THE GATED EXPERTS NETWORK AND TIME SERIES PREDICTION

September 14, 1998

ing. J.Bron and ing. R.M.Zijlstra

Summary

One of the main issues in the research on time series is its prediction. Artificial neural networks are suitable tools for this purpose. Most traditional models are global models, assuming stationary. This ignores the fact that most real–world time series are non–stationary. An important subclass of non–stationary is piecewise stationary, where the series switches between different stationary regimes. Using an artificial neural network for the prediction in each regime, solves the problem of non–stationarity. To predict the transitions between the regimes, an additional artificial neural network can be used, assuming that these transitions are unknown. The gated experts network combines these properties. Key elements are: non–linearity, predicting regime transitions and local predictors for each regime.

The Expectation–Maximization learning algorithm is used to update the free parameters of the network. Our goal is to gain insight in the application of the gated experts network to real–world time series prediction. This is achieved by studying the gated experts network, implementing it in InterAct© and conducting several experiments.

The gated experts network is a reasonable good choice for real–world time series prediction. It uses the experts as local predictors and the gate plausibly allocates the experts to local regions of the input space. The gate splits the input space, but not always as one might expect. This input space splitting depends on the initialization of the weights of the gate and experts. The choice of the free parameters depends on the kind of experiment conducted. This network is a useful tool for analyzing the underlying dynamics of a time series.

The gated experts network can be modified by adding different density functions to individual experts, applying dynamic growth or pruning of the number of experts and hidden units of the experts. The implementation of this network can be extended to include separate tapped delay lines for the experts and the gate, to capture the periodicities in the time series.

Chapter 1

Introduction

One of the main issues in the research on time series is its prediction. In the design of a proper predictor, a careful characterization of the time series has to be developed. Artificial neural networks are suitable tools for this purpose, because of their ability to identify non–linearity in time series. Most traditional models are global models, assuming stationary. This ignores the fact that most real–world time series are non–stationary. An important subclass of non–stationary is piecewise stationary, where the series switches between different stationary regimes. Using an artificial neural network for the prediction in each regime, solves the problem of non–stationarity. To predict the transitions between the regimes, an additional artificial neural network can be used, assuming that these transitions are unknown (hidden). A special form of artificial neural networks, called the gated experts network, combines these properties needed for time series prediction problems. It has the following key elements:

- non–linearity
- predicting regime switching (gate)
- local predictors for each regime (experts)

We want to find the answers to the following questions:

- How well is the gated experts network suited for time series prediction?
- In what way does the gate split the input space and discoveres regimes in a time series?
- How can the free parameters of the gated experts network be determined?
- How can the gated experts network be used for analyzing time series dynamics?

Thus, our goal is to gain insight in the application of the gated experts network to real–world time series prediction. This will be achieved by studying, implementing and testing this network.

First, the basic concepts of time series, artificial neural networks and time series prediction are introduced (Chapter 2). Next, we discuss modular neural networks, found in literature, applied to (real–world) time series prediction (Chapter 3). Then the gated experts network is presented, where the architecture and learning, based on the Expectation Maximization (EM) algorithm, are described in detail (Chapter 4). We continue with the implementation of the gated experts network in the general InterAct© simulation environment (Chapter 5). Next, an overview of the experiments performed with the implementation of the gated experts network is presented (Chapter 6). Finally, our conclusions and directions for further research on time series prediction with gated experts networks are presented (Chapter 7).

Chapter 2

Basic concepts

In this chapter we present the necessary theoretical background of time series analysis, artificial neural networks and time series prediction. For a more extensive study, we refer for time series analysis to Chatfield [1] and for neural networks to Haykin [4]. At the end of this chapter, an illustration of real-world time series prediction using artificial neural networks is presented.

2.1 Time Series

A time series $(X_n, n = 1, 2, \ldots)$ is a collection of observations made sequentially in time. The observations are the outcomes of a process which changes in time under the influence of an unknown stochastical mechanism. Here, *discrete* time series are considered, where the observations are taken at equally sized intervals. Although most real-world time series are continuous, they can be made discrete by a process called *sampling*. An example of a discrete time series is given in Figure 2.1.

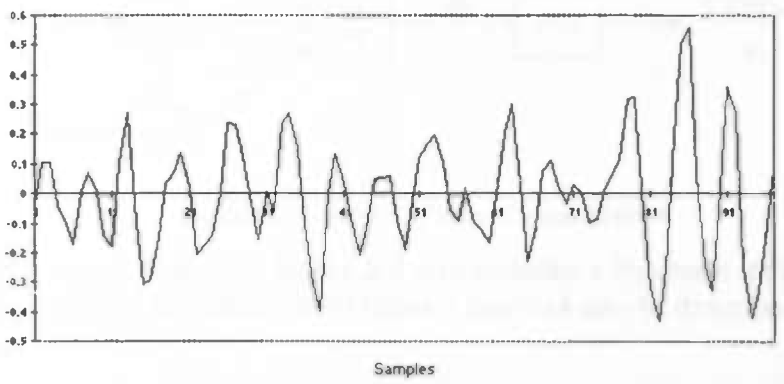

Figure 2.1: *Example of a discrete time series*

Most real-world time series are *stochastic*, in that the future is only partly determined by past values. Exact predictions of these stochastical time series are impossible. To handle stochastical time series we assume the future values to have a probability distribution which is conditioned on knowledge of the past.

A time series is said to be *stationary* if there is no systematic change in mean (there is no trend), if there is no systematic change in variance, and if strictly periodic variations have been removed. Again, real-world time series are mostly *non-stationary*. An important subclass of non-stationary processes are piecewise stationary processes, where the series switches between different stationary regimes. An example of a real-world piecewise stationary time series is presented in section 2.3.

An important issue in time series analysis is the prediction (also called forecasting) of the series, given its past values and possibly additional features. By using more past values or additional features, knowledge of the time series increases. Therefore, a predictor can take into account more knowledge for predicting future values of the series. Univariate methods for prediction only depend on past values, whereas multivariate methods depend on values of one or more additional series or additional features.

To gain some insight in the linear dependence between the values of the two time series {x(1),x(2),..,x(N)} and {y(1),y(2),..,y(N)} at different instances of time, the *cross-correlation function* can be computed (2-1).

$$\rho_{x,y}(\tau) = \frac{\sum_{i=1}^{N-\tau}(x(i) - \bar{x})(y(i-\tau) - \bar{y})}{\sqrt{\sum_{i=1}^{N-\tau}(x(i) - \bar{x})^2 \sum_{i=1}^{N-\tau}(y(i-\tau) - \bar{y})^2}} \tag{2-1}$$

The shifting term τ can be varied to extract information on how the time series {x(1),x(2),..,x(N)} depends linearly on past values (if $\tau > 0$) of the time series {y(1),y(2),..,y(N)}. The number of values in the series is denoted by *N*. By calculating $\rho_{x,x}(\tau)$ the *autocorrelation function* is obtained. This function gives information on the linear dependence of a time series on its own past values.

By examining a time series in the frequency domain rather then in the time domain, periodicities can be found. These periodicities are important as they indicate the period for which it obtains more or less the same values. By the *discrete Fourier transform*, the frequencies of these periods and their amplitudes can be obtained (with Δt as the sampling time).

$$F[f\Delta f] = \sum_{n=0}^{N-1} x[n]e^{[\frac{-i2\pi fn}{N}]}, with \Delta f = \frac{1}{N\Delta t} \tag{2-2}$$

The frequencies $f\Delta f$ for which $arg(F[f\Delta f])$ is relatively large, are suitable candidates for the existing periods in the series.

For the prediction of a time series, a model is created from the real underlying process. Practicaly, this model always remains an approximation of this process. The question here is, how well the model is suited for the specific prediction problem. Many mathematical models have been developed. However, these models often assume a priori that the time series was generated by a linear process, that is the present value of the series depends linearly on its predecessors. In a real–world time series, it often is an impossible task to extract the underlying mathematical model. Another approach is to build a model from observed data by a process of learning from examples. Such an approach can be realized by an artificial neural network (ANN), described in section 2.2.

2.2 Artificial neural networks

An artificial neural network (ANN) can be viewed as a machine that is designed to model the way in which the human brain performs a particular task. An important class of ANNs perform useful computations through a learning process. To achieve good performance, ANNs employ a massive interconnection of simple processing units referred to as *neurons*. Neurons are the building blocks of ANNs and the way in which they are connected determines the architecture. To store knowledge into the ANN, a *learning process* is used.

2.2.1 The Artificial Neuron

A *neuron* can be viewed as a simple information–processing unit. It calculates the sum of its weighted input signals and transforms this sum to the output by means of a *activation function*. Figure 2.2 shows the model of a neuron. The model contains three basic elements:

- A set of *connections*, which are characterized by their weights. This weight is multiplied by the input signal. These connections are considered as uni–directional connections, or *feedforward* connections.
- A *summing* unit for summing the weighted input signals.
- An *activation function* $\varphi(.)$ for transforming the sum into the output of the neuron. In most cases, the output range of a neuron equals [0,1] or [−1,1].

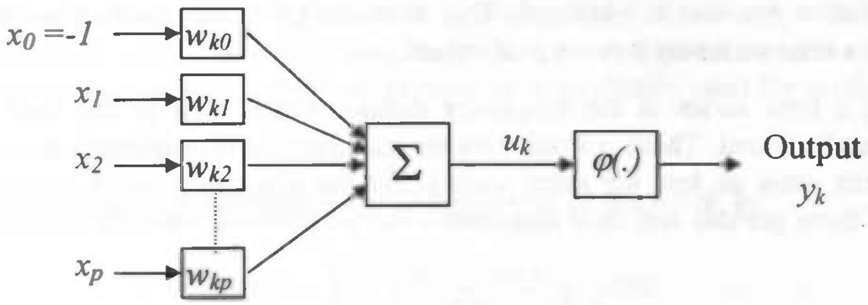

Figure 2.2: *Model of an artificial neuron*

The model of a neuron shown in Figure 2.2 also includes a *bias* term indicated by an input signal x_0 and weight w_{k0}. In mathematical terms, a neuron k may be described by the following two equations

$$u_k = \sum_{j=0}^{p} w_{kj} x_j \tag{2-3}$$

and

$$y = \varphi(u_k) \qquad (2\text{–}4)$$

where $x_1, x_2, \dots, x_p$ are the input signals and $w_{k1}, w_{k2}, \dots, w_{kp}$ are the connection weights of neuron k. Two functions which can be used as activation functions, are:

- The *sigmoid* activation function, which is the most common form of activation function used in ANNs

  $$\varphi(u) = \frac{1}{1 + \exp(-au)} \qquad (2\text{–}5)$$

  where a is the slope parameter. This activation function has a range of [0,1].
- The *hyperbolic tangent* activation function with a range of [−1,1]

  $$\varphi(u) = \tanh(\tfrac{u}{2}) \qquad (2\text{–}6)$$

These two activation functions, (2–5) and (2–6), are non–linear. By using a non–linear activation function we introduce non–linearity in the ANN.

2.2.2 Architecture

As noted, the manner in which the neurons of a neural network are connected, determines the architecture. In this subsection the *multilayer feedforward networks* will be viewed, as illustrated in Figure 2.3.

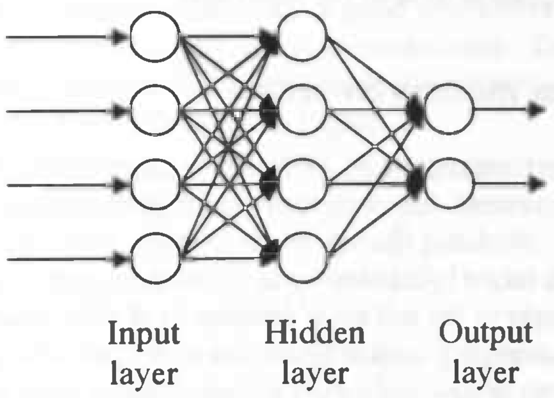

Figure 2.3: *Example of a multilayer feedforward network*

The first layer of neurons is the *input layer*, where the network receives information from the outside world or environment. The second layer is called a *hidden layer*. A multilayer feedforward network contains one or more hidden layers. The last layer is the *output layer*. The number of neurons in each layer can differ. Information flows from the input layer, through the hidden layer(s) to the output layer; hence the term feedforward. Basically, a neural network performs a functional input–output mapping. By presenting the neural network a large set of input–output examples, it learns this mapping by adjusting the weights of the connections. The next section descibes this learning process in more detail.

2.2.3 The learning process

The learning process changes the free parameters of the network. In the model of the artificial neuron (section 2.2.1), these free parameters are the connection weights (including the bias). These changes should eventually increase the performance of the ANN. The learning algorithm steps through the following sequence of events.

The ANN:

- is *stimulated* by the environment, by applying a pattern to the input layer
- undergoes *changes* as a result of this stimulation
- *responds in a new way* to the environment, because of the changes that have occurred in its internal structure

More specifically, an input vector x is presented to the network, together with a desired response vector **d**. The network responds with an output vector y which, typically, is different from the desired response. After calculating the *error* (**d**−**x**) the ANN updates the connection weights to reduce the error. This procedure is repeated for all layers. After the weights are updated the ANN is presented with the next pair of vectors. This type of learning is a form of *supervised* learning, because a *teacher* oversees the learning process by presenting the ANN with a desired response. The most popular learning process used in the neural network community is the *error back–propagation* learning algorithm. An ANN which uses this learning algorithm to adapt its parameters, is called a *Multilayer Perceptron* (MLP). Another type of learning is called *unsupervised* learning, where no teacher is present. In this case, the ANN itself develops a representation of the input data.

2.2.4 Performance measurements

In order to evaluate the performance of an ANN and to compare the performance of different ANNs, this subsection presents the commonly used performance measurements. These measurements must be calculated during the test–cycle (after training) on a test set of input–target patterns. The target is denoted by *d(n)* and the response (output) of the ANN by *y(n)*. The number of patterns in the test set is denoted by *N*. The *mean relative error* and the *mean squared error* are commonly used in neural network applications, the *normalized mean squared error* and the *ratio of squared errors* are specifically used for prediction tasks.

Mean Relative Error (MRE)

The MRE calculates the relative error of the output of the ANN, by

$$MRE = \frac{1}{N}\sum_{n=1}^{N}\left|\frac{y(n) - d(n)}{d(n)}\right| \times 100\% \qquad (2\text{–}7)$$

Mean Squared Error (MSE)

The MSE calculates the average squared error of the output of the ANN, by

$$MSE = \frac{1}{N}\sum_{n=1}^{N}(d(n) - y(n))^2 \qquad (2\text{–}8)$$

Normalized Mean Squared Error (NMSE)

The NMSE is the ratio of mean squared errors of the prediction method represented by an ANN and the method which predicts the mean at every time step, as

$$NMSE = \frac{\sum_{n=1}^{N} (d(n) - y(n))^2}{\sum_{n=1}^{N} (d(n) - \bar{d})^2} \qquad (2\text{–}9)$$

Ratio of Squared Errors (RSE)

The RSE is the ratio of mean squared errors of the prediction method by an ANN and the method which uses the last value as the prediction of the next value.

$$RSE = \frac{\sum_{n=2}^{N} (d(n) - y(n))^2}{\sum_{n=2}^{N} (d(n) - d(n-1))^2} \qquad (2\text{–}10)$$

2.3 Waste–water purification

In the field of waste–water purification, one wants to know the ammonia concentration which is measured every quarter of an hour. The measurements of the ammonia concentration are very expensive, therefore a good alternative would be to predict the ammonia concentration from other (cheaper) measurements. These other values are also measured every quarter of an hour and are related to the ammonia concentration. One week of the ammonia concentration is depicted in Figure 2.4.

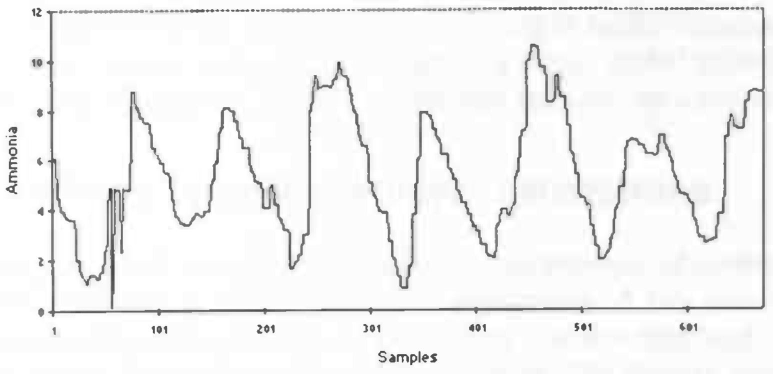

Figure 2.4: *The ammonia concentration of one week*

We see the ammonia concentration as a piecewise stationary time series. It is periodic on a daily and a weekly scale. The series is said to switch between different regimes.

