Opening Pandora's Box, Bottom Side Up

Automated extraction of comprehensible multivariate power functions from real data

Elwin Oost

IAS
University of Amsterdam
IAS Group

Witteveen+Bos

Title page art: "Pandora" (1896) by J. W. Waterhouse (1849-1917)

The story of Pandora

Zeus had Pandora created out of clay. He breathed life into her and demanded that the gods graced her with attractiveness but a deceitful nature. She was given to Epimetheus, brother of Prometheus, whom had stolen fire from Zeus for the benefit of mankind. Epimetheus accepted, oblivious of Prometheus' warning not to accept gifts from Zeus. A box[a] was given as a wedding present, filled with malevolence Zeus had ordered the gods to include. When Pandora opened the box, all contained evils spread into the world. Only expectation, a realistic awareness of the current and probable future situation, was left in the box, which would otherwise have ended all hope in the evil-struck world[b].

[a] A less common but more correct translation would be 'jar'.
[b] Our synopsis is based on Gantz' interpretation of Hesiod (Gantz, 1993).

Opening Pandora's Box, Bottom Side Up

Automated extraction of comprehensible multivariate power functions from real data

Abstract

In this thesis, we will present a method for automated extraction of multivariate, multi-term power functions from large, real data sets. We propose several modifications for the hybrid rule extraction/equation discovery system RF5 to make it applicable for large problems, and present a pruning algorithm to strongly increase the comprehensibility of the found formulas.

Elwin Oost, Amsterdam, August 2002

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

Natural systems often present major challenges to modelers. They often contain complex interactions between the various parameters of the system. Also, non-linear relationships are often found and make matters even more difficult.

Neural networks have a reputation for quickly learning models emulating such systems. Unfortunately, as renowned as standard neural networks (Multi-Layer Perceptrons or MLPs) are for their learning skills, as notorious they are for the difficulty of extracting meaningful relationships between their inputs and outputs. Quoting Mozer and Smolensky (Mozer and Smolensky, 1989),

> *One thing that connectionist networks have in common with brains is that if you open them up and peer inside, all you can see is a big pile of goo.*

This research was performed as my graduate project at Witteveen+Bos, consulting engineers active in the fields of water, infrastructure, environment and construction. Their advanced statistics group uses conventional and multiple regression for modelling complex hydrological systems. These systems often show strong non-linear behavior, and the measurements from which models should be derived usually contain a non-trivial amount of noise.

The company has turned to the use of neural networks for such problems due to their excellent ability to learn complex models. But although the learned networks performed well for many tasks, some clients indicated they would appreciate more insight in the relationships between the inputs and outputs of the network.

1.1 Research question and overview

Is it possible to automatically extract humanly understandable relations between features in large real data sets for which successful neural models were obtained?

The method should be capable of handling data sets with at least about a dozen noisy, mostly real-valued features, optionally including some boolean inputs.

We start by surveying different existing machine learning methods for obtaining insight in relationships directly from data or from fully trained networks. An overview of existing techniques is presented in chapter 2. In chapter 3 we describe the RF5 system, which can extract multivariate power functions and seems a promising solution to our question. Improvements we have developed to solve limitations of the system are presented in chapter 4. Our pruning approach which strongly increases the comprehensibility of the found relationships will be introduced in chapter 5. We combine RF5, the improvements and the pruning algorithm into one automatic procedure in chapter 6 and apply it to a real dataset. We conclude in chapter 7 that our approach works consistently and is capable of automatically providing new insights into feature relationships.

2. BACKGROUND

2.1 Knowledge Discovery in Databases

2.1.1 Overview

Obtaining insight in the relationships embedded in large data sets is a very common problem. Warehouses are looking for hidden patterns of consumer behavior, brokers for future stock prices and climatological researchers for tomorrow's weather.

As such, *knowledge discovery in databases* has become a booming research subject, defined as "the non-trivial process of identifying valid, potentially useful and ultimately understandable patterns in data" (Fayyad et al., 1996). It can be divided in the following nine steps (Brachman and Anand, 1996):

1. Developing an understanding of the application domain, the relevant knowledge and the knowledge discovery goal.
2. Creating a target data set in which the new knowledge should be discovered.
3. Data cleaning and preprocessing.
4. Data reduction and projection.
5. Matching the discovery goal to a particular data mining method.
6. Exploratory analysis and model and hypothesis selection.
7. Data mining.
8. Interpreting mined patterns.
9. Acting on the discovered knowledge

Different KDD projects require different distributions of time required for each step. Also, chances are that during the process steps back will need to be taken when the current data set proves inadequate for discovering the needed knowledge.

2.1.2 Data mining

We will assume all the preparatory work on the data has been done, a data set exists which we believe has the potential of providing us with the required knowledge. In this thesis, we will focus on step 7, providing a method for extracting knowledge from this ready-made data set which we would like to learn more about. The analysis of the found results is also beyond the scope of this thesis.

Step 5, data mining, can again be classified in the following way:

Classification The automated grouping of data in predefined classes such as {*yes, no*} or {*red, green, blue*}. (Example: *using the Boosting algorithm* (Schapire, 1990) *to sort iris species based on their characteristics such as sepal length*)

Regression The automated learning of a function describing data with a continuous output range like a confidence interval *[0–1]* or color intensity percentage *[0–100]*.(Example: *using neural networks*[1] *for predicting house prices given characteristics such as the area crime rate*)

Clustering Grouping objects which characteristics the network considers closely related, without predefined classes (So-called unsupervised instead of the previous, supervised learning methods; example: *using the K-means* (MacQueen, 1967) *search algorithm for clustering different groups of food based on their nutrients*).

Summarization The process of finding a brief, understandable description of the data. This can vary from very simple tables listing the mean and variation to algorithms extracting if-then-else rules (example: *using the C4.5* (Quinlan, 1993) *decision tree builder for classification*) or formulas (example: *using the Bacon* (Langley, 1977) *equation discovery system for regression*) describing the data dynamics. Here, understandability of the model is paramount, while for classification and regression the main focus is accuracy.

Dependency modelling Finding dependencies between features; 'this item has a high chance of increasing when this other item increases'. (Example: *using Bayesian networks* (Pearl, 1988) *to determine the dependencies between sales of different supermarket products*).

Change and deviation analysis Finding trend changes in a database, by analyzing a time series of measurements (Example: *using the Box-Jenkins method* (Box and Jenkins, 1970) *to analyze climatological changes*).

As we have outlined in chapter 1, we are interested in why an observed system behaves the way it does, and thus use *summarization* with the goal of obtaining a comprehensible model of the system. Summarization is again a very broad category describing any kind of analysis describing the data in a brief fashion, whether it is a list of statistical measures or an equation describing dynamics in the data. First, we will turn to *equation discovery*, the most obvious choice for the task at hand.

2.2 Equation discovery

2.2.1 Overview

Equation discovery, the extraction of formulas describing a data set, was pioneered by Bacon (Langley, 1977). It has spawned a large group of algorithms related to it. Bacon is a *generate-and-test* algorithm; it iteratively generates equations, tests how well these perform and uses this knowledge to direct the generation of new conjectures. It requires an interactive test setup; Bacon does not rely on a static set of test data but needs to be able to obtain test results for setups it needs to know more about. The modelled system is supposed to be describable using a single equation.

It has had various incarnations, mainly varying in complexity of formulas (ranging from linear relationships in the first version to the additional use of nominal attributes and consideration of intrinsic properties of the modelled system).

Derived algorithms such as Abacus (Falkenhainer and Michalski, 1986), Coper (Kokar, 1986) and the Ef equation discovery subsystem of Fahrenheit (Zembowicz and Żytkow, 1992) take the unit dimensions of the data into account to generate strictly physically sensible formulas. This of course implies that the units for all features need to be known.

Ef augmented Bacon by providing the user with a scope of applicability of the found laws in the form of the attributes' upper and lower limits. An important improvement is also the ability to work using a static data set (i.e., the algorithm is not interactive). It can only find bivariate formulas. It iteratively builds regression models without the use of domain knowledge, but is better at handling noise.

[1] Described in chapter 2.3.

SDS (Washio and Motoda, 1997) is relatively new algorithm in some ways similar to BACON. It also assumes that a single bivariate formula can describe the system as a whole. The original version also required interaction with the studied system, but this limitation has since been removed (Washio et al., 2000). It uses the scale type of each feature to restrict the search space to sensible solutions.

LAGRANGE (Džeroski and Todorovski, 1994) was the first equation discovery system for ordinary differential equations (ODEs) besides algebraic equations. For ODEs, it used numerical differentiation and was unfortunately relatively sensitive to noise. GOLDHORN (Križman et al., 1999) improved it by using numerical integration and noise filters to obtain a more stable result. LAGRAMGE (Todorovski and Džeroski, 1997) included further quality improvements and the addition of context-free grammar for generation of the possible equations. A second version further improved LAGRAMGE and added context-dependent grammar. Unfortunately, it is still limited to small data sets and generates relatively many spurious or hard-to-interpret formulas on real data sets (Džeroski et al., 1999). Finally, a new technique called PADLES is being developed (Todorovski et al., 2000), which is based on LAGRAMGE and generates partial differential equation (PDE) model structures which may suit the data.

PRET (Bradley et al., 2001) is another generate-and-test algorithm, meant for solving ordinary differential equations. It uses a user-defined domain theory to generate plausible functions. Like BACON, PRET requires direct interaction with the studied system to generate new samples.

ARC (Moulet, 1994) is an combination of ABACUS and BACON (version 3), improving the handling of uncertainties and limiting the search space. KEPLER (Wu and Wang, 1991) and E* (Schaffer, 1993) use a fixed list of possible models to be fitted. IDS (Nordhausen and Langley, 1990) splits up the inputdata space in sections, each having its own equation.

While each of the mentioned algorithms deserves a much longer discussion about its relative merits, an essential limitation of these algorithms for our purposes is that they are all meant for relatively small-scale problems.

2.2.2 Summary

The studied equation discovery algorithms show promise for understanding data sets but are not ideal. Their strengths are

- Direct modelling from the data.
- Different kinds of formula structure possible.
- Continuous models possible, ideal for regression problems.
- Generated models are usually very understandable.

Unfortunately, the studied generate-and-test equation discovery algorithms do not appear to be able to handle data sets as complex as those set out to study. They are not suitable for our problems, due to the following serious limitations:

- Only a very small number of variables allowed (usually 2).
- They are quite sensitive to noise.

We will now turn to *neural networks*, which have successfully been used to create good predictors for complex tasks. In combination with *rule extraction* algorithms we will discuss in section 2.5, they are an alternative to equation discovery to gain insight into relationships in data.

2.3 Neural networks

Following is a brief history of neural networks and the used methodologies as far as they are relevant to this thesis. Interested readers who after reading this section wish to learn more are referred to books with more extensive coverage of these subjects (Bishop, 1995; Ripley, 1996; Kröse and van der Smagt, 1996).

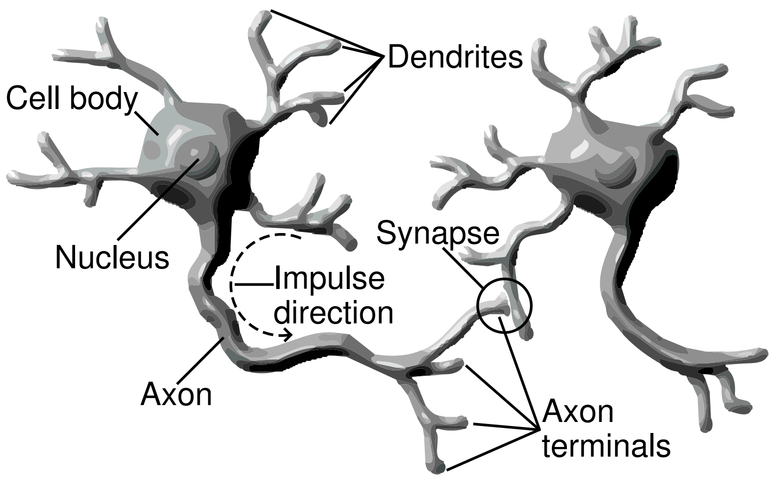


Fig. 2.1: Natural neuron

2.3.1 Introduction

The inspiration for neural networks originates from research on the brain. Describing the whole history is beyond the scope of this thesis, but good overviews have been written on its roots in a wide variety of doctrines including medicine, psychology, behaviorism and associationism (Medler, 1998; Walker, 1992).

At first, the main focus was on gaining insight in the working of the brain by discussing mathematical models believed to represent brain functions. When these models showed promise as practical tools for numerical analysis, the research focus shifted from biological plausibility to mathematical optimality.

Figure 2.1 shows a natural neuron, after which the *nodes*, the main building blocks of neural networks, have been modelled. It receives electrical pulses from its dendrites, which accumulate in the cell body. If the combined pulses have a certain strength, it sends pulses through the axon to its output synapses, which pass them on to the dendrites of other neurons. The network uses a set of *inputs* which receive data from independent variables (*features*) used by the model to create a value called *output*.

The foundation of artificial neural networks (hereafter simply called neural networks) were laid in 1943 (McCullogh and Pitts, 1943), with the proposal to emulate natural neurons using the formula

$$y = \sigma\left(B + \sum_{u=1}^{n_x} w_u x_u\right) \tag{2.1}$$

in which $\vec{x}$ is the vector containing all incoming connections to the neuron. σ is here a so-called *transfer* function, explained below. Here, the *Heaviside* function was used which outputs 1 if its input is above a certain threshold, and -1 if not.

The Perceptron (Rosenblatt, 1958), invented by Rosenblatt in 1958, was the first neural model made to solve real-world tasks (in this case, character recognition). Rosenblatt built a working neural computer based on the Perceptron, called Mark I, which he finished in 1960.

The next important discovery was the 1960 Adaline (Widrow and Hoff, 1960), by Widrow and Hoff. It was similar in design to the Perceptron, but the breakthrough was the use of the *delta rule*, described shortly.

Interest in neural networks declined around 1969. This is commonly attributed to Minsky and Papert's book *Perceptrons* (Minsky and Papert, 1969), but is likely also due to the then spreading fear of 'smart' machines popularized by Clarke and Kubrick's joint project *2001: A Space Odyssey* (Clarke, 1968).

Input 1	*Input 2*	*Target*
False	False	False
False	True	True
True	False	True
True	True	False

Tab. 2.1: XOR function

Minsky and Papert proved that single-layer Perceptrons were unable to learn any non-linearly seperable function, like the xor function shown in table 2.1. In their book, which contained valid critique for single-layer perceptrons, they also stated that according to their intuitions, multi-layer perceptrons would not solve this important limitation.

Most researchers lost their interest and funding for these techniques became about non-existent. Some researchers continued their research though; an important but not at that time widely recognized breakthrough was the development of the *backpropagation algorithm* by Werbos (Werbos, 1974), which provided an efficient way to modify the network weights for network with multiple layers, which did not exist until then.

As we now know, Minsky and Papert's intuitions were incorrect. If we use two layers of Perceptrons on top of each other, non-linear transfer functions (as described below) and a sufficient number of weights (parameters), these Multi-Layer Perceptrons (MLPs) can approximate any function (Hornik et al., 1989). Note that for some problems more layers can still give a practical advantage, sometimes learning faster and requiring less total weights than a two-layer network (Chester, 1990).

Slowly interest emerged again, not only in academic and military institutions but also gaining popularity for civil use. The big breakthrough came through Rumelhart and McClelland's book *Parallel Distributed Processing* (Rumelhart et al., 1986), which popularized the notion of backpropagation and multi-layer perceptrons.

2.3.2 Transfer functions

The transfer function can be any function, but as explained below, for most practical uses of neural networks it is important to have a continuous, completely differentiable function. Over the years, many transfer functions have been proposed (Duch and Jankowski, 1999), but the most prominent ones for neural networks are the

- *Linear* transfer function which uses the sum of all inputs as output.
- *Step* functions return constant values per input interval. Most common are Heaviside (also called the *threshold* function) and *signum* (also called *sign*). They return 1 if the summed input exceeds a certain threshold, and 0 (Heaviside) or -1 (signum) if this is not the case.
- *Sigmoidal* transfer functions which can be seen as linear transfer functions which ends have been bent to obtain asymptotical minimal and maximal values. The ones used most often are the *logarithmic sigmoid*, $y = 1/(\exp(-x) + 1)$, and the *hyperbolic tangent sigmoid*, $y = (2/(\exp(-2x) + 1)) - 1$.

These functions are illustrated in figure 2.2. The attentive reader may have noticed that the step functions are continuous nor differentiable; it works for training the original Perceptron but cannot be used for backpropagation learning.

2.3.3 Multi-Layer Perceptrons (MLPs)

The artificial neurons as described in equation 2.1 can be combined to form a network, in which the output of the neuron may be connected to several other neurons, or act as outputs of the

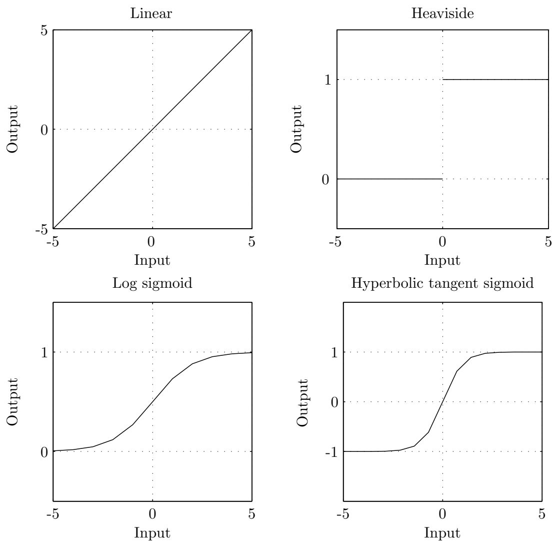


Fig. 2.2: Transfer functions

network. When the connections are strictly hierarchical (moving from node to node following the weights from input to output, there is no possibility of returning to a previously visited node), such a network is called *feed-forward neural network* or *multi-layer perceptron (MLP)*.

To obtain the output values for nodes (here called top nodes) one layer higher than the input nodes,

1. Each top node receives the values of the input nodes multiplied with the weight (parameter) value of the connection between that input and top node.
2. The nodes' *activation* function combines all received values into one value. The summation function is usually applied, and to avoid making this discription more complex than necessary we will limit ourselves to this choice for the moment.
3. Its *transfer* function possibly modifies this value, resulting in its output value.

after which this output may again be used as input for the next, higher layer, if available, in which case we return to the first step.

2.3.4 Supervised learning

If besides having the input samples in the data set, one knows the values one wishes the network to return (called *targets*), one can use *supervised learning* which presents both inputs and targets to the network. Alternatively one can use *unsupervised learning*, where the network is not told what it should output but uses heuristics to find (hopefully interesting) patterns in the data.

For supervised learning, the network is trained using an error measure E derived from the difference between target and actual output of the network. By far the most commonly used error function used for neural networks is the *mean square error*, defined as

$$E = \mathrm{MSE} = \frac{\sum_{i=1}^{n_m}(T_i - y_i)^2}{n_m}$$

which we will adapt in this thesis.

Supervised learning by minimizing MSE can be seen as a descent on a landscape with the dimensionality of the number of weights, and height level of the network error on that position.

To determine how to update the weights, it is useful to determine how the error of the output(s) changes in relation to the weights of the network. The matrix describing this, containing the

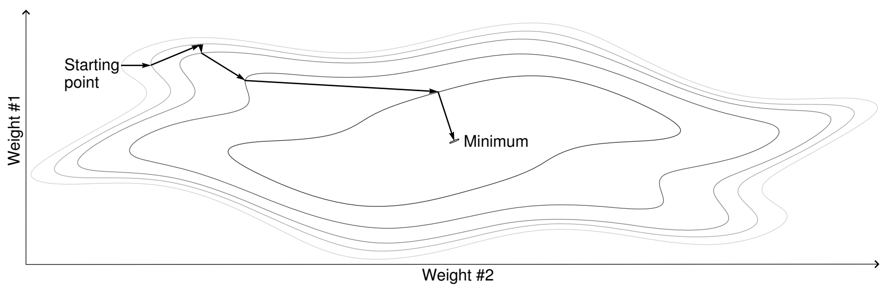


Fig. 2.3: Steepest descent under favorable conditions

derivatives of each output error in relation to each sample is called the *Jacobian matrix.* To ease the understanding, in the following discussion we will use a network of just two weights, to be able to visualize the task in 3D. We begin by defining the landscape:

- the north-south direction being the range of possible values for weight w_1 (north being higher),
- the west-east direction as the range of w_2 (east being higher)
- the height as the error of the function (which we try to minimize)

Training begins on a spot in this landscape which is chosen either at random, or, if feasible, chosen as likely being near the best (lowest) position. Unfortunately we can't see the ground due to a thick fog, but are able to find out the height of the landscape at other locations by doing a high, careful jump and landing there. Using our altimeter we can determine the height of the landscape (the error of the function), which we try to minimize.

First, we choose a direction to try. If we would randomly pick one and descent if the direction is downhill (like *random search*) it might take a very long time. Smarter algorithms use heuristics; the basic heuristic, *steepest descent*, as the name implies always selects the direction with steepest slope. This is equivalent to the negated *gradient.*

The gradient of the error, denoted ∇E, is the vector of the average of the derivatives of the error for all weights w_i. We will later describe how to calculate the gradient. For two weights, the gradient is

$$\nabla E_m = \left[\left(\frac{\partial E_m}{\partial w_1} \right), \left(\frac{\partial E_m}{\partial w_2} \right) \right]$$

while when using the Mean Square Error (MSE) error function, the average gradient $\mathbf{g}$ over all samples is used; $\mathbf{g} = \overline{\nabla E_m}$ for all m.

The gradient is simply the direction of the steepest slope of the landscape at the selected position. As north meant higher values for w_1, a positive partial gradient $\partial \mathbf{g}/\partial w_1$ means that if we would replace w_1 with a slightly higher value, we would go uphill (the network error would become higher), while directly south of the current point the error would become lower. In west-east direction, the same is true for w_2.

As such, unless we have more information available which would make us choose otherwise, it is always best to move directly against the gradient as that is the direction where the steepest error decrease on a small scale can be found. This method is known as *steepest descent.* Mathematically, it is

$$\vec{w}_{i+1} = \vec{w}_i - \lambda \mathbf{g} \tag{2.2}$$

An example is shown in figure 2.3.

2.3.5 Backpropagation

To understand backpropagation, a clear notion of the processing of the network's inputs is vital. Per node connected to the input, the input x is multiplied with the weight w between the input and the node, resulting in node inputs I. These values are combined (we here assume they are summed) to form one value C. This is in turn processed by the transfer function σ resulting in value O. This can in turn be used as input to next layers. In short,

$$x \xrightarrow{\times w} I \xrightarrow{\sum} C \xrightarrow{\sigma} O$$

Werbos proposed correcting the weights in neural networks by using the chain rule to determine the derivative of the error in relation to the individual weights. If we define the derivative of transfer function $\sigma_i^{(t)}(x)$ at the kth layer and ith node to be $\varsigma_i^{(k)}(x)$ and each output node's target value T_i, backpropagation can be described as follows.

