Adaptive Regularization in Neural Network Filters

Course 02455 Advanced Digital Signal Processing

June 20th, 2002

Fares El-Azm	**d970581**
Michael Vinther	**s973971**

1 Table of contents

1 TABLE OF CONTENTS **2**

2 INTRODUCTION **3**

3 BACKGROUND **3**

3.1 Feed-forward networks 4

4 NEURAL NETWORK TRAINING **6**

4.1 Gradient Descent Algorithm 6
4.2 Gauss-Newton Algorithm 8
4.3 Global minimization 9
4.4 Implementation 9
4.5 Generalization 10

5 OPTIMAL BRAIN DAMAGE, OBD **11**

5.1 Implementation 12

6 REGULARIZATION **13**

6.1 Adaptive regularization 13
6.2 Implementation 14

7 RESULTS **14**

7.1 Weight optimization 15
7.2 Optimal brain damage 17
7.3 Adaptive regularization 20

8 CONCLUSION **22**

9 LITERATURE **23**

APPENDIX 1 WEIGHT OPTIMIZATION SOURCE **24**

APPENDIX 2 OBD SOURCE **28**

APPENDIX 3 ADAPTIVE REGULARIZATION SOURCE **32**

2 Introduction

In this report we will give an introduction to training two-layer neural networks. The gradient descent and Gauss-Newton algorithms will be applied for weight optimization, and optimal brain damage (OBD) and adaptive regularization will be tested as methods for network architecture optimization. We will show that the standard gradient descent and Gauss-Newton optimization algorithms can in some cases be improved by making small random jumps when the improvement in an iteration is small.

As none of us have had courses applying neural networks before, we have implemented all the methods from scratch in Matlab and tested them on time series prediction problems.

3 Background

Bernard Widrow was aware of the advantageous properties of adaptive models and pioneered the development of early artificial neural network models in the sixties. Linear adaptive systems and artificial neural network models have proven to be very useful in many applications, such as: system identification, control, speech and image processing, pattern recognition and time-series analysis. A very important property of adaptive models is their ability to learn a signal-processing task from acquired examples of how the task should be solved.

The bulk of theoretical results and algorithms exist for linear systems, but when trying to solve real world problems, systems display very complex behavior and are consequently intrinsically nonlinear. For optimal performance of the nonlinear model, it is required that the model structure is adapted to the specific application. Furthermore, nonlinear models give rise to larger computational complexity, which also is a drawback of dealing with nonlinear models. However artificial neural networks are proving to be more capable of solving significant problems, than the more conventional algorithms, because it is an attempt to approach the functions of the human brain.

McCulloh & Pitts, whom in 1943 proposed a simple parametric nonlinear computational model of a real neuron, initiated the research in artificial neural networks. A neuron or nervous cell can be described as a little computer, which constantly receives information through the *dendrite*, and continuously calculates its state. When the collective input to the neuron exceeds a certain threshold, the neuron switches from an inactive to an active state. This activation is transmitted along the *axon* to other neurons in the network. The transition from the axon signal to another neuron occurs via the *synapses*. The human brain is real neural network, which has 10^{11} neurons and where each neuron has 10^4 connections.

Although, not even state-of-the-art artificial neural networks can provide the capacity of the human brain, the learning and adaptive properties of the networks are of great use and importance in the mentioned applications.

3.1 Feed-forward networks

A simple form of network is a network having a single layer of adaptive weights. The decision boundary in such networks is linear, or more generally hyper planar in higher dimensions. We might expect that such systems do not have optimal performance for many practical applications. Due to the fact that there are limitations in terms of the range of functions, which a single layer network can represent, we need a more complex network structure. According to Bishop, 1995 having two layers with adaptive weights, a network is capable of approximating any continuous functional mapping. Figure 1 is an illustration of the structure of a two layer feed-forward network. The two-layer feed-forward network has d inputs, M hidden neurons and c output neurons. There are no feedback loops present in the network. This ensures that the network outputs can be calculated as explicit functions of the inputs and the weights.

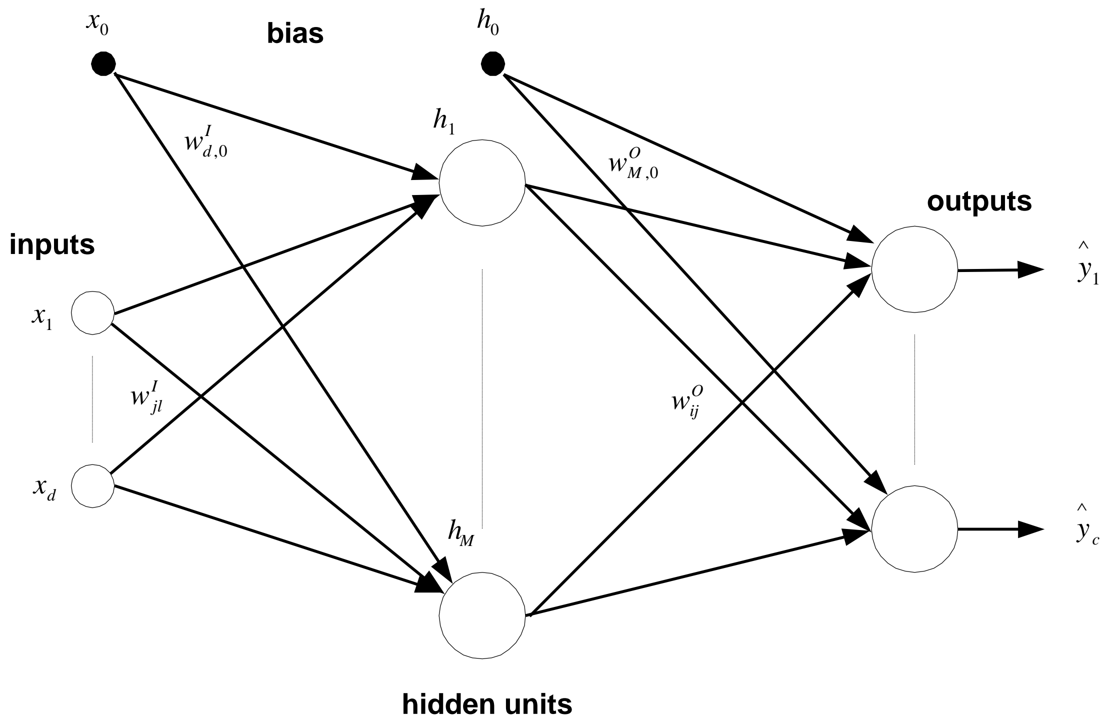


*Figure 1 A feed-forward network having two layers (*d, M, c*) of adaptive weights. The network is a graphical representation of a layered computation.*

The processing in the network in Figure 1, is given by

$$h_j(\mathbf{x}) = \boldsymbol{y}\left(\sum_{l=1}^{d} w^I_{jl} x_l + w^I_{j,0}\right) \tag{1}$$

$$\hat{y}_i(\mathbf{x}) = \boldsymbol{y}\left(\sum_{j=1}^{M} w^O_{ij} h_j(\mathbf{x}) + w^O_{i,0}\right) \tag{2}$$

Here $\mathbf{x} = [1, x_1, \ldots, x_d]$ is the input vector, and $y(u)$ is a nonlinear activation function which usually has a sigmoidal shape. In our implementation of a two-layer neural network we will be using $y(u) = \tanh(u)$. The weights from input l to hidden neuron j and the weights from the hidden neurons j to the output i are denoted respectively as, w_{jl}^{I} and w_{ij}^{O}. The thresholds or bias weights are denoted as $w_{j,0}^{I}$ and $w_{i,0}^{O}$.

The processing can also be expressed as a simple matrix formulation

$$\mathbf{h} = \mathbf{?}\,(\mathbf{W}^{I} \cdot \mathbf{x}) \qquad (3)$$

$$\hat{\mathbf{y}} = \mathbf{?}\,(\mathbf{W}^{O} \cdot \mathbf{h}) = f(\mathbf{x}, \mathbf{w}) \qquad (4)$$

Here $\mathbf{W}^{I}$ is the $(M, d+1)$ input-hidden weight matrix and $\mathbf{W}^{O}$ is the $(c, M+1)$ hidden-output weight matrix. These matrices are given by

$$\mathbf{W}^{I} = \{w_{jl}^{I}\} = \begin{bmatrix} (\mathbf{w}_1^{I})^T \\ \vdots \\ (\mathbf{w}_j^{I})^T \\ \vdots \\ (\mathbf{w}_M^{I})^T \end{bmatrix} \qquad \mathbf{W}^{O} = \{w_{ij}^{O}\} = \begin{bmatrix} (\mathbf{w}_1^{O})^T \\ \vdots \\ (\mathbf{w}_j^{O})^T \\ \vdots \\ (\mathbf{w}_c^{O})^T \end{bmatrix} \qquad (5)$$

where $(\mathbf{w}_j^{I})^T$ is the j'th row of the input-hidden weight matrix and $(\mathbf{w}_i^{O})^T$ is the i'th row of the hidden-output weight matrix. Here $\mathbf{x} = \{x_l\}$ is the (d+1, 1) input vector with $x_0 \equiv 1$, $\mathbf{h} = \{h_j\}$ is the (M+1, 1) hidden vector with $h_0 \equiv 1$ and $\hat{\mathbf{y}} = \{\hat{y}_i\}$ is the (c, 1) output vector. The element-by-element vector activation function is given by $\mathbf{?}\,(\mathbf{u}) = [y(u_1), \ldots, y(u_n)]$.

It can be seen from (4) that the network can be viewed as a nonlinear function $f(\mathbf{x}, \mathbf{w})$ of the input vector and a weight vector $\mathbf{w}$, which contains all the systems weights. The total number of weights in the network m equals $(d+1)M + (M+1)c$.

3.1.1 Signal Processing Applications

Neural networks can be used for classification, identification and time series prediction problems. In this report we will focus on the time series prediction problem. In prediction the goal is to adapt the weights of the network to predict the future values of a time series signal based on the previous samples. The values of the input neurons $\mathbf{x}(k)$ are propagated through the adapted network to predict a future value of the time series signal $\hat{y}(k)$.

4 Neural Network Training

The neural network learns by looking at previous examples of a given series to predict future samples. Therefore a data set of inputs and corresponding outputs has to be available to the training procedure, where the objective is model identification. The network weights are trained to recognize the time structure of the time series.

