a network, the network output is understood as a ‘suggestion’ to the optimum. This is re-calculated with the finite volume solver and the resulting pressure drop indicated in the corresponding tables.

It is important to understand the behavior of the pressure drop due to different shape designs. In figures (4.11), (4.12), and (4.14) (and also in other figures) large recirculation zones appear in the upper and lower part of the deformed channel. This is due to the fact that the deformed region constitutes a diffusor and baffle at the inflow and outflow part of the shifted channel region, respectively. According to the streamline theory in tube flows (cf. e.g. [128,143]), these delaminations influence the pressure, an effect which causes a rise in the objective function. The delamination is responsible for dissipating energy, thus causing a rise in the pressure for the ‘healthy’ flow field. As can be clearly seen in the optimal configuration (4.6) the deformed channel is quite slim, and figure (4.9) shows that in such a slim configuration the flow cannot establish large delaminations. We thus conclude that these delaminations make the optimization process difficult since the slight increase in pressure drop refers to a ‘shallow’ functional coherence between decision variables and objective function. However, if one enlarges the channel in either direction, the pressure drop keeps hanging between values of 6 and 7. Unfortunately, the neural network does not ‘understand’ this. So there exists a bias to inflate the shape since the network tends to propose and chose large parameter vectors. It would be appropriate to circumvent this by using penalty techniques which we actually first employed. But the improvement was negligible in contrast to the introduced complexity and so we present the results which were obtained without penalty functions. Moreover, the latin hypercube sampling method for generating random samples according to a specified distribution guarantees the exploration of the design space to provide a proper training set for the network.

None of the algorithms were able to reach the global minimum (cf figure (4.6) and table 4.1), although the geometries found were quite similar to this global minimum. The pressure drop appears to react very sensitive on parameter variations. However, our optimization results can be considered as successful since the algorithms employed propose shape designs with corresponding pressure drop of 5.58 to 6.10 and designs that all come close the the global optimum.

The simulated annealing resp. evolutionary algorithm approach does not reach the same precision, yet it gives reasonable shape designs close to the optimum with much less function evaluations. We stress the importance of this advantage by considering that real world applications usually imply considerable long evaluation runs.

Table 4.5 indicates the data sets for the NF and SA algorithm. The best result is obtained by the SA algorithm with a pressure drop of 5.5863 which comes closest to the real minimum of 5.5754. The NF data sets are used as training samples for the network. By only randomly varying parameters, the NF data set contains a geometry with pressure drop 5.7698 (cf. figure (4.10)). Table 4.3 and 4.4 give an overview of optimization results from the networks employed. The corresponding geometries are indicated in the last column. In the Bayesian regularization context, convergence is achieved when the effective
number of parameters remains approximately the same which usually takes place after the “sum squared error” achieves its minimum value. This is also indicated for some networks in, e.g. figures (4.7), (4.8). The figures indicate the training error, squared weights, the effective number of parameters, and number of training epochs. One of the advantageous features of the proposed network approach is that the Bayesian regularization networks automatically chooses its parameters. The effective number of parameters gives an insight into the complexity of the network. Following the idea of Bayesian regularization, we achieve a network that rather depicts the causal relationship between parameter vector and objective function instead of ’interpolating’ data. The given figures (4.7), (4.8), show that the effective number of parameters was 562, and 931, respectively. The training data of 415 samples was way too less for training these complex two-layer networks. Although the squared weights converged, the networks were not considered as very successful, also indicated by a ‘jump’ in the iterated number of parameters.

Figures (4.9)-(4.14) correspond to the finally proposed shape designs using 6 design variables as indicated in tables 4.5-4.4. Table 4.3 indicates that the data set with 243 samples yields the best result with a one-layer network using 48 neurons (cf. figure (4.13)). We also gain quite similar shape designs with only 100 training samples (figure (4.14)). Figures (4.9) and (4.17) represent the final optimization result using simulated annealing. In all of the shown channel junctions, we see that the mid-part and upper parts are quite good resembled by the network approach, only the lower part is often oversized. The recirculation zones allow a delamination of the flow and, as mentioned above, may thus
<table>
<thead>
<tr>
<th>Method</th>
<th>Design-variables</th>
<th>Samples</th>
<th>Distribution</th>
<th>Interval/ Std. Dev. $\sigma$</th>
<th>Obj. value in figure</th>
</tr>
</thead>
<tbody>
<tr>
<td>NF</td>
<td>12</td>
<td>104</td>
<td>LHS Uniform</td>
<td>$[-3,3]$</td>
<td>5.8244</td>
</tr>
<tr>
<td>NF</td>
<td>12</td>
<td>91</td>
<td>LHS Uniform</td>
<td>$[-3,3]$</td>
<td>5.8729</td>
</tr>
<tr>
<td>NF</td>
<td>12</td>
<td>220</td>
<td>LHS Gauss</td>
<td>1.0</td>
<td>5.7506</td>
</tr>
<tr>
<td>NF</td>
<td>12</td>
<td>415</td>
<td>cumulated</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ENF</td>
<td>12</td>
<td>138</td>
<td>LHS Uniform</td>
<td>$[-2,2]$</td>
<td>5.7013</td>
</tr>
<tr>
<td>ENF</td>
<td>12</td>
<td>110</td>
<td>LHS Gauss</td>
<td>$[-2,2]$</td>
<td>5.7741</td>
</tr>
<tr>
<td>ENF</td>
<td>12</td>
<td>248</td>
<td>cumulated</td>
<td>$[-2,2]$</td>
<td>5.7013</td>
</tr>
<tr>
<td>SA</td>
<td>12</td>
<td>215</td>
<td>LHS Gauss</td>
<td>-</td>
<td>5.6266</td>
</tr>
<tr>
<td>SA</td>
<td>12</td>
<td>118</td>
<td>LHS Gauss</td>
<td>-</td>
<td>5.6804</td>
</tr>
</tbody>
</table>