1. Set current layer k to the total number of layers, making it point to the highest (output) layer. Compute initial network error measure $\epsilon_i^{(k)} = \varsigma_i^{(k)}\left(C_i^{(k)}\right)(T_i - O_i^{(k)})$ for $i = 1 \ldots n_y^{(k)}$.
2. Change the weights connected from layer k to the previous layer. Loop for all nodes connected to current layer k, $j = 1..n_y^{(k-1)}$,
   - Calculate gradient element $\frac{\partial E}{w_{ij}} = \epsilon_i^{(k)} O_j^{(k-1)}$.
   - Correct current weight w_{ij} by adding $\Delta w_{ij} = \lambda \frac{\partial E}{w_{ij}}$.
   - Calculate new error measure $\epsilon_j^{(k-1)} = \varsigma_j^{(k-1)}\left(C_j^{(k-1)}\right) \sum_{i=1}^{n_y^{(k)}} w_{ij}\epsilon_i^{(k)}$
3. Decrease the current layer number; $k = k - 1$. If $k > 0$, return to step 2.

The algorithm thus efficiently calculates the gradient element by element, according to the chain rule, by exploiting the network structure. It updates the weights using a factor λ of the gradient for that element.

2.4 Training of neural networks

2.4.1 Training functions

Newton's Method

Steepest descent suffers from the *narrow valley* effect; when it encounters a narrow valley, the algorithm steps past the minimum, roughly rotates its step direction repeats this mistake, taking a very long time to converge at the bottom. Also, when the network gets nearer to a minimum, the gradient gets smaller, resulting in smaller steps because these are proportional to the gradient.

More advanced, quadratic numerical optimization methods are based on Newton's method, which assumes the error surface to be quadratic (thus, that the path from the current point to the minimum can be described with a parabola). Under this assumption, the minimum can be found instantly by using the gradient $\mathbf{g}$ (vector with first-order derivatives) and the Hessian $\mathbf{H}$ (matrix with second-order derivatives), using step

$$\mathbf{s} = -\mathbf{H}^{-1}\mathbf{g}$$

In reality, this assumption generally only holds close to minima, but works very well there, while steepest descent often has serious difficulties reaching the minimum. Several problems prevent this algorithm from being commonly used for neural network training though:

- The algorithm only works when the eigenvalues of $\mathbf{H}$ are positive definite (resulting in a continuously decreasing slope towards the minimum).

- For every iteration, both the analytical gradient $\mathbf{g}$ and the Hessian $\mathbf{H}$ need to be calculated.
- The calculation costs of the matrix inversion of $\mathbf{H}$ at every iteration are relatively high.

Thus, in itself the Newton method is not useful as a neural network optimization technique. To avoid the severe limitations of this technique, a group of algorithms called *quasi-Newton* have been developed.

Quasi-Newton

As a viable alternative to the Newton algorithm, a group called quasi-Newton algorithms has been developed. Instead of calculating the Hessian $\mathbf{H}$ at every iteration, the quasi-Newton algorithms build an approximation $\hat{\mathbf{H}}$ by progressively incorporating the information from the gradient at different iterations.

They start off using steepest descent, gradually building the approximated Hessian $\hat{\mathbf{H}}$ and behaving more like the Newton method. It takes n_w steps to create a full approximation, after which the approximation is reset to the Identity matrix $\mathbf{I}$ making the algorithm once again behave like steepest descent. The basic algorithm is displayed here.

1. Initialize counter $i = 1$, weights Φ_i and the Hessian approximation $\hat{\mathbf{H}}_i = \mathbf{I}$ (Identity matrix).
2. Calculate step direction $\mathbf{d}_i = -\hat{\mathbf{H}}_i^{-1}\mathbf{g}_i$.
3. Terminate if stopping criterium is satisfied.
4. Calculate step length λ_i.
5. Update weights $\Phi_{i+1} = \Phi_i + \lambda_i \mathbf{d}_i$.
6. If $i \mod 0 \equiv 0$ then $\hat{\mathbf{H}}_{i+1} = \mathbf{I}$ else update $\hat{\mathbf{H}}_{i+1}$.

As you can see, steps 4 and 6 are open for interpretation. The most popular choice for step 6 is the so-called BFGS-method.

Using current step $\mathbf{s} = \Phi_{i+1} - \Phi_i$ and gradient change $\mathbf{c} = \mathbf{g}_{i+1} - \mathbf{g}_i$, the BFGS update is

$$\hat{\mathbf{H}}_{i+1} = \hat{\mathbf{H}}_i + \frac{\mathbf{c}\mathbf{c}^{\mathrm{T}}}{\mathbf{c}^{\mathrm{T}}\mathbf{s}} - \frac{\hat{\mathbf{H}}_i \mathbf{s}\mathbf{s}^{\mathrm{T}} \hat{\mathbf{H}}_i}{\mathbf{s}^{\mathrm{T}} \hat{\mathbf{H}}_i \mathbf{s}}$$

Using this update would imply that $\hat{\mathbf{H}}$ would need to be inverted at every step, which is computationally very expensive. We can also directly update $\hat{\mathbf{H}}$ in inverted form, using

$$\hat{\mathbf{H}}'_{i+1} = \hat{\mathbf{H}}'_i + \left(1 + \frac{\mathbf{c}^{\mathrm{T}}\hat{\mathbf{H}}'_i\mathbf{c}}{\mathbf{s}^{\mathrm{T}}\mathbf{c}}\right)\frac{\mathbf{s}\mathbf{s}^{\mathrm{T}}}{\mathbf{s}^{\mathrm{T}}\mathbf{c}} - \frac{\mathbf{s}\mathbf{c}^{\mathrm{T}}\hat{\mathbf{H}}'_i + \hat{\mathbf{H}}'_i\mathbf{c}\mathbf{s}^{\mathrm{T}}}{\mathbf{s}^{\mathrm{T}}\mathbf{c}}$$

Limited memory Quasi-Newton

For networks with a very large numbers of weights, the memory cost of this method, $O(n_w)$, may become prohibitive. This is why variants using less memory were developed. The most extreme form is the 'Memoryless' Quasi-Newton method, in which calculation of the Hessian approximation is being skipped and the search direction is calculated using

$$\mathbf{d}_{i+1} = -\mathbf{g}_{i+1} - \frac{\mathbf{s}^{\mathrm{T}}\mathbf{g}_{i+1}}{\mathbf{s}^{\mathrm{T}}\mathbf{c}}\left(1 + \frac{\mathbf{c}^{\mathrm{T}}\mathbf{c}}{\mathbf{s}^{\mathrm{T}}\mathbf{c}}\right)\mathbf{s} + \frac{\mathbf{c}\mathbf{s}^{\mathrm{T}}\mathbf{g}_{i+1} + \mathbf{s}\mathbf{c}^{\mathrm{T}}\mathbf{g}_{i+1}}{\mathbf{s}^{\mathrm{T}}\mathbf{c}}$$

(with the first search direction being $\mathbf{d}_0 = -\mathbf{g}_0$).

Other methods build up only a partial Hessian. An overview of quasi-Newton related algorithms including limited-memory ones has been written by Luenberger (Luenberger, 1984). Limited-memory algorithms do not store the complete Hessian but data of smaller size such as the history of previously taken optimization steps and the gradient change between these steps. From these data an approximation can be constructed. More recently a survey focusing on limited-memory algorithms including recently developed ones has appeared (Nocedal, 1997).

Levenberg-Marquardt

Levenberg-Marquardt (Marquardt, 1963; Moré, 1977; Shepherd, 1997) is a (pseudo-) second order algorithm which has become the algorithm of choice for small to medium-scale non-linear optimization. Here, we will describe its specifics compared to other second-order algorithms.

An efficient way to determine the approximate step length to take to minimize the error along the current search direction, is to use second-order information, that is, the second derivative of the function modelling the error surface. By assuming the error surface to be quadratic, we can calculate the point along the search direction which should be the bottom of the valley.

To do so, we define a linear approximation of the (supposedly quadratic) function for a certain weight vector $\vec{w}_0$:

$$\hat{f}(\vec{x},\vec{w}) = f(\vec{x},\vec{w}_i) + (\vec{w}-\vec{w}_i)^T \nabla f(\vec{x},\vec{w}_i) \tag{2.3}$$

Thus, the mean square error of this function in relation to the target value is

$$\hat{E}(\vec{w}) = (\hat{f}(\vec{x},\vec{w}) - T)^2$$

and its gradient

$$\nabla \hat{E}(\vec{w}) = 2(\hat{f}(\vec{x},\vec{w}) - T)\nabla \hat{f}(\vec{x},\vec{w})$$

Writing out $\hat{f}$ using formula (2.3) and the knowledge that $\nabla \hat{f}(\vec{x},\vec{w}) = \nabla f(\vec{x},\vec{w}_i)$ (since $\hat{f}$ is a linear function of $\vec{w}$) yields

$$\nabla \hat{E}(\vec{w}) = 2(f(\vec{x},\vec{w}_i) + (\vec{w}-\vec{w}_i)^T \nabla f(\vec{x},\vec{w}_i) - T)\nabla f(\vec{x},\vec{w}_i)$$

which should approximate the gradient of the real error function E near point $\vec{w}_i$.

Finally, we rewrite this equation using the average error gradient

$$\mathbf{g} = (f(\vec{x},\vec{w}_i) - T)\nabla f(\vec{x},\vec{w}_i)$$

and an approximation of the *Hessian*, the matrix of second-order derivatives, by the average of the outer products of the gradient;

$$\mathbf{H} = \nabla f(\vec{x},\vec{w}_i) f(\vec{x},\vec{w}_i)$$

Using these, we obtain

$$\nabla \hat{E}(\vec{w}) = 2\mathbf{H}(\vec{w}-\vec{w}_i) + 2\mathbf{g}$$

which is 0 for minima (and maxima). This is the case for

$$\vec{w}_{i+1} = \vec{w}_i - \mathbf{H}^{-1}\mathbf{g} \tag{2.4}$$

when $f(\vec{x},\vec{w}) \approx \hat{f}(\vec{x},\vec{w})$, which is most likely near minima of E. If we are farther away from a minimum, simple gradient descent is a better approach to take, as it is guaranteed to bring the error down.

As such, Levenberg proposed to use a mixture of (2.4) and steepest descent:

$$\vec{w}_{i+1} = \vec{w}_i - (\mathbf{H} + \alpha\mathbf{I})^{-1}\mathbf{g} \tag{2.5}$$

$\mathbf{I}$ being the identity matrix. Thus, a large α causes (2.5) to behave like gradient descent, while a small α makes it behave like the quadratic approximation. α can be seen as a distrust factor of the quadratic approximation.

When the error E goes up after having taken a step, this is caused by the quadratic approximation not working well in this part of the error surface; the gradient descent factor of (2.5) is guaranteed to decrease the error. On decrease, the gain can be attributed to either part.

As such, when the considered step proved counterproductive (E increased), the trust in the quadratic approximation is weakened by increasing α by a factor, and the step is not taken.

When E decreased, the trust is strengthened by increasing α by the same factor, and the step is taken.

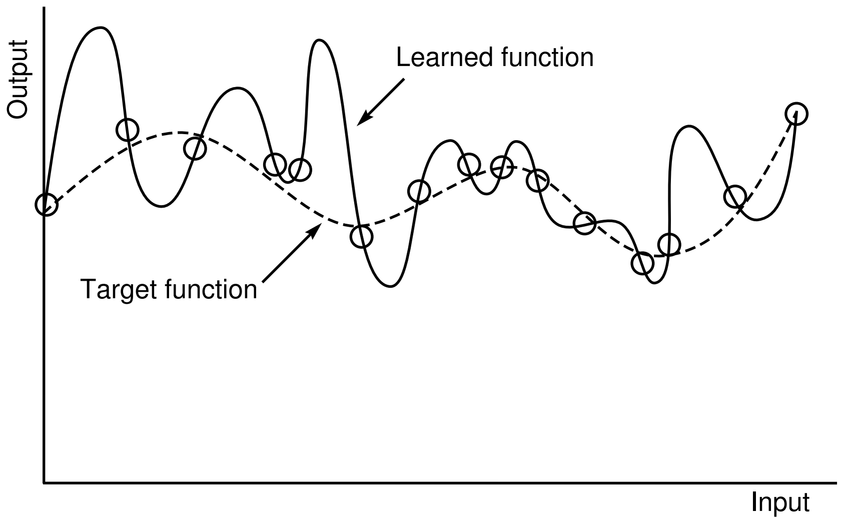


Fig. 2.4: Overfit training (circles denote training samples)

This gradient descent part of the algorithm of Levenberg was improved by Marquardt. He realized that the rough estimate of the second derivative of the error function for each weight, contained in the diagonal of the approximation of the Hessian $\mathbf{H}$, can be used as an indication of how far a step we should take for each weight w, by using the formula

$$\vec{w}_{i+1} = \vec{w}_i - (\mathbf{H} + \alpha\tilde{\mathbf{H}})^{-1}\mathbf{g} \tag{2.6}$$

where $\tilde{\mathbf{H}}$ is the diagonal of $\mathbf{H}$.

2.4.2 Generalization and overfitting

After training, care should be taken to assure the model is not *overfitting*. Overfitting happens when a model is more complex than the system being modelled. An extreme example of overfitting for a system with one input and one output is shown in figure 2.4. Here, the circles denote training samples, which are not completely on the target line due to noise in the data. Although the model does a very good job of modelling the training samples, it is practically useless for predicting the model for input values between those in the training set.

To put it formally, we are looking for models with low *generalization* error, defined as the average error on an infinite number of cases not previously used for building the network:

$$\lim_{n_m \to \infty} \frac{\sum_{m=1}^{n_m} E\left(T_m - y_m\right)}{n_m}$$

Thus, a model with small generalization error is not specific for modelling the training data but able to handle the general system. Note that oversized networks may often not exhibit overfitting but actually have superior generalization, which may be due to the decreased chance of getting trapped in local minima (Lawrence et al., 1997).

2.4.3 Model selection

The behavior of networks of a different structure (for example number of hidden nodes or choice of inputs) can vary a lot. Even identical network structures initialized with different weights can quite possibly produce very different results, due to the training ending in different minima.

The error of a model can be separated in the factors *bias* and *variance*. Bias is a systematic departure from the true answer, while variance is a random fluctuation from it. A perfect fit (no error) will of course have no bias or variance, but perfect models are about non-existent in the real world. Too small models have a relatively large bias due to their inability to capture the complete model, while too big models will have a relatively large variance due to overfitting of the model by also modelling the noise present in the training data.

As such, after a training round it is necessary to compare the quality of the found models. To do so, different model selection methods have been devised.

Hold-out validation

For hold-out validation, the available data are separated in a *training set* which the network is trained on, and a randomly selected *test set* of cases not used for training. The error of the network prediction for this test set is used as a measure for the generalization error. It is not possible to define a constant percentage guaranteed to be enough to reliably measure the generalization error. Work has been done to address this issue (Larsen and Goutte, 1999).

If rare but interesting patterns exist in the data (such as sudden peaks), *stratified* sampling can be used, which divides the data into categories with different properties, after which each category is divided with equal split ratio.

Provided sufficient samples are put in the test set, it gives an unbiased (that is, close to the true value) estimate of the generalization error. Note that if the network structure (i.e., number of hidden nodes and inputs) is also determined with the test set, there is a possibility that there is some overfitting towards the test set. To prevent this, it is best to confirm the generalization by using a third data set called *validation set* (Bishop, 1995).

Cross-Validation

In linear regression, *cross-validation* is often applied for data sets with a small number of examples. It is similar to hold-out validation, but repeats the training of the model a number of times, each time taking out a different part of the data set and with new random parameter (weight) initialization, after which the generalization error is the average of the resulting errors.

The most extreme form is leave-one-out (LOO) cross validation, which takes out one single sample every time. As the experiment needs to be repeated n_m times, this is computationally very expensive. To prevent such extreme training costs, a popular choice is ten-fold cross-validation (10CV) which as the name suggests splits the total data in ten *folds* (parts), thus requiring only ten training rounds.

For non-linear regression, the situation is more complicated. Because of the likelihood of local minima, new training rounds can obtain distinctively different models, making cross-validation strongly less sensible. As such, the non-linear cross validation technique has been proposed (Moody, 1994). To limit the irregularity, here the generalization error is estimated by first training the network on the complete data set, after which cross-validation is used starting retraining from this trained network instead of using new random weights. A more extreme proposal (Kwok and Yeung, 1995) fixes all weights except for the upper layer. We consider it likely that these options, and particularly the latter, underestimate the generalization error, as the network is likely to 'remember' the testset data.

Penalty-based model selection

Penalty-based algorithms are used to approximate the generalization performance of a model, when it is not possible to determine it more reliably (i.e., there are not enough data to form a reliable test/validation set for hold-out validation). Classical examples of these algorithms are AIC (Akaike, 1978), BIC (Schwarz, 1978), MML (Wallace and Boulton, 1968) and MDL (Rissanen, 1983).

They rate models based on their characteristics, typically their training error (MSE) and complexity (i.e., number of weights). They vary in their bias-variance tradeoff, and thus the relative importance ascribed to the model characteristics. Each algorithm has a different definition of what is 'best', making different assumptions about what the user is looking for and about the characteristics of the data.

MDL has gained a particularly strong following in the machine learning community. It is often named a formalization of Occam's Razor. Like MML, it has an information theory background. The definition of 'best' neural network model is here the one which would require the least bits to encode it and its residual error. In this context, it is again formalized in the same manner as other penalty-based network selection algorithms, using a formula with the parameters training error and number of weights.

Early stopping

During training a technique called *early stopping* can be used, aimed at avoiding overfitting, and determining whether sufficient hidden nodes have been added. Here, part of the data is removed from the training data to form a so-called *validation set*. The network is then trained as usual on the remaining data, while at each iteration the error is also being calculated for the validation set.

The idea is that the validation error will decrease as long as the network is mainly in the process of learning the target function, and that it will increase when it has learned the function and is now only learning to model the noise. This would lead to a U-shaped graph of validation error per iteration, where it would be best to stop at the bottom; thus, as soon as an increasing generalization error is detected,training is halted.

If insufficient hidden nodes would have been added, the slope would flatten out, and an extra hidden node should be added to determine whether further improvement compared to this number of hidden nodes is possible.

Unfortunately for many data sets the generalization error doesn't follow this pattern, but instead includes many ups and downs. Using early stopping would likely stop training prematurely, thus decreasing the quality of the final model.

Goodness of fit using R^2

The values of the most common error function, Mean Square Error (MSE), are hard to interpret. One can easily understand that a smaller MSE means a better fit, and MSE 0 a perfect fit, but without analysis of the data set it is impossible to tell whether a MSE of 100 might be considered a good or bad fit. There is no reference model defined as 'worst' other models can be compared with.

A statistic related to MSE is R^2, which is defined as the proportion of total variation of T about its mean $\overline{T}$ accounted for by the network output y. It is a rough measure of how close the sample data lie to the estimated regression line. Mathematically,

$$R^2 = \frac{\sum_{i=1}^{n_m}(T_i - \overline{T})^2}{\sum_{i=1}^{n_m}(y_i - \overline{T})^2}$$

As a way of trading off R^2 fit and model complexity, *adjusted* R^2 has been devised, similar to penalty-based model selection methods for MSE mentioned in section 2.4.3. Its definition is

$$adjusted\ R^2 = R^2 - \frac{(t-1)\left(1-R^2\right)}{n_m - t}$$

in which t is the model size. Like other penalty-based algorithms, it also has a specific bias weighing error and model complexity, and should only be considered a rough indication of the relative qualities of different-sized models (Draper and Smith, 1998).

2.5 Rule extraction

2.5.1 Overview

An important limitation of MLP neural networks is that it is not directly possible to gain understanding of the modelled input-output relations. As an example, consider the formula obtained for a very simple network (two inputs, two outputs and the hyperbolic tangent sigmoid transfer function). It is

$$2\left(1+\exp\left(-2w_{10}^{(2)}-4w_{11}^{(2)}\left(\left(1+\exp\left(-2w_{10}^{(1)}-2w_{11}^{(1)}x_1-2w_{12}^{(1)}x_2\right)\right)^{-1}-\frac{1}{2}\right)\right.\right.$$
$$\left.\left.-4w_{12}^{(2)}\left(\left(1+\exp\left(-2w_{20}^{(1)}-2w_{21}^{(1)}x_1-2w_{22}^{(1)}x_2\right)\right)^{-1}-\frac{1}{2}\right)\right)\right)^{-1}-1 \quad (2.7)$$

In order to obtain models from which it is possible to gain insight in the input-output relationships, *rule extraction* techniques have been devised. These analyze the network weights and/or behavior with the purpose of generating a humanly understandable model similar to the network.

2.5.2 Taxonomy

There is a wide range of different rule extraction techniques, and to allow easier comparison a taxonomy has been proposed (Andrews et al., 1995) based on several different features of these techniques. Here we present this taxonomy including an assessment of the relative importance of each criterium in relation to our subject matter.

1. Expressive power/rule format How understandable are the rules?

   Boolean/propositional if $A \wedge B \wedge \neg C$ then ... else ...

   Fuzzy Using membership functions with overlapping sets. Example: if $(x_1 = low) \wedge (x_2 = high)$ then ...

   First-order logic Has not been used yet. $\forall x_1 \exists x_2$: ... $\rightarrow$ ...

   The rule format is closely related to the comprehensibility*. Although for regression problems a comprehensible formula is preferable, it seems impossible to straightforwardly map the MLP formula to a simpler one; the existing algorithms use various forms of discretization.*

2. Quality What is the error level of the model for the training data?

   Accuracy What is the error level for the testset?

   Fidelity How closely does the ruleset mimic the neural network (will the error levels be distributed in a similar fashion)?

   Consistency Will different runs of the algorithm yield similar results?

   Comprehensibility How many rules are generated, and how long is each rule?

   Only for over-fitted networks a difference between fidelity and accuracy is to be expected. As it is vital not to have over-fitted networks in order to be able to gain insight, this is an important criterium. For non-overfitted networks, accuracy follows from fidelity; a network trained on a training set representative of the test set will yield similar errors for both. Consistency is not very important as long as the results represent the system well. As such, we will only discuss fidelity and overfitting.

3. Translucency How is the network being used to generate the rules?

   Decompositional/Local The weights and structure of the neural network are directly analyzed

   Pedagogical/Global/Oracle/Black box Only input/output pairs are being analysed

   Eclectic A combination of both methods is being used

It is not very important for us whether an algorithm examines the weights inside the network or not, as long as the results are reliable. Of course it is best if an algorithm is pedagogical as we can then use any network structure we like, and only for pedagogical algorithms the extracted rules from MLPs can be guaranteed to give a behavior approximating the network (Tickle et al., 1998)*. Other factors are much more important.*

4. Complexity What is the relationship between the size of the network and required rule extraction calculation time?

   Algorithmic complexity is very important; scientists usually require answers in limited time, and have little use for calculations requiring extensive amounts of time.

