Adaptive Regularization in Neural Network Filters

Course 02455 Advanced Digital Signal Processing

May 23rd, 2002

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

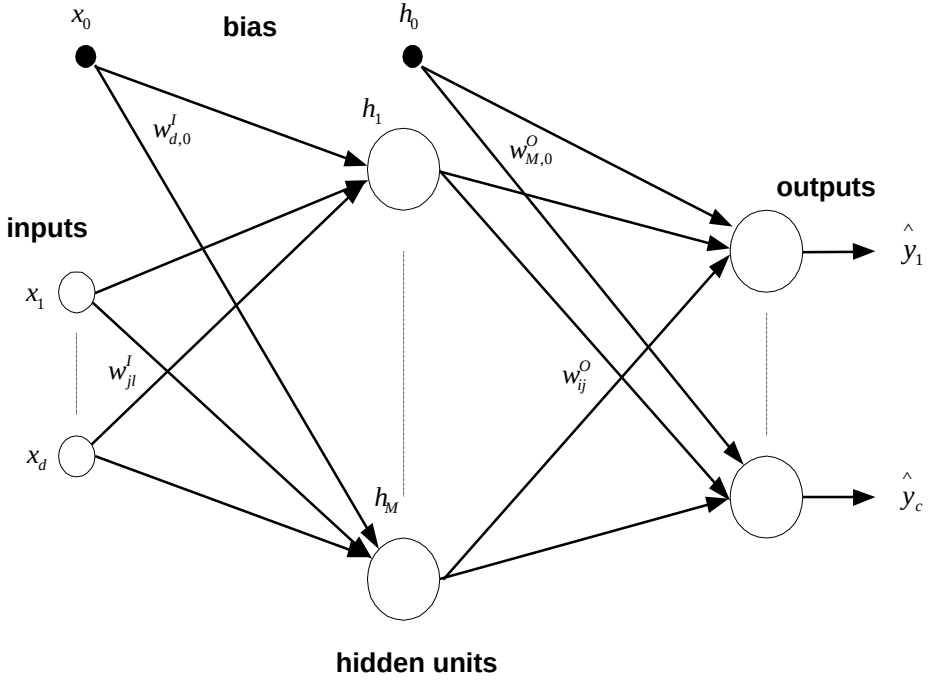
Introduction

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. 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. Artificial neural networks are proving to be more capable of solving significant problems, than more conventional algorithms, because it is an attempt to approach the functions of the human brain.

We have decided to work with time series prediction using a two-layer feed forward network, where 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 We have chosen to implement the neural network with the Gauss-Newton algorithm, Optimal Brain Damage (OBD) and adaptive regularization. To get a better understanding of neural networks, we have also chosen to implement the neural network from scratch using our own code.

Feed-forward Networks

The larger circles are the neurons in the network and the networks degrees of freedom of the network are the total number of weights.



The processing in the network is given by

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

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

Neural Network Training

The neural network learns by looking at previous examples of a given series to predict future samples, i.e. 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.

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

All training techniques require either the computation of the first or second derivatives of the cost function with respect to the weights.

The back-propagation technique provides a computational efficient method of evaluating the first 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.

`DoTraining.m`

Training iterations:

- Find weight update direction based on first derivative (and second derivative approximation)
- Do bisection until cost improvement found:
  - Test network with updated weights
  - If no improvement, divide weight update by two
- If improvement very small:
  - If best weights so far:
    - Store weights
  - Else:
    - Restore best weights
  - Make random jump by adding a small value to all weights

Return best weights

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. An estimate for the generalization ability the MSE can be calculated for a data set, which was not included in the training.

An estimate for the generalization error is given by

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

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. In our implementation we perform a random jump if the improvement in an iteration is very small to avoid getting stuck in a local minimum.

Gradient Descent Algorithm

An approach to determine the weights that minimize the cost function is to use the stochastic algorithm; gradient descent with back-propagation.

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 $\Delta \underline{w}$ is given by the negative gradient of the cost function.