To determine the optimal weights for a prediction problem, one must first define the measure of the quality of the output prediction. The most common is to define a cost function as the sum of the squared differences between the networks predicted output $\hat{y}(k)$ and the expected output $y(k)$. This is commonly known as the mean-squared error (MSE) cost function.

$$S_T(w) = \frac{1}{2N_{Train}} \sum_{k=1}^{N_{Train}} \left(y(k) - \hat{y}(k) \right)^2 = \frac{1}{2N_{Train}} \sum_{k=1}^{N_{Train}} e(k)^2 \tag{6}$$

The cost function is a function of the weights of the network and a training set, where the training set is an extract of the available data in the time series signal. N_{Train} is the number of training examples.

4.1 Gradient Descent Algorithm

One way to determine the weights that minimize the cost function is to use the stochastic algorithm; gradient descent with back-propagation. This is one of the simplest network training algorithms and is also known as steepest descent. With gradient descent the initial weight vector $\mathbf{w}^{(0)}$ is often chosen at random, then with each iteration the weights are updated such that we move a distance in the direction of the greatest rate of decrease of the MSE. This change $\Delta\mathbf{w}$ is given by the negative gradient of the cost function. The gradient of the cost function is given by:

$$\nabla S_T = \frac{\partial S_T(w)}{\partial w} \tag{7}$$

The weights are updated at each iteration as follows:

$$\mathbf{w}^{n+1} = \mathbf{w}^n + \boldsymbol{h}\Delta\mathbf{w}^n = \mathbf{w}^n - \boldsymbol{h}\nabla S_T \tag{8}$$

The parameter *?* is called the learning rate or step-size and controls how big a step is taken in the negative gradient direction. With too large values of *?*, the algorithm may overshoot leading to an increase in MSE and convergence is possibly impossible. If *?* is chosen too small the search for the optimal weights can take an extremely long time, leading to long computation times. The optimum value for *?* will typically change during the course of the minimization of the MSE. A simple way of doing this is by using the line-search method. Here *?* is found by determining the cost function for decreasing values of *?* at each iteration. A decrease in MSE gives the value for *?*.

4.1.1 Back-propagation

Gradient descent training techniques require the computation of the first derivatives of the cost function with respect to the weights in each iteration. As will be shown, the back-propagation technique provides a computational efficient method of evaluating such derivatives. This is done by propagating the error signal $e(k)$ backwards through the network via the errors (*d*) that represent the errors of the individual neurons.

The gradient vector is

$$\frac{\partial S_T(\mathbf{w})}{\partial w_i} = \frac{1}{N_{Train}} \sum_{k=1}^{N_{Train}} \frac{\partial e^2(k)}{\partial \mathbf{w}} = -\frac{1}{N_{Train}} \sum_{k=1}^{N_{Train}} e(k) \frac{\partial \hat{y}_i(k)}{\partial w_i} \qquad (9)$$

By differentiating (6) with respect to the hidden to output weights one gets

$$\frac{\partial \hat{y}_i(k)}{\partial w_{ij}^{O}} = \boldsymbol{y}'\left(u_i^{O}(k)\right) h_j(k) \qquad (10)$$

This leads to

$$\frac{\partial S_T(\mathbf{w})}{\partial w_i} = -\frac{1}{N_{Train}} \sum_{k=1}^{N_{Train}} \boldsymbol{d}_i^{O}(k) h_j(k) \qquad (11)$$

With $\boldsymbol{d}_i^{O}(k) = e(k)\boldsymbol{y}'\left(u_i^{O}(k)\right)$.

The derivatives with respect to the input to hidden weights are found using the chain rule as follows:

$$\frac{\partial \hat{y}_i(k)}{\partial w_{jl}^{I}} = \frac{\partial \hat{y}_i(k)}{\partial h_j(k)} \frac{\partial h_j(k)}{\partial w_{jl}^{I}} = \boldsymbol{y}'\left(u_i^{O}(k)\right) w_{ij}^{O} \boldsymbol{y}'\left(u_j^{I}(k)\right) x_l(k) \qquad (12)$$

This yields

$$\frac{\partial S_T(\mathbf{w})}{\partial w_i} = -\frac{1}{N_{Train}} \sum_{k=1}^{N_{Train}} \boldsymbol{d}_j^{I}(k) x_l(k) \qquad (13)$$

With $\boldsymbol{d}_j^{I}(k) = \boldsymbol{d}_i^{O}(k) w_{ij}^{O} \boldsymbol{y}'\left(u_j^{I}(k)\right)$.

As mentioned, one of the most important features of back-propagation is its computational efficiency. For a neural network with *W* number of weights and biases, a single evaluation of the cost function would require somewhere in the order of *W* operations, in a sufficiently large network.

With the gradient of the cost function found, the weights can be updated as described in (8). The iterative process then repeats its self until a stop criterion is satisfied.

There are two basic stop criteria for the gradient descent algorithm. The first one is to define a maximum number of iterations. The second is to define a small constant $\boldsymbol{t}$ that either has to be larger than: $S_T(\mathbf{w})^n - S_T(\mathbf{w})^{n+1}$ or $\left\|\nabla S_T(\mathbf{w})^n\right\|_2$.

As illustrated in Figure 2, for most points in weights space, the local negative gradient vector $-\nabla S_T$ does not point towards the minimum of the error function. This means that the gradient descent can oscillate across the valley, giving a very slow convergence and a very inefficient procedure.

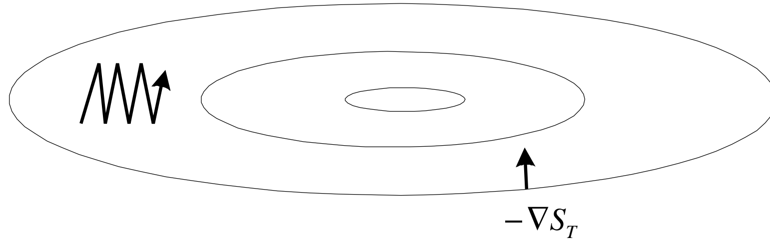


Figure 2 This illustrates the oscillating behavior of the convergence of the gradient descent algorithm

4.2 Gauss-Newton Algorithm

The oscillating behavior of the gradient descent algorithm gives rise to finding a more precise method in which a more direct step towards the cost functions minimum is needed. By a incorporating the second term in the Taylor expansion of the cost function, we get a function of the form

$$S_T(\mathbf{w}) = S_{T0} + \nabla S_T \mathbf{w} + \frac{1}{2}\mathbf{w}^T \mathbf{H}\mathbf{w}$$

The gradient of this cost function is given by

$$\frac{\partial S_T(w)}{\partial w} = \nabla S_T + \mathbf{H}\mathbf{w} \tag{14}$$

and as for the gradient descent algorithm, the cost function is minimized by $\hat{\mathbf{w}}$ and leads to

$$\nabla S_T + \mathbf{H}\hat{\mathbf{w}} = 0 \tag{15}$$

it can be seen from the previous expression that the weights are updated at each iteration as follows

$$\mathbf{w}^{n+1} = \mathbf{w}^n - \boldsymbol{h}\mathbf{H}^{-1}\nabla S_T \tag{16}$$

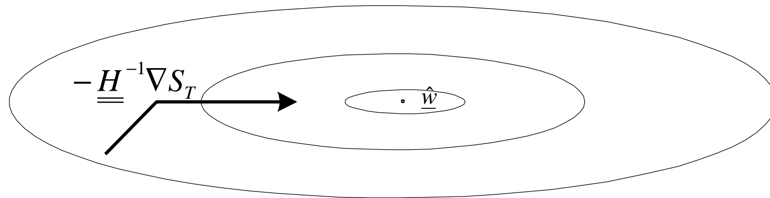


Figure 3 This illustrates the more direct convergence of the Gauss-Newton algorithm

The Hessian matrix $\mathbf{H}$ plays an important role in many aspects of neural computing. In the Gauss-Newton algorithm, the Hessian matrix takes in the considerations of the second-order properties of the cost function. As illustrated in Figure 3, the Newton step ($\mathbf{H}^{-1}\nabla S_T$) points directly towards the minimum of the cost function, which makes the Gauss-Newton algorithm a much more targeted process than the gradient descent algorithm. Furthermore, the inverse of the Hessian matrix is needed in determining the least significant weights in a network as part of network pruning algorithms (e.g. OBD), as well as finding suitable values for the adaptive regularization parameters. The second derivates of the cost function are given by

$$\mathbf{H} = \frac{\partial^2 S_T(\mathbf{w})}{\partial \mathbf{w}_{ij} \partial \mathbf{w}_{jl}} \qquad (17)$$

Evaluating the exact Hessian matrix has shown to be very computationally demanding, that is why many approximations of $\mathbf{H}$ are available. We have tried two methods, the outer product approximation or the Levenberg-Marquardt approximation and the diagonal approximation. The latter method is very computational efficient and its inverse is trivial to evaluate.
In the Levenberg-Marquardt approximation, the Hessian matrix can be written in the form

$$\mathbf{H} = \mathbf{j} \cdot \mathbf{j}^T \qquad (18)$$

where $\mathbf{j}$ (Jacobian) is defined as a vector with elements $\mathbf{j} \equiv \frac{\partial S_T}{\partial w_i}$. Updating the weights in the Gauss-Newton algorithm is as follows

$$\mathbf{w}^{n+1} = \mathbf{w}^n - \boldsymbol{h}(\mathbf{j} \cdot \mathbf{j}^T)^{-1} \mathbf{j} \qquad (19)$$

4.3 Global minimization

The MSE as a function of the weights often has many local minima. When training with the gradient descent or Gauss-Newton algorithm, the minimum found is often one close to the starting point and not necessarily the global minimum. To avoid getting stuck in a local minimum, we perform a random jump if the improvement in an iteration is very small.

In all but the first and last iterations, the improvement in MSE is computed and compared to the minimum improvement, m. If the condition

$$\frac{S_T(\mathbf{w}^{n-1}) - S_T(\mathbf{w}^n)}{S_T(\mathbf{w}^n)} < m \qquad (20)$$

holds, a random jump in the weight space is performed. The jump is made by adding a normal distributed random number with standard deviation max($\mathbf{w}$)·r to each weight. r is a small number determining the jump size, e.g. 0.01.

Before the jump is made, the MSE of the network is compared to that of the best weights found so far. If the current result is not better, meaning that the previous jump lead to a worse result, the best weight are restored and used as starting point for the new jump. If the weights were better, they are saved as the best so far before jumping. The result is that the same weight values might be used as starting point for several jumps, until an improvement is found. Without this mechanism, the jumps could easily lead to a very poor result. After the last iteration, the best weights seen are returned.

4.4 Implementation

Our Matlab implementation of the neural network simulator is divided into a series of sub-functions. Pseudo code for the global minimization function `DoTraining.m` is shown below in Figure 4. This function uses either `TrainNNGradient.m` or `TrainNNGaussNewton.m` to make a step towards the minimum. The actual Matlab code is included in the appendices.