   > "My circuits are now irrevocably committed to calculating the answer to the Ultimate Question of Life, the Universe, and Everything [. . .] but the programme will take me a little while to run." Fook glanced impatiently at his watch. "How long?" he said. "Seven and a half million years," said Deep Thought. (Adams, 1979)

   This is generally not a situation scientists wish to encounter; thus, algorithms of a large complexity can only be used for simple problems. Complexity can depend on a number of factors, which may include (but are not limited to) the number of hidden nodes, inputs and input samples available. The number of hidden nodes and inputs is most often the problem with decompositional algorithms. As the focus of this thesis is large networks with many input patterns, small complexity is an important criterium.

5. Portability Does the method work just for one kind of network structure or does it have a broader application?

   The criteria pedagogical and portability are strongly correlated, as having no restrictions for the network structure implies complete portability. On the other hand, the decompositional and eclectic algorithms by definition expect a certain network structure. A free choice of the layout of a network is preferred as it gives a great deal of freedom, but for our purposes it is only necessary to have support for some network structure as powerful as MLPs.

In appendix A we present a meta-analysis of the different surveys for rule extraction algorithms. Unfortunately, the large majority utilizes *if-then-else* rules which we consider strongly limiting for the task at hand. However, the RF5 algorithm shows promise.

2.5.3 Summary

Like equation discovery, rule extraction from neural networks is an interesting technique, but also not ideal for our purpose. Most rule extraction algorithms partition the input space into sections, which may limit the understandability of relationships in regression problems.

We have discussed the taxonomy of different qualities of rule extraction algorithms and their relevance for extracting understandable rules from noisy data sets. For us, the specific form of the rules is not really important, as long as they are humanly understandable. The quality of the algorithm is definitely important, but quality is generally inversely related with comprehensibility; high quality usually means a lot of rules and thus low comprehensibility, while this is also very important. Translucency in itself is less important, as long as the restrictions do not limit the network's power. Complexity on the other hand certainly is; we require answers in a reasonable amount of time. Finally, portability is no real concern as long as it works for a technique with similar strength as normal MLPs.

Rule extraction strengths are

- Models with relatively many inputs can be tackled.

- Relatively insensitive to noise.

but again, there's a catch:

- Models are often complex and not very understandable.
- Almost all generate if-then-else rules which are not ideal for regression problems
- Indirect modelling; a model of a model is being made. This may in certain cases improve generalization but will most likely result in a lower-quality model.

2.6 Conclusion

We have presented a brief overview of *knowledge discovery in databases* (*KDD*) and focused on the step we will discuss in our thesis, *data mining*. Two methods capable of providing insight in relationships existing in data set were discussed, *equation discovery* and *rule extraction*. The latter technique uses *neural networks*, which were also explained in detail.

There are several approaches we can use to solve the problem of finding humanly understandable relationships in noisy data sets introduced in chapter 1. The methods presented in this chapter provide some answers but none is ideally suited for this task. Our ideal would be a method which combines the best of both the *equation discovery* and *rule extraction* worlds. Equation discovery returns continuous, humanly understandable models and has relatively powerful expressional capabilities. Rule extraction provides a method which is quick, noise-resistant, and can take a relatively large number of inputs. We are interested in combining both strengths while avoiding most drawbacks.

3. RF5

3.1 Introduction

An interesting hybrid which can be classified both as an equation discovery system and rule extraction algorithm is the product unit network, as implemented as equation discovery system in RF5 (Saito and Nakano, 1997a; Saito and Nakano, 1997c).

It combines several desirable features for our data mining task, both from equation discovery and rule extraction systems. It shares with equation discovery systems that it works directly on data, yields continuous formulas and has a relatively powerful expressional capability. The interesting features it shares with rule extraction are a good tolerance for noise and the ability to handle more inputs than the discussed equation discovery systems.

The system was demonstrated (Saito and Nakano, 1997a; Saito and Nakano, 1997c) as functioning with artificial data consisting of up to five related features and four unrelated ones, and with real data consisting of one to two features. It consists of a combination of three techniques;

- Using product unit networks for equation discovery,
- Training them with the BPQ algorithm, and
- Selecting the number of hidden nodes with MDL,

which we will now discuss in detail below.

3.2 Product unit networks in RF5

3.2.1 Product units

The product unit was introduced in 1989 by Durbin and Rumelhart (Durbin and Rumelhart, 1989). They provide a powerful addition to the toolbox of neural network researchers, as they allow higher orders of the unit inputs to be used. Product units are defined as

$$\prod_{u=1}^{n_x} x_u^{p_u} \qquad \text{instead of the regular summation unit} \qquad \sum_{u=1}^{n_x} w_u x_u$$

in which p is a power weight, w a multiplicative weight and n_x the number of inputs. A single product unit can therefore obtain relationships like $y = x_1^{4.14} x_2^{7.18} / \sqrt[1.4]{x_3}$, which would require a network of summation units to approximate. On the other hand, they cannot create sums and as such can't replace summation units. Also, they seem more prune to creating local minima (Leerink et al., 1995).

3.2.2 Product unit networks

Networks should not fully consist of product units, as these would be no more powerful than a single unit. Instead, the most common approach is to have a hidden layer of product units and one summation unit as output, as proposed by Durbin and Rumelhart (Durbin and Rumelhart, 1989). No biases are used in the hidden layer as these would fulfill the same multiplicative role the output weights have. This product unit network (PUN) structure is also the one we will use here. Like regular MLPs (Hornik et al., 1989), such product unit networks are universal approximators

(Leshno et al., 1993). The research focus was mainly on networks relating boolean inputs and outputs, and as such the output of the network was squashed by the used activation functions. The mostly binary focus on product units remained in most publications on product units (Leerink et al., 1995; Leerink et al., 1996; Shi and Eberhart, 1998; Ismail and Engelbrecht, 2001).

Defining n_h as the number of hidden nodes, n_x the number of inputs, w the weights between hidden and output (with w_0 being the bias weight), and p the input weights, this network structure can fit the following kind of functions

$$y = \sum_{h=1}^{n_h} w_h \prod_{u=1}^{n_x} x_u^{p_{hu}} + w_0 \tag{3.1}$$

An example of such a function is:

$$y = 1.23x_1^{-0.73}x_2^{2.34}x_3^0 - 4.56x_1^0x_2^2x_3^{-0.25} + 7.89$$

which can be simplified to

$$y = 1.23x_1^{-0.73}x_2^{2.34} - 4.56x_2^2x_3^{-0.25} + 7.89$$

To prevent output in the complex domain, the attribute vectors should be restricted to non-zero, positive values. If needed and prior knowledge admits it, this can be done by feeding inputs with such attribute vectors $\exp(x)$ instead of x. After training the weights p and w can directly be interpreted as powers and coefficients respectively.

The training of product units has not been without difficulties; the methods tried include random search, different forms of particle swarm optimization, genetic algorithms, leapfrog optimization and simulated annealing, with a number of publications on this subject comparing several of these techniques (Janson and Frenzel, 1993; Leerink et al., 1996; Shi and Eberhart, 1998; Ismail and Engelbrecht, 2001).

In 1997, Saito and Nakano introduced RF5 (Saito and Nakano, 1997a; Saito and Nakano, 1997c), an equation discovery system using these PUNs, their regularization method (Saito and Nakano, 1997b) for the number of hidden nodes based on the MDL principle (Rissanen, 1983), and their quasi-Newton learning algorithm BPQ (Saito and Nakano, 1997d). Saito and Nakano found that the network weights are directly interpretable as an easily understandable multivariate power function, if no activation functions are used.

Saito and Nakano claimed former research focused solely on binary data (Saito and Nakano, 1997a; Saito and Nakano, 1997c). However, Durbin and Rumelhart already gave the example of learning a real-valued circular domain in their 1989 article introducing the product unit (Durbin and Rumelhart, 1989), and Janson and Frenzel published a 1993 article (Janson and Frenzel, 1993) about using them for a real-valued VLSI construction task). Still, their proposal to use it for equation discovery is novel.

Properties

The networks are in the real, continuous domain. Although it has been stated that calculating the complex part of resulting equations (occurring when non-positive inputs are presented to the network) is generally not worth the effort for product units for boolean inputs (Durbin and Rumelhart, 1989), it is essential for equation discovery that correct calculations are being made. It is possible to circumvent the need for complex calculations by restricting inputs to positive values. Table 3.1 shows the valid ranges and domains. Alternatively one could handle complex number calculations, if feasible.

3.3 BPQ

A second element of RF5 is the proposed use of the second-order training algorithm *BPQ* (Saito and Nakano, 1997d), which belongs to the family of Quasi-Newton algorithms discussed in sec-

Item	Domain	Range
Inputs (x)	$\Re$	$\langle 0, \infty\rangle$
Targets (y)	$\Re$	$\langle -\infty, \infty\rangle$
Coefficients (w)	$\Re$	$\langle -\infty, \infty\rangle$
Exponents (p)	$\Re$	$\langle -\infty, \infty\rangle$

Tab. 3.1: Domain and range of the different classes of variables

tion 2.4.1. It is a compromise between a normal Quasi-Newton algorithm and memoryless Quasi-Newton as discussed in section 2.4.1. Instead of building a complete Hessian approximation using n_w weights for step direction determination it only uses a selected number of steps z before restarting using steepest descent, with $z \ll n_w$. As a result of this, it is possible to restrict the memory requirements for the Hessian to $O(zn_w)$ instead of $O({n_w}^2)$.

Instead of the original partial BFGS algorithm mentioned in 2.4.1, which stores step $\mathbf{s}$ and gradient change $\mathbf{c}$, BPQ stores $\mathbf{s}$ and the multiplication of the Hessian approximation $\hat{\mathbf{H}}$ with gradient change $\mathbf{c}$. Also, BPQ determines step length λ by minimizing a second-order Taylor approximation $\hat{y}$ of the network output y,

$$\hat{E}(\lambda) \approx E(0) + \frac{\partial E}{\partial \lambda}(0)\lambda + \frac{1}{2}\frac{\partial^2 E}{\partial^2 \lambda}(0)\lambda^2 \tag{3.2}$$

If $\frac{\partial E}{\partial \lambda}(0) > 0$, the approximation is not working well and we are actually going uphill. In this case the step direction is reset to the negation of the gradient.

When $\frac{\partial E}{\partial \lambda}(0) < 0$ and $\frac{\partial^2 E}{\partial^2 \lambda}(0) > 0$, we are going downhill with flattening slope. In this case BPQ uses the familiar "bottom of parabola" formula $-\frac{b}{2a}$ on equation (3.2), in which $a = \frac{\partial^2 E}{2\partial^2 \lambda}(0)$ and $b = \frac{\partial E}{\partial \lambda}(0)$.

If $\frac{\partial E}{\partial \lambda}(0) < 0$ and $\frac{\partial^2 E}{\partial^2 \lambda}(0) < 0$, we are also going downhill but the slope is steepening. In this case we can't use the previous tactic as it would lead to a top, not a bottom (i.e., higher than the current position). Saito and Nakano here propose to use a different error approximation $\hat{E}$ by using a first-order Taylor approximation of the network output $\hat{y} = y + \lambda\frac{\partial y}{\partial \lambda}$, which can be minimized to obtain λ. It is

$$\hat{E}(\lambda) \approx E(0) + \lambda\frac{\partial E}{\partial \lambda}(0) + \frac{1}{2}\sum_{i=1}^{n_m} \lambda^2 \left(\frac{\partial y}{\partial \lambda}\right)^2$$

minimized at

$$\lambda = -\frac{\frac{\partial E}{\partial \lambda}(0)}{\sum_{i=1}^{n_m}\left(\frac{\partial y}{\partial \lambda}\right)^2}$$

To prevent long steps from being taken, Saito and Nakano propose to use a maximal size one for the total step $\mathbf{s}$ (direction $\mathbf{d}$ times length λ). *We believe a better option might be to restrict the step length relative to the steepness of the gradient (but allowing complete steps when the optimization error is becoming small), because their heuristic might cause a large number of steps when the starting position is far from the minimum.*

Finally, if the resulting step $\lambda\mathbf{d}$ yields an error $E(\lambda) > E(0)$, an existing method (Gill et al., 1981; Battiti, 1989) is being used. If we define t to be the original step length λ and assuming the error surface to be quadratic, we can plot a curve through the current error level $E(0)$ and the error level for the current step length $E(t)$ with gradient $\frac{\partial E}{\partial \lambda}(0)$. This curve has the formula

$$E(\lambda) \approx E(0) + \frac{\partial E}{\partial \lambda}(0)\lambda + \frac{E(t) - E(0) - \frac{\partial E}{\partial \lambda}(0)T}{T^2}\lambda^2$$

and its minimum can be found using

$$\lambda_{i+1} = -\frac{1}{2}\frac{\frac{\partial E}{\partial \lambda}(0)\lambda_i^2}{E(\lambda_i) - E(0) - \frac{\partial E}{\partial \lambda}(0)\lambda_i} \tag{3.3}$$

Because the step direction points downwards, the error is guaranteed to decrease at least for some small λ. As λ is decreased with every iteration of equation 3.3, this algorithm is guaranteed to converge.

3.4 Hidden layer size selection using MDL

The last element of RF5 is the use of MDL for selection of the number of hidden nodes; Saito and Nakano propose to use all available data, add fully connected (i.e., to all inputs and all outputs) hidden nodes and measure generalization error using the MDL criterium

$$\text{MDL} = 0.5n_m \log(\text{MSE}) + 0.5n_w \log(n_m)$$

which in the example from their article correctly chooses the number of hidden units to use. Thus, the models are being selected on the basis of number of hidden nodes with optimal MDL performance.

This is a penalty-based method as described in section 2.4.3, which as described there implies specific assumptions about the data sets are being made. MDL is being used to defines the best model as the one with the shortest total data length of the encoded model and the residuals (Rissanen, 1983).

3.5 Conclusion

In this chapter we have discussed the *RF5* system developed by Saito and Nakano. It consists of *product unit networks*, the *BPQ* training algorithm and *MDL* hidden layer size selection.

RF5 seems a very useful technique, but the data sets in the demonstrated applications are much smaller than those we intend to use it on. Thus, it appears in its current state unusable for use on our data. We should find ways of expanding the applicability of the system to larger tasks.

4. IMPROVEMENTS

4.1 Introduction

Although we consider RF5 a good equation discovery system, we believe in its current state it is not sufficient for the problem we are tackling. We see several possibilities for improvement, which we will discuss here.

4.2 Input normalization

Input *normalization* is the process of translating and scaling the sample vectors for each input.

In the context of neural networks, this is mostly done in order to create a better conditioned error surface; one in which it is more or less equally turbulent in all weight dimensions, not very flat in some and very curved in others.

Some algorithms, like the standard steepest descent, greatly benefit from normalization and will more quickly find the optimum. More advanced algorithms, including the *Levenberg-Marquardt* algorithm used in this thesis, gain less by normalization. Still, especially in the context of product unit networks, there is a very practical need for normalization.

Due to their raising of input values by powers, slight changes in input values can produce drastic changes of the output value. Thus, for $p < -1$ or $p > 1$, badly normalized inputs are also likely to cause overflow or underflow on calculation of the output on a computer. In such cases normalization allows for more accurate calculations, and thus likely for better solutions.

4.2.1 Translation of inputs

Unlike scaling, translation of the input vectors causes fundamental changes to the network which cannot be undone with a change in w or p.

Translation of an input transforms the formula terms wx^p to

$$w(x+t)^p \tag{4.1}$$

If it would be possible to translate an input x without affecting the possible outcome of the network, it should be possible to undo the change in output caused by the translation by finding an appropriate $\{\tilde{w}, \tilde{p}\}$ pair to replace the original $\{w, p\}$ weights, which compensate for the translation of t for any value of x.

Theorem 1. *It is not possible to find a suitable $\{\tilde{w}, \tilde{p}\}$ pair replacing $\{w, p\}$ in wx^p able to compensate for translation of x, for all possible values of x.*

Proof by contradiction. Supposing the theorem is false, the following expression would be valid:

$$\forall t, w, p\ \exists \tilde{w}, \tilde{p}\ \forall x > 0 : \tilde{w}(x+t)^{\tilde{p}} \equiv wx^p$$

Using $t = w = p = 1$ and inputs $\{x_1 = 1, x_2 = 3, x_3 = 7\}$, we first determine $\tilde{w}$ in relation to $\tilde{p}$ for x_1:

$$\begin{aligned} \tilde{w}(x_1+t)^{\tilde{p}} &\equiv wx_1^p \\ \tilde{w}2^{\tilde{p}} &\equiv 1 \\ \tilde{w} &\equiv 2^{-\tilde{p}} \end{aligned} \tag{4.2}$$

Using equation (4.2) in (4.1) for x_2,

$$\begin{aligned}\tilde{w}(x_2+t)^{\tilde{p}} &\equiv wx_2^p \\ 2^{-\tilde{p}}4^{\tilde{p}} &\equiv 3\end{aligned}$$

we can determine the only fitting values of $\tilde{p}$ and $\tilde{w}$ for $\{x_1, x_2\}$

$$\left\{\tilde{p} = {}^2\log(3), \tilde{w} = \frac{1}{3}\right\} \tag{4.3}$$

However, using equation (4.2) in (4.1) for x_3:

$$\begin{aligned}\tilde{w}(x_3+t)^{\tilde{p}} &\equiv wx_3^p \\ 2^{-\tilde{p}}8^{\tilde{p}} &\equiv 7\end{aligned}$$

yields the $\{\tilde{p}, \tilde{w}\}$ pair valid for $\{x_1, x_3\}$:

$$\left\{\tilde{p} = {}^4\log(7), \tilde{w} = \frac{1}{\sqrt{7}}\right\}$$

which is unequal to the pair for $\{x_1, x_2\}$ in (4.3). Thus, models trained on translated inputs cannot be rewritten for untranslated inputs without losing the easily interpretable PUN equation structure. □

4.2.2 Scaling of inputs

The most important part of normalization for our purposes is scaling, since it can give all features the same order of magnitude, preventing large-scale features from gaining an artificial advantage during training. We have found that inputs can be scaled, while un-scaling of the equation found by the PUN is possible to obtain a formula for the original inputs. Since

$$\left(\frac{x}{s}\right)^p \equiv x^p s^{-p}$$

any solution

$$y = \sum_{h=1}^{n_h} w_h \prod_{u=1}^{n_x} \vec{x}_u^{p_{hu}}$$

can be rewritten to

$$y = \sum_{h=1}^{n_h} \tilde{w}_h \prod_{u=1}^{n_x} \left(\frac{\vec{x}_u}{s_u}\right)^{p_{hu}}$$

in which each input $\vec{x}_u$ has been scaled by factor $s_u \neq 0$. In this solution, each

$$\tilde{w}_h = w_h \prod_{u=1}^{n_x} s_u^{p_{hu}}$$

Thus, after finding a solution, the w for unscaled inputs is

$$w_h = \tilde{w}_h / \prod_{u=1}^{n_x} s_u^{p_{hu}} \tag{4.4}$$

which we can use to replace $\tilde{w}$ to obtain the formula for unscaled inputs. An important result is that the inputs are linearly scale-independent, and that any difference in scale will be compensated by the coefficients w. We can scale the inputs within our algorithm for optimal training performance, and afterwards straightforwardly obtain the formula for the original unscaled inputs using formula (4.4).

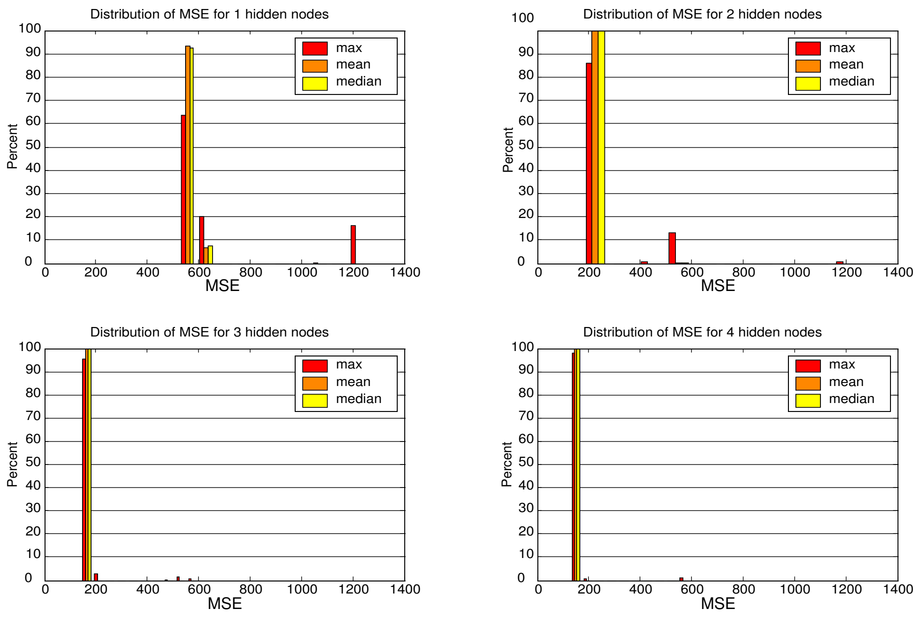


Fig. 4.1: MSE distribution for different scaling methods and network sizes

4.2.3 Comparison of scaling methods

We have studied scaling methods for the inputs both empirically and theoretically. The options we have explored are scaling by mean, median and maximum input value of the training set.

Our empirical study involved data sets with inputs scaled by *maximum, mean* or *median*, in networks using different structures (1 to 4 hidden nodes). These networks were trained on the *Groynes* data set we will discuss in section 6.2.1, using the *Levenberg-Marquardt* training algorithm.

For each network size and each scaling method, a total of 423 trainings were done before these results were aggregated (thus, a grand total of 5076 trainings were performed for this comparison). Per training round and per network structure the network weights were randomly initialized, but the same initial weights were used for each of the three scaling methods to ensure identical starting conditions.

In figure 4.1 the training error distribution is shown. As can be seen, the scaling factors didn't vary that much, but for every network structure the *mean* and *median* scaling consistently returned models with lower or equal error compared to *max* scaling. A smaller difference was found between *mean* and *median* scaling, but in those cases mean scaling returned best results. *Mean* scaling always resulted in a model with the same weight configuration, while the other methods did not.

This indicates that training a network with data scaled on its mean is less likely to result in a network trapped in a local minimum (although these chances remain small otherwise). This makes sense; by dividing by mean the inputs are scaled around 1, the point where the output is the least dependent on correct values of the power weights. Remember that $1^p = 1$, no matter the value of p. The farther away from 1 the input is, the larger the relative error will become and thus the more important the input will be for weight correction.

As scaling inputs with their mean results in these inputs having mean 1, on average each input in each sample will be closest to 1 compared to any other scaling method. Thus, each input will be in the least error-sensitive state and no input will have an artificial advantage caused by scaling, which might otherwise cause a bias in the weight training. This bias could result early in training in a stronger than normal increase of weights connected to the input if a crude correlation between target and input exists, or stronger decrease if not.

4.3 Boolean inputs

4.3.1 Introduction

We will now show the scope of RF5 can be expanded to include boolean inputs. Note that Saito and Nakano have developed a method able to handle nominal (and thus also boolean) values called RF6 (Nakano and Saito, 1999; Saito and Nakano, 2000). This would require a more complex network structure with more weights and larger training difficulty, so we prefer to use RF5 due to its smaller complexity.