Table 4.6: Data Sets used for Test Problem with 12 Design Variables

<table>
<thead>
<tr>
<th>Samples</th>
<th>Design-variables</th>
<th>Neurons</th>
<th>Objective value in figure</th>
</tr>
</thead>
<tbody>
<tr>
<td>104</td>
<td>12</td>
<td>12</td>
<td>6.0320</td>
</tr>
<tr>
<td>104</td>
<td>12</td>
<td>24</td>
<td>6.0942</td>
</tr>
<tr>
<td>104</td>
<td>12</td>
<td>48</td>
<td>5.9812</td>
</tr>
<tr>
<td>220</td>
<td>12</td>
<td>12</td>
<td>6.0074</td>
</tr>
<tr>
<td>220</td>
<td>12</td>
<td>24</td>
<td>5.9623</td>
</tr>
<tr>
<td>220</td>
<td>12</td>
<td>48</td>
<td>5.9147</td>
</tr>
<tr>
<td>415</td>
<td>12</td>
<td>12</td>
<td>5.9560</td>
</tr>
<tr>
<td>415</td>
<td>12</td>
<td>24</td>
<td>5.8704</td>
</tr>
<tr>
<td>415</td>
<td>12</td>
<td>48</td>
<td>6.0739</td>
</tr>
</tbody>
</table>

Table 4.7: One Layer Network, NF algorithm

<table>
<thead>
<tr>
<th>Samples</th>
<th>Design-variables</th>
<th>Neurons</th>
<th>Objective value in figure</th>
</tr>
</thead>
<tbody>
<tr>
<td>220</td>
<td>12</td>
<td>12/2</td>
<td>5.9853</td>
</tr>
<tr>
<td>220</td>
<td>12</td>
<td>12/12</td>
<td>5.7813</td>
</tr>
<tr>
<td>220</td>
<td>12</td>
<td>24/16</td>
<td>6.4583</td>
</tr>
<tr>
<td>415</td>
<td>12</td>
<td>12/2</td>
<td>6.0125</td>
</tr>
<tr>
<td>415</td>
<td>12</td>
<td>16/16</td>
<td>12.6754</td>
</tr>
<tr>
<td>415</td>
<td>12</td>
<td>32/24</td>
<td>6.8091</td>
</tr>
</tbody>
</table>

Table 4.8: Two Layer Network, NF algorithm

<table>
<thead>
<tr>
<th>Method</th>
<th>Design-variables</th>
<th>Samples</th>
<th>Neurons</th>
<th>Objective value in figure</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENF</td>
<td>12</td>
<td>248</td>
<td>12</td>
<td>5.8372</td>
</tr>
<tr>
<td>ENF</td>
<td>12</td>
<td>248</td>
<td>24</td>
<td>5.9056</td>
</tr>
<tr>
<td>ENF</td>
<td>12</td>
<td>248</td>
<td>48</td>
<td>6.2550</td>
</tr>
</tbody>
</table>

Table 4.9: ENF - One Layer Network
4.2 Bayesian Regularization

<table>
<thead>
<tr>
<th>Method</th>
<th>Design-variables</th>
<th>Samples</th>
<th>Neurons</th>
<th>Objective value</th>
<th>in figure</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENF</td>
<td>12</td>
<td>248</td>
<td>12/2</td>
<td>5.8481</td>
<td>-</td>
</tr>
<tr>
<td>ENF</td>
<td>12</td>
<td>248</td>
<td>24/12</td>
<td>diverged</td>
<td>-</td>
</tr>
<tr>
<td>ENF</td>
<td>12</td>
<td>248</td>
<td>16/16</td>
<td>diverged</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 4.10: ENF - Two Layer Network

**Figure 4.7:** Network Performance for 16/16 network on 415 samples.

**Figure 4.8:** Network Performance for 32/24 network on 415 samples.