After an extensive analysis of the data, an ANN is build as the prediction model. Specifically, a multilayer perceptron is used as the architecture with error back–propagation learning. The goal was that 95% of the predictions had a relative error of 10% or less. After training the ANN with different input configurations, a maximum of 25% of the predictions with a relative error of 10% or less was reached [14].

One of the main reasons why the ANN predicts the ammonia concentration poorly, is that it builds a global model, whereas the time series is piecewise stationary. A better approach would be to build several local predictors for the different regimes. Separate ANNs for the prediction of the ammonia concentration on the weekly, daily and hourly scale will have the easier task of modelling less complicated time series. Combining these separate predictions yields the overall prediction of the complete series.

2.4 Discussion

In this chapter we have introduced basic concepts of predicting a time series. In particular, we introduced time series, neural networks and time series prediction. The main point discussed here, is the usage of ANNs for prediction problems. However, the performance of a standard ANN applied to piecewise stationary time series is not adequate. Therefore, the need for applying several predictors to the piecewise stationary series arises.

Chapter 3

Overview modular neural networks

This chapter gives a literature overview of modular neural networks applied to (real-world)time series prediction. Instead of using one global ANN, separate ANNs can be put on different regimes of the time series. Here, several methods used to exploit this principle are described.

3.1 Modular neural networks

Conventional time series models are global models which are appropriate for stationary dynamics. Real-world time series lack the assumption of stationarity. A class of non-stationary time series are the stationarity by parts or multi-stationary time series, that switches between regimes. It is often difficult for a single global model (e.g. the MLP) to learn the switching of regimes. Thus, we motivated the need for a switch predictor.

3.1.1 Combining several multilayer perceptrons

Instead of using one MLP, this paragraph describes the principle of combining several MLPs for a prediction problem in an electric utility application. A key component of the daily operation and planning activities of an electric utility is short-term load forecasting, e.g. the prediction of hourly loads (demand) for the next hour. The electric load has complex and non-linear relationships with several factors such as climatic conditions, past usage patterns, the day of the week and the time of the day. The nature of the load forecasting problem lends itself well to the neural network technology since they can model these complex relationships in the data trough a process of learning. The accuracy of such forecasts has significant economic impact for the utility.

A load forecasting system is described in 1997 by Khotanzad *et al.* [12] known as ANNSTLF (artificial neural network short term load forecaster) which has received wide acceptance by the electric utility industry and presently is being used by 32 utilities across the USA and Canada. The ANNSTLF takes into account the effect of the temperature and the relative humidity on the load. Besides its load forecasting engine, the ANNSTLF contains forecast modules which can generate the hourly temperature and the relative humidity forecasts. The ANNSTLF is based on a multiple neural network strategy that captures various trends in the data. Two generations of ANNSTLF's where developed.

The *first generation* of the ANNSTLF consists of 38 different MLP networks grouped into three modules. These three modules are designed to model weekly, daily and hourly trends of the load. Each module generates a load forecast independent of the other two modules. These three forecasts are adaptively combined to obtain the final forecast. This first generation engine was implemented at more than 20 electric utilities with a satisfactory performance, but the large size of MLP networks and the redundancy of input data used by the various modules gave room for improvement.

The *second generation* engine of the ANNSTLF consists of 24 small size MLP networks (one MLP per hour). Another strategy is now employed; the distinction between forecast indicators for various hours of the day are divided into three categories: past loads, past weather and the forecast weather for the coming day. Some of these indicator variables have a significant impact on the future load of some hours and little effect on other hours. Based on these observations, the hours of the day are divided into four categories: early morning (hours 1 to 9), mid–morning or early afternoon and early night (hours 10 to 14 and 19 to 22), afternoon peak (hours 15 to 18) and late night hours (hours 23 and 24). Holidays are treated like a Saturday or a Sunday by flagging a holiday during on–line operation. This strategy is used because 'holiday' data is relatively sparse in real world historical data.

The electric utility industry considers a forecast accuracy below a mean relative error of 3.0% as quite good. The performance of the second generation engine equals 2.19% for a one day ahead forecast and 3.67% for a seven days ahead forecast. In all cases, the second generation engine outperforms the first generation engine. The application of multiple neural networks to predict a real world time series as done with the ANNSTLF model is a clear motivation for the use of modular neural networks.

3.1.2 Predictive modular neural networks

Kehagias and Petridis [11] introduced in 1994 the Predictive Modular Neural Networks (PREMONN) architecture for time series classification. The PREMONN has a hierarchical structure. The bottom level consists of a bank of linear or non–linear predictor modules. The top level is a decision module that employs (Bayesian) posterior probabilistic or non–probabilistic decision rules. Source switching has to be done in the posterior update rule. Furthermore, off–line training of the predictor modules can be applied, because in some problems the time series sources are known. For problems with unknown sources, an initial module can be trained and the next incoming data can be used for computing the module's posterior probability. If this is high it is used to update the module parameter estimates, otherwise this data is placed in a separate data–pool. Later on, this data–pool is used to train a second module. The incoming data are tested against both modules and if they do not fit either one they are set aside to train a third module etc.

PREMONNs can be applied just as well to classification of stochastic or deterministic time series. Furthermore, PREMONNs can be applied to prediction or classification problems. PREMONNs can also operate online. Convergence to correct classification was proven for various choices of prediction and decision modules. The learning time scales linearly with the number of sources to be learned. Furthermore, the PREMONN is robust to noise.

3.1.3 Mixture of experts

Jordan and Jacobs [7] described in 1996 the modular and hierarchical learning systems with the emphasis on the probabilistic framework. Modular systems allow complex learning problems to be solved by the divide and conquer principle. The focus is on supervised learning, where modular architectures arise when the assumption is made that the data can be described by a collection of functions each of which is defined over a relatively local region of the input space. A modular architecture can model data by allocating different models to different regions of the input space. Modular systems present an interesting credit assignment problem where the learner has no prior knowledge of how to partition the input space. Prior knowledge would implicate certain data 'labels' specifying how to allocate modules to data points.

The EM–algorithm

The EM–algorithm (Dempster *et al.* [2]) solves the credit assignment problem by computing the posterior probabilities that can be thought as estimates of the missing data 'labels'. Jordan and Xu [10] presented in 1995 a theoretical analysis of the EM algorithm for mixtures of experts and hierarchical mixtures of experts, i.e. an iterative approach to maximum likelihood parameter estimation (see section 4.5). The linear convergence of the algorithm can be calculated by an explicit expression for the convergence rate. They also described an acceleration technique that yields a significant speedup in simulation experiments.

3.1.4 Gated experts network

A class of models was presented in 1995 by Weigend *et al.* [16], which are called gated experts networks (GEN). GENs refer to non–linearly gated non–linear experts, first introduced into the neural community in 1991 by Jacobs *et al.* [8] as mixture of experts. The gate can split input space non–linear and the sub–processes can be non–linear through the hidden units of the expert networks. The basic idea behind gated experts is that rather than using a single global model, several local models (the experts) are learned from the data. At the same time, the input space is learned to be split (by the gate). In advance, the splitting of the input space is unknown, because the only information available is the next value of the time series. A blending of supervised and unsupervised learning addresses this issue. The supervised component (expert) learns to predict the next observed value. The unsupervised component (gate) covers the discovery of hidden regimes.

Key elements of the GEN are: non–linear gate and experts, soft–partitioning the input space and adaptive noise levels (variances) of the experts. The noise level parameter of each individual expert is allowed to adapt separately to the data. The expert–specific variances are important for two reasons. The first reason is to facilitate the segmentation, the second to prevent over fitting. The experiments gave positive results in contrast with single networks in three areas, namely (better) prediction, analysis (discovery of hidden regimes) and (less) overfitting. Furthermore, three remarks on improving the performance of the GEN are given: improving generalization through priors on the variances, gating outputs and the weights, improving segmentation through annealing and improving learning through second–order methods.

3.2 Hierarchical modular neural networks

It is noted that it is also possible to build hierarchy into the architecture of a modular network. By extending the divide and conquer principle to the separate modules themselves, hierarchy enters the model. Hierarchical models arise when the assumption is made that the data is well described by a multi–resolution model, a model in which regions are divided recursively into sub–regions.

3.2.1 Hierarchical mixture of experts

Jordan and Jacobs [6] presented in 1994 a tree–structured architecture for supervised learning based on the divide and conquer principle called the hierarchical mixture of experts (HME). The HME architecture is a tree with the linear gating networks at the nonterminals of the tree and the generalized linear expert networks at the leaves of the tree. The outputs of the experts proceed up the tree, being adjusted by the gating network outputs. The statistical model used here, is thus a hierarchical mixture model in which both the mixture coefficients and the mixture components are generalized linear models. Divide and conquer algorithms (splitting a problem into simpler problems and combining the solutions to yield a solution to the complex problem) have convergence times orders of magnitude faster than gradient based neural network algorithms. However, they are also generally variance increasing, therefore a second variance decreasing–device is used: 'soft'–splitting of the input space in stead of 'hard' splits. Learning is accomplished by using the EM–algorithm applied to the HME as a powerful and efficient tool for estimating the network parameters. They also describe an *on–linelearning algorithm* for incremental parameter updating. This on–line algorithm is a stochastic approximation to a Newton–Raphson method rather than a gradient method.

3.2.2 Adaptive hierarchical mixture of experts

Fritsch *et al.* [3] proposed in 1996 an approach to automatically growing and pruning of the HME. A constructive algorithm enables large hierarchies (consisting of several hundreds of experts) to be trained effectively. An evaluation criterion is used to score the experts performance on the training data by splitting the worst expert into a new subtree and copying the weights provided by additional small random permutations (this can be compared to genetic algorithms, authors) to overcome the errors made by this expert. HMEs trained by the automatic growing procedure yield better results than traditional statistic and balanced hierarchies. Pruning bad performing subtrees helps prevent instabilities and singularities in the parameter updates. Specifically, the algorithm allows the HME to use the resources (experts) more efficiently than a standard pre–determined HME architecture. The tree growing algorithm leads to better classification performance (the splitting of input space or continuous density estimators, authors) compared to standard HMEs with an equal number of parameters.

3.3 Discussion

In this chapter we introduced modularity and hierarchy as two important principles in the design of neural networks applied to time series prediction. The reason we selected the GEN, are the nice properties of non–linear gate and experts, soft–partitioning the input space and adaptive noise levels (variances) of the experts. Further research could concentrate on extending this structure to some kind of hierarchy, where the complexity of a process is captured by the divide and conquer principle.

Chapter 4

Theory of the gated experts network

This chapter describes the architecture and the learning algorithm used to update the parameters of the gated experts network and it is based on the article written in 1995 by Weigend et al. [16].

4.1 Introduction

As noted before, many real–world time series are piecewise stationary, where the series switches between different regimes. The basic idea behind the gated experts network (GEN) is to learn several local models (experts) from the data instead of a single global model. Simultaneously, the gate learns to split the input space. This requires blending *supervised* and *unsupervised* learning: the supervised component (expert) learns to predict the next value of the series, and the unsupervised component (gate) discovers the regimes. Summarizing, the key elements of the GEN are:

- non–linear gate and experts
- soft–partitioning the input space
- adaptive noise levels (variances) of the experts

4.2 Architecture

The architecture of the GEN contains K expert networks and one gating network, as depicted in Figure 4.1. The expert networks and the gating network are standard MLPs (section 2.2.3) and are non–linear through the non–linear activation functions of the hidden neurons. The input vector $\mathbf{x}$ denotes the train vector of dimension p and vector $\mathbf{y}_i$ is the output vector of expert i with dimension q (the same dimension as the target vector $\mathbf{d}$). Finally, the outputs of the gating network are denoted by g_i .

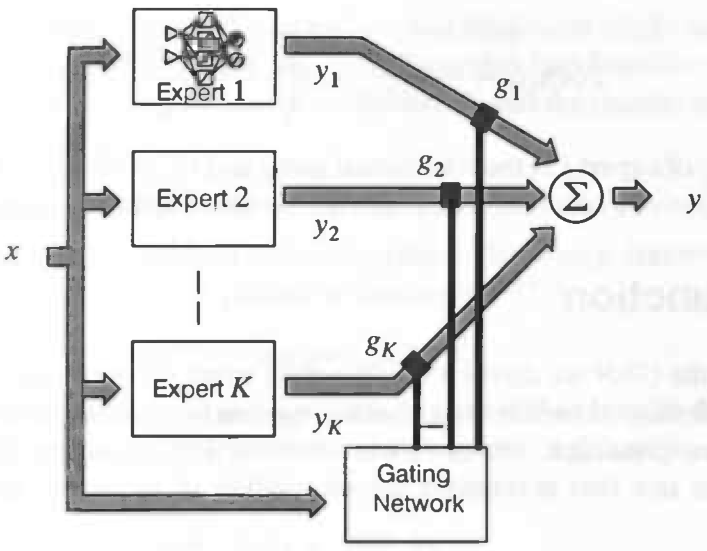

Figure 4.1: *Architecture of the Gated Experts Network*

The total output of the GEN is calculated as

$$\mathbf{y} = \sum_{i=1}^{K} g_i \mathbf{y}_i \tag{4-1}$$

4.3 Probability model

The GEN has to model the underlying probability distribution P(**d**|**x**) of the observed data. The data consists of a set of train patterns {**x**, **d**}. This set is used to adapt the free parameters (weights and variances) of the GEN. It is explicitly assumed that one and only one expert is responsible for each train pattern. The K experts can then be viewed as K ways of observing the target vector **d**, given the input vector **x**. Then, using Bayes' rule, we obtain

$$P(\mathbf{d}|\mathbf{x}) = \sum_{j=1}^{K} P(\mathbf{d}, j|\mathbf{x}) = \sum_{j=1}^{K} P(j|\mathbf{x})P(\mathbf{d}|\mathbf{x}, j) \tag{4-2}$$

The gating network has to learn the probability that a certain train vector **x** was generated by one of the experts. This probability P(j | **x**) is denoted by g_j. Since the outputs of the gating network have a probabilistic nature, they must satisfy the constraints

$$0 \le g_j \le 1 \text{ and } \sum_{j=1}^{K} g_j = 1 \tag{4-3}$$

To meet these constraints (4–3) the activation function of the gating network is defined as

$$g_j = \frac{\exp(u_j)}{\sum_{k=1}^{K} \exp(u_k)} \tag{4-4}$$

with u_i as the activation of output neuron of the gating network. This function is called the *softmax activation function*. The probability that the target vector **d** was generated by expert j given vector x, denoted by P(**d**|**x**,j), is modelled by a multivariate Gaussian density function

$$P(\mathbf{d}|\mathbf{x},j) = \frac{1}{(2\pi)^{q/2}\sigma_j^q}\exp\left(\frac{-\| \mathbf{d} - \mathrm{y}_j(\mathbf{x}) \|^2}{2\sigma_j^2}\right) \tag{4-5}$$

with the output y_j of expert j as the conditional mean and σ^2_j as its variance. This probability model is used to derive a cost function, which we use to adapt the free parameters of the GEN.

4.4 Cost function

In order to train the GEN we derive a cost function using the *maximum likelihood* method. The maximum likelihood method is a classical parameter estimation procedure that views the parameters as quantities, whose values are fixed but unknown. The best estimate is defined to be the one that maximizes the probability of obtaining the samples actually observed.

Using (4–4) and (4–5) and assuming the statistical independence of measurements of each pattern, the product over the likelihoods of the individual patterns indicated by index n are taken to obtain the likelihood function

$$L = \prod_{n=1}^{N}\sum_{j=1}^{K} g_j(\mathbf{x}(n), w_g)\frac{1}{(2\pi)^{q/2}\sigma_j^q}\exp\left(\frac{-\| \mathbf{d}(n) - \mathrm{y}_j(\mathbf{x}(n), w_j) \|^2}{2\sigma_j^2}\right) \tag{4-6}$$

where the parameters of the GEN are denoted by w_g (weight of the gate), w_j (weight of expert j) and σ^2_j (variance of expert j).The cost function C is the logarithm of the likelihood function

$$C = \ln L = \sum_{n=1}^{N}\ln\sum_{j=1}^{K} g_j(\mathbf{x}(n), w_g)\frac{1}{(2\pi)^{q/2}\sigma_j^q}\exp\left(\frac{-\| \mathbf{d}(n) - \mathrm{y}_j(\mathbf{x}(n), w_j) \|^2}{2\sigma_j^2}\right) \tag{4-7}$$

This cost function can be maximized using gradient ascent, but it turns out to be quite hard to learn both the maps of the experts and the splits of the input space through the gating network [6]. Therefore, a different approach will be used; the EM algorithm, as described below.

4.5 The EM-algorithm

This section describes the *Expectation–Maximization algorithm* (EM–algorithm) and its application to the gated experts architecture.