All calculations are made using matrices to take full advantage of Matlab. The weights from the input to the hidden layer are stored in the matrix `Wi`, which has $N_{Hidden} \times (N_{Input}+1)$ elements. The +1 term is the weights from the bias neuron. The $N_{Output} \times (N_{Hidden}+1)$ connections from the hidden to the output layer are stored in `Wo`.

`DoTraining.m`

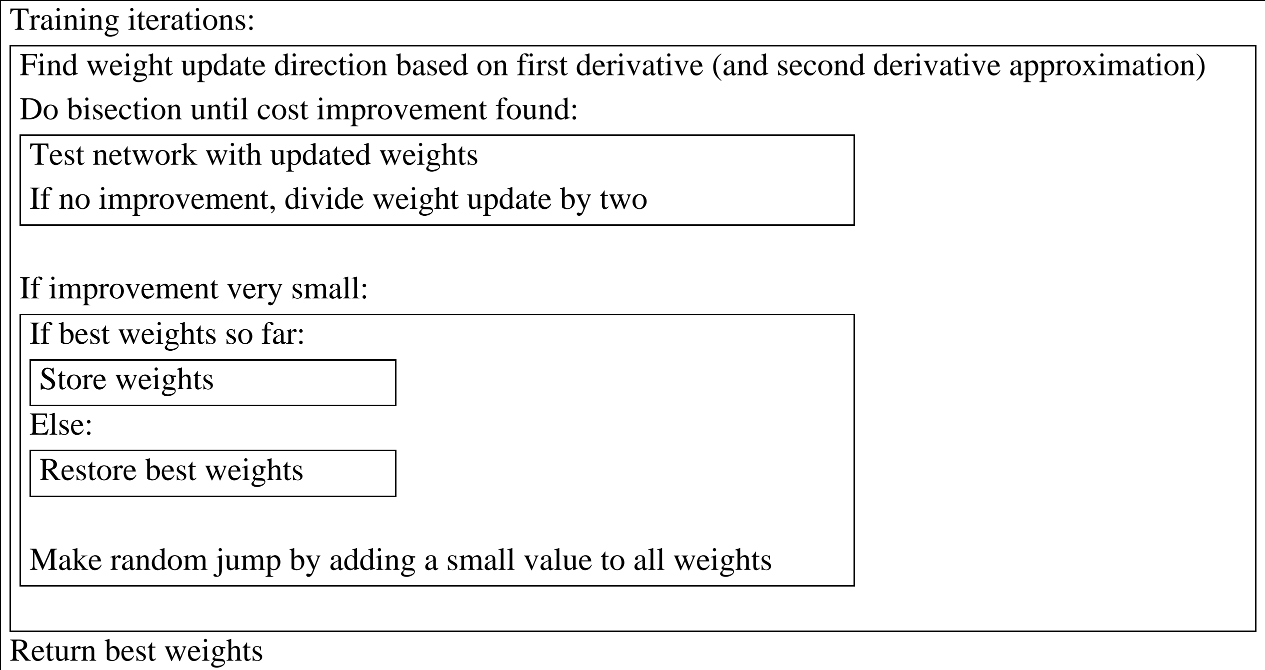


Figure 4: Pseudo code for the global minimization function.

4.5 Generalization

In order to assess the trained networks ability to generate future data (i.e. time series prediction) from a new set of data, we need to evaluate the effectiveness of the network. Looking at the so-called generalization error can do this. Consider that $\hat{\mathbf{w}}$ minimizes the cost function, then the generalization error on a test sample $(\mathbf{x}, y)$ is given by

$$G(\hat{\mathbf{w}}) = E_{\mathbf{x},y}\left[\left(y - f(\mathbf{x}, \hat{\mathbf{w}})\right)^2\right] \qquad (21)$$

Here $E_{\mathbf{x},y}[\cdot]$ denotes ensemble average, or the statistical expectation with respect to both $\mathbf{x}$ and y. The generalization error depends on the number of training data used in the training process. This could lead us to think that an increase in training data gives a decrease of the generalization error. Unfortunately this is not the case; there is a chance of over-fitting or over-training a network with too many weights. A more complex network tends to learn the noise of the training data and does not learn the overall tendency or systematic aspects of the data. This leads to a poor generalization, as would a network with too little number of weights. Figure 5 shows this tendency and it also shows that the optimal network complexity for a given model is where the generalization error is minimal.

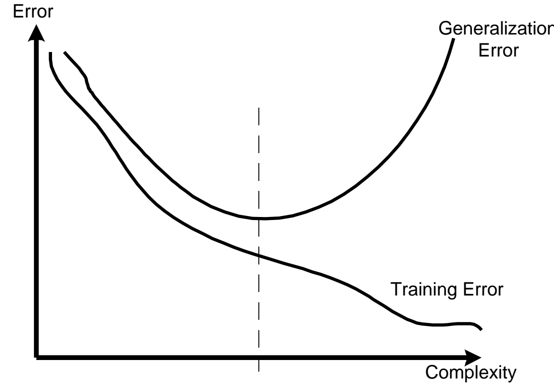


Figure 5 Illustrates the training and generalization error as function of complexity.

An estimate for the generalization error is given by

$$\hat{G}(\hat{\mathbf{w}}) = \frac{1}{N_{Test}} \sum_{k=1}^{N_{Test}} \left(y(k) - f(\mathbf{x}, \hat{\mathbf{w}}) \right)^2 \tag{22}$$

N_{Test} is the number of new test examples. It can be seen that many test examples are needed in order to estimate the generalization reliably. This leads us to incorporate more methods in order to optimize the generalization performance. This can be achieved by architectural optimization such as adaptive regularization and Optimal Brain Damage.

5 Optimal brain damage, OBD

A network structure where all neurons in one layer are connected to all neurons in the next layer might not be optimal. The danger of too big a network is that it might learn the training set too well. It might also learn any noise in the training set. This will give a very small training error, but the ability to generalize is lost.

OBD is a method to determine the effect of each weight on the cost. The change in the cost function as result of removing a weight is called the saliency. A low saliency is assumed to imply that the weight has a negative influence on the generalization skill. The network optimization is done by ranking the weights according to saliency and then removing the least significant ones.

The change in MSE cost when setting a weight to zero can be computed as

$$\boldsymbol{d}S_{\boldsymbol{t}} = \left(\boldsymbol{k} + \frac{1}{2} \frac{\partial^2 S_{\boldsymbol{t}}(\underline{\hat{w}})}{\partial w_j^{\,2}} \right) \hat{w}_j^{\,2}$$

The second derivative of $S_{\boldsymbol{t}}$ can be approximated with the following expression:

$$\frac{\partial^2 S_{\boldsymbol{t}}(\underline{w})}{\partial w_j^{\,2}} \approx \frac{1}{2N_{train}} \sum_{k=1}^{N_{train}} \left(\frac{\partial y(k)}{\partial w_i} \right)^2$$

where $\frac{\partial y(k)}{\partial w_i}$ can be computed using back propagation.

5.1 Implementation

We start with a network where all neurons in one layer are connected to all neurons in the next layer. For each iteration, the network is trained and the weight with least saliency is removed. Then an estimate of the generalization error is computed and compared to the best generalization error seen so far. If an improvement is found the network structure is saved. This continues until only two weights are left, as at least one weight from input to the hidden layer and one from then hidden layer to the output is needed to give a non-constant output. Pseudo code for the algorithm is shown in Figure 6.

Because of the matrix implementation, it is not actually possible to remove a weight; instead the corresponding element in the `Wi` or `Wo` matrices is set to zero. To keep track of which weights have been removed, two mask matrices (`WiMask` and `WoMask`) with the same dimensions as `Wi` and `Wo` containing only ones and zeros are used. They are initiated with ones, and every time a weight is removed the corresponding element is changed from one to zero.

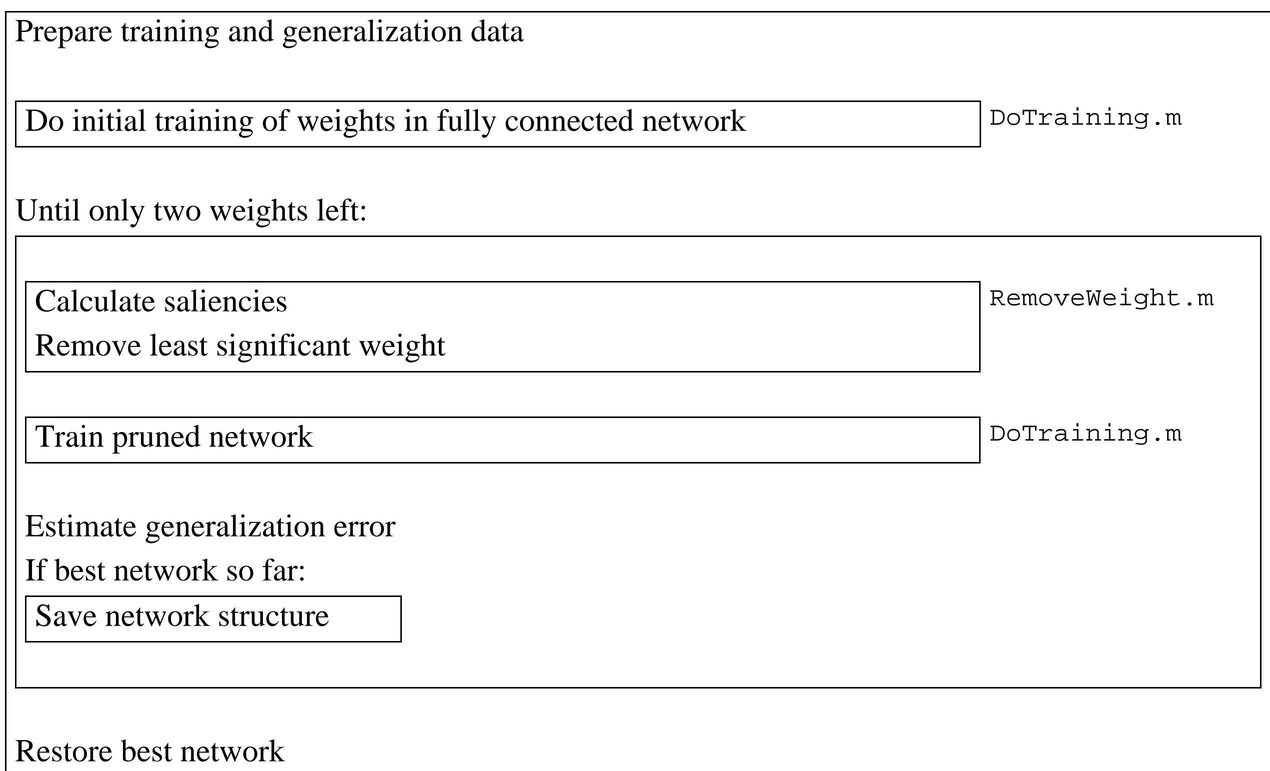


Figure 6: Pseudo code for OBD main program .