The weights are updated at each step as follows:

$$\mathbf{w}^{n+1} = \mathbf{w}^{n} - h\nabla S_T\left(\mathbf{w}^{n}\right)$$

Gauss-Newton Algorithm

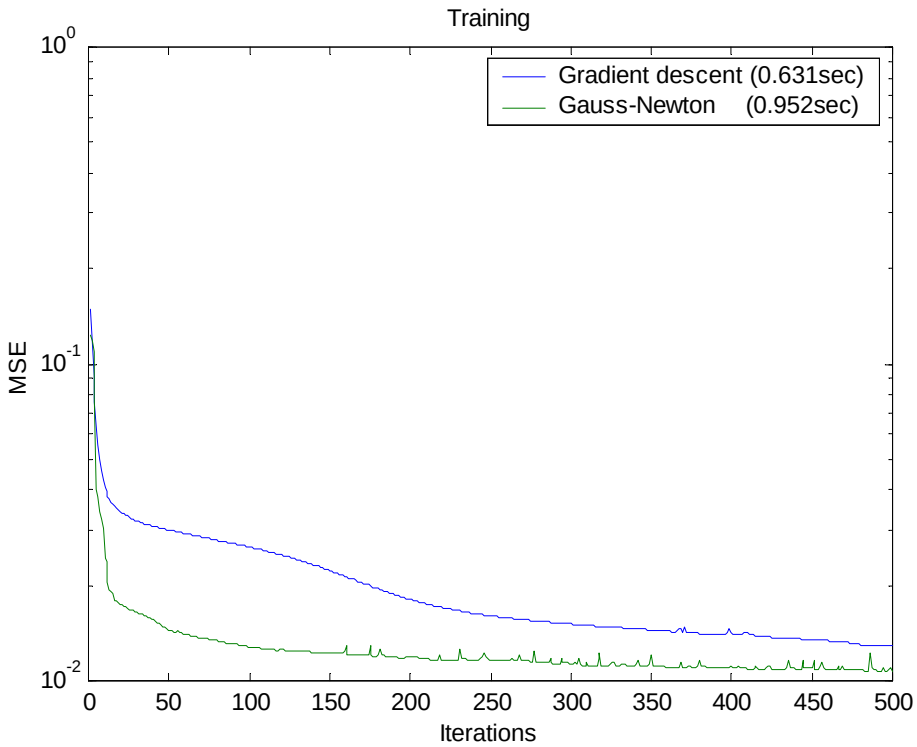
The oscillating behavior of the gradient descent algorithm gives rise to finding a more precise method in which a more direct step towards the cost functions minimum is needed.

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 weights are updated at each iteration as follows:

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



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

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

$$dS_t = \left(k + \frac{1}{2} \frac{\partial^2 S_t(\underline{\hat{w}})}{\partial w_j^{\,2}} \right) \hat{w}_j^{\,2}$$

The second derivative of S_t can be approximated with the following expression:

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

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

Prepare training and generalization data

Do initial training of weights in fully connected network — `DoTraining.m`

Until only two weights left:

- Remove weight with least saliency — `RemoveWeight.m`
- Train pruned network — `DoTraining.m`
- Estimate generalization error
- If best network so far:
  - Save network structure