The *EM–algorithm* is an iterative technique for maximum likelihood estimation. Each iteration of an EM–algorithm is composed of two steps: an *Estimation* step (E step) and a *Maximization* step (M step). The M step involves the maximization of a likelihood function that is redefined in each iteration by the E step. An application of the EM–algorithm begins with the observation that the optimization of the likelihood function $l(\theta;X)$ would be simplified if a set of additional variables, called "missing" or "hidden" variables, were known. In this context, we refer to the observed data set X as "incomplete" and posit a "complete" data set Y that includes the missing set of variables Z. Let $l_c(\theta;Y)$ denote the log likelihood of the complete–data set Y and the original likelihood $l(\theta;X)$ as the likelihood of the incomplete likelihood of data set X. The EM–algorithm first finds the expected value of the complete–data likelihood, given the observed data X and the current model (denoted by θ). This constitutes the *E step*

$$Q(\theta, \theta^{(p)}) = E\left[l_c(\theta; Y)|X\right] \tag{4–8}$$

where p denotes the iteration number of the algorithm. The *M step* maximizes this function Q with respect to θ to find the new parameter estimates $\theta^{(p+1)}$

$$\theta^{(p+1)} = \arg\max_{\theta} Q(\theta, \theta^{(p)}) \tag{4–9}$$

The E step is then repeated to yield an improved estimate of the complete likelihood and the process iterates. Dempster *et al.* [2] proved that an increase in Q implies an increase in the complete likelihood

$$l(\theta^{(p+1)};X) \geq l(\theta^{(p)};X) \tag{4–10}$$

The likelihood l increases monotonically along the sequence of parameter estimates generated by an EM–algorithm. In practice this implies convergence to a local maximum.

4.5.1 Applying EM to the gated experts architecture

The EM–algorithm is based on the assumption that some variables are 'missing'. The missing variables in our model are the probabilities that a given pattern n is generated by expert j. So we introduce an *'indicator variable'* , with the following properties:

- $I_j\,(n) = 1$, if pattern n is generated by expert j
- $I_j(n) = 0$, otherwise

The set of random indicator variables $Z = \{I_j(n);\ j=1,..,K;\ n=1,..,N\}$ constitute the missing data.

4.5.2 The E step

The likelihood of the 'complete data'can now be written as

$$L(D, Z|X, \Theta) = \prod_{n=1}^{N}\prod_{j=1}^{K}\left[g_j(\mathbf{x}(n), w_g)\, P(\mathbf{d}(n)|\mathbf{x}(n), w_j)\right]^{I_j(n)} \tag{4–11}$$

where X and D denote the input and target trainings data, Z the unobserved or missing data and Q the ensemble of parameters $w_g, w_1, w_2,.., w_K, \sigma_1, \sigma_2, .., \sigma_K$. The indicator variables $I_j(n)$ are unknown, hence they need to be estimated. Therefore, we replace the indicator variables by their expected values

$$E\left[I_j(n)|X, D, \Theta\right] = 1 \cdot P\left(I_j = 1|X, D, \Theta\right) + 0 \cdot P\left(I_j = 0|X, D, \Theta\right) = P(j|\mathbf{x}(n), \mathbf{d}(n)) \tag{4–12}$$

Thus, the expectation of $I_j(n)$ is the a posterior probability that the expert j generated the current training pattern n if the input, output and target vectors are known. This posterior probability is denoted by $h_j(n)$ (see Appendix A)

$$E(I_j(n)|X,D,\Theta) = h_j(n) = \frac{g_j(x(n),w_g)\frac{1}{(2\pi)^{q/2}\sigma_j^q}\exp\left(\frac{-\|d(n)-y_j(x(n),w_g)\|^2}{2\sigma_j^2}\right)}{\sum_{k=1}^{K} g_k(x(n),w_g)\frac{1}{(2\pi)^{q/2}\sigma_k^q}\exp\left(\frac{-\|d(n)-y_k(x(n),w_g)\|^2}{2\sigma_k^2}\right)} \tag{4-13}$$

and will be used in the M step.

4.5.3 The M step

Taking the expectation of the logarithm of $L(D,Z|X,\Theta)$, and replacing each I_j by its expected value h_j yields the cost function that includes the assumption of the missing values (see Appendix A)

$$\begin{aligned} C_M &\equiv E[\ln L(D,Z|X,\Theta)] \qquad (4\text{-}14)\\ &= \sum_{n=1}^{N}\sum_{j=1}^{K} h_j(n)\ln\left[g_j(x(n),w_g)\right] - \\ &\quad \frac{1}{2}\sum_{n=1}^{N}\sum_{j=1}^{K} h_j(n)\left(\frac{-\| d(n) - y_j(x(n),w_j)\|^2}{\sigma_j^2} + q\ln(2\pi\sigma_j^2)\right)\end{aligned}$$

By maximizing the cost function C_M, the update rules for the variances for the experts as well as the weights of the experts and the gate can be derived.

4.6 Adapting the gate and experts

In this section we derive the update rules for the GEN. The adaptation of the weights of both the experts and the gate is performed with error back–propagation update rules. To simplify the mathematics used in this section the parameters w_j and w_g are omitted.

4.6.1 The updates for the expert variances

The update for the variances can be computed by setting the partial derivative to zero ($\partial C_M / \partial\sigma_j^2=0$). The variance of expert k that maximizes the cost function is given by (see Appendix A)

$$\sigma_k^2 = \frac{\sum_{n=1}^{N} h_k(n)\ \|\mathbf{d}(n) - \mathbf{y}_k(\mathrm{n})\|^2}{q\sum_{n=1}^{N} h_k(n)} \tag{4-15}$$

Thus, the variance of expert k is the weighted average of the squared errors.

4.6.2 The updates for the expert weights

The weights of the expert networks are updated using the error back–propagation algorithm. In the error back–propagation algorithm, the cost function C_M has to be maximized. The structure of the expert network k is depicted in Figure 4.2.

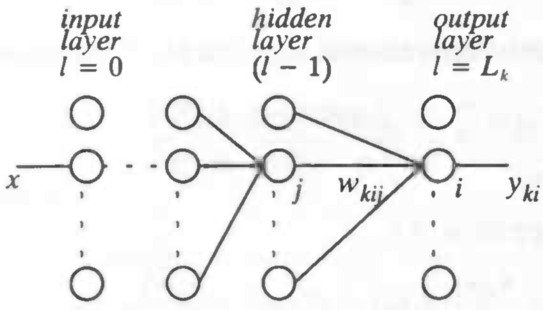

Figure 4.2: *The structure of expert k*

Some notations:

- l : layer index
- L_k : total number of layers (the output layer is layer L_k and the input layer is layer 0)
- q : the number of elements of the target vector **d**
- $\#(l)$: the number of neurons in layer l
- $v^{(l)}_{ki}(n)$: the total input of neuron i in layer l for pattern n
- $y^{(l)}_{ki}(n)$: the output of neuron i in layer l for pattern n
- φ_{ki} : the activation function for neuron i
- $w^{(l)}_{kij}$: the weight from neuron j to neuron i
- η_1 : learn speed of the experts

All neurons in the hidden layers and the output layer receive input from a bias (as described in section 2.2.1). Each neuron i has a corresponding weight $w^{(l)}_{ki0}$. The bias always has the index zero.

The total input of neuron i in layer l is the weighted sum of all the outputs from the previous layer $(l-1)$

$$v^{(l)}_{ki}(n) = \sum_{j=0}^{\#(l-1)} w^{(l)}_{kij}(n) y^{(l-1)}_{kj}(n) \qquad (4\text{–}16)$$

The output of neuron i in layer l is calculated by

$$y^{(l)}_{ki}(n) = \varphi_{ki}(v^{(l)}_{ki}(n)) \qquad (4\text{–}17)$$

To derive the update rule for the weight $w^{(l)}_{kij}$, the error signal for output neuron i in layer $l=L$ will be derived first, where $d_i(n)$ denotes the i–th element of the target vector **d**.

$$e_{ki}(n) = d_i(n) - y_{ki}^{(l)}(n) \tag{4-18}$$

The weight change is calculated by

$$\Delta w_{kij}^{(l)} = \eta_1 \frac{\partial C_M(n)}{\partial w_{kij}^{(l)}(n)} \tag{4-19}$$

Using the chain rule, the partial derivation in equation (4–19) is expressed as

$$\frac{\partial C_M(n)}{\partial w_{kij}^{(l)}(n)} = \frac{\partial C_M(n)}{\partial y_{ki}^{(l)}(n)} \cdot \frac{\partial y_{ki}^{(l)}(n)}{\partial v_{ki}^{(l)}(n)} \cdot \frac{\partial v_{ki}^{(l)}(n)}{\partial w_{kij}^{(l)}(n)} \tag{4-20}$$

The first term yields (see Appendix A)

$$\frac{\partial C_M(n)}{\partial y_{ki}^{(l)}(n)} = \frac{h_k(n)}{\sigma_k^2}\left[d_{ki}(n) - y_{ki}^{(l)}(n)\right] = \frac{h_k(n)}{\sigma_k^2} e_{ki}(n) \tag{4-21}$$

and the second term

$$\frac{\partial y_{ki}^{(l)}(n)}{\partial v_{ki}^{(l)}(n)} = \varphi'_{ki}(v_{ki}^{(l)}(n)) \tag{4-22}$$

and the third term

$$\frac{\partial v_{ki}^{(l)}(n)}{\partial w_{kij}^{(l)}(n)} = y_{kj}^{(l-1)}(n) \tag{4-23}$$

Combining the equations (4–21), (4–22) and (4–23) yields

$$\Delta w_{kij}^{(l)} = \eta_1 \frac{\partial C_M(n)}{\partial w_{kij}^{(l)}(n)} = \frac{\eta_1}{\sigma_k^2} h_k(n) e_{ki}(n) \varphi'_{ki}(v_{ki}^{(l)}(n)) y_{kj}^{(l-1)}(n) \tag{4-24}$$

The learning rule (4–21) adjusts the weights of expert k such that the output $\mathbf{y}_k$ moves towards the desired output $\mathbf{d}$. Three factors modulate the weight change:

- $h_k(n)$, which modulates the weight change proportional to the importance of expert k for pattern n
- $1/\sigma^2_j$, which modulates the learning according to the general noise level in the regime of expert k. If the average squared error in the regime is large, the influence of the error on the weight update is scaled down. If the regime has little noise, small differences in the error are exaggerated by dividing by a small variance
- $e_{ki}(n)$, which is the usual difference between the desired value and the output value of neuron i of expert k

The weight update rules for the output and hidden neurons are different, so these rules are defined separately. To illustrate the form of the update rules, a choice is made for the activation functions of the hidden and output neurons.

Output layer $l=L_k$:

The *sigmoid activation function* is used for the output neurons , which is defined by

$$y_{ki}^{(l)}(n) = \varphi_{ki}(v_{ki}^{(l)}(n)) = \frac{1}{1 + \exp(-v_{ki}^{(l)}(n))} \tag{4-25}$$

with its derivative

$$\varphi'_{ki}(v_{ki}^{(l)}(n)) = y_{ki}(n)\left[1 - y_{ki}^{(l)}(n)\right] \tag{4-26}$$

The local gradient for neuron *i* in the output layer is calculated by

$$\delta_{ki}^{(l)}(n) = \frac{1}{\sigma_k^2} h_k(n) e_{ki}(n) y_{ki}^{(l)}(n)\left[1 - y_{ki}^{(l)}(n)\right] \tag{4-27}$$

and the weight update rule

$$\Delta w_{kij}^{(l)}(n) = \eta_1 \delta_{ki}^{(l)}(n)\, y_{kj}^{(l-1)}(n) \tag{4-28}$$

Hidden layer $0<l<L_k$:

For the neurons in the hidden layers the *hyperbolic tangent activation function* will be used, which is defined by

$$y_{ki}^{(l)}(n) = \varphi_{ki}(v_{ki}^{(l)}(n)) = \tanh(v_{ki}^{(l)}(n)) \tag{4-29}$$

with its derivative

$$\varphi'_{ki}(v_{ki}^{(l)}(n)) = (1 - y_{ki}^{(l)}(n)^2) \tag{4-30}$$

The local gradient for neuron *i* in a hidden layer is calculated by

$$\delta_{ki}^{(l)}(n) = (1 - y_{ki}^{(l)}(n)^2) \sum_{j=0}^{\#(l+1)} \delta_{kj}^{(l+1)}(n) w_{kij}^{(l+1)}(n) \tag{4-31}$$

and the weight update rule

$$\Delta w_{kij}^{(l)}(n) = \eta_1 \delta_{ki}^{(l)}(n)\, y_{kj}^{(l-1)}(n) \tag{4-32}$$

4.6.3 The updates for the gate weights

The weights of the gating network are updated using error back-propagation. With the error back-propagation algorithm, the cost function C_M will be maximized. The structure of the gating network is depicted in the Figure 4.3.

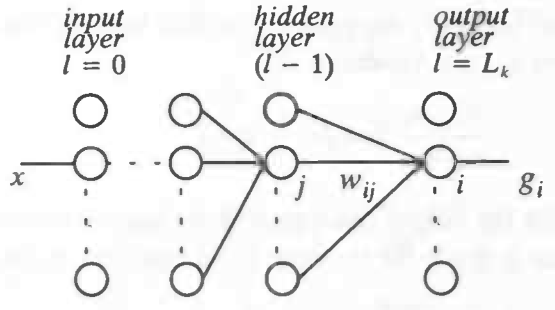

Figure 4.3: *The structure of the gate*

Some notations:

- l : layer index
- L_g : total number of layers (output layer as layer L_g and the input layer as layer 0)
- K : number of outputs of the gate (equals the number of experts)
- $\#(l)$: the number of neurons in layer l
- $u^{(l)}{}_i(n)$: the total input of neuron i in layer l for pattern n
- $y^{(l)}{}_i(n)$: the output of neuron i in layer l for pattern n
- $g_i\,(n)$: the output of neuron i in the output layer for pattern n (the a priori probability)
- $w^{(l)}{}_{ij}$: the weight from neuron j to neuron i
- η_2 : learn speed of the *gating* network

The softmax function is used as the activation function of the output neurons (for layer $l{=}L_g$)

$$g_i(n) = \frac{\exp(u_i^{(l)}(n))}{\sum_{j=1}^{K}\exp(u_j^{(l)}(n))} \tag{4-33}$$

The total input of neuron i in layer l is the weighted sum of all the outputs from the previous layer $(l-1)$

$$u_i^{(l)}(n) = \sum_{j=0}^{\#(l-1)} w_{ij}^{(l)}(n)y_j^{(l-1)}(n) \tag{4-34}$$

The weight change is calculated by

$$\Delta w_{ij}^{(l)} = \eta_2\frac{\partial C_M}{\partial w_{ij}^{(l)}(n)} \tag{4-35}$$

For neuron i in the output layer $(l{=}L_g)$ and using the chain rule, the partial derivative in equation (4–35) can be calculated by

$$\frac{\partial C_M(n)}{\partial w_{ij}^{(l)}(n)} = \frac{\partial C_M(n)}{\partial u_i^{(l)}(n)}\cdot\frac{\partial u_i^{(l)}(n)}{\partial w_{ij}^{(l)}(n)} = \frac{\partial C_M(n)}{\partial u_i^{(l)}(n)}y_j^{(l-1)}(n) \tag{4-36}$$

Observing that the a posterior probability $h_i(n)$ in the cost function C_M is calculated in the E–step, and the net input $u^{(l)}{}_i(n)$ only depends on the first term of this cost function (through $g_i(n)$) the partial derivative is (see Appendix A)

$$\frac{\partial C_M(n)}{\partial u_i^{(l)}(n)} = h_i(n) - g_i(n) \tag{4-37}$$

The weight update rules for the output neurons and the hidden neurons are to be calculated separately. Again, a choice is made for the activation function of the hidden neurons.

Output layer $l{=}L_g$:

The local gradient for neuron i in the output layer equals

$$\delta_i^{(l)}(n) = h_i(n) - g_i(n) \tag{4-38}$$

and the weight update rule

$$\Delta w_{ij}^{(l)} = \eta_2 \delta_i^{(l)}(n) y_j^{(l-1)}(n) \tag{4-39}$$

Hidden layer $0<l<L_g$:

For the neurons in the hidden layers the *hyperbolic tangent activation function* will be used, see (4–29) and (4–30). The local gradient for neuron i in a hidden layer equals

$$\delta_i^{(l)}(n) = (1 - y_i^{(l)}(n)^2) \sum_{j=0}^{\#(l+1)} \delta_j^{(l+1)}(n) w_{ij}^{(l+1)}(n) \tag{4-40}$$

and the weight update rule

$$\Delta w_{ij}^{(l)} = \eta_2 \delta_i^{(l)}(n) y_j^{(l-1)}(n) \tag{4-41}$$

4.7 Summary of the Algorithm

The algorithm for updating the weights of the experts and the gate and the variances of the experts is summarized below.