6 Regularization

Altering the training by augmenting the cost function $S_T(\mathbf{w})$ with a penalty term, which penalizes the high magnitude weights and help to prevent over-training. The augmented cost function is given by

$$C_T(\mathbf{w}) = S_T(\mathbf{w}) + \boldsymbol{k}\sum_i w_i^2 \tag{23}$$

where $\boldsymbol{k} = \dfrac{\boldsymbol{a}}{N_{Train}}$. This simple method of regularization is called weight decay.

The regularization term forces the magnitudes of the weights towards zero, as the cost or error is large for large weights. The effect is somewhat the same as in OBD, because unnecessary weights will become very small. It also causes the cost function to contain less curvature, i.e. smoothing. This decreases the number of local minima and hence improves the performance of the weight optimization algorithm. The resulting cost function is a compromise between fitting the data and minimizing the penalty term.

6.1 Adaptive regularization

In adaptive regularization a validation dataset is required to optimize the regularization parameters. This dataset is taken from the same dataset as the training and generalization set and is usually of a smaller size than the two latter sets. The estimation of the regularization parameters can be done more systematic than guessing the *?* as in the weight decay method. With the validation set, the search and evaluation of the optimal *?* is found by using the gradient descent approach on the cross-validation estimate

$$\hat{\Gamma} = \frac{1}{K}\sum_{j=1}^{K} S_{v_j}(\hat{\mathbf{w}}_j) \tag{24}$$

In (24), $S_{v_j}(\hat{\mathbf{w}}_j)$ is the cost function of the validation set, where $\hat{\mathbf{w}}_j$ are the weights that minimize the augmented cost function (23). *K* is the number of validation set there are available. In our method *K* is unity for simplicity. By updating regularization parameters as follows

$$\mathbf{?}_{n+1} = \mathbf{?}_n - \boldsymbol{h}\frac{\partial\hat{\Gamma}}{\partial\boldsymbol{k}}\left(\hat{w}(\mathbf{?}_n)\right) \tag{25}$$

the optimal regularization parameters can be found. The vector $\mathbf{?} = \left[\boldsymbol{k}^I, \boldsymbol{k}^O\right]$, gives the input-to-hidden and hidden-to-output regularization parameters.

The gradient of the cross-validation error equals

$$\frac{\partial\hat{\Gamma}}{\partial\mathbf{?}}(\mathbf{?}) = \frac{1}{K}\sum_{j=1}^{K}\frac{\partial S_{v_j}}{\partial\mathbf{?}}(\hat{\mathbf{w}}_j) \tag{26}$$

where $\dfrac{\partial S_{v_j}}{\partial\mathbf{?}}(\hat{\mathbf{w}}_j) = -2(\hat{\mathbf{w}}_j)\cdot\mathbf{g}_j$. $\mathbf{g}_j$ is a vector that is given by $\mathbf{g}_j = \mathbf{H}_j^{-1}(\hat{\mathbf{w}}_j)\cdot\dfrac{\partial S_{v_j}(\hat{\mathbf{w}}_j)}{\partial\mathbf{w}}$

6.2 Implementation

The first step is to train the network with some initial value of $\mathbf{k}$. Then a series of training iterations are made which each makes a gradient descent step and bisection. The optimization process is very time-consuming, because every change to $\mathbf{k}$ requires retraining of the weights. The sub function, which performs the optimization, is shown in Figure 7.

FindRegulizers.m

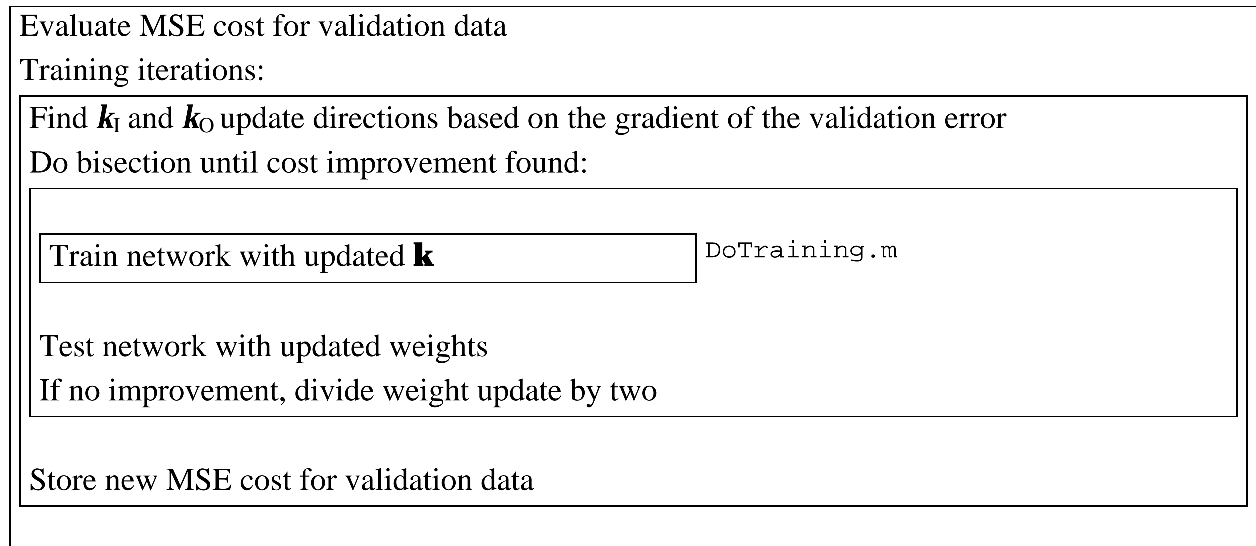


Figure 7: Pseudo code for adaptive regularization algorithm.

7 Results

Most figures presented here are *the* result from running a test program but rather *a* result. Because of both the random initialization of the weights and the small random jumps made in the training, the results varied a great deal.

The sunspot activity from 1700 to 1979 sampled every year was our primary test data set from time series prediction (see Figure 8). The period is approximately 11 years. This simple time series only required a small network, which was important to avoid too much waiting time during the simulations.

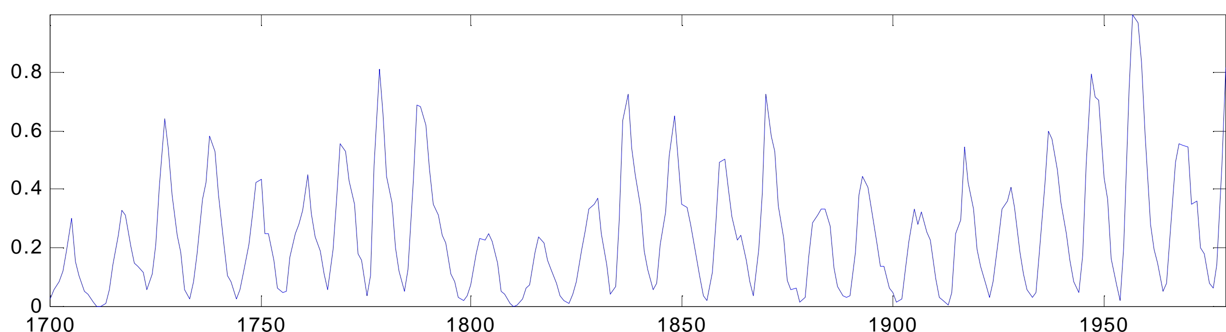


Figure 8: The sunspot activity from 1700 to 1979 was our primary training set.

7.1 Weight optimization

An example of the performance of gradient descent and the Gauss-Newton algorithm is shown in Figure 9. In this example Gauss-Newton converged much faster, but this was not always the case. It was generally faster, but often the difference was small. Gauss-Newton iterations were, however, always more time consuming. For larger networks, the time difference between Gauss-Newton and the gradient descent algorithm grows. This is resulted by the evaluation and inversion of the Hessian matrix in the Gauss-Newton algorithm.

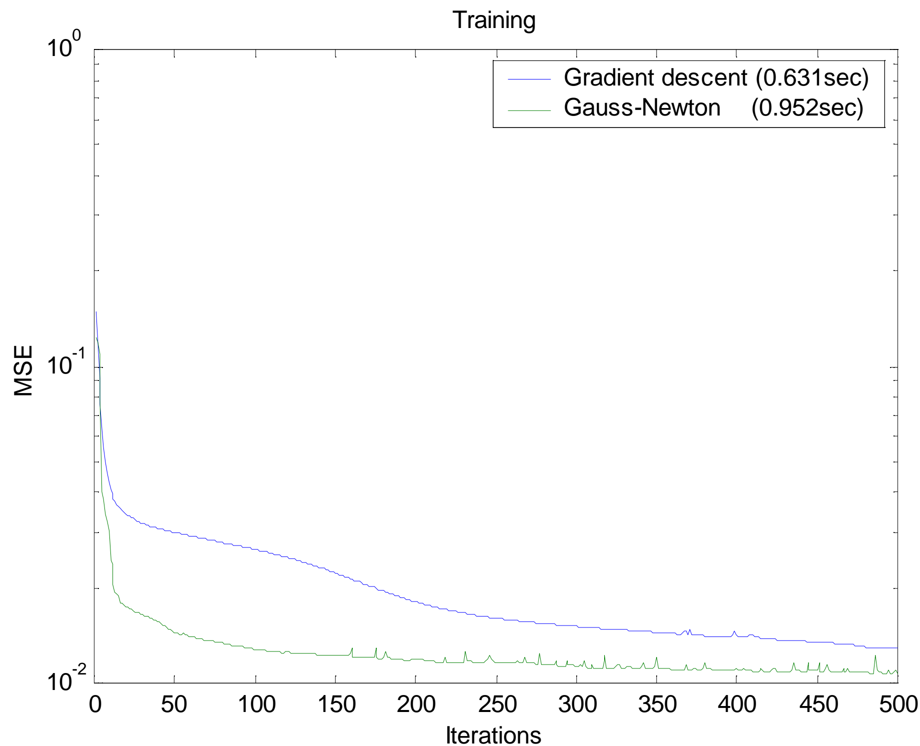


Figure 9: Comparison of gradient descent and Gauss-Newton weight optimization. The training was done on the sunspot data with 11 input and 4 hidden neurons. The "noise" in the MSE is caused by the random jumps.