It is quite possible the behavior of the network is a mixture of different formulas, dependent on the state of certain boolean parameters. Consider for example the water level in a sink, dependent on the open/closed states of the tap and the sink. For systems like this the ability to switch formulas depending on boolean inputs is essential. An example input vector would be

$$\langle \mathbf{TapOpen}, \mathbf{SinkOpen}, \mathbf{CurrentWaterLevel} \rangle$$

While former research with boolean values for PUNs was focused on boolean input-output mapping, this functionality allows the network to indicate different formula for different truth values; each term in the hidden node can be (partly) switched on and off.

4.3.2 Boolean-to-real mapping

We have found that booleans mapped to different numeric values can be successfully used in product unit networks to learn to separate different learned functions. In the following discussion we will represent the value chosen for true as B_T and the one chosen for false as B_F.

Consider a network with two inputs, x_1 being a boolean and x_2 a continuous value. We can map input x_1 with truth value *true* to value $B_T = 1\frac{1}{3}$, and *false* to $B_F = \frac{2}{3}$. Any two different, positive values will work; these values were chosen because they have mean one following our recommendation in section 4.2.3, and a factor two difference making further explanation easy (it is recommended not to choose two values very near to each other as this may cause computational difficulties in separating the functions for true and false inputs).

When the power weight for this input would learn the value 0, the *true* and *false* cases would yield the same x^p value (1), and as such the input would be deemed irrelevant. For power 1, *false* would yield $\frac{1}{2}$ the value of *true*, as this is the ratio B_F/B_T. For power 2 this would become $\frac{1}{4}$th, power 3 $\frac{1}{8}$th and so on, following a $(B_F/B_T)^p$ pattern.

The p value learned by the network thus defines how different the coefficients will be. If for example a hidden node term is valid for one *true* and not for *false*, the learned power for data sets without noise would become ∞, but for real data sets as high as needed to separate up to the signal-to-noise ratio S which determines how strong the average formula signal is in comparison to the noise in the measurements.

Thus, the smaller the learned p, the less important the boolean input is. Large p values will cause the result of x^p for *false* input values to be negligible in comparison to the result for *true* input values, for all practical purposes making the factor behave like a true boolean function switching the connected hidden node on and off. Conversely, if the boolean factor has only partial influence, the training will assign only a small positive value to p resulting in a non-neglectable term for this hidden node, whether this input is true or false.

Negated relationships (partly) inhibiting a hidden unit can be created by using negative p powers. Here, small negative weights (close to zero) partially inhibit it, while large negative weights cause near-complete inhibition.

Example

As an example, suppose we are looking for a function describing some system, for which the formula depends on a boolean. When the boolean value is *true*, the target formula $y = 4x_2^2$ applies, and otherwise $y = 5x_2^{-3}$.

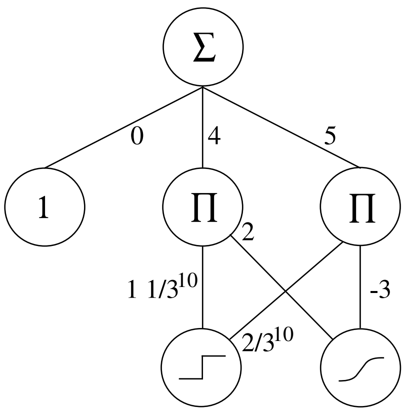


Fig. 4.2: Mixed boolean and continuous input PUN

If the data set we would train the PUN on would be perfectly noise-free (and no roundoff errors would occur), training would never complete. It is not possible to completely inhibit a node in this way; $B_F^p \neq 0$ for any p. In any real data set this is no problem because as soon as the function has been learned accurately enough that noise in the data prevents it from further improvement, training is stopped as no further progress can be made.

Suppose we have a very clear signal, with relative signal strength $S = 1024$ compared the noise (thus, signal-to-noise ratio 1024 : 1). Due to this clear signal separation, the resulting equation would require a very crisp boundary separating the two different functions using the boolean input. As the noise level is less than one per mil of the signal, both functions can be quite accurately found in the training data; only below this noise level it becomes impossible to separate these. If our algorithm performs well, the functions should thus be very well separated here.

The value the network will learn for the power weight p between a boolean input and a hidden node depends on the (unknown) signal strength S and the ratio B_T/B_F. We will here use $B_T = 1\frac{1}{3}$ and $B_F = \frac{2}{3}$ as input values for *true* and *false* respectively.

Then, the minimal constant raising the boolean inputs up to the level needed to separate the functions up to the signal-to-noise ratio is $q = {}^{(B_T/B_F)}\log(S) = {}^{(1\frac{1}{3}/\frac{2}{3})}\log(1024) = 10$. This means the boolean input should be raised to the tenth power to separate the functions as well as the noise level allows.

The functions can be modelled using a PUN with mapped-boolean input x_1 and continuous input x_2, yielding equation

$$y = \frac{4}{B_T^q} x_1^q x_2^2 + \frac{5}{B_F^{-q}} x_1^{-q} x_2^{-3}$$

as shown in figure 4.2. When $x_1 = B_T$, the first function term precisely returns the formula required for *true* input samples, while the second term returns only an extremely small value. When $x_1 = B_F$, the opposite happens; the first term becomes roughly zero, and the second term returns the formula required for input *false*.

4.3.3 Multiple biases

When a analyzed system consists of a different power functions separable using boolean inputs, these functions can of course have different added constants. While the PUN only has one bias, hidden nodes can also act as boolean-dependent added biases. They can do so when the PUN bias w_0 is trained to be one of the bias values to be modelled, while such a hidden node (one per different bias value) learns coefficient weights w_i set to the difference between that bias and w_0, while its continuous inputs are approximately 0. The hidden node thus returns a constant equal

x_1	x_2	5% noise	10% noise
true	*true*	$3519.161x_3^{-3.009}$	$3584.827x_3^{-3.023}$
true	*false*	$20.498x_3^{1.984}$	$19.301x_3^{2.007}$
false	*true*	$1.009x_3^{3.996}$	$0.815x_3^{4.104}$
false	*false*	$100.694x_3^{0.994}$	$100.138x_3^{0.988}$

Tab. 4.1: Formulas discovered by PUN

to the difference between the bias and required bias, depending on the values the boolean input weights have learned.

Example

For example, if we add a bias to above-mentioned formula to obtain $y = 4x_2^2 + 50$ and $y = 5x_2^{-3} + 150$, the learned model could become

$$y = \frac{4}{B_T^{10}} x_1^{10} x_2^2 + \frac{5}{B_F^{-10}} x_1^{-10} x_2^{-3} + \frac{100}{B_F^{-10}} x_1^{-10} x_2^0 + 50$$

which would always add 50 as constant term, but also add 100 in case the boolean input x_1 receives the $\frac{2}{3}$ mapped false value. Conversely, the network could also learn bias 150 and extract 100 in case x_1 receives the mapped true value.

4.3.4 Multiple booleans

This capability of PUNs is not restricted to one boolean input; an unlimited number of boolean inputs can be used. Each hidden unit can have a different activation based on the boolean inputs. Using the previously mentioned power-coefficient combinations, the boolean inputs can be considered to be combined with the logical operator `AND`. Each input can be used directly, first negated using `NOT` or it always returns true making the boolean input irrelevant for that hidden node (always returning false is also possible but of course not very useful, as it would result in a never-active hidden unit). Combinations of i boolean inputs can at maximum fit 2^i different functions.

Example experiment

As another example, consider the function set with boolean inputs x_1,x_2 and continuous input x_3:

$$y = \begin{cases} x_1 = true & \begin{cases} x_2 = true & \rightarrow 3500x_3^{-3} \\ x_2 = false & \rightarrow 20x_3^2 \end{cases} \\ x_1 = false & \begin{cases} x_2 = true & \rightarrow x_3^4 \\ x_2 = false & \rightarrow 100x_3 \end{cases} \end{cases} \tag{4.5}$$

which can be modelled in a four-node PUN as

$$y = \frac{3500}{B_T^{2q}} x_1^q x_2^q x_3^{-3} + \frac{20}{B_T^q B_F^{-q}} x_1^q x_2^{-q} x_3^2 + \frac{1}{B_F^{-q} B_T^q} x_1^{-q} x_2^q x_3^4 + \frac{100}{B_F^{-2q}} x_1^{-q} x_2^{-q} x_3$$

The PUN is indeed able to recover distinct formulas in this way. A test training for the formulas in (4.5) using 1000 samples per formula, in which the inputs were corrupted with 5% and 10% normally distributed noise, yielded the formulas described in table 4.1, which clearly closely resemble the original formulas from (4.5).

Here, the hidden unit with the largest contribution was considered the right formula; as described above the other hidden units all contributed as well. However, their contribution was very

Boolean inputs		5% noise				10% noise			
x_1	x_2	HN1	HN2	HN3	HN4	HN1	HN2	HN3	HN4
$true$	$true$	100.00	0.00	0.00	0.00	99.99	0.00	0.01	0.00
$true$	$false$	0.00	100.00	0.00	0.00	0.01	99.39	0.00	0.60
$false$	$true$	0.00	0.00	100.00	0.00	0.62	0.00	99.32	0.06
$false$	$false$	0.00	0.00	0.00	100.00	0.00	0.39	0.20	99.41

Tab. 4.2: Node activation distribution (%) per input combination and noise level

small, and thus clearly attributable to the noise in the data. The relative activations levels of the hidden nodes, sorted in the order of function set 4.5, is shown in table 4.2.

If the value of the power weight of a boolean is low and thus brings a less sharp distinction, one can straightforwardly extract the different formulas for the different combinations of values of the booleans. To do so, all that is needed is to multiply each coefficient by the product of the x^p factors for each connection between a boolean input and the hidden node belonging to that coefficient, and then removing these factors from the resulting equation.

4.3.5 Conclusion

In conclusion, boolean inputs can be used by PUNs to create clearly distinct formulas. The formula represented by each hidden node per boolean value can be determined by removing the boolean input factor x^p and multiplying the hidden node coefficient with B_T^p to obtain the formula for input value *true*, and with B_F^p for *false*. This can be done in an iterative way if multiple boolean inputs exist.

4.4 Analytical weight determination

If we would be able to set the weights of the network analytically, even as a first approximation, this might speed up and improve the solution finding process. We found it to be possible for cases where an *oracle* (a model which can provide target values for requested input vectors, such as a previously trained neural network) is present. Using our technique, the network can have an unlimited number of inputs, but only one hidden node.

When inputs are set to one, the PUN equation is simplified as its power weight for that input becomes irrelevant (as 1^p remains 1 for any p). We exploit this fact by feeding the network with inputs 1 except for one input which we feed different values. This means an unlimited number of inputs can be handled using this method.

We have found that it is possible to reduce formula complexity by using a smart choice for the interval between given input values x; best results were obtained when their magnitude differed by a factor exp(2). Thus, we choose input values $x_i = \exp(2^{i-1})$, with $i = \{1, 2, 3 \ldots\}$; any real positive value can be used though.

In these formulas, we have also included a scaling factor s which can be used to scale the input values to the input domain. This is important to evenly distribute the samples over the input domain and avoid using input values outside this domain.

4.4.1 Calculation

A number of different samples equal to the degrees of freedom in the network (i.e., number of weights) is required to estimate all parameters. Thus, for one hidden node, we need three measurements. When all inputs except for the one under investigation are set to value one, the resulting PUN formula is $y_i = w_0 + w_1 \exp(2^{i-1}p_1)$ which we can rewrite to

$$w_1 = \frac{y_i - w_0}{\exp\left(2^{i-1}p_1 s\right)}$$

to yield for $i = \{1, 2, 3\}$

$$w_1 = (y_1 - w_0)\exp(-p_1 s) \tag{4.6}$$
$$= (y_2 - w_0)\exp(-2p_1 s) \tag{4.7}$$
$$= (y_3 - w_0)\exp(-4p_1 s) \tag{4.8}$$

By equating one of these formulas for w with one for $w+1$, we obtain a new equation which can be rewritten to obtain the value of other variables, of the general form

$$p_1 = \frac{\log\left(\frac{y_{i+1}-w_0}{y_i-w_0}\right)}{is}$$

Equating formulas (4.6) and (4.7) and rewriting it to obtain a formula for w_1, and repeating this for formulas (4.7) and (4.8), we find

$$p_1 = \frac{\log\left(\frac{y_2-w_0}{y_1-w_0}\right)}{s} \text{ and } \frac{\log\left(\frac{y_3-w_0}{y_2-w_0}\right)}{2s}$$

which we can again equate. This can finally be rewritten to a formula for bias w_0, resulting in the rather complicated formula

$$w_0 = \frac{y_2^2 + y_2y_1 - 2y_3y_1 + y_2\sqrt{-3y_2^2 + 4y_2y_3 + 2y_2y_1 - 4y_3y_1 + y_1^2}}{3y_2 - 2y_3 - y_1 + \sqrt{-3y_2^2 + 4y_2y_3 + 2y_2y_1 - 4y_3y_1 + y_1^2}} \tag{4.9}$$

4.4.2 Results

We have performed several experiments on this technique. Although working well when no noise is present, it is extremely sensitive to noise, often causing the formula below the square root in equation (4.9) to become negative, resulting in imaginary numbers. This sensitivity to noise also results in unusable initializations in case the system cannot be perfectly described using multivariate power functions. Also, the limitation of using one hidden node is a serious impedance for use of this technique. Thus, we do not consider it viable for initializing PUNs.

4.5 Breadth-First search

In the limited number of papers published on product units, much focus has been on local minima (Leerink et al., 1995; Ismail and Engelbrecht, 2001). In our experience this problem only arises for small synthetic problems. For real data, we got very consistent well-generalizing minima with training, on par with MLP training performance. A possible explanation for the difficulties presented in the literature is that local minima may happen more for problems having boolean outputs.

For cases where such problems would arise, we propose to use a breadth-first search approach which can be summarized as follows:

1. *Quick model generation*
   Do a number (say ten) of brief (say one-tenth of the normal training epochs) successive trainings, resetting the weights to random for every new round.

2. *Model selection*
   Select the network with the lowest validation error (or other method as described in section 2.4.3).

3. *Further training*
   Continue training it using normal, longer training rounds.

The few cases where we experienced local minima were solved using this method. Large real data sets are unlikely to require this technique due to their smoother error surface and decreased likelihood of local minima with more dimensions. Instead of doing a fixed number of training rounds one can also repeat an unspecified number of times until the validation error drops below a certain threshold.

4.6 Levenberg-Marquardt training

We propose to use Levenberg-Marquardt instead of a quasi-Newton algorithm such as BPQ, as it has shown to be superior to quasi-Newton for many real-data tasks (Shepherd, 1997; Sarle, 2002). Because of its empirical success it is unanimously recommended as default optimization algorithm for non-linear regression by the authors of the major mathematical analysis suites (Wolfram, 1999; SAS Institute, 1999; Demuth and Beale, 2000; StatSoft, 2002).

Levenberg-Marquardt is ideally suited for the equation discovery task; its limitations of use are no problem:

- **Memory**: Levenberg-Marquardt should not be used with large number of weights[1], since it requires calculation and storage of the Hessian with a size proportional to the squared total number of weights. *Equation discovery shouldn't start with such huge networks as the resulting equations are extremely unlikely to be humanly comprehensible.*

- **Residuals**: It does not work well for systems with large residuals (that is, a large remaining error after the network is optimally trained). *When a data set causes large residuals it means the found equation is a very bad model from which no real insight can be gained. Thus, large residuals already indicate we should take a step back in the knowledge discovery process to increase the data set quality and/or reassess the applicability of the network for this problem.*

Although we could not implement BPQ in time to enable a direct comparison, we have compared its ancestor, BFGS Quasi-Newton, with Levenberg-Marquardt on both synthetic and real data sets. We used PUNs with up to 100 weights, which is a relatively large number of weights for a PUN used for equation discovery. For data sets yielding small residuals (which is required for obtaining a meaningful equation), the test results were indeed favorable for Levenberg-Marquardt. Since BPQ is developed to decrease the memory use by trading off some performance, we conclude Levenberg-Marquardt is preferable over BPQ for equation discovery.

4.7 Hidden layer size selection using test set

Unlike the other PUN network layers, the size of the hidden layer is not fixed and needs to be chosen. As discussed in section 3.4, Saito and Nakano propose to use MDL to do so. The advantage is that all samples can be used for training. As we have discussed in section 2.4.3, such penalty-based methods make a specific bias-variance tradeoff, assuming a specific relationship between the network error and network size.

According to the 'No Free Lunch' theorem (Wolpert and Macready, 1997), for any algorithm which fares well on a data set there exists another data set on which it will perform badly. It is an open issue how common such situations will be for real-world data sets. Penalty-based algorithms seem quite sensitive to the problem since they use very limited knowledge about the data sets. Good discussions of the limitations of penalty-based algorithms have been written (Webb, 1994; Webb, 1996; Kearns et al., 1997; Domingos, 1999).

Whenever sufficient data is available, we consider using a test and validation set to obtain a reliable assessment of the generalization error is preferable. but when limited data is available, a reliable test or validation set may not be possible to create. We believe the validation set can

[1] An estimate of the maximum number of weights can only be given in relation to a specific computer architecture and problem. A modest Athlon 700 with 256MB RAM could easily handle some hundreds of weights with Levenberg-Marquardt for the problems we have encountered.

under certain conditions be equal to the test set. When it is used very sparingly (such as using it just a couple of times for comparing different completely trained models, and not systematically such as using it for early stopping) we assess the chance of overfitting extremely small for most situations.

For rule extraction the absence of a precise generalization indication is not necessarily a problem. As the main objective is insight in the underlying natural systems, we regard querying domain experts to be preferable. They can decide which equations make sense given the current knowledge in the field. Quoting Henderson and Velleman (Henderson and Velleman, 1981),

> *The data analyst knows more than the computer (...) failure to use that knowledge produces inadequate data analysis.*

There always remains uncertainty about the optimal number of hidden nodes. Except for the theoretical possibility that we would have found a perfect model (i.e., zero remaining error, which is extremely unlikely) there is no way of telling how many hidden nodes should be used. This is true for PUNs as well as MLPs. It is quite possible that a certain signal in the data can only be modelled using a couple of hidden nodes. Then adding nodes up to one less than this required number of nodes will not yield an increasing performance. Similarly (and also like MLPs), it is impossible to determine how many rounds one should initialize and train to obtain the best possible performance; one can never guarantee the network is not stuck in a local minimum. Such minima, when far from the global minimum, are likely to diminish the reliability of any extracted formula. When near to the global minimum, extracted equations are likely to still give valuable insight about the system.

The number of training steps should also be chosen. In case a network takes a long time to converge, the network will have found a plateau (area in the weight space where the error hardly decreases with training). It can have a minimum, but also be followed by a steep error decrease after it has been crossed. (Too) large networks can require early stopping as discussed in section 2.4.3 to prevent overfitting.

When multiple global minima with similar error levels but distinctively different weight configurations exist, the system can be described in different ways. This may be an indication that there are input correlations with interchangeable (combinations of) inputs.

To assess the number of required initialize-and-train rounds before the best model is most likely found, the user can consider the number of minima found during training; if many different local minima are found the network error surface is likely to be very convoluted and as such more rounds than otherwise are recommended. As the learning rate is adaptive for Levenberg-Marquardt, defining the current balance between steepest descent and Gauss-Newton, the starting value is not very important. We have used value 0.01, as suggested by Demuth and Beale (Demuth and Beale, 2000).

4.8 Conclusion

In this chapter, we have suggested several improvements of the RF5 algorithm. We have proposed scaling of each input feature with its mean, to improve network training performance. We have proved an other normalization technique, input translation, to be impossible for PUNs without making the resulting formulas significantly less understandable. PUNs including boolean inputs were shown to be able to accurately map different boolean input combinations to different regression functions. Analytical initialization of the weights was proved to be theoretically possible for one hidden node and unlimited number of inputs, but unusable for patterns containing even small amounts of noise, and badly scalable due to lack of support for multiple hidden nodes. A breadth-first search method has been proposed which often resulted in obtaining satisfactory results much quicker than the usual depth-first search strategy. Levenberg-Marquardt training was proposed instead of Quasi-Newton methods such as BPQ since the algorithm has superior characteristics for training PUNs for equation discovery. Finally, we proposed to not always use MDL but to default to using validation and test sets for generalization assessment.

5. NETWORK SIZE OPTIMIZATION

5.1 Introduction

In chapter 4 we have proposed techniques which expand the usability of PUNs to larger problems. One serious problem remains; the formulas extracted from the PUNs are often very large, and as such hard to interpret. We would prefer to be able to have finer control of the network size than the selection of number of nodes proposed by Saito and Nakano as discussed in section 3.4. In this chapter we will propose a way to fine-tune the network size to optimize the comprehensibility of the resulting formula.

We propose to train networks with different numbers of hidden nodes (up to maximal computationally feasible complexity or until no improvement seems to be gained by adding more nodes) and compare the formulas resulting after pruning these. Pruning can strongly increase of the comprehensibility of the resulting formulas. The product unit network always yields a function with n_h terms each consisting of n_x factors $x_j^{p_{ij}}$, and a bias. It is likely that not all of these terms are important; for example, if a power p_{ij} equals 0, it can immediately be discarded as the outcome will always be 1.

For two different reasons, it is very important to minimize the size of PUNs as long as its accuracy is not seriously decreased:

- The function is much more likely to overfit using a full network, as it is unlikely all hidden units will require all terms with all features.
- The resulting function will be a lot less comprehensible for the user, particularly for networks with many features.

Besides training custom-sized networks using prior knowledge, there are two ways to create networks without full layer connections; *construction* and *pruning*. Construction starts off with an empty network and adds connections until it considers the network 'finished'. Pruning starts with a network with fully connected layers and prunes away connections until the network is considered small enough.

5.2 Arguments for construction

Which size optimization method is best is as yet a controversial issue in the machine learning community. We will discuss some arguments for construction in relation to the pruning approach (Kwok and Yeung, 1997), following with a brief assessment of its impact on PUN pruning:

- No difficulty determining starting network size.
  For our purposes this is not a serious issue, as pruning can also be done in a stepwise fashion, starting out with pruning factors from one hidden node, then training a new network with two hidden nodes and pruning factors from that one, raising complexity until either computation limitations or common sense (persistent lack of improvement) determine a maximum.
- Computation costs are economical as the network starts small and only increases up to the final network size instead of starting too big.
  This argument assumes solutions will be found at the same rate as pruning algorithms, which we consider unlikely, as discussed in section 5.3.

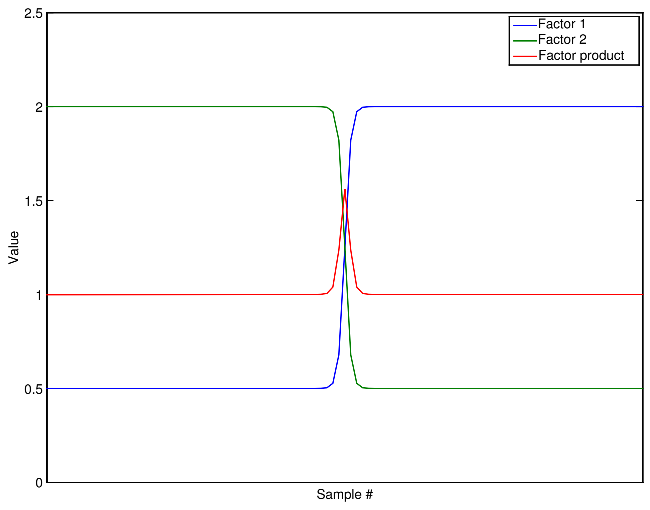


Fig. 5.1: Difficult pruning situation

- As retraining for all possibilities is usually too expensive, pruning algorithms can only approximate the generalization error per connection removal and thus may introduce large errors.
  The better a pruning algorithm can estimate the expected increase in error, the less it will be subject to this problem. We consider pruning algorithm for PUNs strongly less sensitive to this problem compared to MLP pruning algorithms, due to the mathematical structure of PUNs.