Initialization : Assign initial small values (uniformly distributed) to the weights of the different experts and the gate.

The E–step: For each data pair (x(n),**d**(n)) compute

$$y_{ki}^{(l)}(n) = \varphi_{ki}\left[\sum_{j=0}^{\#(l-1)} w_{kij}^{(l)}(n) y_{kj}^{(l-1)}(n)\right] \quad \textit{(for all layers l)} \tag{4-42}$$

$$\mathbf{y}_k(n) = \left[y_{k1}^{L_k}(n), y_{k2}^{L_k}(n), \ldots, y_{kq}^{L_k}(n)\right]^T \tag{4-43}$$

$$\mathbf{y} = \sum_{i=1}^{K} g_i \mathbf{y}_i \tag{4-44}$$

$$u_i^{(l)}(n) = \sum_{j=0}^{\#(l-1)} w_{ij}^{(l)}(n) y_j^{(l-1)}(n) \tag{4-45}$$

$$g_i(n) = \frac{\exp(u_i^{(L_g)}(n))}{\sum_{j=1}^{K} \exp(u_j^{(L_g)}(n))} \tag{4-46}$$

$$h_j(n) = \frac{g_j(n) \frac{1}{(2\pi)^{q/2}\sigma_j^q} \exp\left(\frac{-\|\mathbf{d}(n) - \mathbf{y}_j(n)\|^2}{2\sigma_j^2}\right)}{\sum_{k=1}^{K} g_k(n) \frac{1}{(2\pi)^{q/2}\sigma_k^q} \exp\left(\frac{-\|\mathbf{d}(n) - \mathbf{y}_k(n)\|^2}{2\sigma_k^2}\right)} \tag{4-47}$$

The M–step

For each data pair (x(n),d(n)) and posterior probability $h_j(n)$, update the parameters of each expert and the gate.

Adaption of the variances. For each expert k adapt the variance by

$$\sigma_k^2 = \frac{\sum_{n=1}^{N} h_k(n) \, \| \mathbf{d}(n) - \mathbf{y}_k(n) \|^2}{q \sum_{n=1}^{N} h_k(n)} \tag{4–48}$$

Adaption of the weights of the experts. For each expert k through all patterns, adapt the weights by

Output layer $l = L_k$:

$$e_{ki}(n) = d_i(n) - y_{ki}^{(L_k)}(n) \tag{4–49}$$

$$\delta_{kj}^{(l)}(n) = \frac{1}{\sigma_k^2} h_k(n) e_{ki}(n) y_{ki}^{(l)}(n) \left[1 - y_{ki}^{(l)}(n) \right] \tag{4–50}$$

$$\Delta w_{kij}^{(l)}(n) = \eta_1 \delta_{ki}^{(l)}(n) \, y_{ki}^{(l-1)}(n) \tag{4–51}$$

Hidden layer $0<l<L_k$:

$$\delta_{ki}^{(l)}(n) = (1 - y_{ki}^{(l)}(n)^2) \sum_{j=0}^{\#(l+1)} \delta_{kj}^{(l+1)}(n) w_{kij}^{(l+1)}(n) \tag{4–52}$$

$$\Delta w_{kij}^{(l)}(n) = \eta_1 \delta_{ki}^{(l)}(n) \, y_{kj}^{(l-1)}(n) \tag{4–53}$$

Adaption of the weights of the gate. For each pattern n adapt the weights by

Output layer $l=L_g$:

$$\delta_i^{(l)}(n) = h_i(n) - g_i(n) \tag{4–54}$$

$$\Delta w_{ij}^{(l)}(n) = \eta_2 \delta_i^{(l)}(n) y_j^{(l-1)}(n) \tag{4–55}$$

Hidden layer $0<l<L_g$:

$$\delta_i^{(l)}(n) = (1 - y_i^{(l)}(n)^2) \sum_{j=0}^{\#(l+1)} \delta_j^{(l+1)}(n) w_{ij}^{(l+1)}(n) \tag{4–56}$$

$$\Delta w_{ij}^{(l)}(n) = \eta_2 \delta_i^{(l)}(n) y_j^{(l-1)}(n) \tag{4–57}$$

4.8 Discussion

In this chapter we introduced the architecture of the GEN and the EM–algorithm as its learning process. The assumption that the data has a Gaussian distribution could be replaced by other distributions. The mathematical theory serves as the basis of the implementation.

Chapter 5

Implementation of the gated experts network

This chapter describes the implementation of the gated experts network in the program InterAct©. Further, we discuss some implementation problems and solutions.

5.1 The implementation environment InterAct©

InterAct©, developed at the Rijks*universiteit* Groningen, is a general simulation environment for creating, training and testing artificial neural networks. One can use this environment by writing an application program that interacts with InterAct©. The simulation environment combines an user–interface with a library. With the user–interface the computations made by the application program can be controlled and observed. The library contains routines or calls for input/output processing, creating data structures, learning rules and observations. A user can develop an application program in the language C by using these calls. Additional functionality can also be implemented by dropping in user specific routines.

5.2 General design of the gated experts network

Attempts have been made to implement the GEN as one complete structure in InterAct©. This turned out to be very problematic due to the nature of the EM–algorithm. Namely, all the posterior probabilities for all experts and patterns must be computed before learning can begin. The output neurons of the experts must have access to these posterior probabilities in order to calculate the local gradient. By trying to store all the GEN parameters in the structure itself, the clarity of the structure will decrease due to the resulting complicated structures. Too many tricks should be performed to comply to this philosophy. Therefore another direction was followed. The gate and the experts are implemented as separate structures (section 5.2.1). The EM–algorithm is implemented by using a mixture of standard InterAct© calls and solutions to application specific needs, e.g. the implementation of the softmax activation function (section 5.3.1).

5.2.1 Creating the network structure

To simplify the overall structure of the GEN, the different experts and the gate are implemented as separate networks. An implication of this strategy is that InterAct© has to switch between these networks, because only one network can be active at the same time. Unfortunately, this makes the overall performance slower. To create the separate networks for each expert and the gate, two routines are written. The experts are created by applying the following code fragment.

```
void expert_Screate_nets(
    long nr_experts,    /* number of experts to create */
    long nr_inputs,     /* number of input neurons of each expert */
    long nr_hiddens,    /* number of hidden neurons of each expert */
    long nr_outputs,    /* number of output neurons of each expert */
    net_Sid_t *ids,     /* network identification array of the expert */
    status_St *status)  /* check if the call was successful */
```

The gate is created by the following call

```
void gate_Screate_net(
    long nr_inputs,       /* number of input neurons of the gate*/
    long nr_hiddens,      /* number of hidden neurons of the gate */
    long nr_outputs,      /* number of outputs of the gate */
    net_Sid_t *id,        /* network identification of the gate */
    status_St *status)    /* check if the call was successful */
```

The next call creates the complete GEN and the posterior array.

```
void gen_Screate_net(
   pattern_Slist_id_t       list_id,             /* pattern list for GEN */
   long               nr_experts,            /* number of experts */
   long               nr_experts_hiddens,    /* number of expert hiddens */
   long               nr_gate_hiddens,       /* number of gate hiddens */
   status_St          *status )              /* check call succes */
```

5.2.2 Coding the EM-algorithm

The EM-algorithm consists of two steps which are implemented as separate calls. The first call executes a complete expectation calculation and fills the posterior array, based on the pattern list referenced by `list_id`.

```
void calc_SExpectation_step(
  pattern_Slist_id_t   list_id,          /* pattern list for testset */
  long                 nr_experts,       /* number of experts */
  status_St            *status     )     /* check call succes */
```

The second call performs the maximization step by first calculating the new variances, then updating the weights of the experts and finally updating the weights of the gate.

```
void calc_SMaximization_step(
  pattern_Slist_id_t   list_id,          /* pattern list for testset */
  long                 nr_experts,       /* number of experts */
  status_St            *status     )     /* check call succes */
```

5.3 Coding problems and solutions

This section describes several problems and their solutions encountered while implementing the GEN in InterAct©.

5.3.1 Absence of the softmax activation function

In InterAct© the softmax activation function is not yet available. It is required to set the activation function type at the creation of a neuron. For this, the sigmoid activation function is used. In order to use the softmax with these limitations, this function is implemented as an application specific function:

```
void softmax(
    long    nr_outputs, /* number of gate outputs */
    out_St  *gate_out ) /* array containing the gate output values */
```

This function is used after the evaluation of the output neurons of the gate network, instead of the standard InterAct© call `get_Soutput_group`. This function sets the value of the output neurons externally to the corresponding softmax value and returns these values in the array `*gate_out`.

5.3.2 Using sigmoids as output activation functions for the gate

Because we use sigmoids as the activation functions for the output neurons in the gate and we want to use the standard error back–propagation learning rule for these output neurons, we have a problem. After setting the output values with the softmax function, the error back–propagation learning rule multiplies the error with the derivative of the sigmoid. To overcome this faulty multiplication, the target for the gate is recalculated:

```
gate_targets[i]= (posterior[pattern_id-1][i]-gate_outputs[i]) /
(gate_outputs[i]*(1.0-gate_outputs[i])) + gate_outputs[i];
```

This will compensate for the derivative. Using a linear activation function would make this recalculation unnecessary, but this is not implemented yet.
A consequence of the recalculation of the target is, that it introduces a potential division by zero. To tackle this, a lower bound on the difference between $h_i(n)$ and g_i is introduced:

```
if( fabs((posterior[pattern_id-1][i]-gate_outputs[i]))>0.0001 )
    gate_targets[i]= (posterior[pattern_id-1][i]-gate_outputs[i]) /
    (gate_outputs[i]*(1.0-gate_outputs[i])) + gate_outputs[i];
else
    gate_targets[i]=posterior[pattern_id-1][i];
```

5.3.3 Limiting the weighted sum of the gate outputs

In the softmax activation function the weighted sum is exponentiated. To avoid numerical instability, i.e. maximum overflow, the maximum of this weighted sum is limited to [–80, 80]. Using these values for the limit, encountered maximum overflow errors were resolved.

```
get_Sstatus_neuron( neuron1, &ns, &s );
if( ns.sum <-80.0) ns.sum=-80;
if( ns.sum >80.0) ns.sum=80;
gate_out[i] = ns.sum;
sum+=exp(gate_out[i]);
```

5.3.4 Learning of the experts

As described in chapter 4, the local gradients of the output neurons of the experts include the posterior probability. Since the standard error–back propagation learning rule is used, the next code fragment is used to include these terms:

```
set_Sinput( inputs, nr_inputs, &s );
calc_Sstart_evalu( 0L, hidden_Slist, &s);
calc_Sstart_evalu( 0L, output_Slist, &s );
set_Starget( targets, nr_targets, &s );
change_Slearn( error_back_St, learn_rate*
posterior[pattern_id-1][expert_nr], momentum, 0.0, 0.0, 0.0, &s );
calc_Sstart_learn( 0L, output_Slist, &s );
change_Slearn( error_back_St,learn_rate, momentum, 0.0, 0.0, 0.0, &s );
calc_Sstart_learn( 0L, hidden_Slist, &s );
```

5.3.5 The influence of scaling on the variance

The target values of the experts are scalled to a range of [0.1, 0.9] to confine the target values to the linear area of the sigmoid function. The variances of the expert are not correct due to this scaling. To overcome this problem, linear activation functions were used for the output neurons, but the performance of the GEN decreased. Therefore this option was omitted. Instead, the variance can be scaled back to its correct value.

5.4 Discussion

In this chapter we described the implementation of the GEN in InterAct$^{©}$. For a detailed description of the complete source code of the GEN, the reader is referred to Appendix B.

Chapter 6

Experiments

This chapter presents an overview of the experiments performed with the implementation of the GEN. In the gated experts architecture, the values of several parameters are chosen before conducting the experiments. The parameters are: the number of expert networks, the number of hiddens for the expert networks and the gate, the learning rate and learn momentum of the expert networks and the gate. Examples are presented and the outcomes of the experiments are studied and evaluated.

6.1 Introduction

The experiments are conducted by using the implementation of the GEN. The GEN consists of a gate and a certain number of experts (depending on the experiment). The gate and experts consist of an input and an output layer. The use of a hidden layer and the number of hidden neurons depend on the experiments. In general a time series data–file contains *N* patterns and is split in an equally sized train and test set. The train set is used to train the GEN and the generalization performance of the trained GEN is tested on the test set (test cycle). Both the sets are *not* permuted (the original time sequence is not disturbed), because when using the trained GEN in practice, past values of a time series are presented by means of a delay line. A delay line consists of several unit–delay operators z^{-1} whose purpose is to delay a signal by one time unit ($x_{t-1} = z^{-1}x_t$). This delay–line can be seen as placeholders for past values of a time series. Another reason is that the learning behavior is influenced by the sequence of train patterns presented to the GEN. The following experiments are conducted.

- Testing the gate: 'Splitting two Gaussian distributions' (Section 6.2)
- Repeating Weigend's experiment: 'Mixture of two non–linear processes' (Section 6.3)
- Analyzing the gate: 'Santa Fe Laser Data' (Section 6.4)
- Performance on real world data: 'Waste Water Purification' (Section 6.5)

The next sections present the conducted experiments.

6.2 Splitting two Gaussian distributions

The objective of the experiment of the splitting of two Gaussian distributions is to see whether the gate is able to split the input space and correctly assign the input patterns to the experts. This experiment can be seen as a test case for the implementation of the GEN. For this experiment a series of values is selected at random from two Gaussian distributions, which serve as the input values. The GEN has to classify these input values into two classes: one for each Gaussian distribution. The distributions are characterized as follows:

- $x_1(n) \sim N(-1.0, 0.1)$ negative class C1
- $x_2(n) \sim N(1.0, 0.1)$ positive class C2

In Figure 6.1 the sequence of values used to test the GEN is depicted.

Because this experiment is trivial, two experts are used, each with one input, one sigmoidal output neuron and no hidden neurons. The gate has one input neuron and two output neurons. In Table 6.1 the GEN parameters for this experiment are shown.

	Gate	Experts
Number of networks	1	2
Input neurons	1	1
Output neurons	2	1
Hidden neurons	0	0
Learning rate	0.1	0.2
Learn momentum	0.1	0.1

Table 6.1: The GEN parameters

6.2.1 Results

Several trials of this experiment where conducted. In some of the trials the gate converged to one output being one (one expert was active for the entire test set). One reason for this behavior is that the classification task is trivial, in that only one expert can handle this classification by itself. In the other trials the gate correctly splits the input space into two classes as depicted in Figure 6.2 and Figure 6.3.

We have observed that the initialization of the weights of the gate and the expert networks are responsible for the convergence to one expert for both classes or one expert for each class. This initialization determines how the gate splits the input space before training begins. If the weights of the gate are randomly set to such values that for most of the train patterns one gate output is frequently higher then the other gate output, the corresponding expert receives more training information then the other expert. Hence, the performance of that expert increases faster and the gate converges to one output being active. The initialization of the weights of the experts can also result in this behaviour, i.e. one expert is frequently better than the other and thus the gate converges to this expert.