To test the effect of the random jumps in the optimization, we plotted the MSE after a fixed number of iterations with varying jump size (r) and minimum improvement (m), see equation (20) . The plots were produced by first training the network to some point were the improvement is small and saving these weights. For each change in r or m, the weights are restored and the 100 iterations more were made. In Figure 10 both r and m were changed, and the Gauss-Newton algorithm was used. Figure 11 and Figure 12 show the effect of changing r with gradient descent and Gauss-Newton, respectively.

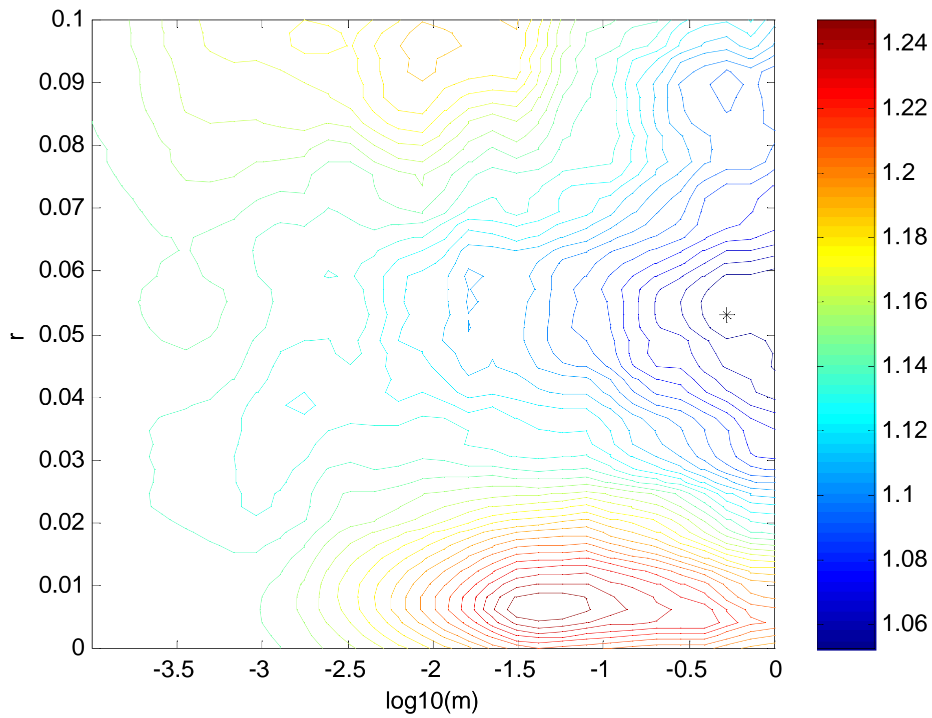


Figure 10: MSE for different jump sizes and minimum improvements using the Gauss-Newton algorithm. 100 iterations were made on a network containing 21 weights.

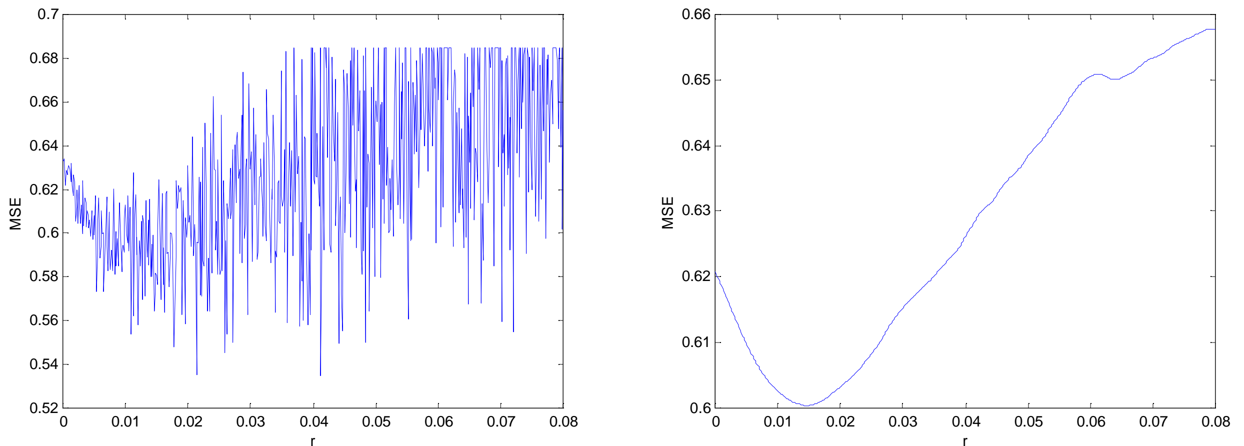


Figure 11: MSE after 100 gradient descent training iterations with different random jump sizes. A random jump was made in all iterations. The plot to the right is low-pass filtered to emphasize the shape of the curve. Before the tests, the network was trained to MSE 0.685, which explains the clipping of the peaks.

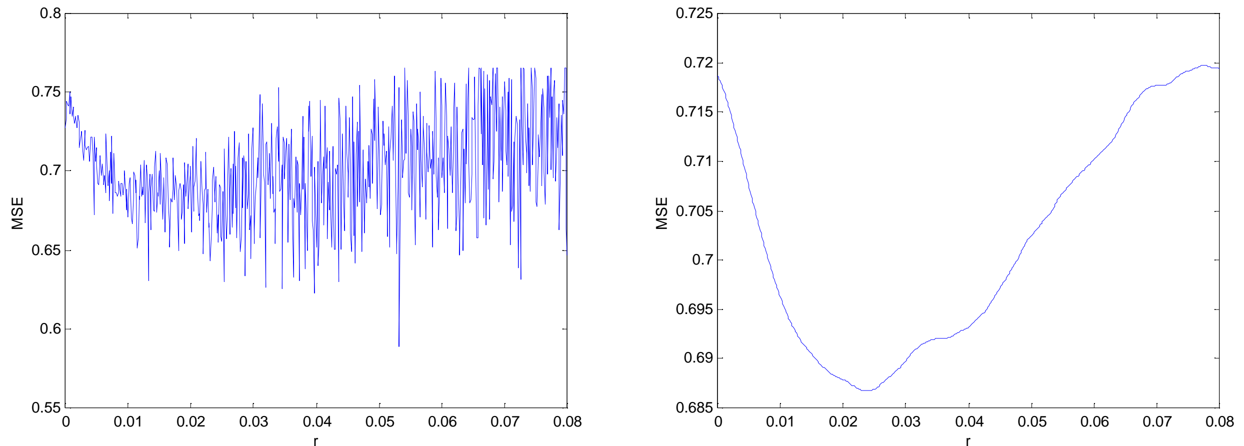


Figure 12: MSE after 100 Gauss-Newton training iterations with different random jump sizes. A random jump was made in all iterations. Again, a low-pass filtered version of the plot is included.

All the plots were made with very small r steps and then smoothed to emphasize the effect. They clearly show that small jumps (r>0) improve the performance for both optimization methods. The result in Figure 10 was not quite what we expected: The best MSE was achieved with m close to 1 ($\log_{10}(m)$ close to 0), and as the improvement never gets this big, it actually means that a jump is made at every iteration. That jumping usually gives an improvement no matter the size of the improvement was seen in many tests with different networks and number of iterations.

We were unable to find a good explanation to this interesting result. Our anticipation was that a small m value, e.g. 1% would be optimal. Maybe the jumps themselves give an improvement, or maybe the jumps sometimes produce a better staring point for the minimization algorithms. Also remember that a jump can never make the MSE worse because we save the best weight so far before the jump, it can only cause some iterations to be wasted.

The other results presented here with gradient descent optimization were produced with r = 0.01 and a small m value.

7.2 Optimal brain damage

We start with a network where all neurons in one layer are connected to all neurons in the next layer. In each iteration, the network is trained and the weight with least saliency is removed. As weights are removed, the training error is expected to increase while the generalization error decreases to some point when the optimal network structure is found. This effect in seen in Figure 13, however the improvement is not big.

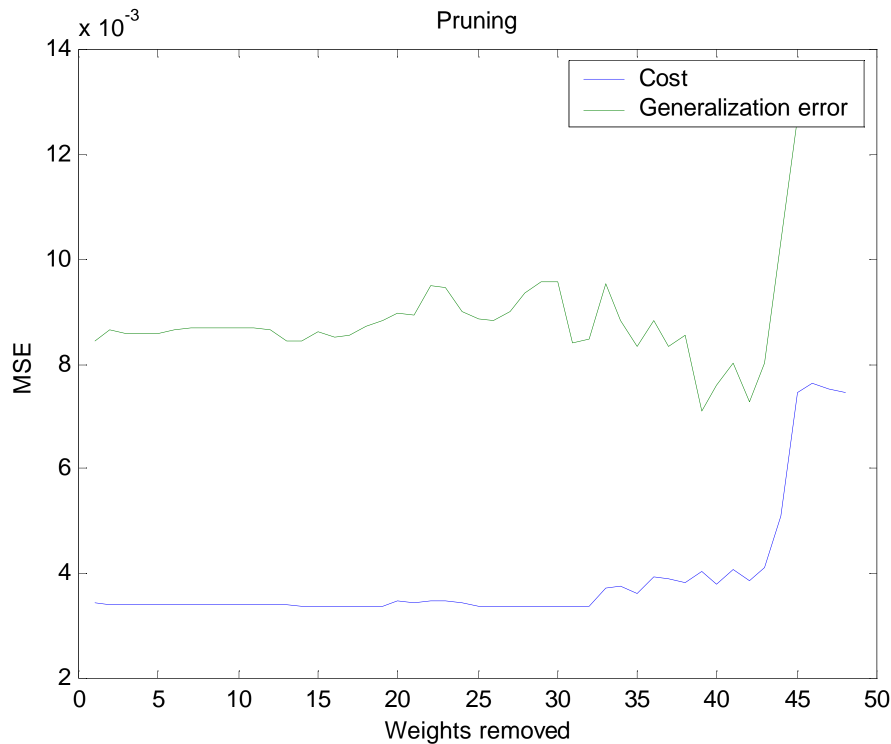


Figure 13: Development in training cost and generalization error when removing weights

The network is initiated with random weights, and as the weight training algorithms are not perfect two pruning runs can end up with very different networks and different predictions. Figure 14 to Figure 16 show three runs starting with 11 input and 4 hidden neurons. Note that the bias neurons are not shown. The samples shown in black correspond to the 11 input neurons. A prediction of 60 samples using the two networks is shown in magenta.