- More likely to find smaller solutions, as constructions starts small and networks with different sizes can perform similarly.
  This is a possibility, though strongly less so if one starts pruning from networks with a progressively increasing number of hidden nodes.

The last point is illustrated in figure 5.1. The first feature has value 0.5 for almost the first half of the training samples and ramps up to 2 for the latter half, while the second feature is its inverse. The output of a PUN hidden node for these factors with powers one is also displayed. It nearly always has value one except for a small peak halfway the sample space. Removing the two factors together would not cause any real increase in MSE for most networks, but the error of removing one single factor without the other (and without retraining) would likely cause a severe MSE change. Thus, in such cases the importance of the related factors appears higher than justified.

5.3 Arguments for pruning

Although we have not found any paper in the pruning literature dealing with arguments against constructive algorithms in relation to those for pruning, we will here present some critique we have about these techniques.

- Severe search space limitation
  After the network size has been increased, constructive algorithms continue their search exactly where it ended before this increase (with the exception of the addition of extra weights). As such, the weight configuration of new networks will only start from the minima in the weight space for the smaller network (again except for the added weights), which will severely limit the explored weight configurations.

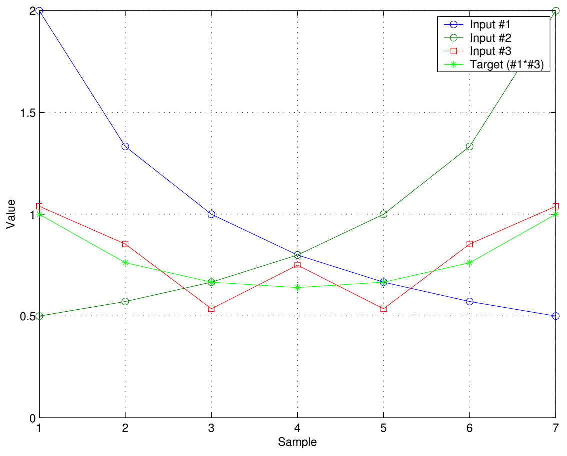


Fig. 5.2: Difficult construction situation

- Difficulty of finding complex relationships
  It may well be that a relation can be described by a combination of features or hidden nodes, while every single feature or hidden node separately does not contain useful information, causing the algorithm to make incorrect addition decisions. In this case the right combination will only be found by chance, likely causing strongly oversized networks.

The latter point is illustrated in figure 5.2. Consider a network with three potential features and (conveniently scaled) samples. The first feature samples follow a $1/t$ pattern, those for the second one its inverse over the sampled time frame and the third a W-shaped pattern. The target function can be perfectly fitted as the product of the (first powers of the) first two features (thus, a PUN connected to features one and two, with all weights 1 and bias 0). On making the first connection the construction algorithm will begin with feature three, as it resembles the target function more than either of the other functions alone. While we believe pruning to be a more natural choice for the non-linear problems with complex feature interactions neural networks are usually applied to, we have also attempted the use of a construction algorithm.

5.4 Network construction

Construction algorithms can be divided in two classes:

- *Coarse-grained* The algorithm adds a fully connected hidden node (i.e., connected to the output node and all feature nodes). Thus, the algorithm only selects the number of hidden nodes and is the most basic form of construction algorithm.

- *Fine-grained* Only one or few connection between an feature and hidden node is added, or a hidden node with one or a few features.

To improve the results of the construction algorithm, we propose to add a pruning step after the construction; if the pruning would not improve the network quality one could simply keep the final constructed network.

5.4.1 Coarse-grained construction

Coarse-grained construction is a very basic technique, adding a complete hidden node at a time. Ash introduced it with his *Dynamic Node Creation* algorithm (Ash, 1989), as shown in figure 5.3. Here, Ω denotes the complete network, that is, structure and weights. $E(\Omega)$ is the average error for all samples using this network.

```
1. Initialize network $\Omega_0$ with only a bias $\overline{T}$, time step $t = 1$.
2. Generate $\Omega_t = \Omega_{t-1}$ with an added fully connected hidden node (*we set the coefficient weight* $w = 0$) and small random values for the new input weights. Train it and save its error.
3. If $E(\Omega_t) < E(\Omega_{t-1})$ for the testset: $t = t + 1$, return to step 2.
```

Fig. 5.3: Dynamic node creation

```
1. Initialize network $\Omega_0$ with only a bias $\overline{T}$, time step $t = 1$, current hidden nodes $n_h = 0$.
2. For $u = 1$ to $n_x$:
    - For $h = 1$ to $n_h$: Generate $\Omega_{\ni hu} = \Omega_{t-1}$ with an added connection between hidden node $h$ and feature $u$, train it and save its error.
    - Generate $\Omega_{\ni n_h+1u} = \Omega_{t-1}$ with extra hidden node $n_h+1$ connected to feature $u$, train it and save its error.
3. Save $\Omega_t$ as the network with lowest error from step 2.
4. If $E(\Omega_t) < E(\Omega_{t-1})$ for the testset: $n_h = n_{h\Omega_t}$, $t = t + 1$, return to step 2.
5. Prune the network.
```

Fig. 5.4: Stepwise construction algorithm

The algorithm simply adds new hidden nodes until error criteria have been satisfied; in this implementation, that means that a (local) minimum error has been reached. It is only recommended for simple problems with relatively few features; otherwise training would still result in many connections being created making the formula more complex. Otherwise, it would still be advisable to switch to the constructive algorithm discussed in section 5.4.2.

5.4.2 Fine-grained construction

In case the network has many features and/or the problem is complex (requiring many hidden nodes), coarse-grained construction is not ideal. Here, it is recommended to build the network in a more fine-grained way. Several methods have been proposed in the literature, as well as a taxonomy (Kwok and Yeung, 1997).

A straightforward implementation is shown in figure 5.4. It generates PUNs with one extra connection in the bottom (power) layer for every feature and hidden node between no connection exists as yet, as well as models with an extra hidden node which in each model is connected to a different feature node. As retraining is necessary for every created model, this algorithm does not scale well; our motivation for creating it was to study how node creation works for PUNs.

Unfortunately, the algorithm performed quite badly, even though we expected it to fare well except for the high computational overhead. For several problems we have tested, the algorithm did not converge on a satisfactory solution. Instead, it created dozens of hidden nodes each with one power connection (thus, one x^p factor), while the target formulas only used a small number of hidden nodes.

As the heuristics behind other constructive algorithms are mainly aimed at improving the scalability at the cost at as little as possible accuracy, we do not believe they hold much promise for PUN networks.

5.5 Existing pruning methods

The most reliable way to create a well-performing network is simply training a completely connected network. The computational complexity of pruning such a network can be prohibitive when there are many connections (due to many features and/or hidden nodes). It is best if the pruning algorithm already starts off with a network which is roughly the right size.

Thus, for networks with a small number of features, it is best to simply use an algorithm adding completely connected hidden nodes. For tasks with many features, for which pruning the resulting fully connected network is likely to be a practical impossibility, one could use the *constructive* algorithm to build a network connection by connection until it is roughly the right size (most likely still a little larger than necessary). In both cases it would be best to prune the resulting network afterwards (when pruning does not yield better models one can keep the current model).

5.5.1 Weight decay

As was mentioned in section 5.1, factors with power weight $p = 0$ can be immediately discarded. Factors with powers $p \approx 0$ are also likely to be removable without significant decrease in performance. The same applies to regular feedforward neural networks; in both cases the output is likely to have little effect on the shape of the output. *Weight decay* (Weigend et al., 1991) is based on this observation, and rates weights according to their distance from zero $|p_{ij}|$.

Although a strong correlation exists, it is not right to simply define the importance of the factors as this distance, as the shape of the feature over the sample space is also important. To illustrate this, consider the feature vectors fed to inputs x_1 and x_2

- $f_1 : [0.1, 0.2, 0.1, 3.6]$ for input x_1, and
- $f_2 : [0.9, 1.0, 1.1, 1.0]$ for input x_2,

with respective powers $p_1 = 1$, and $p_2 = 2$. Ordering the factors x^p according to their power p would clearly denote $x_2^{p_2}$ to be twice as important. The error increase due to removing input 1, $E(\Omega \not\ni f_1) \approx 2.2898$, is much larger than that for removing feature 2, $E(\Omega \not\ni f_2) \approx 0.9018 \times 10^{-4}$. Despite the squared output of input 2 due to p_2 it remains quite monotonic compared to factor 1.

5.5.2 Sensitivity-based pruning (SBP)

Sensitivity-based pruning (Moody and Utans, 1992) ranks features as pruning candidates according to the error value for networks in which the selected feature has been replaced by its mean. The *saliency*, defined as $MSE_{original} - MSE_{pruned}$, is calculated, after which the feature with the smallest saliency is removed.

The disadvantage of this method is that it is rather coarse-grained, removing complete feature nodes. For PUNs it thus removes all x^p factors in all terms for the selected p. Also, the top layer weights are not adjusted to re-balance the hidden units after removing the feature node, while this may well improve the pruning results.

5.5.3 Using variance

A better way to determine the importance is to sort on the contribution variance of each weight (Sietsma and Dow, 1991). This method has been developed for standard MLP networks, measuring the variance of $x \times w$, but can be readily applied to PUNs by measuring x^p.

Here, smaller variance indicates a flatter shape of the output of the factor, and thus less influence on the shape of the network output. The factor with the smallest variance would therefore be a prime candidate for pruning.

For the cases presented in section 5.5.1, the variances are $\mathrm{var}(\vec{x}_1) = 9.02$ and $\mathrm{var}(\vec{x}_2) = 0.0802$, thus clearly indicating feature node x_2 should be pruned. Using variance can also be deceptive. If we take

- $f_1 : [0.1, 0.2, 0.1, 3.6]$ for input x_1 and
- $f_3 : [1.99, 0.01, 1.99, 0.01]$ for input x_3

and both feature nodes are raised to power $p = 1$, we obtain $\text{var}(f_1) = 9.02$ and $\text{var}(f_3) \approx 3.9598$. Thus, although feature 1 is considered most important here (and thus the least interesting target for pruning), the minimum possible MSE when it is removed is $E(\Omega \not\ni \vec{x}_1) \approx 0.1226^{e-2}$ while $E(\Omega \not\ni \vec{x}_3) \approx 0.647^{e-1}$. Here, the shape of factor 3 is the culprit; the small value at the end almost cancels out the extreme non-monotonicity in factor 1.

5.5.4 Skeletonization

This algorithm (Mozer and Smolensky, 1989) is focused on removing whole units at a time. The underlying idea behind it is to add a gating variable with a range of 0 for no output to 1 for complete normal output, and to determine the importance of the unit using an approximation of the derivative of the error with respect to this variable for values 0 and 1. As this approximation fluctuated wildly in their experiments, they proposed using an exponentially decaying average of the derivative.

Since this algorithm prunes complete nodes at a time and we are looking for a pruning method for single connections, we do not consider it for further analysis. The fact that the approximation already fluctuates wildly for MLPs makes us doubt it would work satisfactory for the more sensitive PUNs though.

5.5.5 Optimal Brain Damage (OBD)

This algorithm (Le Cun et al., 1990) requires the network to be trained to a minimum; the gradient needs to be zero for the algorithm to work. It ranks pruning candidates based on *saliency*, a measure of the second-order derivative of the network output error in relation to the weight w_i, defined as

$$\mathit{saliency}(w_i) = \frac{1}{2}\mathbf{H}_{ii} w_i^2$$

A second-order Taylor series is used to approximate the network error, which is only likely to be valid for reasonably small changes, which setting the weight to zero hardly seems to be. The error is estimated directly without any retraining being done. Their experiments indicated no resulting problems though.

Here $\mathbf{H}_{ii}$ is approximated by assuming the Hessian to be diagonal, which means that the principal axes of the network need to be parallel to the weight axes. This is often not the case in nonlinear optimization, and considered a weakness of OBD. Also, the criterium that the network should be trained completely to the bottom of a minimum can be difficult to achieve for large problems.

5.5.6 Optimal Brain Surgeon (OBS)

To avoid the diagonality assumption of the Hessian in the Optimal Brain Damage algorithm, the *Optimal Brain Surgeon* (Hassibi and Stork, 1993) algorithm was developed. It works in a similar fashion but completely calculates the Hessian, and uses a validation set to stop pruning once the validation error begins to rise.

One of the major drawbacks of this algorithm is the high computational overhead of calculating the Hessian every time a weight is pruned. Also, like OBD the network needs to be completely at the bottom of a minimum, while for large problems this is often not easy to achieve in practice.

In our experience this makes the algorithm a lot less usable, as we have often encountered very long plateaus which appeared to be close to the true minimum (in such cases the prediction accuracy was already very good and MLP training also could not yield better models). Hence, training was stopped before a minimum was found. In such cases OBS (and OBD) cannot not be used.

5.6 Enhanced Sensitivity-based Pruning (ESP)

5.6.1 Introduction

The pruning algorithms discussed in section 5.5 were all developed for MLPs, and as such none exploit the specific characteristics of PUNs. Its structure can be used to obtain better estimates of the importance of a connection. To do so, we have developed the Enhanced Sensitivity-based Pruning algorithm. It is an enhanced form of sensitivity-based pruning which can prune per single x^p factor instead of per feature (which would remove all x^p factors for that specific x from all hidden nodes), and can yield much better errors estimations for PUNs.

As we have discussed in section 3.2.2, the output of the PUN per hidden node consists of a coefficient w multiplied with the x^p factors for every connected input, thus yielding a $wx_1^{p_1}x_2^{p_2}x_3^{p_3}$ term for a hidden node connected to three inputs. Thus, replacing a x^p factor by a constant is the same as removing the factor and multiplying the w of that node with this constant. We will only consider pruning of the p weights, as pruning the w weights means pruning a whole hidden unit at a time. This can still be done in a stepwise fashion by pruning all p weights.

5.6.2 Calculation

Per factor x^p we will determine the MSE without it, adjusting the network bias and related coefficient to minimize the error increase. Since the output of the network is a direct summation of the bias w_0 and hidden node outputs multiplied by their respective coefficients w, the problem can be rephrased as finding the optimal $\{w'_i, w'_0\}$ pair replacing original $\{w_i, w_0\}$ pair to obtain output y closest to target T.

Per term i, per x^p factor j, we can determine the coefficients w'_i and w'_0 in the replacement function

$$y' = w'_0 + w'_i \prod_{\substack{u=1\\u\neq j}}^{n_{x\,i}} x_{mu}^{p_{ju}} + \sum_{\substack{h=1\\h\neq i}}^{n_h} w_h \prod_{u=1}^{n_{x\,h}} x_{mu}^{p_{hu}} \tag{5.1}$$

for the original PUN as defined in equation 3.1, which minimizes the resulting error $E(T - y')$. If E is the *mean square error* function, this problem can be solved by using standard statistical regression to analytically obtain the optimal solution possible without retraining.

We define α_m as the remaining activation value of the hidden node containing the pruned connection for sample m, and α_m as the value which the sum of the hidden node and bias should have to result in the target. This results in the equations

$$\begin{aligned}\alpha_m &= \prod_{\substack{u=1\\u\neq j}}^{n_{x\,i}} x_{mu}^{p_{ju}}\\ \beta_m &= T_m - \sum_{\substack{h=1\\h\neq i}}^{n_h} w_h \prod_{u=1}^{n_{x\,h}} x_{mu}^{p_{hu}}\end{aligned}$$

which constitute vectors $\vec{\alpha}$ and $\vec{\beta}$ containing α_m and β_m for all samples m. We can use these in the linear regression equation

$$w'_i\vec{\alpha} + w'_0 = \vec{\beta}$$

In the special case $n_{x\,i} = 1$ (the last feature connection to hidden node i is being removed) this simplifies to

$$w'_0 = \vec{\beta}$$

As has been mentioned, minimization of such a formula can be done with standard linear regression techniques (Draper and Smith, 1998). The optimal regression coefficient w'_i can be

```
1. Initialize number of left connections $n = n_h \times n_x$ and network $\Omega_n = \Omega$.
2. Create a list $A_n$ of the weighted connections $p_{ij} \in \Omega_n$ between hidden nodes $i$ and feature nodes $j$.
3. For each weight $p_{ij}$ in list $A_n$, calculate error $E(\Omega_{\not\ni p_{ij}})$, which is the error of network $\Omega_n$ without connection $p_{ij}$, in which $w'_i$ and $w'_0$ are updated using equations (5.2) and (5.3).
4. Select the $p_{ij}$ connection with the smallest error.
5. If $\tilde{E}(\Omega_{\not\ni p_{ij}}) \leq \tilde{E}(\Omega_t)^\dagger$ continue to step 6. Otherwise, first
   (a) Retrain the network.
   (b) Repeat steps 3 and 4.
   (c) If $E(\Omega_{\not\ni p_{ij}}) < E(\Omega_n)$ we have escaped a local minimum. In this case,
       i. While $n < n_h \times n_x$ and $E(\Omega_{\not\ni p_{ij}}) \leq E(\Omega_{n+1})$, $n = n + 1$.
       ii. Set the weights of $\Omega_n$ to those of $\Omega_{\not\ni p_{ij}}$, replacing the value of connections pruned between these two steps with 0.
       iii. Train $\Omega_n$ and abort this loop to go to step 2.
6. Prune selected connection weight $p_{ij}$. Set $A_{n-1} = A_n \not\ni p_{ij}$ and $n = n - 1$. If $n > 1$ return to step 3.
```

† $\tilde{E}$ being any error function for the model as a whole, as discussed in section 2.4.2.

Fig. 5.5: PUN pruning algorithm

calculated using

$$w'_i = \begin{cases} \frac{\operatorname{cov}(\vec{\alpha},\vec{\beta})}{\operatorname{var}(\vec{\alpha})} = \frac{(\vec{\alpha}^T\vec{\beta})/n_m - \overline{\alpha\beta}}{(\vec{\alpha}^T\vec{\alpha})/n_m - \overline{\alpha}^2} = \frac{(\vec{\alpha}^T\vec{\beta}) - n_m\overline{\alpha\beta}}{(\vec{\alpha}^T\vec{\alpha}) - n_m\overline{\alpha}^2} & \text{for } n_{xi} > 1 \\ \text{inapplicable} & \text{for } n_{xi} = 1 \end{cases} \tag{5.2}$$

and optimal intercept (bias) w'_0 with

$$w'_0 = \begin{cases} \overline{\beta} - w'_i\overline{\alpha} & \text{for } n_{xi} > 1 \\ \overline{\beta} & \text{for } n_{xi} = 1 \end{cases} \tag{5.3}$$

Thus, we can use (5.2) and (5.3) in (5.1) to establish a list of least errors without retraining. With training these errors could likely still become lower, but they still give a very good measure of the current importance of the individual features for the network.

When the error has hardly increased, the network already did not appear to use the pruned factor. Retraining is in such cases less likely to yield a significant performance gain compared to large error increases. As training costs time, we would like to avoid unnecessary trainings. To decide whether retraining is necessary, we propose to directly accept the new pruned network if the increase in error is less than inversely proportional to its new size; thus, if a network is pruned from 5 to 4 nodes, to skip retraining if the error increase is smaller than $\frac{5}{4}$. Figure 5.5 shows the pruning algorithm. Briefly put, for every factor the MSE is calculated of networks without it with optimized coefficients, the network with the smallest resulting MSE is selected and this process is repeated for all remaining factors.

5.6.3 Example use

To demonstrate the training of PUNs on synthetic data sets, we have chosen to simulate a hydrological formula. To show that the PUN can indeed discover actual natural laws when available in

	MSE			Iterations		Time (s)	
Nodes	Min.	Avg.	SD	Avg.	SD	Avg.	SD
1	46.35	98.68	45.69	47.46	20.97	0.83	0.38
2	3.87×10^{-2}	70.54	123.75	75.01	43.68	1.57	0.96
3	4.80×10^{-2}	80.54	241.48	132.11	74.65	3.23	1.88

Tab. 5.1: Learning statistics for the modified Hochstein formula

the data, an artificial data set was created to model the 'modified Hochstein' formula (Schulze, 1999), defined as

$$y = r_1 U_s \left(\left(\frac{A_c}{A_w} \right)^{1\frac{1}{4}} - 1 \right) b' + s_1$$

in which r_1 and s_1 are constants, and the other factors variables. The formula can thus be written as a PUN power function with six x^p factors,

$$s_1 + r_1 U_s^1 A_c^{1\frac{1}{4}} A_w^{-1\frac{1}{4}} b'^1 - r_1 U_s^1 b'^1$$

in which the bias would learn s_1, and the coefficient weights r_1.

For training, we used tiny training rounds of 10 epochs, resetting the weights when a round yielded less than 1% MSE decrease. The tests were performed on a 700Mhz Athlon machine, which is a quite modest computer nowadays but still returned the correct answer within one second.

The setup of the test was comparable to the synthetic test of Saito and Nakano, except that we have raised the bar for the network in a couple of ways. We used a data set with 15 (instead of 9 as used by Saito and Nakano) features, 4 of which are actually used in the formula, while the remaining features contained random noise. The other 11 For the feature value distribution, we opted for a normal distribution instead of the uniform distribution apparently used in Saito and Nakano's work, as we believe parameter distributions in real datasets are unlikely to be uniform and more likely to have a normal distribution.

Also, we added noise to the targets with standard deviation 0.2 instead of 0.1. The sample size remained 200. To select the number of hidden nodes, we have chosen to extract 20% of these for a test set, instead of training on all and using the MDL criterium.

For data generation, we used constants $r_1 = 3$ and $s_1 = 2$. The results from 100 trials for 1 to 3 hidden units are summarized in table 5.1. The result for the function with smallest generalization error (two hidden nodes) has 30 x^p factors, so the function should definitely be cleared up.

The abbreviation of the function can be simplified by rounding off after the second digit and removing weights with resulting value zero (Oost et al., 2002), resulting in

$$y = 2.06 + 3.06 U_s^{1.00} A_c^{1.25} A_w^{-1.25} b'^{1.00} - 3.07 U_s^{0.99} A_c^{0.01} A_w^{-0.01} b'^{1.00}$$

which is very close to the true formula. As we explained previously, this is not a mathematically sound way to reduce its complexity, and indeed causes two (albeit small) invalid terms here.

Using the pruning algorithm outlined in chapter 5.6, we obtained the pruning history shown in figure 5.6. The red line indicates the training error, the green line the test error. As you can see, the training and generalization error approach eachother until all but six nodes are pruned away (which are precisely the six used in the network), after which both show a sharp increase in error.