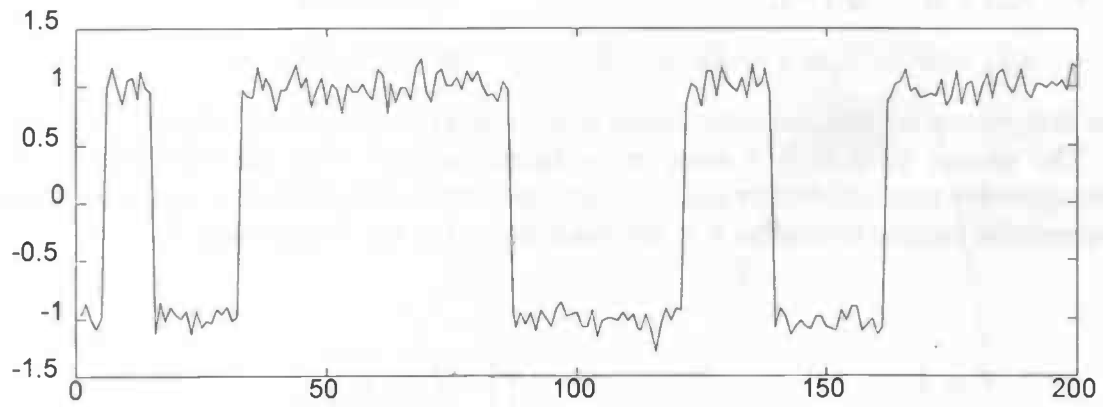

Figure 6.1: *The sequence of values in the test set*

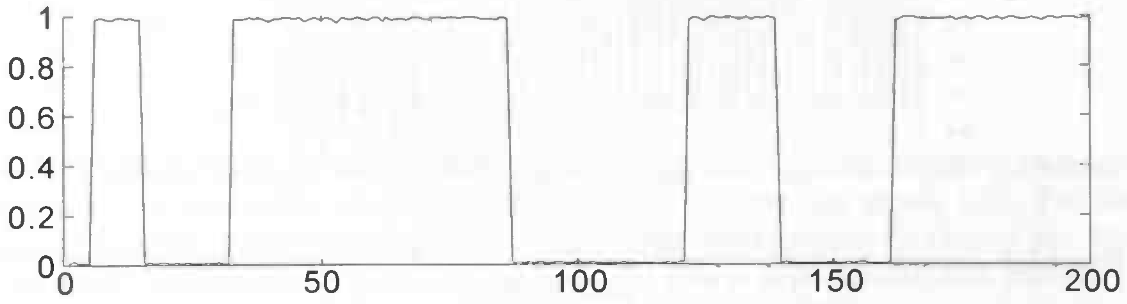

Figure 6.2: *Gate output 1*

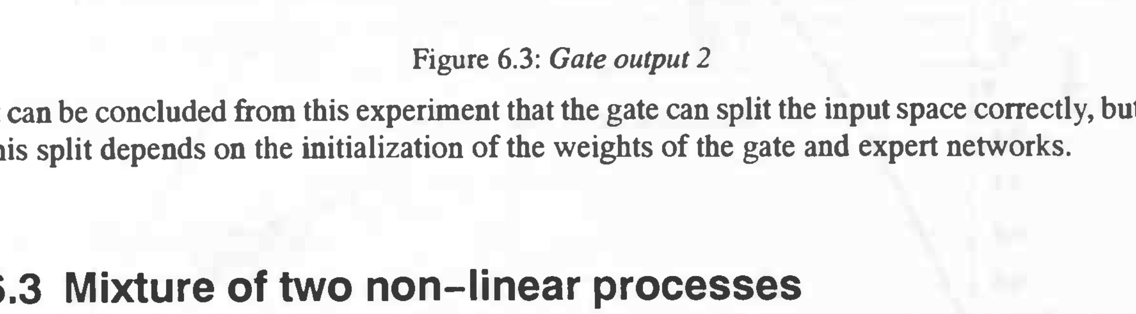

Figure 6.3: *Gate output 2*

It can be concluded from this experiment that the gate can split the input space correctly, but this split depends on the initialization of the weights of the gate and expert networks.

6.3 Mixture of two non-linear processes

The following experiment is the same as the computer-generated experiment used in Weigend *et al.*[16]. It is a time series prediction problem where the series switches (at random) between regimes. There are two processes, mathematically described as:

- $x_{t+1} = 2(1 - x_t^2) - 1,$ if switch = 1
- $x_{t+1} = \tanh(-1.2x_t + e_t+1),\ e_t \sim N(0, 0.1)$ if switch = 0

The first process is a deterministic chaotic process: it is the quadratic map on the interval [−1, 1]. The second process is a noisy non-chaotic process: it is the composition of an autoregressive process of order one, with Gaussian noise of variance 0.1, squashed through a hyperbolic tangent to confine it to the same interval as the first process.

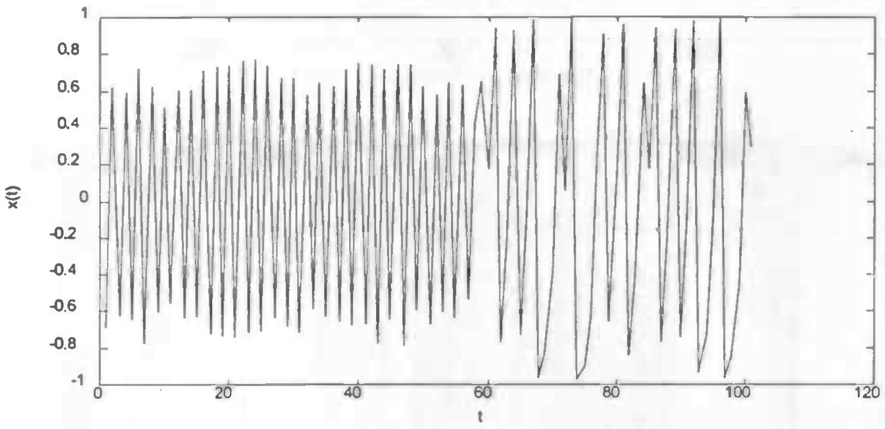

Figure 6.4: *Segment of the process with the left half the noisy tanh process and the right half the quadratic map*

In Figure 6.4 it is shown that the two processes give the same appearance in time domain, but they are actually very different, as evidenced by the two return plots (scatter plot) in Figure 6.5 and Figure 6.6.

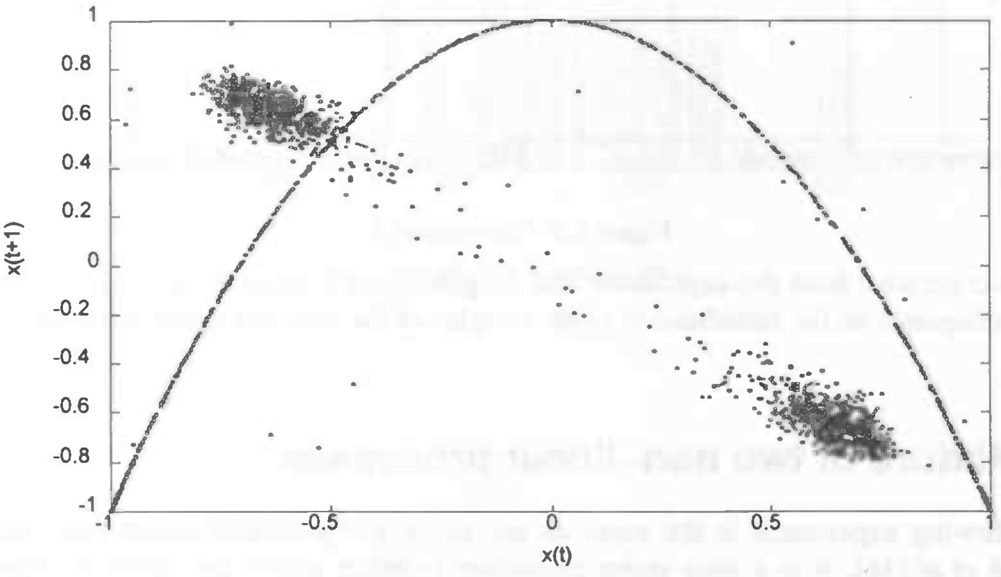

Figure 6.5: *Two−dimensionalreturn plot of x(t+1) and x(t)*

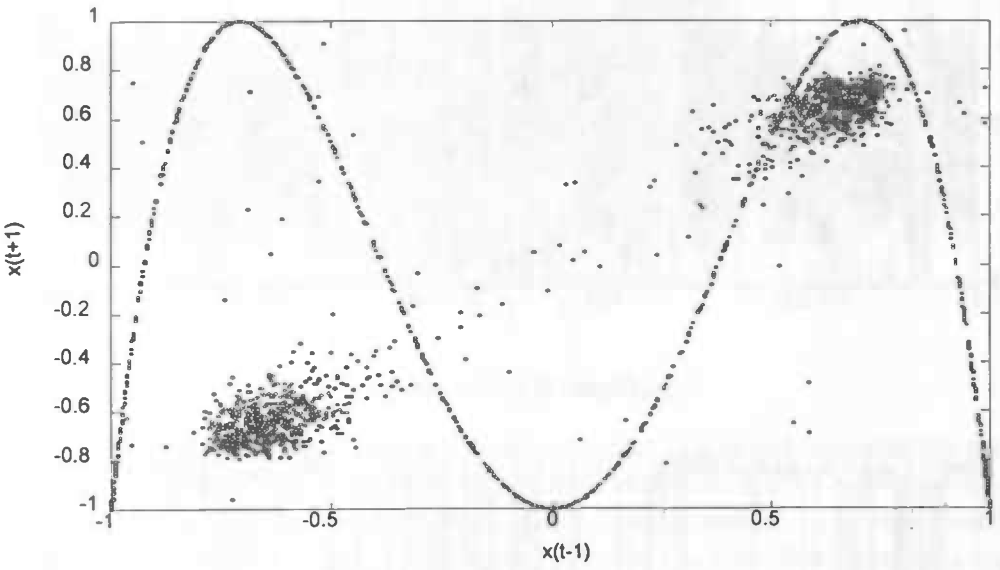

Figure 6.6: *Two-dimensionalreturn plot of x(t+1) and x(t−1)*

The switching between the two processes occurs at random; the exact switching times are unknown. The probability of switching between the two process equals 0.02. For the experiment 1000 patterns are used for the train set and 1000 patterns for the test set. The architecture contains three experts and one gate. The goal is to predict the next point x_{t+1}. Both the gate and the experts have access to the two past values of the series $\{x_t, x_{t-1}\}$. Each expert contains 12 hidden neurons, and the gate contains 15 hidden neurons. The output neurons of the experts are sigmoids.

	Gate	Experts
Number of networks	1	3
Input neurons	2	2
Output neurons	3	1
Hidden neurons	15	12
Learn speed	0.1	0.4
Learn momentum	0.1	0.1

Table 6.2: The GEN parameters

6.3.1 Results

Again, several experiments where conducted. Not one of the experiments converged to two surviving experts, as one might expect from the fact that the times series contains two processes. The gate did not split the input space as described in Weigend *et al.* [16]. Instead, Figures 6.8, 6.9 and 6.10 show a typical split made by the gate for the test set (Figure 6.7).

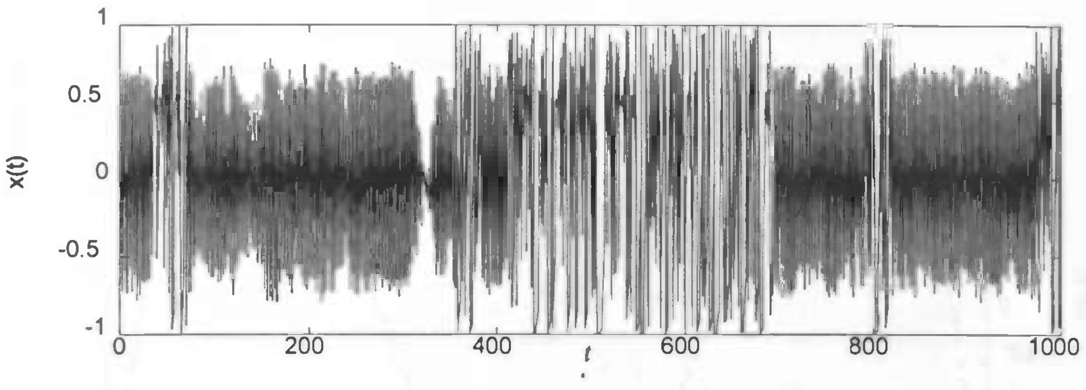

Figure 6.7: *Test set*

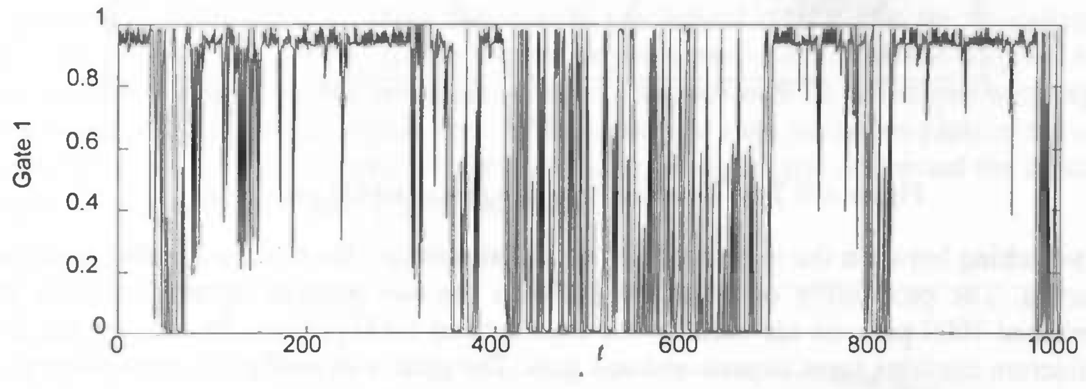

Figure 6.8: *Gate output 1*

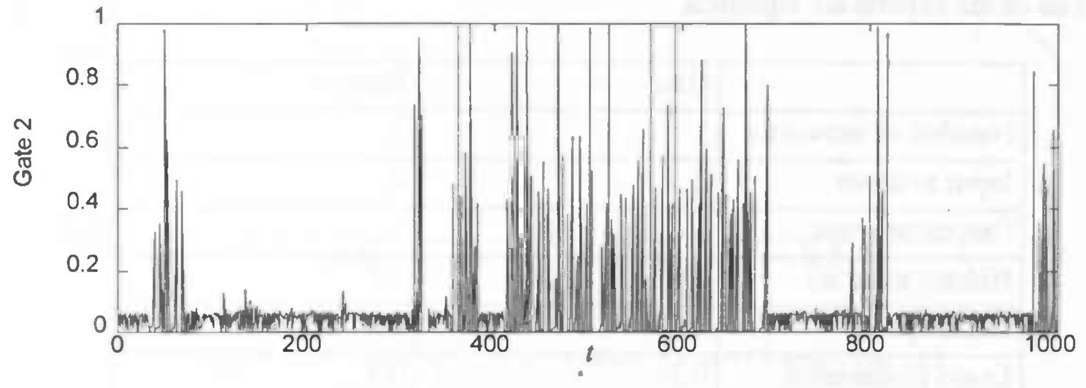

Figure 6.9: *Gate output 2*

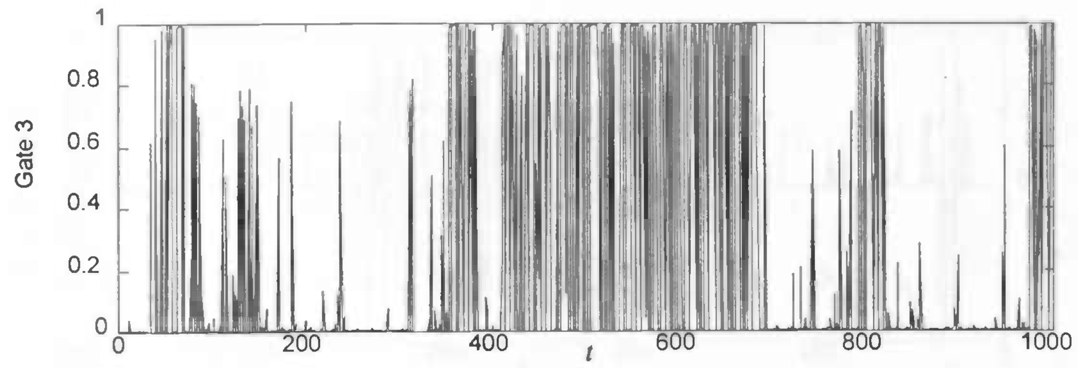

Figure 6.10: *Gate output 3*

Figure 6.8 shows that output 1 of the gate (and so expert 1) is largely responsible for the noisy tanh process. The quadratic map areas of the time series are shared between experts 2 and 3. To evidence this observation the outputs of the experts, multiplied by their corresponding gate outputs, are plotted ($y_j \cdot g_j$). This allowes us to see the contribution of the experts to the complete output signal (i.e. the output of the complete GEN).