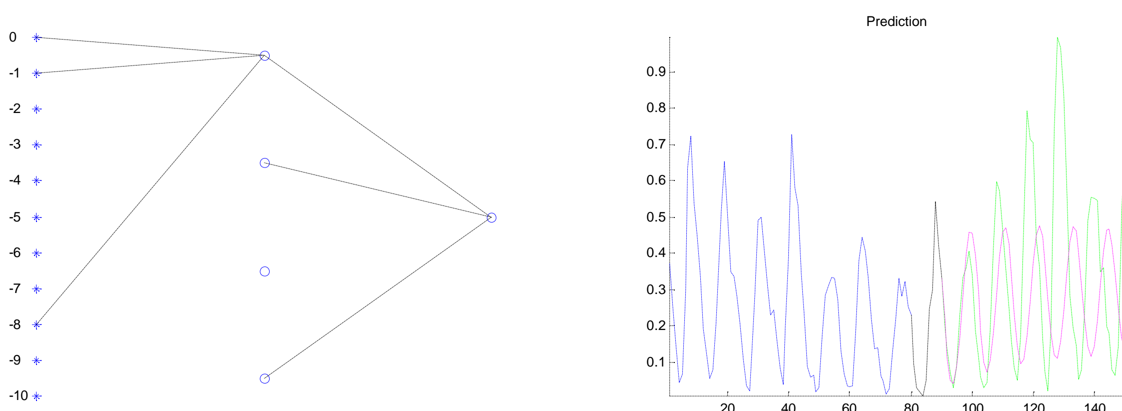
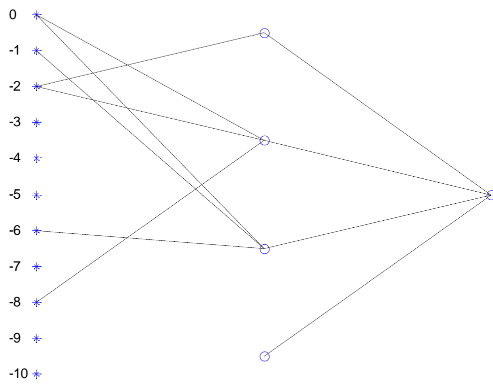


Figure 14. Prediction using pruned network with 10 weights. Note that two of the weights are pretty useless.



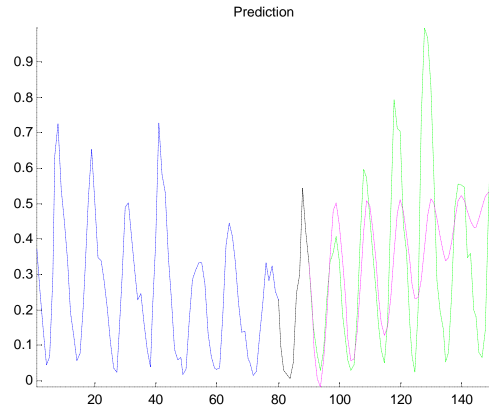
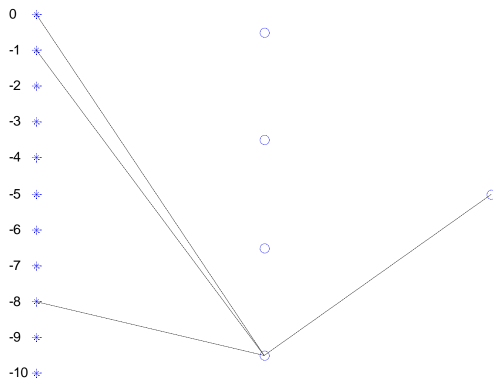


Figure 15. Prediction using pruned network with 12, weights.



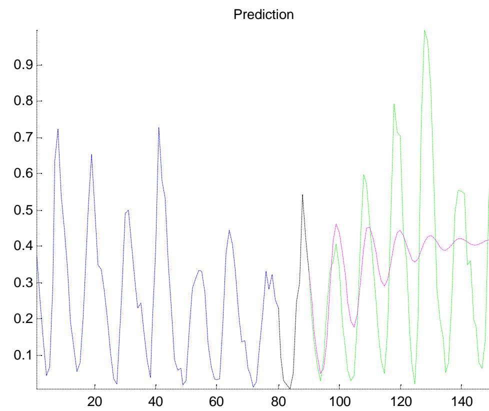
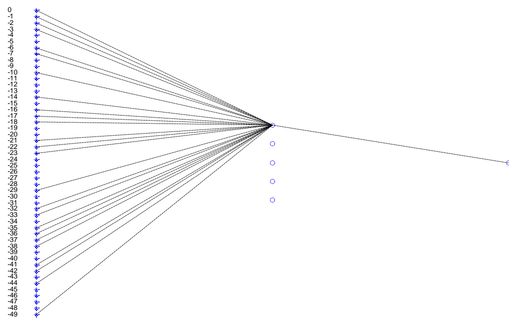


Figure 16. Prediction using pruned network with 6 weights.



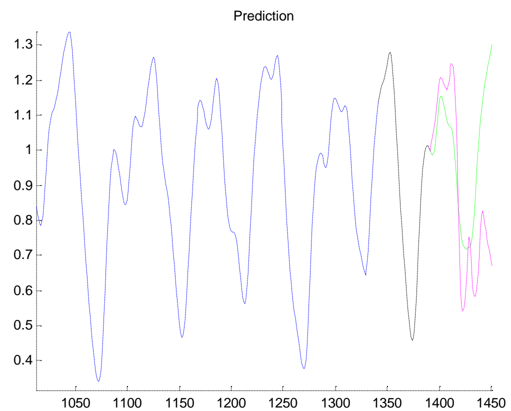


Figure 17. Prediction using pruned network with 28 weights. Mackey glass data set.

An example with a more complex data set, the Mackey glass differential equations is shown in Figure 17. The initial network here has 50 input neurons and 5 hidden. As in the examples above, the prediction only is only valid in the first few samples. Most of our tests are done with

smaller networks and simpler test data because the pruning is very time consuming. The pruning and training of this network took more than 7 minutes on a 1.6 GHz Ahtlon PC.

The OBD method successfully removed most of the weights in the network and at the same time improved the generalization error. There was not a great improvement in the prediction and generally the predictions were only valid a few samples into the future.

7.3 Adaptive regularization

It is important to use good $\mathbf{k}$ values because the wrong values can produce very poor results. In Figure 18 $\mathbf{k}$ is chosen very large and this results in a nearly constant output. This happens because the weights will decay to zero and the squared output difference term of the cost function will have no influence on training the network.

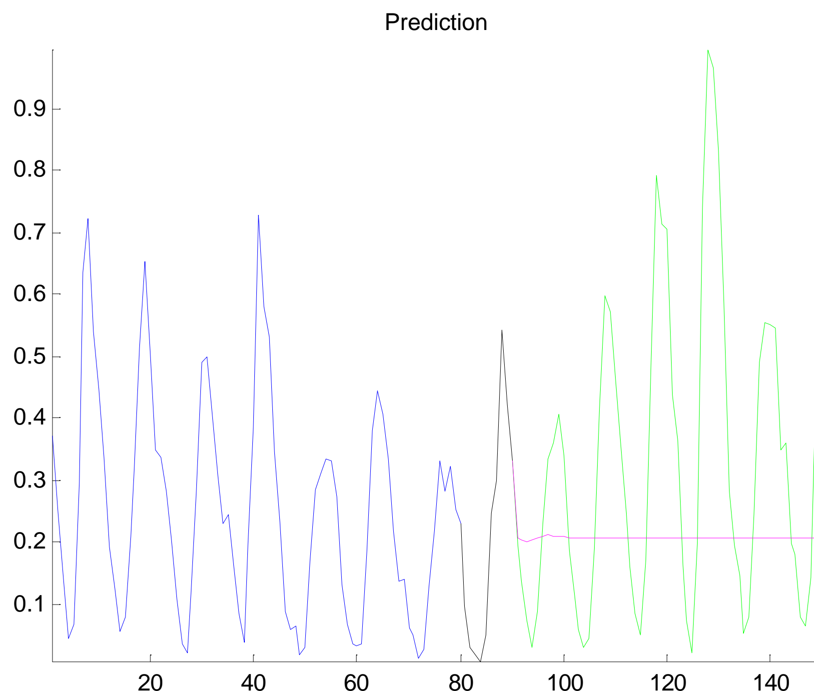


Figure 18: If the regularization parameters are chosen too large, the weights will become close to zero and the network output will be constant.

To test our algorithm for finding the optimal $\mathbf{k}$, we plotted the search path in Figure 19. The contour plot shows the MSE for $\mathbf{k}$ values in the search area. In this example it can be seen that the search goes from an area with high MSE to an area with low MSE, as expected. Each $\times$ represents a gradient descent step, and the step size is seen to decrease as we approach the minimum. The gradient descent would sometimes produce $\mathbf{k}$ values that were negative so we had to insert a line in the code to set them to zero in this case. This effect can also be seen in the example below, where the last part of the search is on the vertical axes.

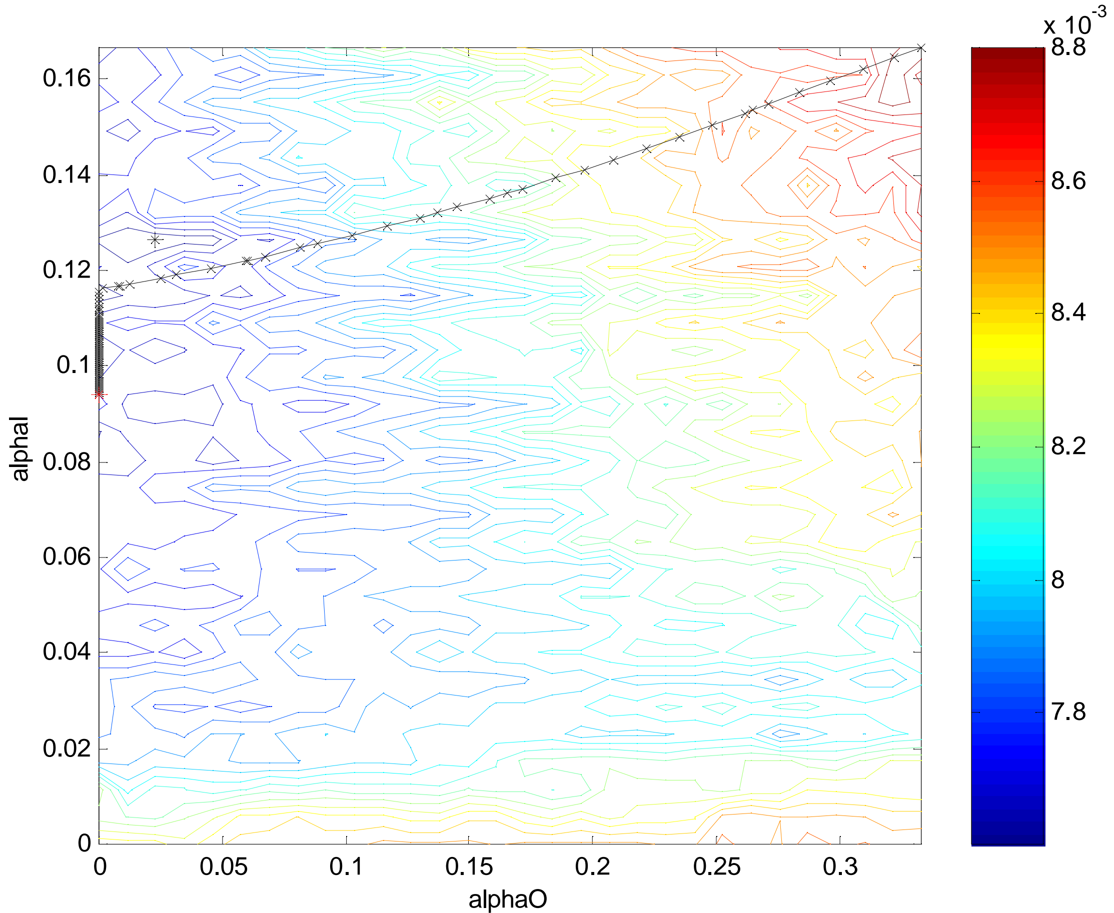


Figure 19: Optimization of $\mathbf{a} = \mathbf{k} \cdot N_{traint}$ *for at network with 8 input neurons and 2 hidden neurons. The contour plot is the MSE cost function for the sunspot data set.*