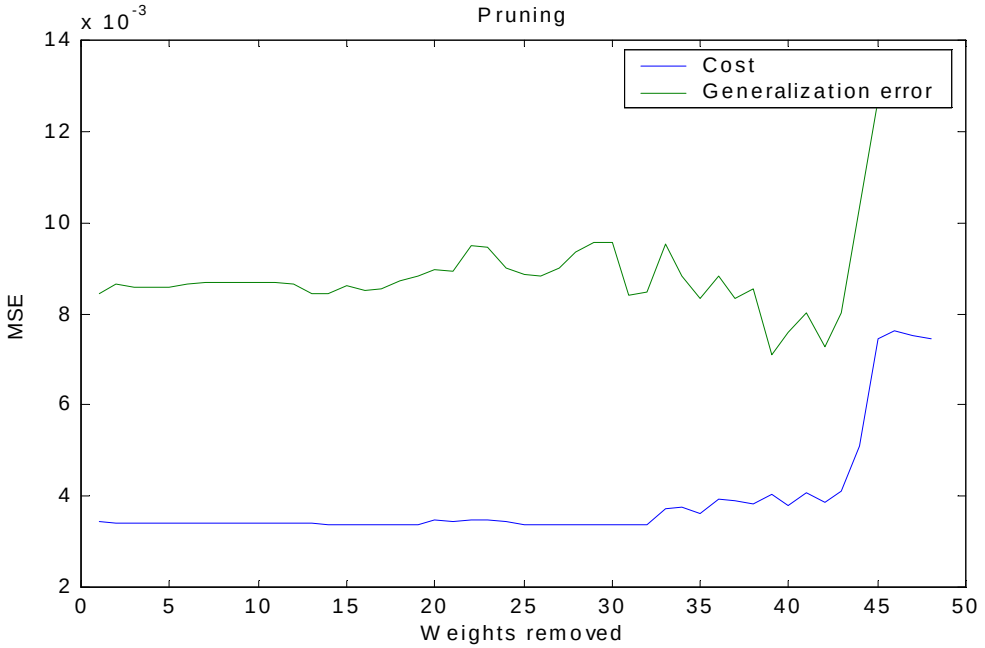
Restore best network

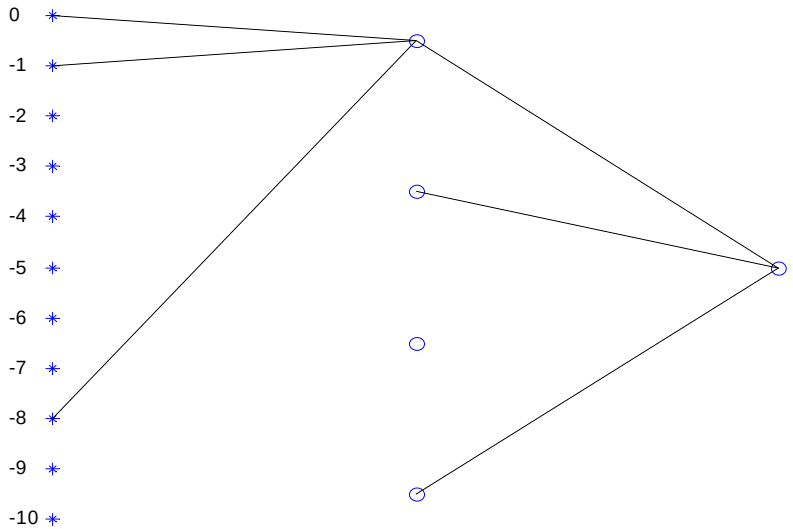
`RemoveWeight.m`

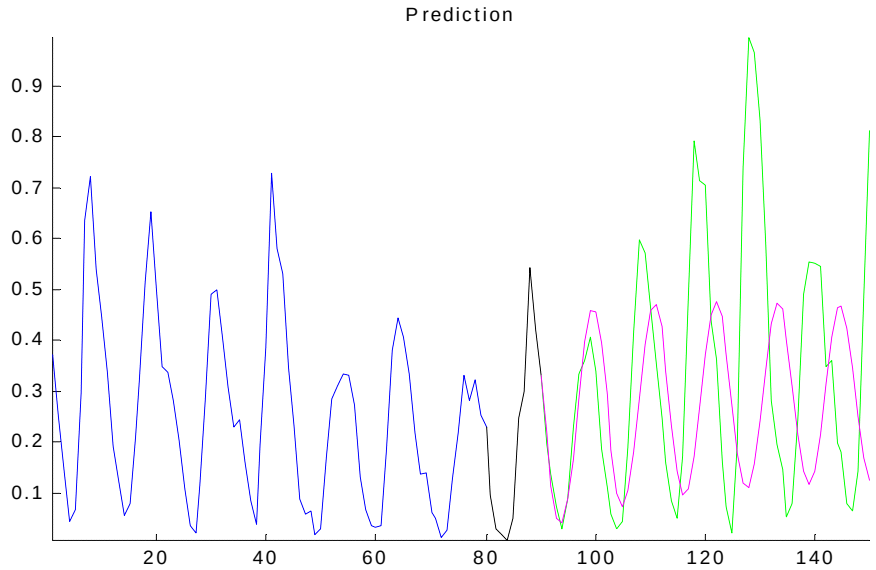
Calculate saliencies
Sort weights by saliency
Remove least significant weight