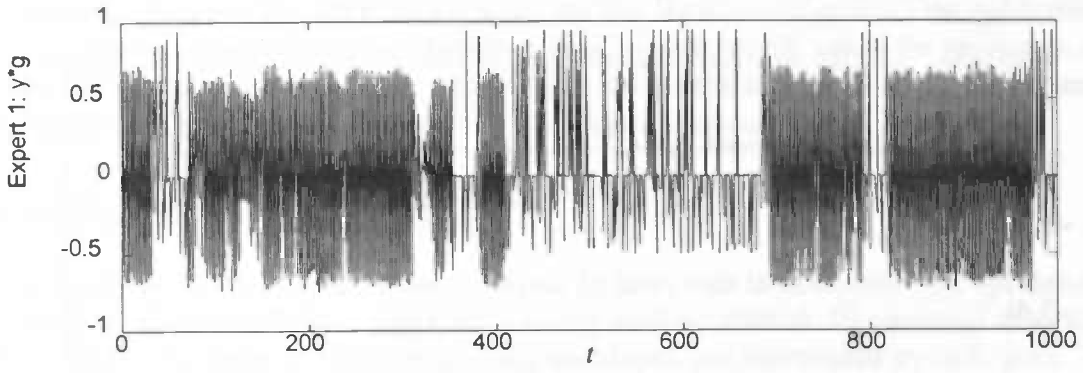

Figure 6.11: *Expert 1: $y_1 * g_1$*

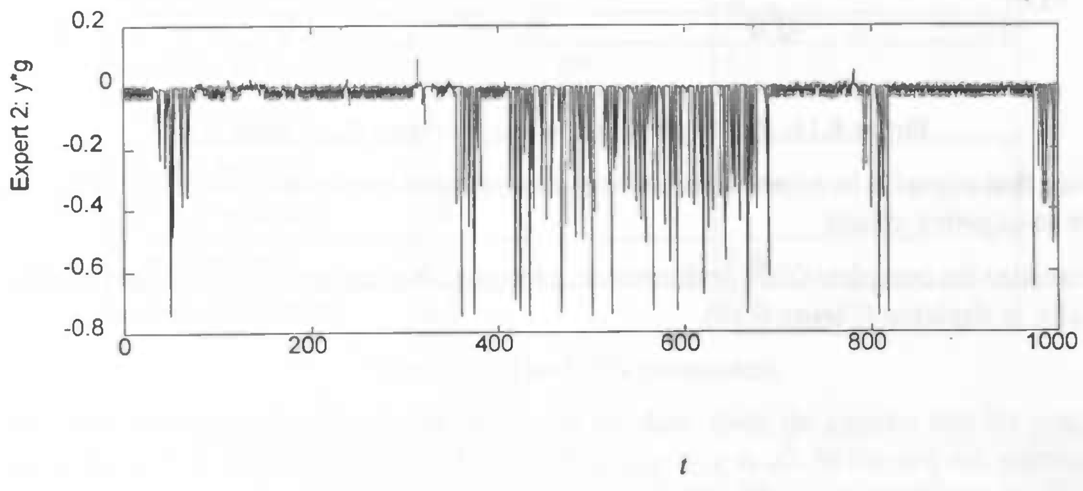

Figure 6.12: *Expert 2: $y_2 * g_2$*

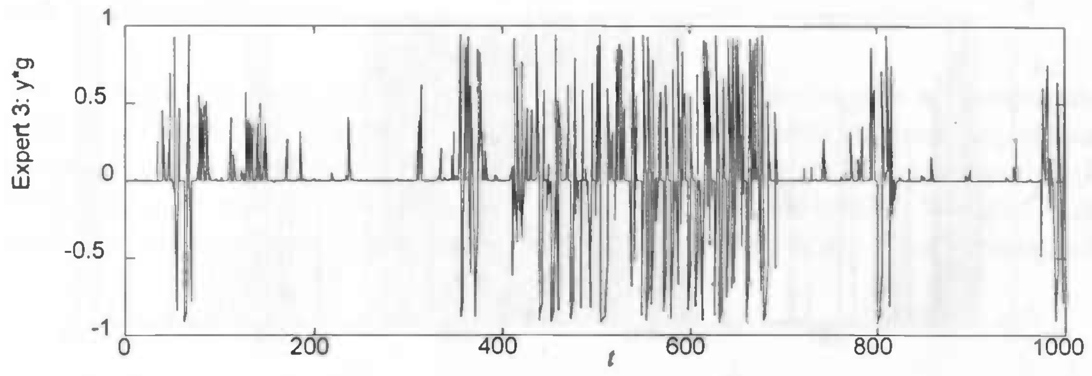

Figure 6.13: *Expert 3: $y_3 * g_3$*

Figures 6.11, 6.12 and 6.13 show that expert 1 is indeed responsible for the noisy tanh process, expert 2 concentrates (partly) on the negative part of the quadratic map and expert 3 is partly responsible for the complete quadratic map process. To investigate why expert 2 concentrates partly on the negative part of the quadratic map, the return plots of the expert output (multiplied with the gate output) and the input signal *x(t)* can reveal the functional input–output mapping expert 2 is responsible for.

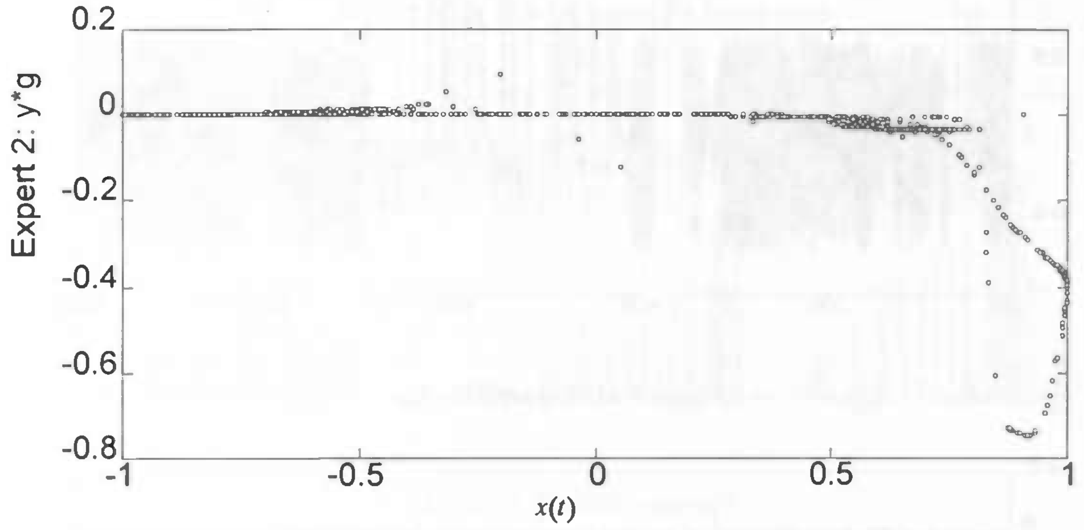

Figure 6.14: *Return plot of the output of expert 2 and input x(t)*

We see that expert 2 becomes active when the quadratic map moves from a positive high value to negative values.

To visualize the complete GEN performance, a return plot of the total GEN output y and input signal x_t is depicted (Figure 6.15).

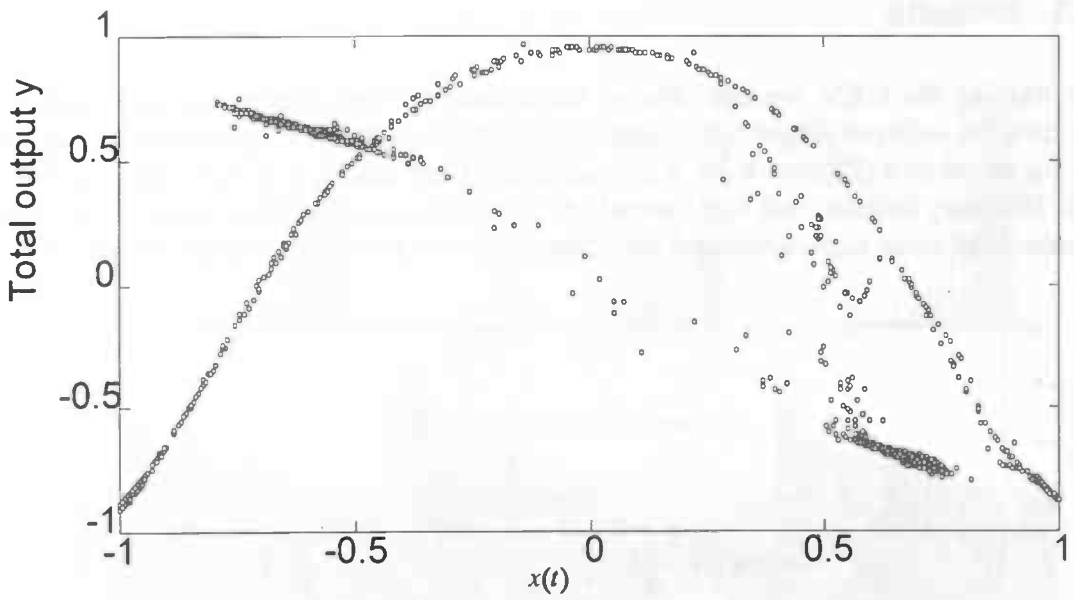

Figure 6.15: *Return plot of the total output y and x(t)*

Comparing Figure 6.15 with Figure 6.5, we see that the network predicts the quadratic map reasonably good except for a small positive area (x(t)=[0.4,0.6]) where the prediction of the noisy tanh process influences the prediction of the quadratic map. Further, the variance of the noisy tanh prediction is smaller than the original variance.

6.4 Santa Fe laser data

The objective of the experiment of the Santa Fe laser data is to examine the specialization of the gate. The Santa Fe laser data is a frequently used benchmark. It consists of chaotic laser data, where the areas of steadily growing envelopes are interrupted by collapses. These collapses are part of the internal dynamics of the laser. The prediction of these collapses is our main goal. In Table 6.1 the GEN parameters for this experiment are shown.

	Gate	Experts
Number of networks	1	6
Input neurons	5	5
Output neurons	6	1
Hidden neurons	10	5
Learning rate	0.1	0.4
Learn momentum	0.1	0.1

Table 6.3: The GEN parameters

The GEN has to predict the next point x_{t+1} of the data. Both the experts and the gate have access to the five past values of the data $\{x_t, x_{t-1}, x_{t-2}, x_{t-3}, x_{t-4}\}$. In the test set, presented in Figure 6.16., the collapses of the laser are clearly visable. The main goal is to see if one of the experts specializes in predicting these collapses.

6.4.1 Results

After training the GEN, we can observe the following. Two experts are responsible for predicting the collapse (Figures 6.17 and 6.19), and three experts are predicting the steadily growing envelopes (Figures 6.20, 6.21 and 6.22). Gate output 2 is very inactive (Figure 6.18). This may indicate that five instead of six experts would suffice. Weigend *et.al.* [16] performed the same experiment and also concluded that five or six experts would suffice.

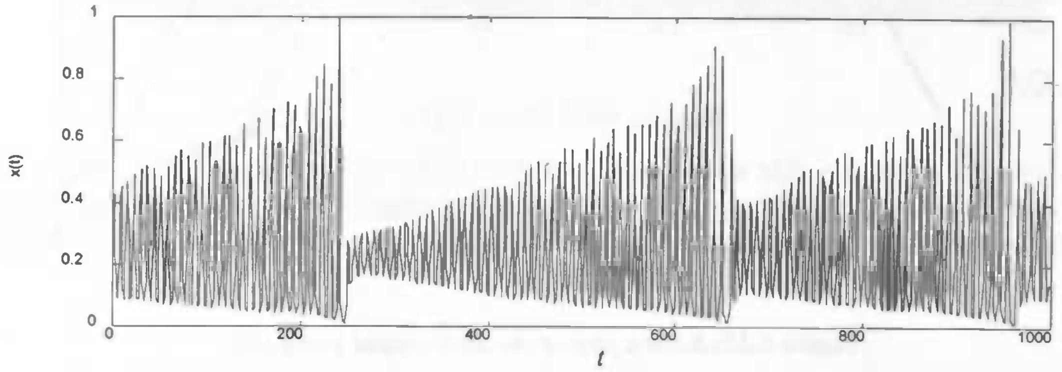

Figure 6.16: *Test set Santa Fe Laser data*

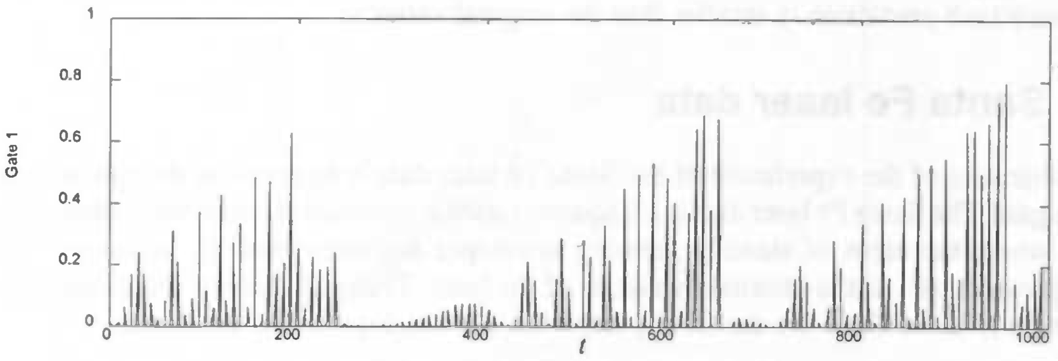

Figure 6.17: *Gate output 1*

Figure 6.18: *Gate output 2*

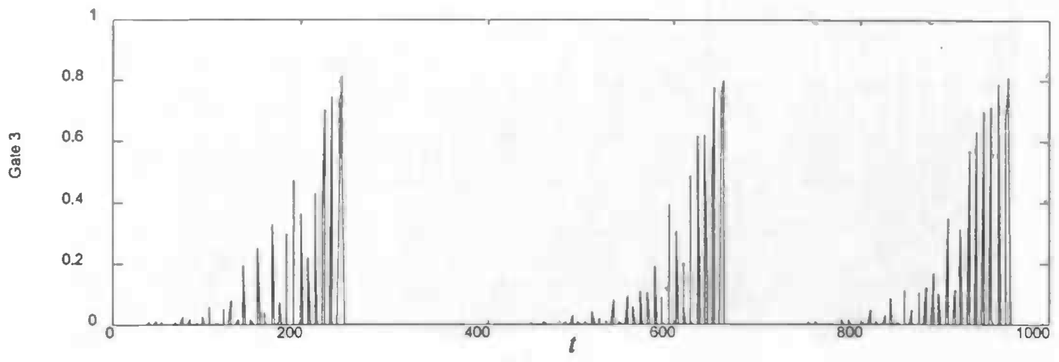

Figure 6.19: *Gate output 3*

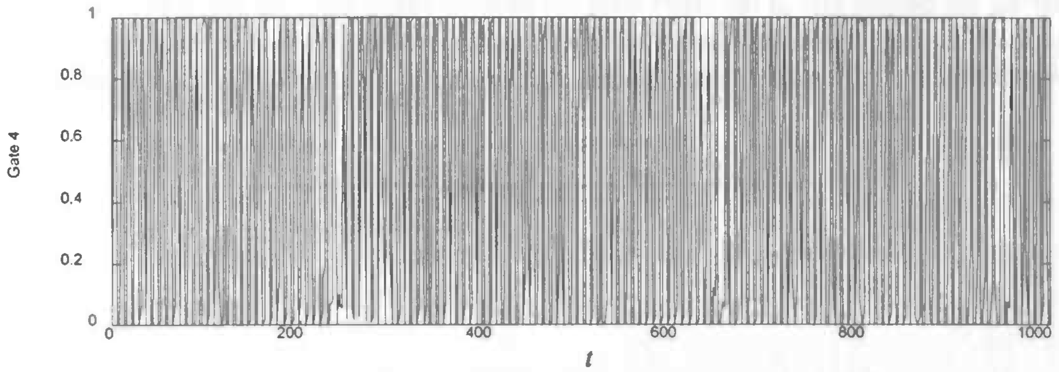

Figure 6.20: *Gate output 4*

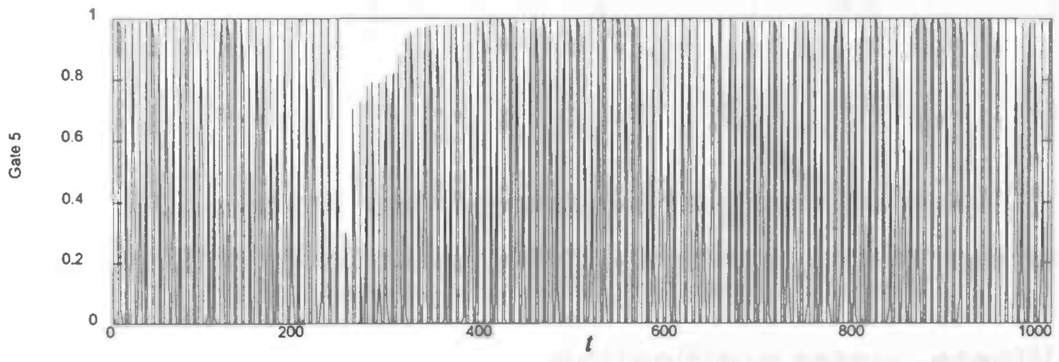

Figure 6.21: *Gate output 5*

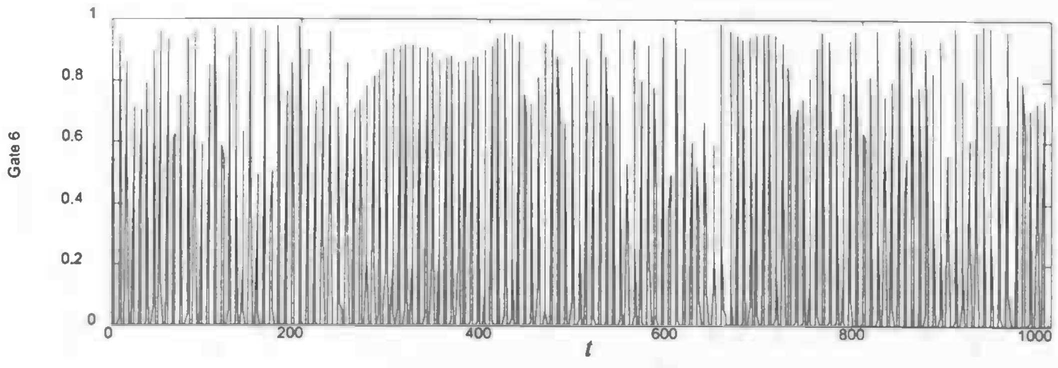

Figure 6.22: *Gate output 6*

Comparing the output y(t) of the GEN with the desired value x(t), we see in Figure 6.23 that the prediction relates almost linearly with the desired value (crosscorrelation of 0.9724), except for values of x(t) higher than 0.6 where the prediction is smaller then the desired value.