Thus, we obtain the formula

$$2.06 + 3.06 U_s^{1.00} A_c^{1.25} A_w^{-1.25} b'^{1.00} - 3.07 U_s^{0.99} b'^{1.00}$$

which is indeed practically identical to the original formula. For the test set of both the 'sloppy' and precise pruning method $R^2 \approx 1.00$, indicating an approximately perfect fit. Further training using this minimal network structure yields the 'perfect' formula

$$2.00 + 3.00 U_s^{1.00} A_c^{1.25} A_w^{-1.25} b'^{1.00} - 3.00 U_s^{1.00} b'^{1.00}$$

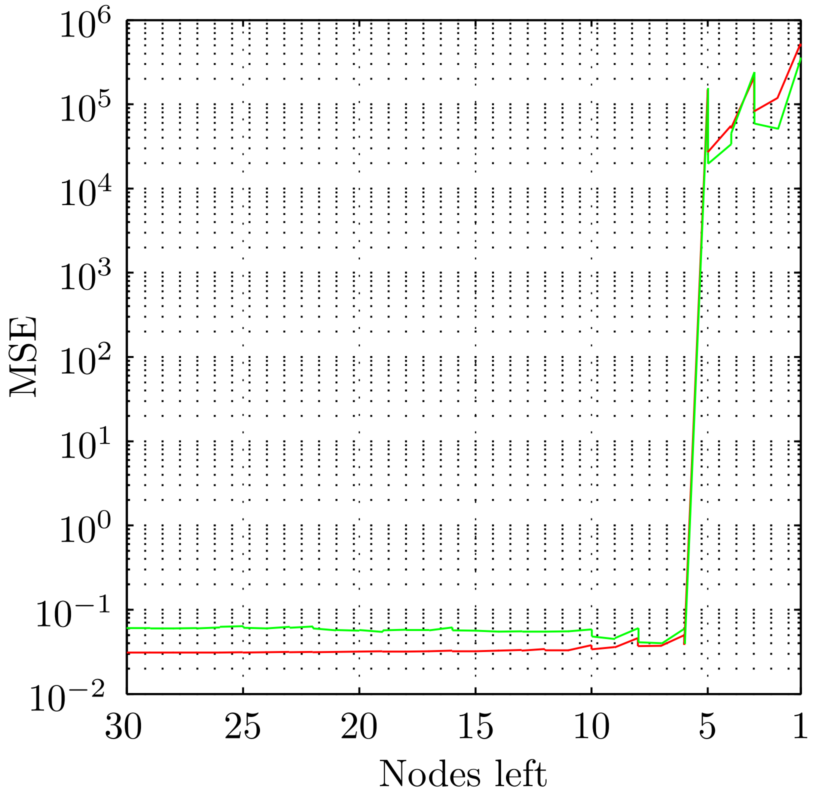


Fig. 5.6: Pruning history for Hochstein dataset

5.7 Conclusion

In this chapter we have discussed the necessity of limiting the network size to yield understandable formulas. We have presented the arguments for both the *construction* and *pruning* methods made for this, and presented existing algorithms for both. Unfortunately, most of these algorithms were specifically created for MLPs. Experiments were done, in which the tested constructive algorithms yielded less than satisfactory results. Pruning algorithms were also discussed, and a new algorithm specifically designed for pruning single weights from PUNs called *Enhanced Sensitivity-based Pruning (ESP)* introduced. ESP was tested on an artificial data set with many irrelevant features and noise, with very good results.

6. PRACTICAL USE AND EXPERIMENTS

6.1 Introduction

Now that we have completed discussing the techniques underlying our equation discovery system, we will present the complete framework of selected techniques from chapters 3 to 5 being part of it. We will then use our method on real data sets. The formulas discovered by training the PUN and progressively pruning the trained network will finally be analyzed and presented to domain experts,to study the capability of the algorithm to reveal unknown relationships hidden in the data.

6.2 Training procedure

1. *Conversion of booleans*
   Any boolean input values should be mapped to any two different, positive real values, as described in §4.3.

2. *Scaling*
   The inputs should be scaled to prevent high-valued inputs from gaining unfair advantage compared to others. Full normalization is not possible since the extracted formulas would then become a lot less comprehensible. We consider scaling to mean 1 the soundest way of scaling. This is described in §4.2.

3. *Training initialization*
   Define a maximum number of hidden units used v, and total number of times each network size should be initialized and trained j. Create vector A of size v containing the validation errors for the best currently found networks, in which each element is initialized as ∞. Also create vector B of equal size, containing these best found networks, initialized using $\emptyset$. Set current number of nodes $n = 1$.

4. *Network initialization*
   Initialize the current network Ω_n, which has n hidden nodes. The weights should be initialized to values between the minimal and maximal expected power. If no knowledge is available, we suggest using small values, for example uniform random samples in domain [-1. . . 1].

5. *Optimization*
   Train the network using Levenberg-Marquardt. Here, the following methodology can be used:

   (a) *Training*
   Train using a predefined number of epochs, large enough to consider further improvement unlikely if no real progress is being made during the training (we have used 500 epochs per training).

   (b) *Breadth-first (optional)*
   Select the best out of a number of brief trainings (say ten trainings each with one tenth of the length of a normal training), using the one with the best validation error for the next step (See section 4.5).

(c) *Repeat while progressing*
Return to step 5a to continue training if the last round yielded a training error decrease larger than a certain percentage (we have used 1%).

6. *Performance comparison*
Calculate the validation error E_v using a validation set or other criterium discussed in section 2.4.3.

```
If $E_v < A_n$: Set $A_n = E_v$ and $B_n = \Omega_n$.
    If $i < v$: Create new network $\Omega_{n+1}$ with identical weights as $\Omega_n$, and one extra hidden
        node with hidden weight 0 and input weights initialized as in step 4. Go to step 7.
    else: Continue with step 8.
Else: Increase $i$. If $i < v$, take step 4 and continue to step 7. Else, go to step 8.
```

7. *Perform another training (optional)*
Decrease j. If $j > 0$, return to step 5.

8. *Pruning (optional)*
Select network from B with the minimum A value, and begin the pruning procedure described in chapter 5.6.

Using this method, we combine the advantage of reusing knowledge about potential successful weight configurations gained by training smaller networks with the wider search. To summarize, data preprocessing is done in steps 1 and 2; the PUN is initialized in steps 3 and 4, trained in step 5 and compared with previous trained PUNs in step 6, after which the process may be repeated with step 7. Finally, the best found PUN is optionally pruned in step 8.

6.2.1 Preliminary experiment: Groynes dataset

As a first case we have used the *groynes* data set, which according to the domain knowledge should have embedded the modified Hochstein formula discussed in section 5.6.3. Unfortunately this data set is of questionable quality containing few samples and many unreliable measurements. Despite numerous attempts it appears impossible to create a MLP with low MSE and high R^2 for it.

The PUN network trained on this data set with each feature scaled to its mean very consistently found the same solutions, as illustrated by the MSE distribution in figure 4.1, indicating few (if any) local minima. The quality of the resulting predictor was equivalent to that of the MLP networks. The extracted formulas did not represent the presumed modified Hochstein formula. Since the data set the formulas can only be considered representative of the system when the error is low, no conclusion can be drawn from this result.

6.3 Experiment: Chloride dataset

6.3.1 Background

Recently, the decision has been made to change the regulation of the Haringvliet sluices; starting January first 2005, a small opening will be created, which will remain for a period of five years. This will not be a permanent situation.

Before this change can occur, it is important that the current situation is well-understood and that a clear monitoring plan has been defined. This plan will track the results of this policy change for the Northern Delta Basin, which are important in order to be able to reliable compensate affected parties.

A specific modelling task requested by the Rijkswaterstaat RIZA institute coordinating this study, was the salt distributions (amounts of chloride) near Alblasserdam (Witteveen+Bos, 2001). This can be used to determine the water flow, as saltwater from the North Sea has higher chloride levels than freshwater from the Rhine[1]. Because the complex interactions in the system proved very

[1] Then again, Rhine freshwater may be considered a contradictio in terminis.

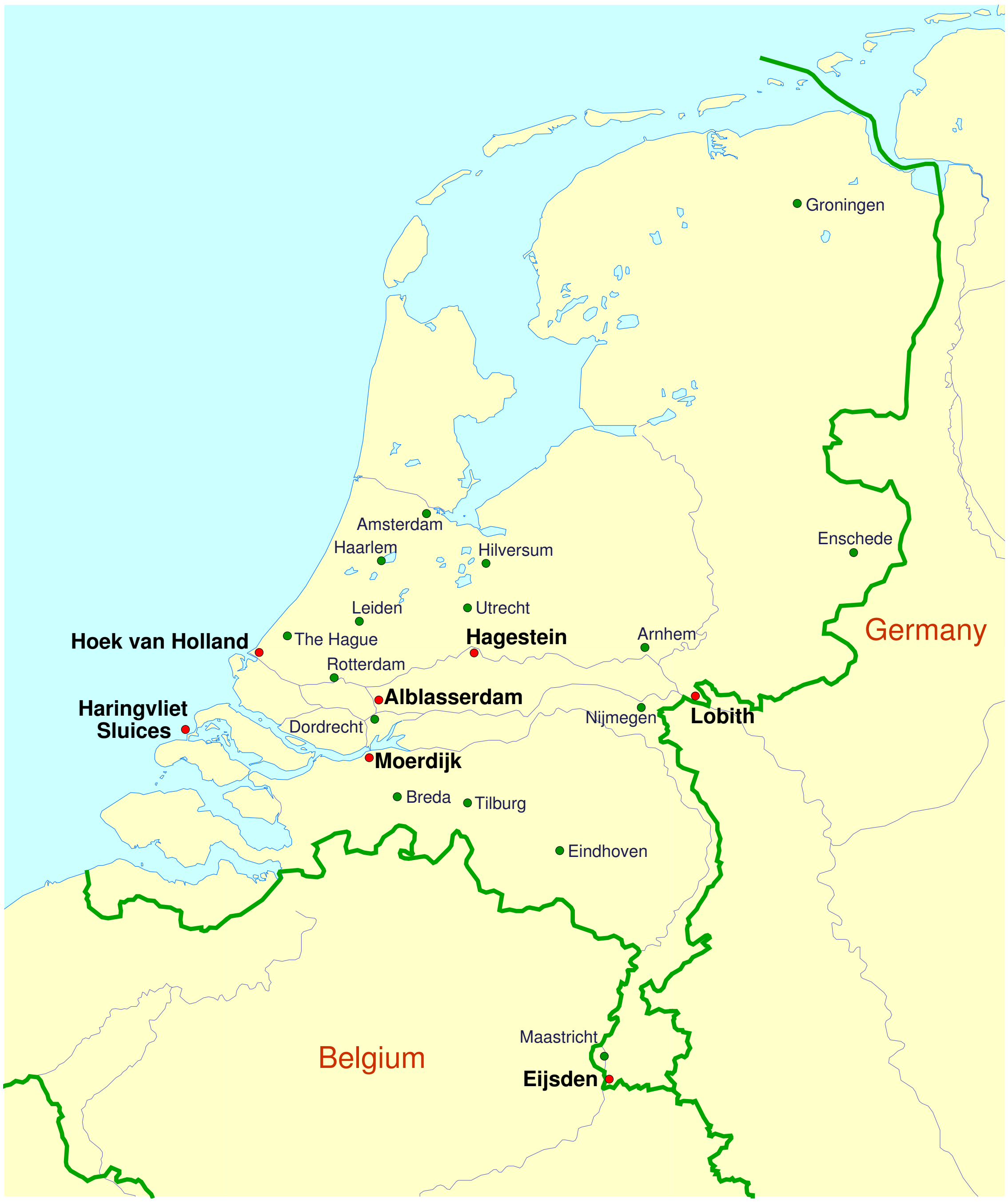


Fig. 6.1: Chloride data sampling locations

difficult to model using regular statistical methods, MLP neural networks were used to analyze the system, with good results. As explained previously, it is very hard to determine causal relationships between the inputs and outputs of neural networks, while this would certainly be of value for insight in the natural system.

Variable	Description	Unit
$Cl_{H,t-x}$	Chloride levels at Hoek van Holland, $x = \{3, 4\}$ hours ago	mg/l
$H_{H,t-x}$	Water level at Hoek van Holland, $x = \{2, 3, 4\}$ hours ago	$m\ NAP$
$H_{M,t}$	Current water level at Moerdijk	$m\ NAP$
HL	$\max(H_H) - \min(H_M)$ for current tide	m
$Cl_{L,t-x}$	Chloride level at Lobith, $x = \{47, 48, 49\}$ hours ago	mg/l
$Q_{W,t-x}$	Run-off at Hagestein, $x = \{12, 34\}$ hours ago	m^3/s
$Cl_{E,t-48}$	Chloride level at Eijsden, 48 hours ago	mg/l
$Ws_{H,t}$	Current wind speed at Hoek van Holland	m/s
d_{HVS}	Current state of Haringvliet sluice, open (1) or closed (0)	$boolean$

Tab. 6.1: Predictor variables for the Chloride data set

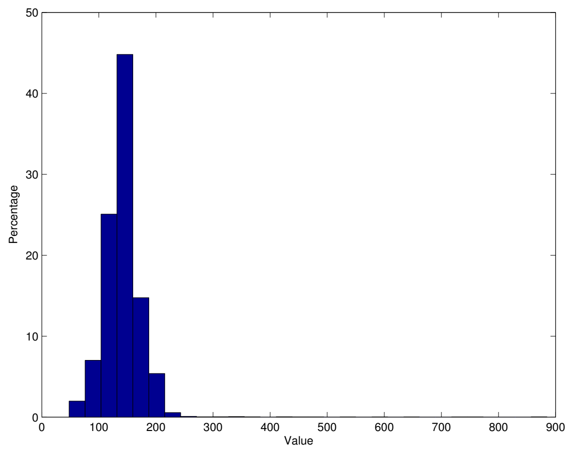


Fig. 6.2: Chloride target distribution

6.3.2 Approach

Figure 6.1 shows the sampling locations for data points throughout the Netherlands. The inputs for the model are shown in table 6.1, and a histogram of the target values in figure 6.2. The histogram clearly shows the irregular distribution of the chloride level, with extremely rare huge peaks.

Some adjustments needed to be made to make the data suitable for training PUNs. First, we converted the boolean to real values as described in section 4.3. Secondly, some data with unit $m\ NAP$ were non-positive and needed to be adapted for PUN use[2]. One possibility would be to convert such values x to values $\exp(x)$, another to use a different reference point than NAP to obtain strictly positive data. Following expert advise the latter approach was chosen. We used the approach outlined in chapter 5.6, trained networks of increasing sizes and pruning them factor by factor.

6.3.3 Results

In figure 6.3 the pruning log is shown for the best networks found with respectively 2, 3 and 4 hidden nodes. As there were fifteen inputs, the total number of factors for unpruned PUNs with 2, 3 and 4 nodes were 30, 45 respectively 60. We show the MSE as well as the adjusted R^2. In order to obtain a clearer view of the error activity, only the part of the graph where the training or test error actually changes is shown; left of these graphs the errors stay practically identical to

[2] NAP is a Dutch unit for water level relative to a reference point called Nieuw Amsterdams Peil (New Amsterdam Level).

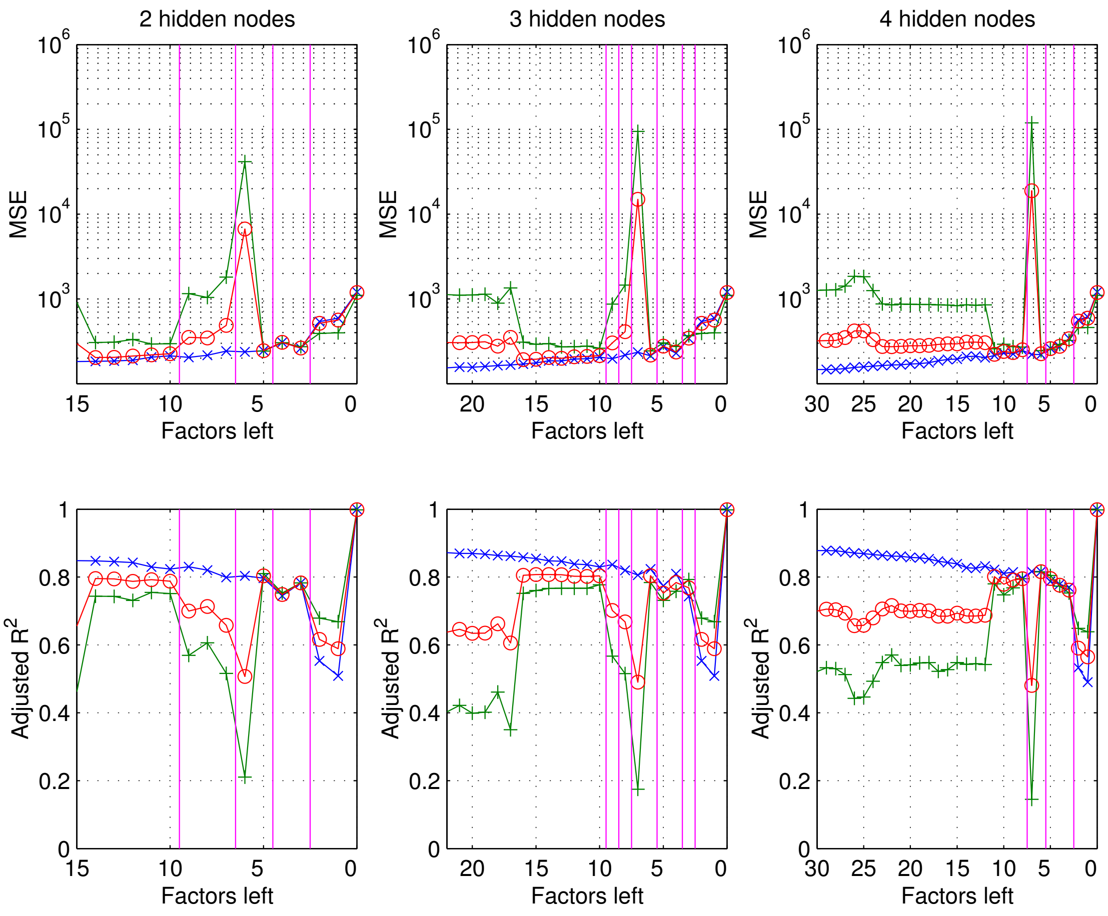


Fig. 6.3: Chloride pruning errors

the left border of each graph. The test error is represented as a $+$, the train error as a $\times$ and the error of the total data set as $\circ$.

An interesting result was that during pruning all of the different-sized networks converged to practically identical formulas, which we consider a verification of the robustness of our pruning methodology. The error levels follow a nearly identical path for the last couple of hidden nodes. Where they converge, at around 5 factors, the formulas from these networks are practically identical, and the generalization error the lowest.

Table 6.2 shows the formulas obtained in the last five pruning steps, for networks initialized with two, three and four hidden nodes. The MSE is given, as well as the adjusted R^2 to give a rough indication of the relative model qualities. In this section, I will refer to the resulting equations as $n(m)$, n being the initial number of hidden nodes and m the number of x^p steps left.

6.3.4 Interpretation

As you can see, the pruned networks all converge to a near-identical solution; the last three steps are completely identical with the slight exception of $Cl_{L,t-47}$ instead of $Cl_{L,t-49}$ being chosen for the 4-node PUN.

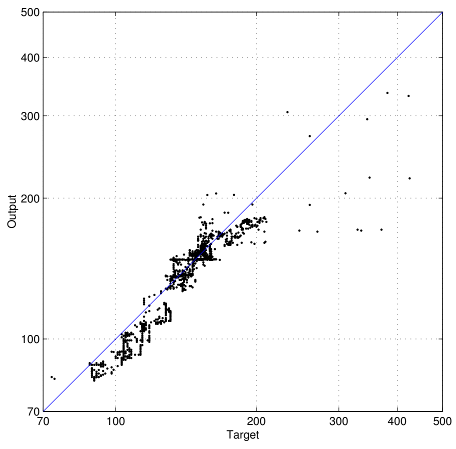
Note that although formulas 2(3) and 3(3) contain identical inputs, the resulting weights are different, resulting in a better MSE fit for equation 2(3). The network may have become trapped in a local minimum or have found a very long plateau; still, in all three cases the pruning algorithm consistently chose to prune $H_{M,t0}$.

#	MSE	aR^2	Formula (2-node PUN)
5	242.36	0.81	$44.01 + 0.50\,Cl_{L,t-49}^{1.08} + 1.44 \times 10^{81}\,Q_{W,t-12}^{-30.32}\,H_{M,t0}^{15.68}\,H_{H,t-3}^{2.81}\,H_{H,t-2}^{-2.30}$
4	305.05	0.75	$46.87 + 0.40\,Cl_{L,t-49}^{1.12} + 2.20 \times 10^{106}\,Q_{W,t-12}^{-38.63}\,H_{M,t0}^{19.86}\,H_{H,t-3}^{3.0}$
3	272.91	0.79	$46.55 + 0.40\,Cl_{L,t-49}^{1.12} + 3.15 \times 10^{107}\,Q_{W,t-12}^{-38.63}\,H_{M,t0}^{19.86}$
2	392.40	0.68	$46.41 + 0.41\,Cl_{L,t-49}^{1.12} + 3.73 \times 10^{116}\,Q_{W,t-12}^{-39.94}$
1	397.95	0.67	$47.26 + 0.41\,Cl_{L,t-49}^{1.12}$

#	MSE	aR^2	Formula (3-node PUN)
5	302.53	0.73	$-4.04 \times 10^{6} + 2.57 \times 10^{-5}\,Cl_{L,t-49}^{2.82} + 9.27 \times 10^{112}\,Q_{W,t-12}^{-41.10}\,H_{M,t0}^{21.16}\,H_{H,t-3}^{3.12} + 4.04 \times 10^{6}\,Q_{W,t-34}^{-7.01\times10^{-6}}$
4	274.30	0.76	$-4.04 \times 10^{6} + 2.70 \times 10^{-5}\,Cl_{L,t-49}^{2.81} + 1.51 \times 10^{114}\,Q_{W,t-12}^{-41.10}\,H_{M,t0}^{21.16} + 4.04 \times 10^{6}\,Q_{W,t-34}^{-7.01\times10^{-6}}$
3	368.53	0.79	$114.38 + 2.42 \times 10^{-5}\,Cl_{L,t-49}^{2.83} + 2.91 \times 10^{117}\,Q_{W,t-12}^{-42.94}\,H_{M,t0}^{27.63}$
2	392.40	0.68	$46.41 + 0.41\,Cl_{L,t-49}^{1.12} + 3.73 \times 10^{116}\,Q_{W,t-12}^{-39.94}$
1	397.95	0.67	$47.26 + 0.41\,Cl_{L,t-49}^{1.12}$

#	MSE	aR^2	Formula (4-node PUN)
5	255.20	0.80	$-1.06 \times 10^{6} + 1.06 \times 10^{6}\,Cl_{L,t-47}^{6.54\times10^{-5}} + 4.76 \times 10^{117}\,Q_{W,t-12}^{-43.28}\,H_{M,t0}^{29.66} + 1.89 \times 10^{-11}\,Cl_{L,t-49}^{5.36} + 9.16 \times 10^{-57}\,Cl_{H,t-3}^{13.58}$
4	291.17	0.77	$-1.06 \times 10^{6} + 1.06 \times 10^{6}\,Cl_{L,t-47}^{6.54\times10^{-5}} + 4.76 \times 10^{117}\,Q_{W,t-12}^{-43.28}\,H_{M,t0}^{29.66} + 1.89 \times 10^{-11}\,Cl_{L,t-49}^{5.36}$
3	323.99	0.75	$-1.06 \times 10^{6} + 1.06 \times 10^{6}\,Cl_{L,t-47}^{6.54\times10^{-5}} + 4.76 \times 10^{117}\,Q_{W,t-12}^{-43.28}\,H_{M,t0}^{29.66}$
2	456.96	0.65	$-1.06 \times 10^{6} + 1.06 \times 10^{6}\,Cl_{L,t-47}^{8.79\times10^{-5}} + 2.81 \times 10^{117}\,Q_{W,t-12}^{-40.25}$
1	459.86	0.64	$-1.06 \times 10^{6} + 1.06 \times 10^{6}\,Cl_{L,t-47}^{8.79\times10^{-5}}$

Tab. 6.2: Chloride formulas



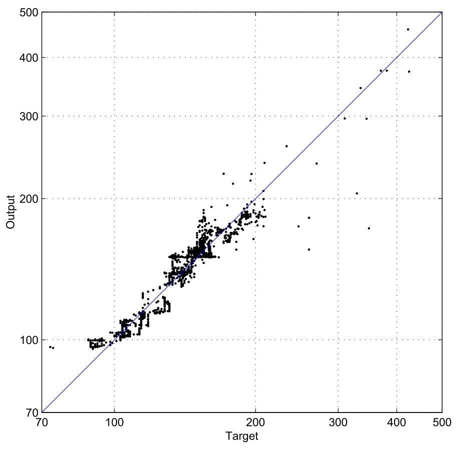


Fig. 6.4: Targets vs. output three- (left) and five- (right) parameter model

The functions 5 have a very small power for $Cl_{L,t-47}$, resulting in a near-constant output very slightly increasing from 1 with input level. The large coefficient magnifies these values while the bias translates them back to the correct output range. The final result is an activation curve for input to output which is roughly linear but slightly boosts mid-domain input levels. This yields less performance than time frame $t-49$ as used in the other formulas. The decision of the pruning algorithm to keep time frame $t-47$ instead of $t-49$ may have been caused by a local minimum in the pruning process or may have yielded superior results in more complex formulas.