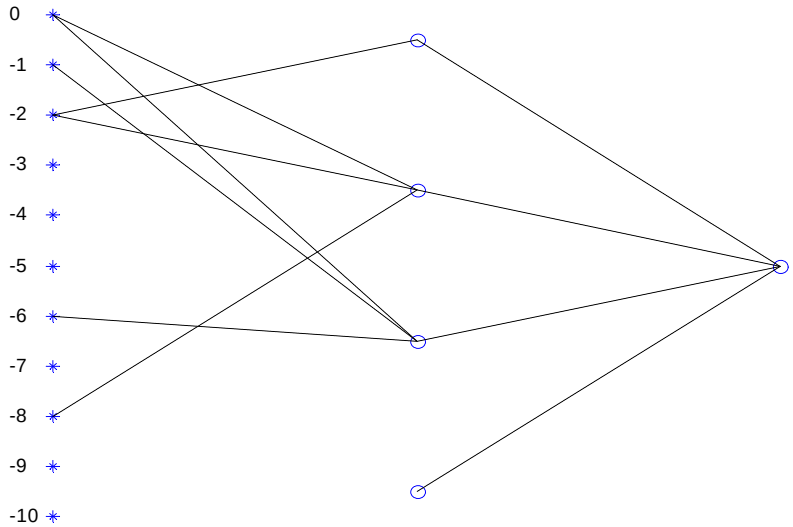
We start with a network where all neurons in one layer are connected to all neurons in the next layer. In each iteration, the network is trained and the weight with least saliency is removed. As weights are removed, the training error is expected to increase while the generalization error decreases to some point when the optimal network structure is found.

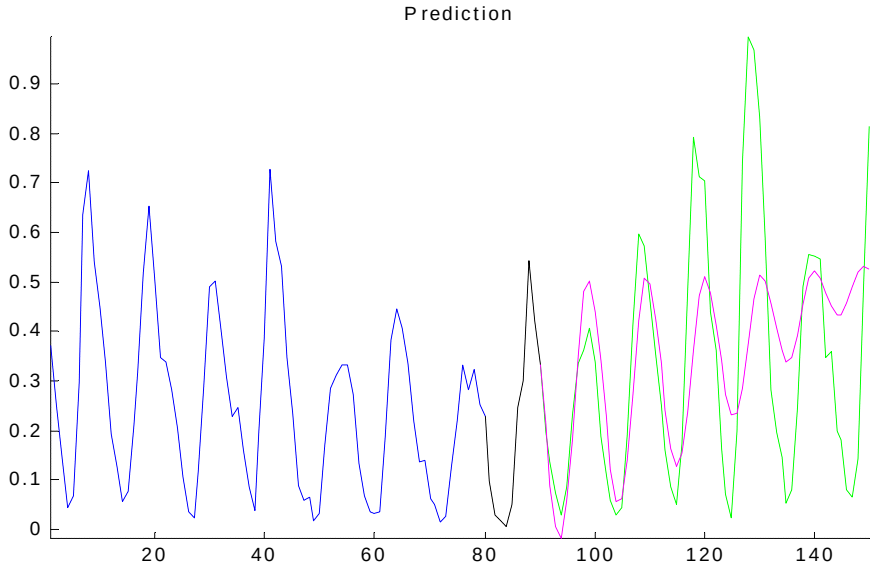
Development in training cost and generalization error when removing weights. The plot show pruning of a network with 11 input and 4 hidden neurons using the sunspot data. The best generalization is seen to be achieved after removing about 40 weights.