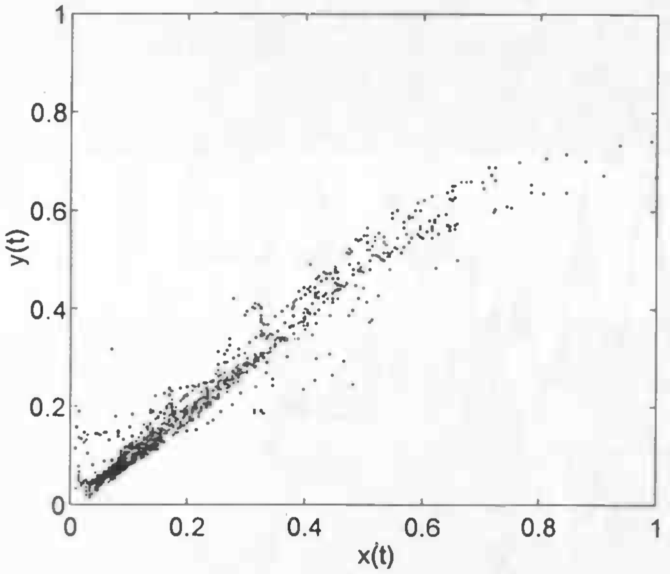

Figure 6.23: *Return plot of GEN output and the desired value*

6.5 Waste–water purification

In this experiment we study the performance of the GEN on the prediction of a multivariate real–world time series. This series arises from an industrial application of waste–water purification (described in section 2.3). The ammonia concentration has to be predicted by using temperature and three other concentrations in the water (namely oxygen and the so–called auc and influent measurements), which are related to the ammonia concentration. After an extensive analysis of the data [14], a clear daily period was found. In order to make this information available to the GEN, a sine representing the daily rhythm and the type of day (differentiating between weekends and working days) is included in the train set. This results in an input vector of 12 values for every time step (one quarter of an hour). Table 6.4 presents the network parameters used for this experiment.

	Gate	Experts
Number of networks	1	4
Input neurons	12	12
Output neurons	4	1
Hidden neurons	20	15
Learning rate	0.7	0.7
Learn momentum	0.1	0.1

Table 6.4: The GEN parameters

6.5.1 Results

The experiment is repeated many times and every run resulted in a MSE of about 0.04. The test set is depicted in Figure 6.24.

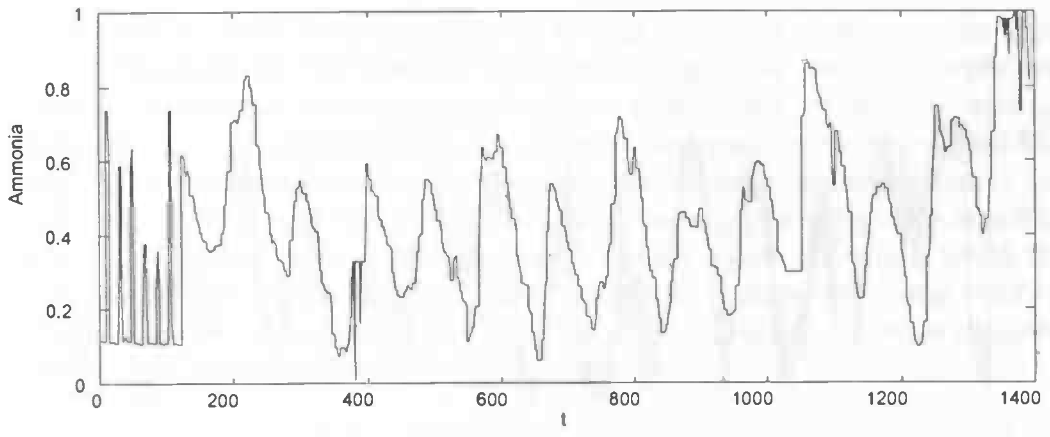

Figure 6.24: *The ammonia concentration in the first half of the test set*

Although the GEN performed poorly in predicting the ammonia concentration, it discovered the periodic behavior (Figure 6.25).

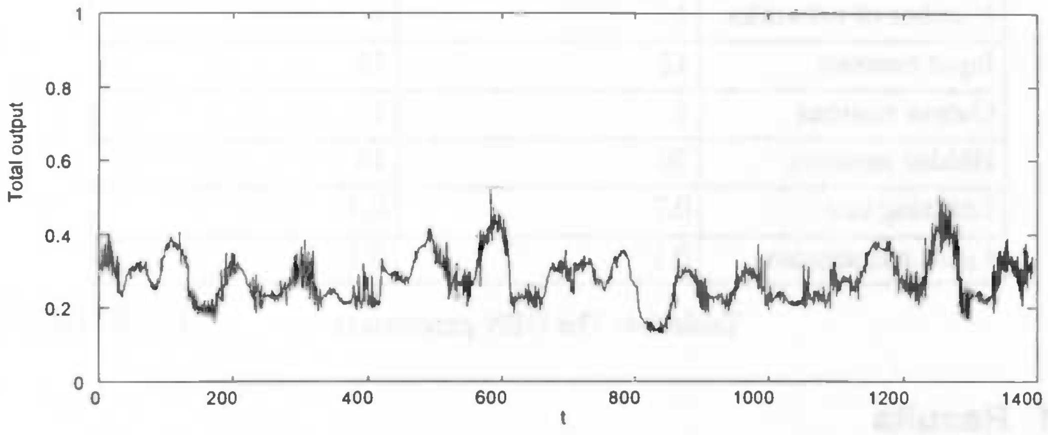

Figure 6.25: *The total output of GEN*

Analyzing output 2 of the gate, we see in Figure 6.26 that this output is responsible for the prediction of the peaks in the series.

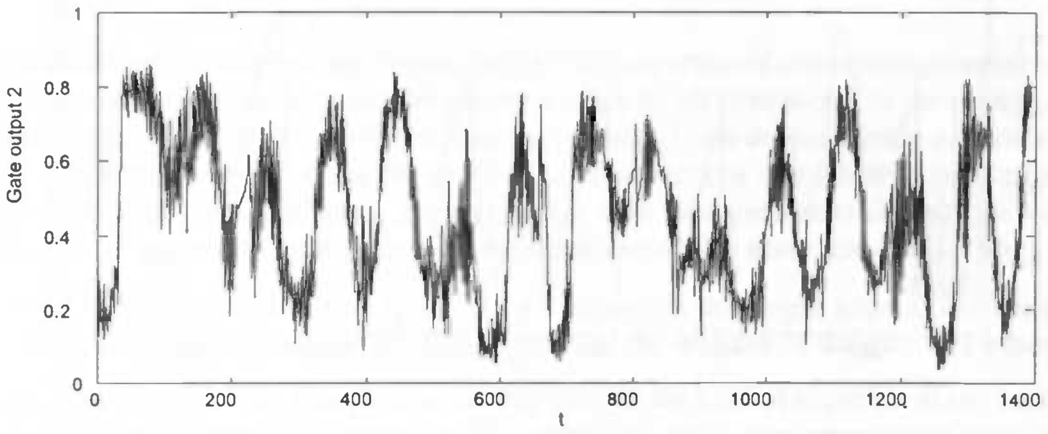

Figure 6.26: *Gate output 2*

To present an indication of the performance of the GEN, the resulting error of the prediction on the test set is depicted in Figure 6.27 and is very simular to the test set (Figure 6.24). A possible reason of the very low performance is that to few past values of the related data is used for the prediction. By applying more past values (more information) to the GEN a higher performance can be expected.

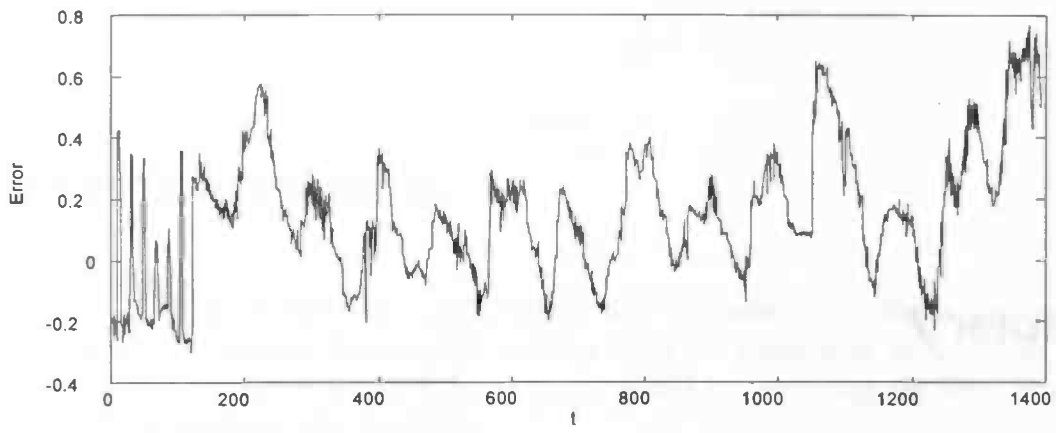

Figure 6.27: *The error made by GEN*

6.6 Discussion

In this chapter we have used the GEN for several prediction problems. The choice of parameters depends on the kind of experiment conducted and are experimentally determined. For instance, if the number of regimes were known, we used the same number of experts. We have observed that the gate can split the input space correctly, but this split depends on the initialization of the weights of the gate and expert networks. Future research has to be performed on this initialization phenomenon to gain more insight in this matter. Examining the relation between the gate outputs and the inputs, the way in which the gate splits the input space can be analyzed. This way, hidden regimes in a times series can be uncovered. Thus, besides using the GEN for time series prediction, it also is an useful tool for analyzing the underlying dynamics of a time series.

Chapter 7

Conclusions and further research

This chapter presents conclusions and directions for further research on prediction of time series with gated experts networks.

We selected the gated experts network, for its nice properties of non–linear gate and experts, soft–partitioning the input space and adaptive noise levels (variances) of the experts. These properties are found useful for times series prediction. From the conducted experiments we have empirically observed that the gated experts network is a reasonable good choice for real–world time series prediction. It uses the experts as local predictors and the gate plausible allocates the experts to local regions of the input space. Our main conclusions are:

- The gate splits the input space, but not always as one might expect. The splitting of input space depends on the initialization of the weights of the gate and experts.
- The choice of the free parameters depends on the kind of experiment conducted and are experimentally determined by repeating the same experiment with varying parameters.
- The learn rate of the gate should be small compared to the learn rate of the experts to yield a better segmentation of the input space.
- By examining the relation between the gate outputs and the inputs of the gated experts network, the way in which the gate splits the input space can be analyzed. This way, hidden regimes in a time series can be uncovered. Thus, the gated experts network is a useful tool for analyzing the underlying dynamics of a time series.

Further research could concentrate on extending this structure to some kind of hierarchy, where the complexity of a process is captured by the divide and conquer principle. The gated experts network can be modified by adding different density functions to individual experts, applying dynamic growth of the number of experts and hidden units of the experts. The implementation of the gated experts network can be extended to include separate tapped delay lines, with different values for the delay step size and the length of the delay line, for the experts and the gate to capture the periodicities in the time series.

Acknowledgements

We like to thank R.Venema, J.A.G.Nijhuis and L.Spaanenburg for their comments, suggestions and support. Further, we want to thank J.P.Kelderman of Grontmij BV for introducing the waste-water purification problem and for supplying the data. Finally, we want to thank everyone else involved.

References

[1] Chatfield, C., *The Analysis of Time Series: An introduction.* Chapman & Hall, 5th edition, 1996.

[2] Dempster, A.P., Laird, N.M. and Rubin, D.B., "Maximum likelihood from incomplete data via the EM algorithm", *Journal of the Royal Statistic Society*, Series B 39, 1977.

[3] Fritsch, J. and Finke, M and Waibel, A., "Adaptively growing Hierarchical Mixtures of Experts (HME)", Carnegie Mellon University, Pittsburg, 1996.

[4] Haykin, S., *Neural Networks: A Comprehensive Foundation*, MacMillan, New York, NY, 1994.

[5] Husmeier, D. and Taylor, J.G., "Predicting Conditional Probability Densities of Stationary Stochastic Time Series", *Neural Networks*, Vol.10, No. 3, 1997.

[6] Jordan, M.I. and Jacobs, R.A., "Hierarchical Mixtures of Experts and the EM Algorithm", *Neural Computation*, Vol.6, 1994.

[7] Jordan, M and Jacobs, R.A. "Modular and Hierarchical Learning Systems", *The handbook of Brain Theory and Neural Networks*, MIT Press, Camebridge, MA, 1996.

[8] Jacobs, R.A., Jordan, M.I., Nowlan, S.J., and Hinton, G.E. "Adaptive Mixtures of Local Experts", *Neural Computation*, 3:79–87, 1991

[9] Jacobs, R.A., Peng, F. and Tanner, M.A., "A Bayesian Approach to Model Selection in Hierarchical Mixtures-of-Experts Architectures", *Neural Networks*, Vol.10, No.2, 1997.

[10] Jordan, M.I. and Xu, L., "Convergence Results for the EM Approach to Mixtures of Experts Archtectures", *Neural Networks*, Vol.8, No.9, 1995.

[11] Kehagias, A. and Petridis, V., "Predictive Modular Neural Networks for Time Series Classification" *Neural Networks*, Vol.10, No.1, 1997.

[12] Khotanzad, A., Afkhami, R., "ANNSTLF – A Neural Network Based Electric Load Forecasting System", *IEEE Transaction on Neural Networks*, Vol.8, No. 4, July 1997.

[13] LeBaron, B. and Weigend, A.S., "A Bootstrap of the Effect of Data Splitting on Financial Time Series", *IEEE Transaction on Neural Networks*, Vol.9, No. 1, January 1998.

[14] Venema, R.S., Bron, J., Zijlstra R.M., Nijhuis, J.A.G, Spaanenburg, L., "Using neural networks for waste–water purification", *Proceedings 12th International Symposium Computer Science for Environmental Protection UI '98*, Bremen, Germany, 1998

[15] Weigend, A.S., "Time Series Analysis and Prediction using Gated Experts with Application to Energy Demand Forecasts", *Applied Artificial Intelligence*, Vol.10, 1996.

[16] Weigend, A.S. and Mangeas, M. and Srivastava, A.N., "Non–Linear Gated Experts for Time Series: Discovering Regimes and Avoiding Overfitting", *International Journal of Neural Systems*, Vol. 6, No. 4, 1995.

[17] Weigend, A.S. and Nix, D.A., "Predictions with Confidence Intervals (Local Error Bars)", *ICONIP'94*, Seoul, 1994.