The MSE contour plot was made by slowly altering the $\mathbf{k}$ in the order shown in Figure 20. After each change in $\mathbf{k}$ the network had to be retrained starting with the weights from the previous $\mathbf{k}$. If we had started the training with random weights for each point the MSE contour plot would have been to noisy to see where the optimal point was. On the other hand, if we had computed all lines from left to right it would have resulted in a large MSE every time we started a new line.

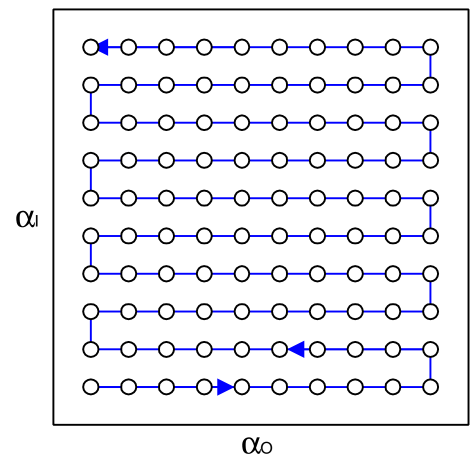


Figure 20: To minimize the noise in MSE in Figure 19, we had to move $\mathbf{a}_I$ *and* $\mathbf{a}_O$ *in this order to avoid jumping from the end of one line to the beginning of the next. The circles correspond to the* $\mathbf{k}$ *values where the MSE is evaluated.*

The search in Figure 19 looks very convincing, but we have seen other examples were the search for $\mathbf{k}$ did not find very good values. This leads us to believe that there might be an error in our implementation, but we have been unable to locate it. Even in the cases where the optimal $\mathbf{k}$ values were found, the prediction was not better than without regularization.

8 Conclusion

Most of our results were generated using the gradient descent algorithm because of the difference in running time, and often the difference in performance after a given number of iterations was very small. The noise we added when the training got stuck in a local minimum proved to be very effective, especially when retraining a network after removing a weight in OBD or adjusting $\mathbf{k}$ in adaptive regularization. To our surprise we saw that it almost always gave better results to make small jumps, no matter the size of the improvement.
To improve the effect of the random jumps, a better way of selecting the jump size is needed. One approach might be to use simulated annealing, and make the jumps get smaller and smaller.

The results when running our test programs varied very much because of the random weight initialization. We had to look at the tendency of many results to see the effect of our experiments. As expected, the predictions made with our test data were only good a few samples into the future. Apart from that, only the period of a periodic signal could be found, and there are easier ways to extract period times than using neural networks.

OBD very effectively reduced the network size, but in our matrix implementation it did not reduce the computation time because we only set the weights to zero and therefore still included them in the calculations. Because of the varying results, the effect of adaptive regularization was hard to see. As the optimization method used for the regularization parameters required re-training several times at every iteration, the optimization process was very time consuming. Neither OBD nor adaptive regularization gave a noticeable improvement in the predictions with our test data, and only little improvement in mean square error.

9 Literature

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

Introduction to Artificial Neural Networks
Jan Larsen
IMM, DTU, November 1999

Adaptive regularization in Neural Network Modeling
Jan Larsen, Claus Svarer, Lars Nonboe Andersen and Lars Kai Hansen
IMM, DTU

Appendix 1 Weight optimization source

The source code for the important functions is presented in appendix 1 to 3. All test programs have not been included as there is quite a bit of code.

```
% Evaluate cost
function E=Cost(y,Ty,Wi,Wo,alphaI,alphaO);
E = (y-Ty)*(y-Ty)'+sum(sum(Wi.^2))*alphaI+sum(sum(Wo.^2))*alphaO;

% Find response for input x
function [y,h]=EvaluateNN(x,Wi,Wo)
h = tanh(Wi*x);
h = [h;ones(1,size(x,2))];
y = Wo*h;

% Find response for input x, append threshold neuron values of 1
function [y,h]=EvaluateNN1(x,Wi,Wo)
Threshold = ones(1,size(x,2));
h = tanh(Wi*[x;Threshold]);
y = Wo*[h;Threshold];
```

```
% Update weights using gradient descent and bisection
function [Wi,Wo,eta] = TrainNNGradient(Tx,Ty,Wi,Wo,eta,alphaI,alphaO,WiMask,
    WoMask)

nTrain = size(Tx,2);
Threshold = ones(1,nTrain);
Tx = [Tx;Threshold];

% Evaluate network
uI = Wi*Tx; % Activation og hidden layer
h = tanh(uI); % Hidden layer transfer function
dh = 1-h.*h;
h = [h;ones(1,size(h,2))]; % Append threshold neuron to hidden layer
uO = Wo*h;
y = uO; % Output layer transfer function (linear)
e = Ty-y;

% Find derivatives at output layer
dO = e; %.*dActivation(uO);
dSdWo = 1/nTrain * (dO*h' + alphaO*Wo);
% Find derivatives at hidden layer
dI = Wo(:,1:size(Wo,2)-1)'*dO.*dh;
dSdWi = 1/nTrain * (dI*Tx' + alphaI*Wi);

if nargin>=8
  dSdWi = dSdWi.*WiMask;
end
if nargin>=9
  dSdWo = dSdWo.*WoMask;
end

% Find good eta by bisection
E = Cost(y,Ty,Wi,Wo,alphaI,alphaO);
while Cost(EvaluateNN(Tx,Wi+eta*dSdWi,Wo+eta*dSdWo),Ty,Wi+eta*dSdWi,
    Wo+eta*dSdWo, alphaI,alphaO)>E
  eta = eta*0.5;
end

% Update weights
Wo = Wo+eta*dSdWo;
Wi = Wi+eta*dSdWi;
```

```
% Update weights using Gauss-Newton and bisection
function [Wi,Wo,eta] = TrainNNGaussNewton(Tx,Ty,Wi,Wo,eta,alphaI,alphaO,
    WiMask,WoMask)

nTrain = size(Tx,2);
Threshold = ones(1,nTrain);

Tx = [Tx;Threshold];

% Evaluate network
uI = Wi*Tx; % Activation og hidden layer
h = tanh(uI); % Hidden layer transfer function
dh = 1-h.*h;
h = [h;ones(1,size(h,2))]; % Append threshold neuron to hidden layer
uO = Wo*h;
y = uO; % Output layer transfer function (linear)
e = Ty-y;

% Find derivatives at output layer
dO = e; %.*dActivation(uO);
dWo = 1/nTrain * (dO*h' + alphaO*Wo);

J = reshape(dWo,1,size(dWo,1)*size(dWo,2));
H = J'*J; % Second derivative approximation
H = H+diag(ones(1,length(J))*max(max(H))*1e-4); % Make H non-singular

WoStep = reshape(H\J',size(dWo,1),size(dWo,2));

% Find derivatives at hidden layer
dI = Wo(:,1:size(Wo,2)-1)'*dO.*dh;

dWi = 1/nTrain * (dI*Tx' + alphaI*Wi);

J = reshape(dWi,1,size(dWi,1)*size(dWi,2));
H = J'*J; % Second derivative approximation
H = H+diag(ones(1,length(J))*max(max(H))*1e-4); % Make H non-singular

WiStep = reshape(H\J',size(dWi,1),size(dWi,2));

if nargin>=8
  WiStep = WiStep.*WiMask;
end
if nargin>=9
  WoStep = WoStep.*WoMask;
end

% Find good eta by bisection
E = Cost(y,Ty,Wi,Wo,alphaI,alphaO);
while Cost(EvaluateNN(Tx,Wi+eta*WiStep,Wo+eta*WoStep),
    Ty,Wi+eta*WiStep,Wo+eta*WoStep,alphaI,alphaO)>E
  eta = eta*0.5;
end

% Update weights
Wo = Wo+eta*WoStep;
Wi = Wi+eta*WiStep;
```

```
% Do IterCount training iterations (Jump=r, MinStep=m)
function [Wi,Wo,E,ET,G] = DoTraining(Tx,Ty,Gx,Gy,Wi,Wo,WiMask,WoMask,
    eta,alphaI,alphaO,IterCount,Jump,MinStep)

if nargin<13
  Jump = 0.01;
end
if nargin<14
  MinStep = 0.0001;
end

BestE = inf;

Wi = WiMask.*Wi;
Wo = WoMask.*Wo;

JumpCount = -1;

for i=1:IterCount
  [Wi,Wo,eta] = TrainNNGradient(Tx,Ty,Wi,Wo,eta*2,alphaI,
      alphaO,WiMask,WoMask);
  %[Wi,Wo,eta] = TrainNNGaussNewton(Tx,Ty,Wi,Wo,eta*2,alphaI,alphaO,
      WiMask,WoMask);

  y = EvaluateNN1(Tx,Wi,Wo);
  LastE = Cost(y,Ty,Wi,Wo,alphaI,alphaO)/length(y);
  E(i) = LastE;
  ET(i) = Cost(y,Ty,0,0,0,0)/length(y);
  if ~isempty(Gx)
    y = EvaluateNN1(Gx,Wi,Wo);
    G(i) = Cost(y,Gy,0,0,0,0)/length(y);
  end

  if (i>1 & (E(i-1)-E(i))/E(i)<MinStep) | i==IterCount
    if LastE<BestE % Improvement found
      BestE = LastE;
      BestWi = Wi;
      BestWo = Wo;
    else % No improvement, restore weights
      Wi = BestWi;
      Wo = BestWo;
    end
    Wi = Wi+WiMask.*randn(size(WiMask))*max(max(Wi))*Jump;
    Wo = Wo+WoMask.*randn(size(WoMask))*max(max(Wo))*Jump;
    JumpCount = JumpCount+1;
  end
end
Wi = BestWi;
Wo = BestWo;
```