Formula 2(3) and 2(5) are the most interesting formulas, since they have the lowest generalization errors. They are plotted in figure 6.4. The error of 2(5) is lower than that of 2(3), but the formula is less easy to understand.

In formula 2(3) the predicted chloride level is most strongly dependent on the upstream chloride level at the German border, as can be expected. The power of 1.12 indicates that peaks of $Cl_{L,t-49}$ have become extra high relative to lower values of this variable.

Another clearly existing but less obvious relationship is run-off at Hagestein 12 hours $Q_{W,t-12}$ ago and current water level at Moerdijk $H_{M,t0}$. A combination of very low run-off at Hagestein followed 12 hours later by relatively high water levels at Moerdijk yields extra high chloride peaks at Alblasserdam.

As has been mentioned before, water level does not have a physically logical unit; it relies on an arbitrary reference point. As such, the resulting equation in itself is not the 'true' underlying formula of the system, but it gives a clear indication of the existing relationships between the inputs, which can be used to study other formula hypotheses.

The high powers and low coefficients result in a mostly flat line, with high values only occurring at very low run-off and very high water level. This can be trivially shown by rewriting the term

$$\frac{H_{M,t0}^{20}}{Q_{W,t-12}^{40}} \text{ to } \left(\frac{H_{M,t0}}{Q_{W,t-12}^{2}}\right)^{20}$$

resulting in a more intuitive formula; a $H_{M,t0}/Q_{W,t-12}^2$ relationship extremely exponentially bent. After scaling using the extremely small coefficient, it is approximately zero for most situations and returns high values in some extreme cases.

The RIZA institute considered the HL feature likely to be of particular importance, because it is considered a measure of salt penetration. Surprisingly, this feature did not appear in the found formulas. We turned to domain experts from Witteveen+Bos for their opinion on this subject, which considered it a very interesting result. The network may have found a relation able to more precisely predict chloride levels than HL. HL is not as adaptive as it remains a static value between tide changes, hence it cannot explain the (potentially strong) chloride fluctuations within a tide. Further study of the found equation can thus yield a better insight in the relationships of the features.

In formula 2(5), the ratio of the powers of $Q_{W,t-12} : H_{M,t0}$ is again $-2 : 1$. This is in line with our above-mentioned assumption of a negated quadratic relationship between these two inputs. The term now also includes the factors $H_{H,t-3}^{2.81} H_{H,t-2}^{-2.30}$, which is interesting as H_H and H_M, now both present in the formula, are the two features determining HL.

6.4 Conclusion

The performance of our proposed equation discovery system was studied for two real data sets. One of these data sets was of low quality containing few samples and large measurement errors. Here, the error of the test set prediction for the trained PUN was practically identical to that found using an MLP, and too high to consider the equation found reliable. The other data set was of high quality, with many samples and small measurement errors. Here, the PUN test set prediction error was again on par with the MLP error, and low enough to warrant further study.

Pruning PUNs with different amounts of hidden nodes trained on this data set resulted in convergence to near-identical formulas for the last steps in the pruning process, indicating the consistency of the pruning algorithm. According to domain experts the extracted formulas indicated sensible relationships, and an interesting potential replacement function for a currently used parameter was found.

7. CONCLUSION

7.1 Summary

In this thesis, we have studied different techniques for gaining insight in large data sets with complex interactions. We have given an overview of *equation discovery* and *rule extraction*, as well as the models rule extraction is dependent on, *neural networks*, and the most common neural network configuration, multi-layer perceptrons (MLPs) . Both techniques were found to have desirable properties, but neither provides a clear optimal solution.

We focused on RF5, which can be described both as equation discovery and rule extraction system, and yields multivariate power functions. Its three elements *product unit networks* (PUNs), the *BPQ* training algorithm and *MDL* hidden layer size selection were described. The technique was found to be very promising, but some limitations would need to be addressed.

Several improvements and additions were suggested. Normalization by scaling to an attribute's mean was found to work well, while translation of attributes was proved to be unhelpful. PUNs were shown to be able to separate different functions based on mapped boolean attributes. Analytical weight initialization was studied but did not yield good results. A breadth-first search strategy was proposed which strongly decreased training time. The *Levenberg-Marquardt* training algorithm was proposed instead of BPQ, and use of a validation set instead of MDL for hidden node layer size selection.

Network size optimization techniques were studied which might strongly increase formula understandability. Tests on *constructive* algorithms did not yield good results. *Pruning* algorithms were discussed but none considered good candidates for PUN pruning. A new pruning algorithm (ESP) was introduced and tested on artificial data, with very good results.

A combination of techniques was proposed for obtaining understandable rules from PUNs. Two real data sets were analyzed using this system; one small data set with unreliable measurements, and one large data set of higher quality. Both yielded predictors of similar quality as MLPs. The first data set quality was insufficient for obtaining reliable rules, the latter did yield good predictors for differently sized networks. When pruned using our Enhanced Sensitivity-based Pruning algorithm these network consistently returned near-identical formulas. According to domain experts the extracted formulas indicated sensible relationships, and an interesting replacement function which potentially amends a currently used parameter was found.

7.2 Future work

7.2.1 Constructive algorithms

We feel more study is warranted on the subject of constructive algorithms (particularly in combination with pruning algorithms). Although we have done a number of simulations for simple constructive algorithms, we consider more complex construction algorithms likely to be more fruitful than the dissatisfactory construction results from obtained so far.

7.2.2 Alternative weight initializations

While other researchers have proposed to use high-valued initializations for networks with product units (Leerink et al., 1995), we consider small initializations to be preferable as these are likely closer to the real formula to be found than formulas with many high powers. We have done some simulations to verify these intuitions which affirmed it, but more extensive tests are recommended on this subject.

7.2.3 Extended unpruning capabilities

Currently, unpruning is done when local minima have been overcome up to the last prune step with higher error. It may be interesting to re-add previously pruned connections to the networks found in the last few pruning steps, particularly the ones which caused a relatively large error increase when pruned. We regard this particularly important when local minima have been found during the pruning process.

7.2.4 Model selection methods

The described pruning algorithm can use any model selection method. When the test set / validation set approach we recommend is not viable due to limited samples, we consider a more extensive test of different model selection methods and the threshold used to require retraining a likely direction for provide valuable insights.

7.2.5 Application of techniques on RF6

RF6 (Nakano and Saito, 1999; Saito and Nakano, 2000) is the successor of RF5 for nominal inputs. We have not used it in this thesis as this extra feature has not been necessary for our purposes while it does increase the network complexity. In our understanding most of our techniques will also be applicable to RF6 though.

7.3 Conclusion

In conclusion, we have a system for extracting insightful relations capable of handling noisy data with complex attribute interactions. The motivation behind it is opposite to the more common 'end of pipe' reasoning commencing with default techniques taken for granted and converting their results towards the solution; instead, the focus has been on the solution, and which (possibly new) techniques are best suited for providing it.

Using this inverse strategy we can open the neural network 'black box' avoiding the problems we would otherwise face, and instead obtain valuable insight into modelled systems. We have shown this technique works, and are confident it has a bright future ahead of it.

A. RULE EXTRACTION ALGORITHMS

Table A.1 lists the properties of the rule extraction algorithms as reported in the different rule extraction surveys. Readers interested in algorithms are referred to the survey(s) describing the algorithm as well as the original publication. The source survey keys are listed in table A.2, and the keys describing the properties of the algorithms in table A.3. Unfortunately only limited data is available about many algorithms; missing data are represented with a question mark.

Name	Year	Type	Restrictions	Complexity	Fidelity	Comprehensibility	Reference
Subset; KT^{abef}	1988+	D^{a}	D^{a}	$-\ldots+^{a}$ $-:2^{n_x}$^{bd}	$-\ldots+^{a}$	$-\ldots+^{ad}$	(Saito and Nakano, 1988; Fu, 1991; Towell and Shavlik, 1993; Fu, 1994)
RuleNet^{abde}	1991	D^{a}	Bs^{a}	$-^{a}$	$\pm^{a}$	$+^{a}$	(McMillan et al., 1991; McMillan et al., 1992)
Untitled^{b}	1993	D^{b}	K^{b}	?	?	?	(Tresp et al., 1993)
DFA From Recurrent ANN^{a}	1993	D^{a}	R^{a}	$+^{a}$	$\pm^{a}$	$-^{a}$	(Giles and Omlin, 1993)
VIA^{abde}	1993	P^{a}	C^{a}	$-^{a}$	$-^{a}$	$-^{a}$	(Thrun, 1993)
KBANN/M-of-N^{abdef}	1993	D^{a}	D^{a}K^{d}	$-^{a}$	$+^{a}$	$+^{a}$	(Towell and Shavlik, 1993)
BRAINNE^{ade}	1994	Px^{b}	NT^{b}	?	?	?	(Sestito and Dillon, 1994)
DEDEC^{abde}	1994	E^{a}	D^{a}	?	$+^{a}$		(Tickle et al., 1994)
REAL^{abde}	1994	E^{a}	D^{a}	$\pm^{a}$	$+^{a}$	?	(Craven and Shavlik, 1994)
RULENEG^{abde}	1994	P^{a}	B^{a}	+: $n_m + n_x \ldots$^{a}	$+^{a}$	$-^{a}$	(Pop et al., 1994)
RULEX^{abde}	1994	D^{a}	C^{a}	$++^{a}$	$+^{a}$	$+^{a}$	(Andrews and Geva, 1994)
Untitled^{a}	1995	D^{a}	B^{a}	$-^{a}$	$+^{a}$	$+^{a}$	(Sestiano and Liu, 1995)
NNES^{d}	1995	?	?	?	$--^{d}$	$-^{d}$	(Medina and Pratt, 1995)
BIO-RE	1996	P^{a}	B^{a}	$-:2^{n_x}$^{a}	$+^{a}$	$-^{a}$	(Taha and Ghosh, 1996)
Combo^{c}	1996	D^{c}	B^{c}	$-:2^{n_x}$^{c}	$+^{c}$	$+^{c}$	(Krishnan, 1996)
Full-RE^{a}	1996	D^{a}	C^{a}	$\pm^{a}$	$+^{a}$	$+^{a}$	(Taha and Ghosh, 1996)
Interval Analysis	1996	P^{c}	C^{c}	?	$+^{c}$	$-^{c}$	(Filer et al., 1996)
Partial-RE^{a}	1996	D^{a}	D^{a}	$+^{a}$	$\pm^{a}$	$-^{a}$	(Taha and Ghosh, 1996)
Relevance Information^{a}	1996	P^{a}	D^{a}	$+^{a}$	?	$-^{a}$	(Johnson et al., 1996)
RF5^{c}	1996	D^{c}	N^{c}	+: $n_h + n_x$^{c}	$+^{c}$	$+^{c}$	(Saito and Nakano, 1996)

Successive Regularization^e	1996	D^e	?	$+^z$	?	?	(Ishikawa, 1996; Ishikawa, 2000)
Trepan^{ce}	1996	P^c	C^c	$+^c$: ${n_m}^2 {n_x}^3 v^2$	$+^c$	$+^c$	(Craven, 1996)
Untitled^c	1997	D^c	BR^c	?	$+^c$	?	(Schellhammer et al., 1998)
NeuroLinear^a	1997	D^a	C^a	$-^a$	$+^a$	$+^a$	(Setiono and Liu, 1997)
RX^c	1997	D^c	1HI^c	?	$+^c$	?	(Lu et al., 1996; Setiono, 1997)
Untitled^a	1998	D^a	RI^a	$-^a$	$+^a$	$+^a$	(Neumann, 1998b)
Untitled^f	1998	E^f	?	?	?	$-^f$	(d'Avila Garcez et al., 1998; d'Avila Garcez et al., 2001)
ANN-DT^e	1999	P^e	?	?	?	?	(Schmitz et al., 1999)
M-of-N3^e	2000	D^e	B^e	?	?	?	(Setiono, 2000)
Untitled	2000	D^z	C^z	$+^z$	?	?	(Tsukimoto, 2000)
GLARE	1999	D^z	B	?	?	?	(Gupta et al., 1999)
Induce-Net	1999	D^z	D	$+^z$	?	?	(Fu, 1999)

Tab. A.1: Rule extraction methods meta-analysis

Note	Reference
a	(Neumann, 1998a)
b	(Andrews et al., 1995)
c	(Tickle et al., 1998)
d	(Boz, 1995)
e	(Duch, 2000)
f	(Darbari, 2001)
z	*Claim by author*

Tab. A.2: References

Key	Description
Type	
D	Decompositional
E	Eclectic
P	Pedagogical
Px	Pedagogical but extending input layer
Restrictions	
B	Boolean inputs
C	Continuous inputs
D	Discrete inputs
K	Knowledge-based
N	Custom network type
R	Recurrent networks
T	Structural modification/retraining
$-\ldots+$	Not good…good

Tab. A.3: Property legend

B. GLOSSARY

Attribute: See *feature.*

Backpropagation: The most well-known neural network optimization technique, in which steps of fixed length in the direction of the negated gradient are taken.

Data mining: The term data mining is somewhat overloaded. It sometimes refers to the whole process of knowledge discovery and sometimes to the specific machine learning phase. [ML:]

Definite (positive- or negative-): If the eigenvalues of the Hessian $\mathbf{H}(\mathbf{w})$ are positive definite (strictly positive, so not including zero), $\mathbf{w}$ is a *strong minimum;* negative definite means strictly negative eigenvalues, making $\mathbf{w}$ a *strong maximum.*

Delta rule: See *backpropagation.*

Epoch: See *iteration.*

Equation discovery: The area of machine learning that develops methods for automated discovery of quantitative laws, expressed in the form of equations, in collections of measured data (Langley et al., 1987).

Factor: Small part of a formula. For PUNs, this thesis reserves the term for an attribute raised to the power of an attribute weight; x^p.

Feature: Independent variable, fed into an *input* of a *neural network.* Used to predict a dependent variable (*output*), usually along with other features.

Fully connected: In this thesis strictly used for networks with all normal connections (i.e., all connections between neighboring layers). In the literature sometimes used for networks also including non-standard connections between non-neighboring layers.

Generalization error: Average error of the network for unseen cases; measures how reliable the model is when used on data outside the training set.

Generalized delta rule: See *backpropagation.*

Global (minimum): *Neural network* weight configuration having the lowest possible error for the trained data set.

Gradient $\mathbf{g}$: Vector containing the derivatives of the error function for every *weight* in the network.

Hessian (matrix): $n_w \times n_w$ square matrix containing the second-order derivatives of the network error E for *weight* set $\mathbf{w}$ in relation to its individual *weights*;

$$\mathbf{H}_{ij} = \frac{\partial^2 E(\mathbf{w})}{\partial w_i \partial w_j}$$

for continuous error functions this matrix is symmetric, since $\partial w_i \partial w_j \equiv \partial w_j \partial w_i$.

Hidden layer: Group of *hidden nodes.*

Hidden (neuron/node/unit): *Neuron* which is not an output but sends its output value to another neuron.

Indefinite: If the eigenvalues of the Hessian $\mathbf{H}(\mathbf{w})$ are indefinite, $\mathbf{w}$ is not a minimum or a maximum.

Input: Unit in a neural network receiving data from a specific *feature* from the data set.

Iteration: Single optimization step.

Jacobian (matrix): $n_y \times n_w$ matrix containing the derivatives of the network outputs in relation to the *weights* (in the context of neural networks).

KDD: Knowledge Discovery in Databases. First defined as *the non-trivial extraction of implicit, previously unknown and potentially useful information in data* (Frawley and Batheus, 1991), later redefined as *the non-trivial process of identifying valid, potentially useful and ultimately understandable patterns in data* (Fayyad et al., 1996).

Learning rate: See *step length.*

Line search: Used in numerical optimization to determine the *step length* along the *search direction* to minimize the error.

Local (minimum): Neural network error *minimum* which is not the global minimum.

Maximum: Location in the *weight* space from which it is not possible to move to a location with higher error without first decreasing it (note that we not usually would like to increase the error).

Minimum: Location in the *weight* space from which it is not possible to move to a location with lower error without first increasing it.

MLP: See *Multi-Layer Perceptron.*

Multi-Layer Perceptron: Most commonly used *neural network* type. Discussed in chapter 2.3.

Network: See *neural network.*

Neural network: Numerical optimization model with many different possible configurations. Unless specified otherwise, this thesis reserves this term for *Multi-Layer Perceptrons.*

Neuron: See *node.*

Node: Elementary element of a neural network, combining (usually) multiple attribute values to form one output value.

Off-line (training): Numerical optimization in which the *weights* are updated per batch instead of per training sample.

On-line (training): Numerical optimization in which the *weights* are updated per training sample instead of per batch.

Output: Dependent variable; value generated by the network based on its *attribute* values, which should ideally be identical to its *target* for those attribute values.

Overfitting: An overfitting network's function is more complex than the actual function of the real system; it also partly models the noise in the training data making it perform worse on unseen data.

Rule extraction: The process of generating (generally humanly comprehensible) rules approximating the behavior of a network.

Search direction: See *step direction.*

Semi-definite (positive- or negative-): If the eigenvalues of the Hessian $\mathbf{H}(\mathbf{w})$ are positive semi-definite (non-negative, including zero), $\mathbf{w}$ is a minimum; negative definite means strictly negative eigenvalues, making $\mathbf{w}$ a maximum.

Stationary point: Point in the weight space where the *gradient* is zero.

Steepest descent: Optimization technique identical to *backpropagation*, except for its use of a *line search* for its *step length* instead of a fixed *step length*.

Step $\mathbf{s}$: Change in the network weight configuration taken to decrease the network error; step length λ multiplied with step direction $\mathbf{d}$.

Step direction $\mathbf{d}$: Direction in which the optimization algorithm travels to attempt minimizing the error. For *backpropagation* this is always the negated *gradient*.

Step length λ: Distance the optimization algorithm travels over the error landscape.

Strong (minimum/maximum): Minimum or maximum not surrounded by weight space locations with the same error.

Supervised learning: Neural network learning method in which the network should learn to give a predefined output per input sample.

Target: Value a network is supposed to return given a certain set of inputs.

Term: Part of a formula limited by summation or substraction operators. For PUNs, this amounts to the output of a hidden node multiplied by its output weight (coefficient); an example for a hidden node with three inputs is $wx_1^{p_1}x_2^{p_2}x_3^{p_3}$.

Test set: A set of examples used only to assess the performance (generalization) of a fully-specified classifier (Ripley, 1996).

Training rate: See *step length*.

Training set: A set of examples used for learning, that is to fit the parameters (i.e., weights) of the classifier (Ripley, 1996).

Unit: See *neuron*.

Unsupervised learning: Neural network learning method in which the network should learn to find relationships in data without being told which output it should generate for each specific input sample.

Validation set: A set of examples used to tune the architecture of a classifier, for example to choose the number of hidden units in a neural network (Ripley, 1996).

Weak (minimum/maximum): *Minimum* or *maximum* connected to *weight* space locations with the same error.

Weight: Parameter of the neural network.

BIBLIOGRAPHY

Adams, D. (1979). *The hitchhiker's guide to the galaxy.* Harmony Books, New York.

Akaike, H. (1978). A Bayesian analysis of the minimum AIC procedure. *Annals of the institute of statistical mathematics*, 30A:9–14.

Andrews, R., Diederich, J., and Tickle, A. B. (1995). A survey and critique of techniques for extracting rules from trained artificial neural networks. *Knowledge-Based Systems*, 8:373–389.

Andrews, R. and Geva, S. (1994). Rule extraction from a constrained error back propagation MLP. In *Proceedings of the 5th Australian Conference on Neural Networks, Brisbane, Queensland, Australia*, pages 9–12.

Ash, T. (1989). Dynamic node creation in backpropagation networks. *Connection Science*, 1(4):365–375.

Battiti, R. (1989). Accelerated back-propagation learning: Two optimization methods. *Complex systems*, 3(4):331–342.

Bishop, C. M. (1995). *Neural networks for pattern recognition.* Oxford University Press.

Box, G. E. P. and Jenkins, G. M. (1970). *Time series analysis: Forecasting and control.* Holden-Day, San Francisco.

Boz, O. (1995). *Knowledge integration and rule extraction in neural networks.* PhD thesis, EECS Department, Lehigh University, US.

Brachman, R. and Anand, T. (1996). The process of knowledge discovery in databases: A human-centered approach. In Fayyad, U. M., Piatetsky-Shapiro, G., Smyth, P., and Uthurusamy, R., editors, *Advances in Knowledge Discovery and Data Mining*, pages 37–58. MIT Press.

Bradley, E., Easley, M., and Stolle, R. (2001). Reasoning about nonlinear system identification. *Artificial Intelligence*, 133:139–188.

Chester, D. L. (1990). Why two hidden layers are better than one. In *Proceedings of the 1990 International Joint Conference on Neural Networks (IJCNN-90)*, volume 1, pages 265–268. Lawrence Erlbaum.