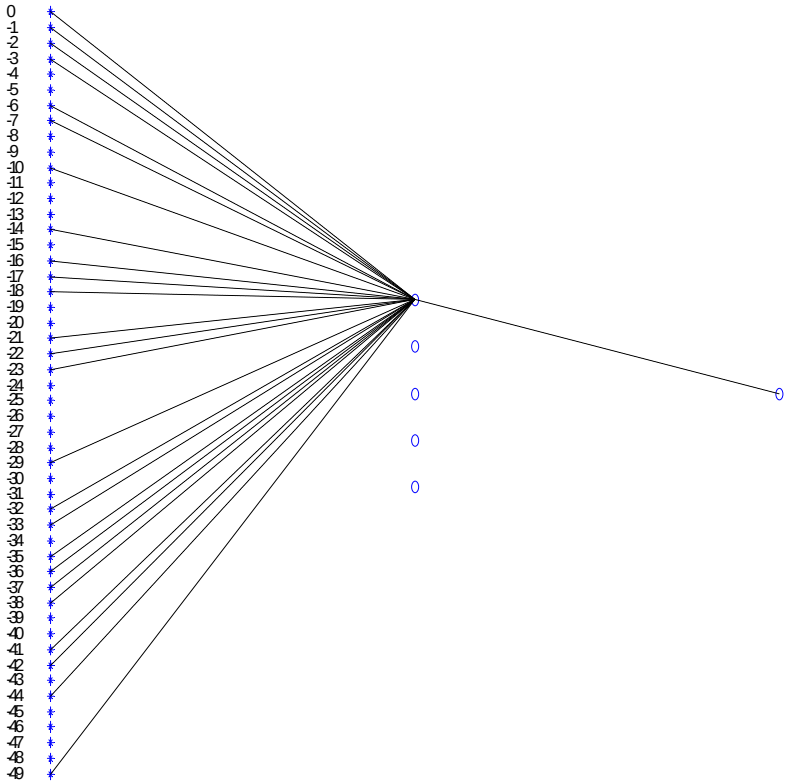


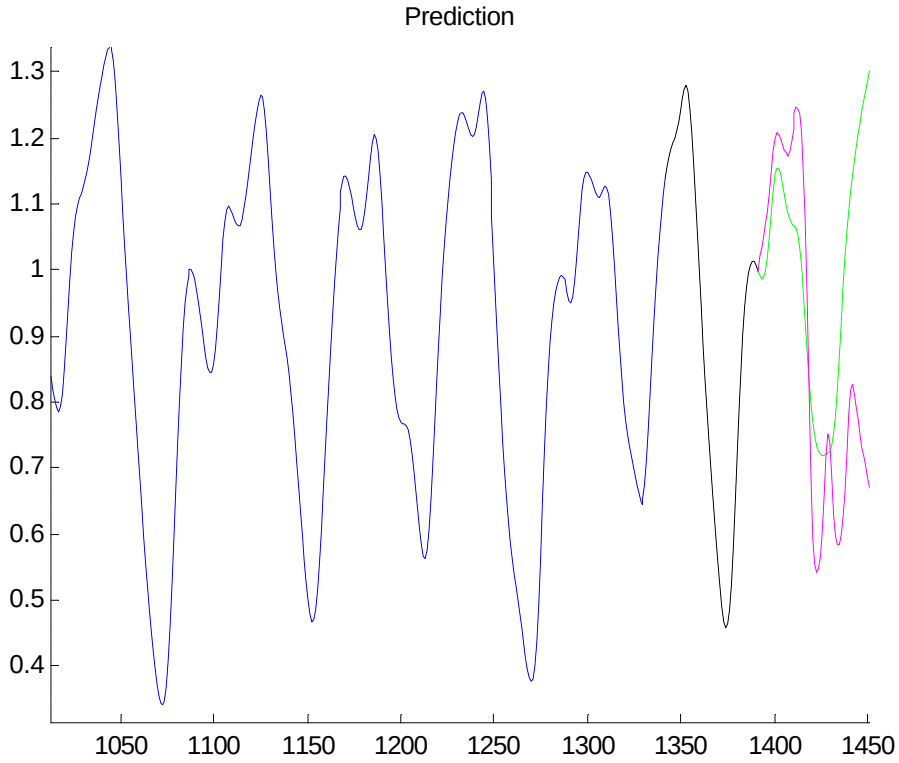






The network is initiated with random weights, and as the weight training algorithms are not perfect two pruning runs can end up with very different networks and different predictions. The figures show two runs starting with 11 input and 4 hidden neurons. Note that the bias neurons are not shown. The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