Appendix 2 OBD source

```
% Remove weight with least saliency
function [WiMask,WoMask] = RemoveWeight(Tx,Ty,Wi,Wo,alphaI,alphaO,
    WiMask,WoMask)

nTrain = size(Tx,2);
WiCount = size(Wi,1)*size(Wi,2);
WoCount = size(Wo,1)*size(Wo,2);
Threshold = ones(1,nTrain);

% Evaluate network
Tx = [Tx;Threshold];
uI = Wi*Tx; % Activation of hidden layer
h = tanh(uI); % Hidden layer transfer function

% Pseudo Hessain
[r c] = size(Wo);
dO = ones(nTrain,r);
dI = (1 - h.^2) .* (dO * Wo(:,1:c-1))';
h = [h;Threshold]; % Append threshold neuron to hidden layer

% Pseudo Hessian elements for the output weights
ddWo = ((h.^2)*dO)';
% Pseudo Hessian elements for the input weights
ddWi = (dI.^2)*(Tx.^2)';

% Add second derivatives of weight decay term
ddWi = ddWi + alphaI;
ddWo = ddWo + alphaO;

% Calculate saliencies for the input weights
Si = (alphaI + 0.5*ddWi).*(Wi.^2);
% Calculate saliencies for the output weights
So = (alphaO + 0.5*ddWo).*(Wo.^2);

% Set removed weight saliencies to big values
WoMask = reshape(WoMask,1,WoCount);
So = (1-WoMask)*1e100 + reshape(So,1,WoCount); % Output saliencies
WiMask = reshape(WiMask,1,WiCount);
Si = (1-WiMask)*1e100 + reshape(Si,1,WiCount); % Input saliencies

% Find minimum
[MinSo,MinSoI] = min(So);
[MinSi,MinSiI] = min(Si);
if length(find(WoMask>0.5))==1
  MinSo = 1e200; % Dont't remove the last one
end
if length(find(WiMask>0.5))==1
  MinSi = 1e200; % Dont't remove the last one
end
if MinSo<MinSi
  WoMask(MinSoI) = 0;
else
  WiMask(MinSiI) = 0;
end

WiMask = reshape(WiMask,size(Wi));
WoMask = reshape(WoMask,size(Wo));
```

```
%Prepare for prediction using the sunspot data set
function [nInput,nHidden,nOutput,Tx,Ty,Gx,Gy] = PredictionSunspot

nInput = 8;
nHidden = 2;
nOutput = 1;

Samples = 120;

G = load('sunspot.txt');
G = G(:,2);

S = G(1:Samples+nInput);
for s=1:length(S)-nInput
  for i=1:nInput
    Tx(i,s) = S(s+i-1);
  end
  Ty(s) = S(s+nInput);
end

G = G(Samples:length(G));
for s=1:length(G)-nInput
  for i=1:nInput
    Gx(i,s) = G(s+i-1);
  end
  Gy(s) = G(s+nInput);
end

% Do Count samples forward prediction
function y = ForwardPrediction(Input,Wi,Wo,Count);

y = zeros(1,Count);
for i=1:Count
  y(i) = EvaluateNN1(Input',Wi,Wo);
  Input = [Input(2:length(Input)) y(i)];
end
```

```
% OBD main test program
function OBDmain

[nInput,nHidden,nOutput,Tx,Ty,Gx,Gy] = PredictionSunspot;

BestE = inf;

StartWeights = 1;
etaStart = 10;

alphaI = 0.0;
alphaO = 0.0;

Iterations = [1:1000];

% Input-hidden weights including threshold
Wi = randn(nHidden,nInput+1)*StartWeights;
% Hidden-output weights including threshold
Wo = randn(nOutput,nHidden+1)*StartWeights;

WiMask = ones(size(Wi));
WoMask = ones(size(Wo));

WeightCount = size(Wi,1)*size(Wi,2)+size(Wo,1)*size(Wo,2);
WeightRemove = [1:WeightCount-2];

% Do initial training
[Wi,Wo,E,ET,G] = DoTraining(Tx,Ty,Gx,Gy,Wi,Wo,WiMask,WoMask,
    etaStart,alphaI,alphaO,length(Iterations));

figure
semilogy(Iterations,E,Iterations,ET,Iterations,G);
title('Initial training')
legend('Cost','Cost without weight decay','Generalization error')
xlabel('Iterations')
ylabel('MSE')
clear E ET G
drawnow

% Remove weights
figure
for i=WeightRemove
  [WiMask,WoMask] = RemoveWeight(Tx,Ty,Wi,Wo,alphaI,alphaO,WiMask,WoMask);

  [Wi,Wo,Ei] = DoTraining(Tx,Ty,Gx,Gy,Wi,Wo,WiMask,WoMask,
      etaStart,alphaI,alphaO,150);

  plot(Ei), drawnow

  y = EvaluateNN1(Tx,Wi,Wo);
  E(i) = Cost(y,Ty,Wi,Wo,alphaI,alphaO)/length(y); % Cost with weight decay

  ET(i) = Cost(y,Ty,0,0,0,0)/length(y); % Cost without weight decay

  y = EvaluateNN1(Gx,Wi,Wo);
  G(i) = Cost(y,Gy,0,0,0,0)/length(y); % Generalization error

  NewE = G(i);
  if NewE<=BestE % Save best netwotk structure
    DrawNet(Wi,Wo);
```

```
    drawnow
    BestWi = Wi;
    BestWo = Wo;
    BestWiMask = WiMask;
    BestWoMask = WoMask;
    BestE = NewE;
  end

  WeightsLeft = WeightCount-i
end

% Show improvement
if alphaI==0 & alphaO==0
  plot(WeightRemove,E,WeightRemove,G);
  legend('Cost','Generalization error')
else
  plot(WeightRemove,E,WeightRemove,ET,WeightRemove,G);
  legend('Cost','Cost without weight decay','Generalization error')
end
title('Pruning')
xlabel('Weights removed')
ylabel('MSE')

% Restore best network
Wi = BestWi;
Wo = BestWo;
WiMask = BestWiMask;
WoMask = BestWoMask;

% Show network
figure
DrawNet(Wi,Wo);
OptimalWeightCount = length(find(WiMask>0.5))+length(find(WoMask>0.5))

% Do forward prediction
figure
hold on
Forward = 60;

BackI = 1:length(Gy)-Forward-nInput+1;
InputI = length(Gy)-Forward-nInput+1:length(Gy)-Forward;
ForwardI = length(Gy)-Forward:length(Gy);

plot(BackI,Gy(BackI),'b')
plot(ForwardI,Gy(ForwardI),'g')
plot(InputI,Gy(InputI),'k')
y = ForwardPrediction(Gy(InputI),Wi,Wo,Forward);
plot(ForwardI,[Gy(ForwardI(1)) y],'m')

hold off
axis tight
title('Prediction')
```

Appendix 3 Adaptive regularization source

```
% Find optimal regularization parameters and return search path
function [alphaI,alphaO,Wi,Wo,alphaPath] = FindRegulizers(Tx,Ty,Vx,Vy,Wi,
    Wo,WiMask,WoMask,etaStart,alphaI,alphaO)

nValidation = size(Vx,2);
Threshold = ones(1,nValidation);

Vx = [Vx;Threshold];

iterations = 50;

L = Cost(EvaluateNN(Vx,Wi,Wo),Vy,0,0,0,0);
for i=1:iterations
  % Evaluate network
  uI = Wi*Vx;       % Activation og hidden layer
  h = tanh(uI);     % Hidden layer transfer function
  dh = 1-h.*h;
  h = [h;ones(1,size(h,2))]; % Append threshold neuron to hidden layer
  uO = Wo*h;
  y = uO;           % Output layer transfer function (linear)
  e = Vy-y;

  % Find derivatives at output layer
  dO = e; %.*dActivation(uO);
  dWo = (dO*h' + alphaO*Wo);

  % Find derivatives at hidden layer
  dI = Wo(:,1:size(Wo,2)-1)'*dO.*dh;
  dWi = dI*Vx' + alphaI*Wi;

  dWi = dWi.*WiMask;
  dWo = dWo.*WoMask;

  dWiV = dWi;
  dWoV = dWo;
  [d1,d2]=size(dWiV);

  Ji = reshape(dWiV,1,d1*d2);
  Hi = Ji'*Ji;
  DiV = reshape((Hi+1e-6*eye(d1*d2)*max(max(Hi)))\Ji',d1,d2);
  Ho = dWoV'*dWoV;
  DoV = (Ho+1e-6*eye(length(dWoV))*max(max(Ho)))\dWoV';
  [x y] = size(DiV);
  DiV = reshape(DiV,x*y,1);


  X_ValidI = 2*reshape(Wi,1,d1*d2)*DiV/(d1*d2);
  X_ValidO = 2*Wo*DoV/length(Wo);

  % Update alphaO and do bisection to find etaV

  etaV = 0.1;
  AlphaReset = 0;

  for j=1:20
    newAlphaO = alphaO - etaV*X_ValidO;
```

```
    if newAlphaO < 0
      newAlphaO = AlphaReset;
    end
    newAlphaI = alphaI - etaV*X_ValidI;
    if newAlphaI <= 0
      newAlphaI = AlphaReset;
    end

    [Wi,Wo,E] = DoTraining(Tx,Ty,[],[],Wi,Wo,WiMask,WoMask,
        etaStart,newAlphaI,newAlphaO,80);
    if Cost(EvaluateNN(Vx,Wi,Wo),Vy,0,0,0,0)<L
      break
    end
    etaV = etaV*0.5;
    newAlphaO = alphaO;
    newAlphaI = alphaI;
  end
  alphaO = newAlphaO;
  alphaI = newAlphaI;

  L = Cost(EvaluateNN(Vx,Wi,Wo),Vy,0,0,0,0);
  LPath(i) = L;
  alphaPath(1,i) = alphaI;
  alphaPath(2,i) = alphaO;
  subplot(2,1,1)
  plot(alphaPath(2,:),alphaPath(1,:),'-xk'); hold on
  plot(alphaPath(2,end),alphaPath(1,end),'*r'); drawnow
  hold off
  subplot(2,1,2)
  plot(LPath)
end
```