Clarke, A. C. (1968). *2001: A Space Odyssey.* New American Library, Inc., New York.

Craven, M. and Shavlik, J. (1994). Using sampling and queries to extract rules from trained neural networks. In *Proceedings of the 11th International Conference on Machine Learning (ICML-94)*, pages 37–45. Morgan Kaufman.

Craven, M. (1996). *Extracting comprehensible models from trained neural networks.* PhD thesis, Department of Computer Sciences, University of Wisconsin-Madison, USA.

Darbari, A. (2001). Rule extraction from trained ANN: A survey. Technical Report WV-2000-03, Knowledge Representation and Reasoning Group, Department of Computer Science, Dresden University of Technology, Dresden, Germany. Draft.

d'Avila Garcez, A., Broda, K., and Gabbay, D. M. (1998). Symbolic knowledge extraction from trained neural networks: A new approach. Technical Report TR-98-014, Department of Computing, Imperial College, London, UK.

d'Avila Garcez, A., Broda, K., and Gabbay, D. M. (2001). Symbolic knowledge extraction from trained neural networks: A sound approach. *Artificial Intelligence*, 125(1–2):155–207.

Demuth, H. and Beale, M. (2000). *Neural network toolbox user's guide.* The MathWorks, Inc., Natick, Massachusetts, USA.

Domingos, P. (1999). The role of Occam's razor in knowledge discovery. *Data Mining and Knowledge Discovery*, 3(4):409–425.

Draper, N. and Smith, H. (1998). *Applied Regression Analysis, Third Edition.* John Wiley and Sons.

Duch, W. and Jankowski, N. (1999). Survey of neural transfer functions. *Neural Computing Surveys*, 2:163–213.

Duch, W. (2000). Neural knowledge extraction notes from extraction of knowledge from data using computational intelligence methods.

Durbin, R. and Rumelhart, D. E. (1989). Product units: A computationally powerful and biologically plausible extension to backpropagation networks. *Neural Computation*, 1(1):133–142.

Džeroski, S., Todorovski, L., Bratko, I., Kompare, B., and Križman, V. (1999). Equation discovery with ecological applications. In Fielding, A. H., editor, *Machine Learning Methods for Ecological Applications*, pages 185–207. Kluwer Academic Publishers, Boston, MA, USA.

Džeroski, S. and Todorovski, L. (1994). Discovering dynamics: From inductive logic programming to machine discovery. *Journal of Intelligent Information Systems*, 4:89–108.

Falkenhainer, B. C. and Michalski, R. S. (1986). Integrating quantitative and qualitative discovery: the ABACUS system. *Machine Learning*, 1(4):367–401.

Fayyad, U. M., Piatetsky-Shapiro, G., and Smyth, P. (1996). From data mining to knowledge discovery: An overview. In Fayyad, U. M., Piatetsky-Shapiro, G., Smyth, P., and Uthurusamy, R., editors, *Advances in Knowledge Discovery and Data Mining*, pages 1–34. MIT Press.

Filer, R., Sethi, I., and Austin, J. (1996). A comparison between two rule extraction methods for continuous input data. In *Proceedings of the NIPS '97 Rule Extraction From Trained Artificial Neural Networks Workshop, Queensland University of Technology, Australia*, pages 38–45.

Frawley, W. and Batheus, C. (1991). Knowledge discovery in databases: An overview. In Piatetsky-Shapiro, G. and Frawley, W., editors, *Knowledge Discovery in Databases*, pages 1–27. MIT Press.

Fu, L. (1991). Rule learning by searching on adapted nets. In *Proceedings of the 9th National Conference on Artificial Intelligence, Anaheim, California*, pages 590–595.

Fu, L. (1994). Rule generation from neural networks. *IEEE Transactions on Systems, Man and Cybernetics*, 24(8):1114–1124.

Fu, L. (1999). Knowledge discovery by inductive neural networks. *IEEE Transactions on Knowledge and Data Engineering*, 11(6):992–998.

Gantz, T. (1993). *Early greek myth: A guide to literary and artistic sources.* Johns Hopkins University Press.

Giles, C. and Omlin, C. W. (1993). Extraction, insertion and refinement of production rules in recurrent neural networks. *Connection Science*, 5(3-4):307.

Gill, P., Murray, W., and Wright, M. (1981). *Practical optimization.* Academic Press, London, UK.

Gupta, A., Park, S., and Lam, S. M. (1999). Generalized analytic rule extraction for feedforward neural networks. *IEEE Transactions on Knowledge and Data Engineering*, 11(6):985–991.

Hassibi, B. and Stork, D. G. (1993). Second order derivatives for network pruning: Optimal brain surgeon. In Hanson, S. J., Cowan, J. D., and Giles, C. L., editors, *Advances in Neural Information Processing Systems*, volume 5, pages 164–171. Morgan Kaufman.

Henderson, H. V. and Velleman, P. F. (1981). Building multiple regression models interactively. *Biometrics*, 37(2):391–411.

Hornik, K., Stinchcombe, M., and White, H. (1989). Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366.

Ishikawa, M. (1996). Rule extraction by successive regularization. In *Proceedings of the 1996 International Conference on Neural Networks (ICNN-96)*, pages 1139–1143.

Ishikawa, M. (2000). Rule extraction by successive regularization. *Neural Networks*, 13:1171–1183.

Ismail, A. and Engelbrecht, A. P. (2001). Global optimization algorithms for training product unit neural networks. In *Proceedings of the 2000 International Joint Conference on Neural Networks (IJCNN-00)*, volume 1, pages 132–137.

Janson, D. and Frenzel, J. (1993). Training product unit neural networks with genetic algorithms. *IEEE Expert*, 8(5):26–33.

Johnson, G., Nealon, L., and Lindsay, R. O. (1996). Using relevance information in the acquisition of rules from a neural network. In *Proceedings of the Rule Extraction From Trained Artificial Neural Networks Workshop, Queensland University of Technology, Australia*, pages 68–80.

Kearns, M. J., Mansour, Y., Ng, A. Y., and Ron, D. (1997). An experimental and theoretical comparison of model selection methods. *Machine Learning*, 27(1):7–50.

Kokar, M. M. (1986). Determining arguments of invariant functional descriptions. *Machine Learning*, 1(4):403–422.

Krishnan, R. (1996). A systematic method for decompositional rule extraction from neural networks. In *Proceedings of the NIPS '97 Rule Extraction From Trained Artificial Neural Networks Workshop, Queensland University of Technology, Australia*, pages 38–45.

Križman, V., Gams, M., and Džeroski, S. (1999). Discovering dynamics from data. In *Proceedings of the Information society conference, Warehouses and data mining*, pages 119–124, Ljubljana, Slovenija.

Kröse, B. and van der Smagt, P. (1996). *An introduction to neural networks.* University of Amsterdam, The Netherlands.

Kwok, T. and Yeung, D. (1995). Efficient cross-validation for feedforward neural networks. In *Proceedings of the 1995 International Conference on Neural Networks (ICNN-95)*, volume 5, pages 2789–2794, Perth, Western Australia.

Kwok, T. and Yeung, D. (1997). Constructive algorithms for structure learning in feedforward neural networks for regression problems. *IEEE Transactions on Neural Networks*, 8(3):630–645.

Langley, P., Simon, H. A., Bradshaw, G. L., and Żytkow, J. M. (1987). *Scientific discovery.* MIT Press.

Langley, P. (1977). Bacon: A production system that discovers empirical laws. In *Proceedings of the 5th International Joint Conference on Artificial Intelligence (IJCAI-77)*, page 344. Morgan Kaufman.

Larsen, J. and Goutte, C. (1999). On optimal data split for generalization estimation and model selection. In Hu, Y. H., Larsen, J., Wilson, E., and Douglas, S., editors, *Proceedings of the IEEE Workshop on Neural Networks for Signal Processing IX*, pages 225–234, Piscataway, New Jersey, USA. IEEE.

Lawrence, S., Giles, C. L., and Tsoi, A. C. (1997). Lessons in neural network training: Overfitting may be harder than expected. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*, pages 540–545. AAAI Press, Menlo Park, California, USA.

Leerink, L. R., Giles, C. L., Horne, B. G., and Jabri, M. A. (1995). Learning with product units. In Tesauro, G., Touretzky, D., and Leen, T., editors, *Advances in Neural Information Processing Systems*, volume 7, pages 537–544. The MIT Press.

Leerink, L., Giles, C., Horne, B., and Jabri, M. (1996). Product unit learning. Technical Report CS-TR-3503, University of Maryland Institute for Advanced Computer Studies.

Leshno, M., Lin, V. Y., Pinkus, A., and Schocken, S. (1993). Multilayer feedforward networks with a nonpolynomial activation function can approximate any function. *Neural Networks*, 6(6):861–867.

Le Cun, Y., Denker, J. S., and Solla, S. A. (1990). Optimal brain damage. In Touretzky, D. S., editor, *Advances in Neural Information Processing Systems*, volume 2, pages 598–605, San Mateo, CA, USA. Morgan Kaufman.

Luenberger, D. (1984). *Linear and non-linear programming.* Addison-Wesley, Reading, MA, USA.

Lu, H., Setiono, R., and Liu, H. (1996). Effective data mining using neural networks. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):957–961.

MacQueen, J. B. (1967). Some methods for classification and analysis of multivariate observations. In *Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, volume 1, pages 281–297. University of California Press.

Marquardt, D. W. (1963). An algorithm for least-squares estimation of non-linear parameters. *Journal for the Society of Industrial and Applied Mathematics*, 11(2):431–441.

McCullogh, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biophysics*, 5:115–133.

McMillan, C., Mozer, M. C., and Smolensky, P. (1991). The connectionist scientist game: Rule extraction and refinement in a neural network,. In *Proceedings of the 13th Annual Conference of the Cognitive Science Society, Hillsdale, NJ, USA.*, pages 424–430.

McMillan, C., Mozer, M. C., and Smolensky, P. (1992). Rule induction through integrated symbolic and subsymbolic processing. In Moody, J. E., Hanson, S. J., and Lippmann, R. P., editors, *Advances in Neural Information Processing Systems*, volume 4, pages 969–976. Morgan Kaufman.

Medina, C. and Pratt, L. Y. (1995). NNES: A neural network explanation system for transforming trained neural networks into decision trees using neural networks. Submitted to the 14th International Joint Conference on Artificial Intelligence (IJCAI-95).

Medler, D. A. (1998). A brief history of connectionism. *Neural Computing Surveys*, 1:61–101.

Minsky, M. and Papert, S. (1969). *Perceptrons: An introduction to computational geometry*. MIT Press, Cambridge, MA, USA.

Moody, J. and Utans, J. (1992). Principled architecture selection for neural networks: Application to corporate bond rating prediction. In Moody, J. E., Hanson, S. J., and Lippmann, R. P., editors, *Advances in Neural Information Processing Systems*, volume 4, pages 683–690. Morgan Kaufman.

Moody, J. (1994). Prediction risk and architecture selection for neural networks. In Cherkassky, V., Friedman, J. H., and Wechsler, H., editors, *From Statistics to Neural Networks: Theory and Pattern Recognition Applications*, NATO ASI Series F, pages 147–165. Springer-Verlag, New York, NY, USA.

Moré, J. J. (1977). The Levenberg-Marquardt algorithm: Implementation and theory. In Watson, G. A., editor, *Numerical Analysis*, number 630 in Lecture Notes in Mathematics, pages 105–116. Springer Verlag.

Moulet, M. (1994). Iterative model construction with regression. In Cohn, A. G., editor, *Proceedings of the 11th European Conference on Artificial Intelligence (ECAI-94)*, pages 448–452, Amsterdam, The Netherlands. John Wiley and Sons.

Mozer, M. and Smolensky, P. (1989). Using relevance to reduce network size automatically. *Connection Science*, 1:3–16.

Nakano, R. and Saito, K. (1999). Discovery of a set of nominally conditioned polynomials. In Arikawa, S. and Furukawa, K., editors, *Proceedings of the 2nd International Conference on Discovery Science (DS-99)*, volume 1721 of *Lecture Notes on Artificial Intelligence*, pages 287–298, Berlin. Springer Verlag.

Neumann, J. (1998a). Classification and evaluation of algorithms for rule extraction from artificial neural networks. PhD Summer Project, ICCS, Division of Informatics, University of Edinburgh, United Kingdom.

Neumann, J. (1998b). A comparison of two algorithms for extracting rules from artificial neural networks. PhD Term project.

Nocedal, J. (1997). Large scale unconstrained optimization. In Watson, A. and Duff, I., editors, *The state of the art in numerical analysis*, pages 311–338. Oxford university press.

Nordhausen, B. and Langley, P. (1990). A robust approach to numeric discovery. In *Proceedings of the 7th International Conference on Machine Learning (ICML-90)*, pages 411–418.

Oost, E. M., ten Hagen, S., and Schulze, F. (2002). Extracting multivariate power functions from complex data sets. In *Proceedings of the Belgian-Dutch AI Conference (BNAIC-02)*.

Pearl, J. (1988). *Probabilistic reasoning in intelligent systems: networks of plausible inference*. Morgan Kaufman.

Pop, E., Hayward, R., and Diederich, J. (1994). RULENEG: extracting rules from a trained ANN by stepwise negation. Technical report, Neurocomputing Research Centre, Queensland University of Technology, Australia.

Quinlan, J. R. (1993). *C4.5: Programs for machine learning*. Morgan Kaufman.

Ripley, B. D. (1996). *Pattern recognition and neural networks*. Cambridge University Press.

Rissanen, J. (1983). A universal prior for integers and estimation by Minimum Description Length. *Annals of Statistics*, 11(2):416–431.

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

Rumelhart, D. E., McClelland, J. L., and the PDP Research Group, editors (1986). *Parallel Distributed Processing*. Volume 1 and 2. MIT Press.

Saito, K. and Nakano, R. (1988). Medical diagnostic expert system based on PDP model. In *Proceedings of the 1988 International Conference on Neural Networks (ICNN-88)*, pages 255–262.

Saito, K. and Nakano, R. (1996). Law discovery using neural networks. In *Proceedings of the NIPS '96 Rule Extraction From Trained Artificial Neural Networks Workshop*, pages 62–69.

Saito, K. and Nakano, R. (1997a). Law discovery using neural networks. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 1078–1083, San Francisco.

Saito, K. and Nakano, R. (1997b). MDL regularizer: a new regularizer based on MDL principle. In *Proceedings of the 1997 International Conference on Neural Networks (ICNN-97)*, pages 1833–1838.

Saito, K. and Nakano, R. (1997c). Numeric law discovery using neural networks. In *Proceedings of the 4th International Conference on Neural Information Processing (ICONIP-97)*, pages 843–846.

Saito, K. and Nakano, R. (1997d). Partial BFGS update and efficient step-length calculation for three-layer neural networks. *Neural Computation*, 9(1):123–141.

Saito, K. and Nakano, R. (2000). Discovery of nominally conditioned polynomials using neural networks, vector quantizers and decision trees. In Arikawa, S. and Morishita, S., editors, *Proceedings of the 3rd International Conference on Discovery Science (DS-00)*, volume 1967 of *Lecture Notes on Artificial Intelligence*, pages 325–329. Springer Verlag.

Sarle, W. S., editor (2002). *Neural network frequently asked questions*. Periodic posting to the Usenet newsgroup comp.ai.neural-nets. URL: ftp://ftp.sas.com/pub/neural/FAQ.html.

SAS Institute (1999). *SAS/IML user's guide, version 8*. SAS Institute, Cary, North Carolina, USA.

Schaffer, C. (1993). Bivariate scientific function finding in a sampled, real-data testbed. *Machine Learning*, 12:167–183.

Schapire, R. E. (1990). The strength of weak learnability. *Machine Learning*, 5(2):197–227.

Schellhammer, I., Diederich, J., Towsey, M., and Brugman, C. (1998). Knowledge extraction and recurrent neural networks: An analysis of an Elman network trained on a natural language learning task. In Powers, D., editor, *Proceedings of the Joint Conference on New Methods in Language Processing and Computational Natural Language Learning, Sydney, Australia*, pages 73–78, New Jersey, USA. Association for Computational Linguistics.

Schmitz, G. P. J., Aldrich, C., and Gouws, F. S. (1999). ANN-DT: An algorithm for extraction of decision trees from artificial neural networks. *IEEE Transactions on Neural Networks*, 10(6):1392–1401.

Schulze, F. H. (1999). *Rijkswaterstaat RIZA: Kunstmatige neurale netwerken toegepast op sedimenttransport in kribvakken langs de Waal, gevoeligheidsanalyse, fase 1*. Witteveen+Bos, P.O. Box 233, 7400AE Deventer, NL.

Schwarz, G. (1978). Estimating the dimension of a model. *Annals of statistics*, 6:461–464.

Sestiano, R. and Liu, H. (1995). Understanding neural networks via rule extraction. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 480–487.

Sestito, S. and Dillon, S. T. (1994). *Automated knowledge acquisition.* Prentice Hall, Australia.

Setiono, R. and Liu, H. (1997). Neurolinear: From neural networks to oblique decision rules. *Neurocomputing*, 17(1):1–24.

Setiono, R. (1997). Extracting rules from neural networks by pruning and hidden-unit splitting,. *Neural Computation*, 9(1):205–225.

Setiono, R. (2000). Extracting M of N rules from trained neural networks. *IEEE Transactions on Neural Networks*, 11:512–519.

Shepherd, A. J. (1997). *Second-order methods for neural networks: Fast and reliable training methods for multi-layer perceptrons.* Perspectives in Neural Computing. Springer Verlag.

Shi, Y. and Eberhart, R. (1998). A modified particle swarm optimizer. In *Proceedings of the IEEE International Conference of Evolutionary Computation, Anchorage, Alaska, USA.*

Sietsma, J. and Dow, R. J. F. (1991). Creating artificial neural networks that generalize. *Neural Networks*, 4(1):67–69.

StatSoft (2002). *Electronic statistics textbook.* StatSoft, Inc., Tulsa, Oklahoma, USA. URL: http://www.statsoft.com/textbook/stathome.html.

Taha, I. and Ghosh, J. (1996). Symbolic interpretation of artificial neural networks. *IEEE Transactions on Knowledge and Data Engineering*, 11(3):448–463.

Thrun, S. (1993). Extracting provably correct rules from artificial neural networks. Technical Report IAI-TR-93-5, Institut fur Informatik III, Universität Bonn, Germany.

Tickle, A. B., Andrews, R., Golea, M., and Diederich, J. (1998). The truth will come to light: Directions and challenges in extracting the knowledge embedded within trained artificial neural networks. *IEEE Transactions on Neural Networks*, 9(6):1057–1068.

Tickle, A., Orlowski, M., and Diederich, J. (1994). Dedec: decision detection by rule extraction from neural networks. Technical Report 96-01-05, Neurocomputing Research Centre, Queensland University of Technology, Australia.

Todorovski, L., Džeroski, S., Srinivasan, A., Whiteley, J., and Gavaghan, D. (2000). Discovering the structure of partial differential equations from example behavior. In *Proceedings of the 17th International Conference on Machine Learning (ICML-00)*, pages 991–998.

Todorovski, L. and Džeroski, S. (1997). Declarative bias in equation discovery. In *Proceedings of the 14th International Conference on Machine Learning (ICML-97)*, pages 376–384.

Towell, G. and Shavlik, J. (1993). The extraction of refined rules from knowledge-based neural networks. *IEEE Transactions on Machine Learning*, 13:71–101.

Tresp, V., Hollatz, J., and Ahmad, S. (1993). Network structuring and training using rule-based knowledge. In Hanson, S. J., Cowan, J. D., and Giles, C. L., editors, *Advances in Neural Information Processing Systems*, volume 5, pages 871–878. Morgan Kaufman.

Tsukimoto, H. (2000). Extracting rules from trained neural networks. *IEEE Transactions on Neural Networks*, 11(2):377–389.

Walker, S. F. (1992). A brief history of connectionism and its psychological implications. In *Connectionism in Context*, pages 123–144. Springer-Verlag, Berlin.

Wallace, C. S. and Boulton, D. M. (1968). An information measure for classification. *Computer Journal*, 11:185–194.

Washio, T., Motoda, H., and Niwa, Y. (2000). Enhancing the plausibility of law equation discovery. In *Proceedings of the 17th International Conference on Machine Learning (ICML-00)*, pages 1127–1134.

Washio, T. and Motoda, H. (1997). Discovering admissible models of complex systems based on scale-types and identity constraints. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, volume 2, pages 810–817.

Webb, G. I. (1994). Generality is more significant than complexity: towards an alternative to Occam's razor. In Zhang, C., Debenham, J., and Lukose, D., editors, *Proceedings of the 7th Australian Joint Conference on Artificial Intelligence (AI-94)*, pages 60–67, Armidale, Australia. World Scientific.

Webb, G. I. (1996). Further experimental evidence against the utility of Occam's razor. *Journal of Artificial Intelligence Research*, 4:397–417.

Weigend, A. S., Rumelhart, D. E., and Huberman, B. A. (1991). Generalization by weight-elimination with application to forecasting. In Lippmann, R. P., Moody, J. E., and Touretzky, D. S., editors, *Advances in Neural Information Processing Systems*, volume 3, pages 875–882. Morgan Kaufman.

Werbos, P. J. (1974). *Beyond regression: New tools for prediction and analysis in the behavioral sciences*. PhD thesis, Harvard University, Cambridge, MA.

Widrow, B. and Hoff, M. (1960). Adaptive switching circuits. In *IRE WESCON Convention Record*, volume 4, pages 96–104. IRE, New York.

Witteveen+Bos (2001). *Opzetten van Neurale Netwerken voor het voorspellen van chloride-concentraties in het Noordelijk Deltabekken. Voorstudie ter vergroting van het inzicht in de bruikbaarheid van Neurale Netwerken voor het vaststellen van de uitgangssituatie ten aanzien van de chloride-concentraties in het NDB in het kader van implementatie ander beheer van de Haringvlietsluizen (De Kier)*. Witteveen+Bos, P.O. Box 233, 7400AE Deventer, NL.

Wolfram, S. (1999). *The Mathematica book*. Cambridge University Press, UK.

Wolpert, D. and Macready, W. (1997). No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82.

Wu, Y. and Wang, S. (1991). Discovering functional relationships from observational data. In Piatetsky-Shapiro, G. and Frawley, W., editors, *Knowledge Discovery in Databases*, pages 55–70. MIT Press.

Zembowicz, R. and Żytkow, J. M. (1992). Discovery of equations: experimental evaluation of convergence. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI-92)*, pages 70–75, Cambridge, MA, USA. MIT Press.

INDEX

ACKNOWLEDGEMENTS

My graduation research has been a very enjoyable experience, and I thank everybody who supported it.

I'd like to thank my thesis supervisor *Stephan ten Hagen* for his invaluable input, his encouragement to publish my work and being a great guy; *Floris Schulze*, the *Advanced statistics* team leader at Witteveen+Bos, for providing a great working environment and his interest in my work; *Marcus Bergman* and *Johan van der Wielen*, the team staff, for all their feedback and for being great colleagues to work with; my father and mother for their suggestions and overall support, and finally *Yvette* for her support and being who she is.