This is an example with a more complex data set, the Mackey glass differential equations. 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.

Adaptive regularization

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

$$C_T(\mathbf{w}) = S_T(\mathbf{w}) + k^I \sum_j {w^I}_j^2 + k^O \sum_j {w^O}_j^2$$

The regularization terms forces the magnitudes of the weights towards zero, as the error is large for large weights. The effect is somewhat the same as in OBD, because un-necessary 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 optimal regularization parameters can be found using a gradient descent approach similar to the one used to find the weights. With a validation data set, the search and evaluation of the optimal k = [k^I , k^O] are found by minimizing the cross-validation estimate

$$\hat{\Gamma} = S_v(\hat{\mathbf{w}})$$

$S_{v_j}(\hat{\mathbf{w}}_j)$ is the cost function of the validation set, where $\hat{\mathbf{w}}_j$ are the weights that minimize the augmented cost function $C_T(\mathbf{w})$. By updating regularization parameters as follows

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

the optimal regularization parameters can be found.

`FindRegulizers.m`

Evaluate MSE cost for validation data
Training iterations:

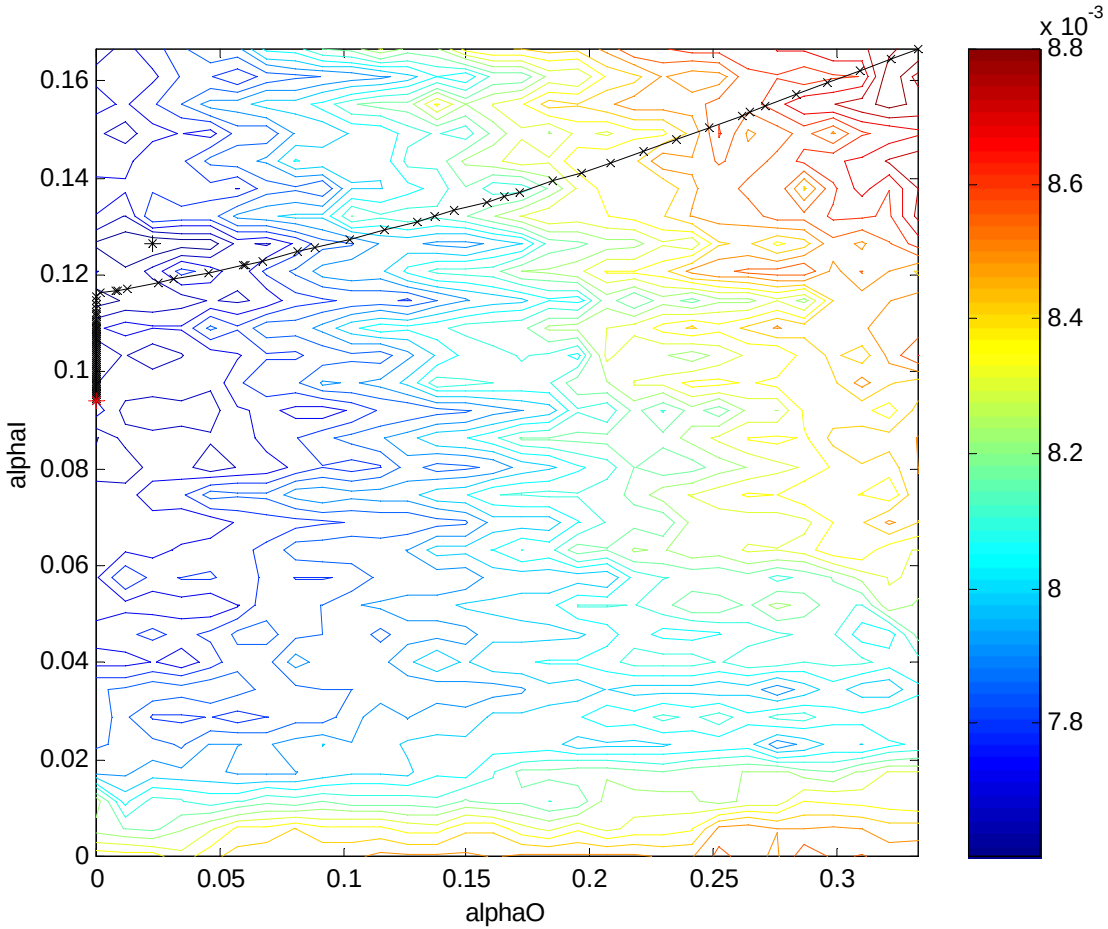
Find k_I and k_O update directions
Do bisection until cost improvement found:

Train network with updated k — `DoTraining.m`

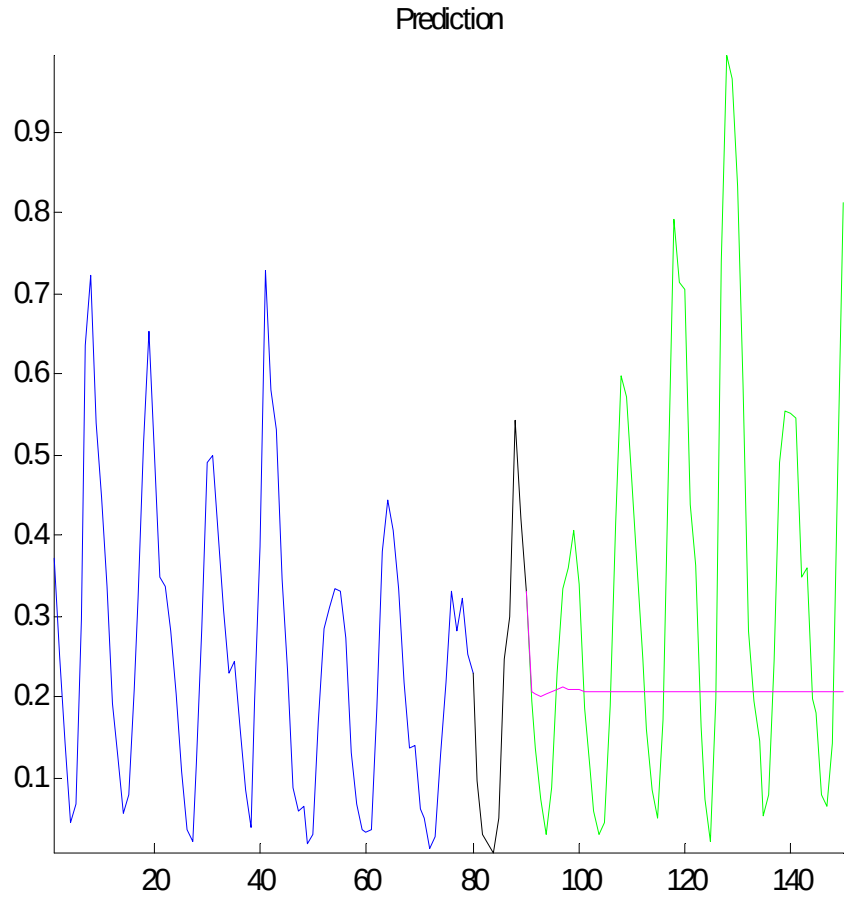
Test network with updated weights
If no improvement, divide weight update by two

Store new MSE cost for validation data

The adaptive regularization can be combined with OBD so that the network is trained with optimized regularization parameters every time a weight is removed.



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



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

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 k in adaptive regularization.

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.

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