Appendix A

The derivation of the posterior probability

$$E\left[I_j(n)|X, D, \Theta\right] \quad \text{(A-1)}$$

$$= \mathrm{h_j(n)} \quad \text{(A-2)}$$

$$= P(\mathrm{j}|\mathrm{x}(n), \mathbf{d}(n)) \quad \text{(A-3)}$$

$$= \frac{P(\mathrm{j}, \mathbf{d}(n)|\mathrm{x}(n))}{P(\mathbf{d}(n)|\mathrm{x}(n))} \quad \text{(A-4)}$$

$$= \frac{g_k(\mathbf{x}(n), w_g)\, P(\mathbf{d}(n)|\mathbf{x}(n), w_j)}{\sum_{k=1}^{K} g_k(\mathbf{x}(n), w_g)\, P(\mathbf{d}(n)|\mathbf{x}(n), w_k)} \quad \text{(A-5)}$$

$$= \frac{g_j(\boldsymbol{x}(n), w_g) \frac{1}{(2\pi)^{q/2}\sigma_j^q} \exp\left(\frac{-\|d(n) - y_j(x(n), w_g)\|^2}{2\sigma_j^2}\right)}{\sum_{k=1}^{K} g_k(\boldsymbol{x}(n), w_g) \frac{1}{(2\pi)^{q/2}\sigma_k^q} \exp\left(\frac{-\|d(n) - y_k(x(n), w_g)\|^2}{2\sigma_k^2}\right)} \quad \text{(A-6)}$$

The derivation of the cost function

$$\mathrm{C_M} \quad \text{(A-7)}$$

$$= \mathrm{E}\,[\ln \mathrm{L}(\mathrm{D}, \mathrm{Z}|\mathrm{X}, \Theta)] \quad \text{(A-8)}$$

$$= \ln\left[\prod_{n=1}^{N}\prod_{j=1}^{K}\left[g_j(\mathbf{x}(n), w_g)\, P(\mathbf{d}(n)|\mathbf{x}(n), w_j)\right]^{h_j(n)}\right] \quad \text{(A-9)}$$

$$= \sum_{n=1}^{N}\sum_{j=1}^{K} h_j(n) \ln\left[g_j(\mathbf{x}(n), w_g)\, P(\mathbf{d}(n)|\mathbf{x}(n), w_j)\right] \quad \text{(A-10)}$$

$$= \sum_{n=1}^{N}\sum_{j=1}^{K} h_j(n) \ln\left[g_j(\mathbf{x}(n), w_g) \frac{1}{(2\pi)^{q/2}\sigma_j^q} \exp\left(\frac{-\|\, d(n) - y_j(x(n), w_j)\,\|^2}{2\sigma_j^2}\right)\right] \quad \text{(A-11)}$$

$$= \sum_{n=1}^{N}\sum_{j=1}^{K} h_j(n)\left(\ln g_j(\mathbf{x}(n), w_g) - \frac{\|\,\mathbf{d}(n) - \mathbf{y}_j(\mathbf{x}(n), w_j)\,\|^2}{2\sigma_j^2} + \frac{q}{2}\ln(2\pi\sigma_j^2)\right) \quad \text{(A-12)}$$

$$= \sum_{n=1}^{N}\sum_{j=1}^{K} h_j(n)\ln\left[g_j(\mathbf{x}(n), w_g)\right] - \tag{A-13}$$

$$\frac{1}{2}\sum_{n=1}^{N}\sum_{j=1}^{K} h_j(n)\left(\frac{-\| \mathbf{d}(n) - \mathbf{y}_j(\mathbf{x}(n), w_j) \|^2}{\sigma_j^2} + q\ln(2\pi\sigma_j^2)\right)$$

The derivation of the local gradient of the gate

$$\frac{\partial C_M}{\partial u_i^{(l)}(n)} \tag{A-14}$$

$$= \sum_{k=1}^{K} h_k(n)\frac{\partial}{\partial u_i^{(l)}(n)}\ln g_k(n) \tag{A-15}$$

$$= \sum_{k=1}^{K} h_k(n)\frac{\partial}{\partial u_i^{(l)}(n)}\ln\left[u_k^{(l)}(n) - \ln\sum_{j=1}^{K}\exp u_j^{(l)}(n)\right] \tag{A-16}$$

$$= h_i(n)\left[1 - \frac{exp(u_i^{(l)})}{\sum_{j=1}^{K} exp(u_k^{(l)})}\right] + \sum_{k\neq 1}^{K} h_k(n)\frac{\partial}{\partial u_i^{(l)}(n)}\ln\left[u_k^{(l)}(n) - \ln\sum_{j=1}^{K}\exp u_j^{(l)}(n)\right] \tag{A-17}$$

$$= h_i(n)(1 - g_i(n)) + \sum_{k\neq i}^{K} h_k(n)(0 - g_i(n)) \tag{A-18}$$

$$= h_i(n) - h_i(n)g_i(n) - \sum_{k\neq i}^{K} h_k(n)g_i(n) \tag{A-19}$$

$$= h_i(n) - \sum_{k=1}^{K} h_k(n)g_i(n) \tag{A-20}$$

$$= h_i(n) - g_i(n)\sum_{k=1}^{K} h_k(n) \tag{A-21}$$

$$= h_i(n) - g_i(n) \tag{A-22}$$

The derivation of the variance update

$$\frac{\partial C_M}{\partial \sigma_k^2} = 0 \tag{A-23}$$

$$\equiv \frac{\sum_{n=1}^{N}\left(h_k(n)\,\| d(n) - y_k(\mathbf{x}(n), w_k)\|^2\right) - \sigma_k^2\, q \sum_{n=1}^{N} h_k(n)}{(\sigma_k^2)^2} = 0 \tag{A-24}$$

$$\equiv \sum_{n=1}^{N}\left(h_k(n)\,\| d(n) - y_k(\mathbf{x}(n), w_k)\|^2\right) - \sigma_k^2\, q \sum_{n=1}^{N} h_k(n) = 0 \tag{A-25}$$

$$\equiv \frac{\sum_{n=1}^{N}\left(h_k(n)\,\| d(n) - y_k(\mathbf{x}(n), w_k)\|^2\right)}{q \sum_{n=1}^{N} h_k(n)} = \sigma_k^2 \tag{A-26}$$

Appendix B

```
/**********************************************************************/
/**********************************************************************/
#include "interact.h"
#include "constants_int.h"
#include <math.h>
#include <stdlib.h>
#include <string.h>
#define   max(x,y) (((x)>(y))?(x):(y))

#define   MAX_LAYERS     4L
#define   LAYER_DIST     40L
#define   NEURON_X_DIST  20L
#define   NEURON_Y_DIST  20L
#define   MAX_EXPERTS    (INTERACT_MAX_NETWORKS-1)
/*
  These global variables are needed for the gated experts network
*/
net_Sid_t       expert_id[MAX_EXPERTS];
net_Sid_t       gate_id;
double          expert_Variance[MAX_EXPERTS];
double          **posterior;
learn_Spar_t    learn_rate=0.7, momentum=0.0;
observation_Sargs_net_status_t    oa;
observation_Sargs_net_structure_toas;

/*
  This function creates the posterior matrix.
  Parameters:
                nr_patterns    : the total number of patterns
                nr_experts     : the number of expert networks
*/
void create_posterior(  long nr_patterns,
                        long nr_experts)
{
  long i,j;

  posterior = (double **)malloc(sizeof(double*)* nr_patterns);
  if( posterior==NULL ) exit(-2 );
  for(i=0;i<nr_patterns;i++){
    posterior[i]=(double  *)malloc( sizeof(double ) * nr_experts );
    if( posterior[i]==NULL ) exit(-2 );
    for(j=0;j<nr_experts;j++)
      posterior[i][j]=0.0;
  }
}
/*
   This function loads the pattern database file
   Parameters:
                 name      : name of the pdbf file
   Result:
                 list_id   : identifier of the pattern list
*/
pattern_Slist_id_t MyLoadPatterns (char name[])
{
   status_St             status;
   pattern_Slist_id_t    list_id;

   pattern_Slist_load (name, &list_id, &status);
   printf ("Loading database file: %s\n", name);        return (list_id);
}
/*
   This function splits a pattern list into a train and test set
   Parameters:
                 list_id     : identifier of the pattern list that needs to be split
                 trn_set     : pointer to the train set
                 tst_set     : pointer to the test set
   Remark:
                   first, the pattern list is permuted
*/
void MySplitPatterns (  pattern_Slist_id_t list_id,
                        pattern_Slist_id_t *trn_set,
                        pattern_Slist_id_t *tst_set)
{
   status_St      status;
   long           selector[INTERACT_MAX_PATTERN_PATTERNS];
   long           ind, nr_trn, nr_tst;
   pattern_Slist_info_t info;
   pattern_Slist_get_info (list_id, &info, &status);
   nr_trn = info.nr_patterns/2;
   nr_tst = info.nr_patterns - nr_trn;  /* Select patterns with ids 1..nr_trn */
   for (ind = 0;  ind < nr_trn;  ind++)
      selector[ind] = ind + 1;
   pattern_Slist_select_rows (list_id, selector, nr_trn, trn_set, &status);
   /* Select patterns with ids nr_trn+1..info.nr_patterns */
   for (ind = 0;  ind < nr_tst; ind++)
      selector[ind] = nr_trn + ind +1;
   pattern_Slist_select_rows (list_id, selector, nr_tst, tst_set, &status);
}

void initialize_Sexpert (net_Sid_t net_id)
{
   status_St     status;

   net_Sset_working_net( net_id , &status);     set_Sbias_random (0L,neu-
ron_Slist,-1.0,1.0,0.0,0.6,rand_Suniform,&status);
   set_Sweight_random (0L, 0L, neuron_Slist, neuron_Slist, 1, -1.0, 1.0, 0.0,
                           0.6, rand_Suniform,&status);
}

void initialize_Sgate (net_Sid_t net_id)
```

```
{
   status_St     status;

   net_Sset_working_net( net_id ,&status );     set_Sbias_random (0L,neu-
ron_Slist,-1.0,1.0,0.0,0.6,rand_Suniform,&status);
   set_Sweight_random (0L, 0L, neuron_Slist, neuron_Slist, 1, -1.0, 1.0, 0.0,
                            0.6, rand_Suniform,&status);
}

void expert_Screate_nets( long nr_experts,
                          long nr_inputs,
                          long nr_hiddens,
                          long nr_outputs,
                          net_Sid_t *ids,
                          status_St *status)
{
  status_St           s;
  long                height;
  long                x_off, y_off;
  long                k;
  neuron_Sid_t        neuron_id;
   learn_Spar_t       learn_rate=0.9, momentum=0.1;
   net_Sid_t          net_id;
  char                str[20]; status->all = status_Sok;

  height = max( max( nr_inputs, nr_hiddens), nr_outputs );

   for(k=0;k<nr_experts;k++){
     expert_Variance[k]=1.0;
     strcpy(str,"Expert ");
     net_Sadd(str,&net_id, &s );
     default_Sthres( sigdet_sum_c2_St, 0.0, 0.0, 0.0, 0.0, 0.0, &s );
     default_Slearn( error_back_St, learn_rate, momentum, 0.0, 0.0, 0.0, &s );
     net_Sset_working_net( net_id ,&s);
     ids[k]=net_id;
     printf("\nNet id: %i", net_id);          /* add input layer */
     default_Skind( kind_Sinput_neuron, &s );
     x_off = 0;
     y_off = ((height - nr_inputs) * NEURON_Y_DIST ) / 2;
     default_Scoord_incr( 0L, NEURON_Y_DIST, 0L, &s );
     neuron_Sadd( neuron_Sauto_t, 0L, x_off, y_off, 0L, nr_inputs, &neuron_id, &s );
     x_off += LAYER_DIST;
     /* add hidden layer */
     default_Skind( kind_Shidden_neuron, &s );
      y_off = ((height - nr_hiddens) * NEURON_Y_DIST ) /2;
     default_Scoord_incr( 0L, NEURON_Y_DIST, 0L, &s );
     neuron_Sadd( neuron_Sauto_t, 0L, x_off, y_off, 0L, nr_hiddens, &neuron_id, &s );
     x_off +=LAYER_DIST;     /* add output layer */
     default_Sthres( sigdet_sum_c2_St, 0.0, 0.0, 0.0, 0.0, 0.0, &s );
     default_Skind( kind_Soutput_neuron, &s );
     y_off = ((height - nr_outputs) * NEURON_Y_DIST ) / 2;
     neuron_Sadd( neuron_Sauto_t, 0L, x_off, y_off, 0L, nr_outputs, &neuron_id, &s );
     /* add connections*/
     connection_Sadd(0L, 0L, input_Slist, hidden_Slist, 1, 0.0, connection_Ssingle, &s );
     connection_Sadd(0L, 0L, hidden_Slist, output_Slist, 1, 0.0, connection_Ssingle, &s );
     calc_Sset_mode( calc_Ssyn, 1L, 1L, &s );
```

```
      initialize_Sexpert (net_id);
    }
  }

  void gate_Screate_net(    long nr_inputs,
                            long nr_hiddens,
                            long nr_outputs,
                            net_Sid_t *id,
                            status_St *status)
  {
    status_St           s;
    long                height;
    long                x_off, y_off;
    neuron_Sid_t        neuron_id;
    learn_Spar_t        learn_rate=0.9, momentum=0.1;
    net_Sid_t           net_id;

    status->all = status_Sok;
    height = max( max( nr_inputs, nr_hiddens), nr_outputs );

    net_Sadd("Gate",&net_id, &s );
    default_Sthres( sigdet_sum_c2_St, 0.0, 0.0, 0.0, 0.0, 0.0, &s );
    default_Slearn( error_back_St, learn_rate, momentum, 0.0, 0.0, 0.0, &s );
    net_Sset_working_net( net_id ,&s);
    *id=net_id; /* add input layer */
    default_Skind( kind_Sinput_neuron, &s );
    x_off = 0;
    y_off = ((height - nr_inputs) * NEURON_Y_DIST ) / 2;
    default_Scoord_incr( 0L, NEURON_Y_DIST, 0L, &s );
    neuron_Sadd( neuron_Sauto_t, 0L, x_off, y_off, 0L, nr_inputs, &neuron_id, &s );
    x_off += LAYER_DIST;
    /* add hidden layer */
    default_Skind( kind_Shidden_neuron, &s );
    y_off = ((height - nr_hiddens) * NEURON_Y_DIST ) /2;
    default_Scoord_incr( 0L, NEURON_Y_DIST, 0L, &s );
    neuron_Sadd( neuron_Sauto_t, 0L, x_off, y_off, 0L, nr_hiddens, &neuron_id, &s );
    x_off +=LAYER_DIST;
    /* add output layer */
    default_Sthres( sigdet_sum_c2_St, 0.0, 0.0, 0.0, 0.0, 0.0, &s );
    default_Skind( kind_Soutput_neuron, &s );
    y_off = ((height - nr_outputs) * NEURON_Y_DIST ) / 2;
    neuron_Sadd( neuron_Sauto_t, 0L, x_off, y_off, 0L, nr_outputs, &neuron_id, &s );
    /* add connections*/
    connection_Sadd(0L, 0L, input_Slist, hidden_Slist, 1, 0.0, connection_Ssingle, &s );
    connection_Sadd(0L, 0L, hidden_Slist, output_Slist, 1, 0.0, connection_Ssingle, &s );
    calc_Sset_mode( calc_Ssyn, 1L, 1L, &s );
    initialize_Sgate (net_id);
  }


  void gen_Screate_net(   pattern_Slist_id_t    list_id,
                          long                  nr_experts,
                          long                  nr_experts_hiddens,
                          long                  nr_gate_hiddens,
                          status_St             *status )
  {
```

```
  pattern_Slist_info_t  info;
  long                    nr_inputs,
                          nr_outputs,
                          nr_patterns;
  pattern_Slist_get_info (list_id, &info, status);
  nr_inputs = info.nr_inputs;
  nr_outputs = info.nr_targets;
  nr_patterns = info.nr_patterns;
  expert_Screate_nets( nr_experts, nr_inputs, nr_experts_hiddens, nr_outputs,
                       expert_id, status);
  gate_Screate_net( nr_inputs, nr_gate_hiddens, nr_experts,&gate_id, status );
  create_posterior(  nr_patterns,  nr_experts);
}

void MySetObsevations ( net_Sid_t net_id , char name[])
{
  status_St                           status;
  observation_Sid_t                   obs;
  observation_Sargs_net_status_t      *sdata;
  observation_Sdata_t                 data;
  static long                         x_off=0, y_off=0;

  net_Sset_working_net( net_id ,&status); sdata = &data.net_status_args;

  sdata->group_id       = neuron_Slist;
  sdata->min_value      = -1.0;
  sdata->max_value      = 1.0;
  sdata->eval_incre     = 1;
  sdata->status         = observation_Sstatus_out_level;
  sdata->figure         = observation_Sfigure_circle;
  sdata->size           = 10;
  sdata->show_id        = bool_Sfalse;
  sdata->show_name      = bool_Sfalse;
  sdata->show_legend    = bool_Strue;
  sdata->auto_scaling   = bool_Strue;
  sdata->visual         = observation_Scolor_table_jet;

  observation_Sset(       5+x_off, 5+y_off, 300, 200, name,
                          observation_Skind_net_status,
                          &data, &obs, &status);
  x_off+=30; y_off+=30;
}

void ObsTargetOutput (pattern_Slist_id_t list_id)
{
  status_St     status;
  neuron_Sid_t  id;

  get_Sid_neuron (get_Soption_first, 0L, output_Slist, "", &id, &status);

  ObservationUseKind (observation_Skind_target_output);
  ObsSetArg (ObsTimeType, 0);
  ObsSetArg (ObsTimeStep, 10);
  ObsSetArg (ObsNeuron, id);
  ObsSetArg (ObsPatternListId, list_id);
  ObsSetArg (ObsColumn, 1);
  ObsSetArg (ObsShowDifference, 1);
  ObsSetArg (ObsShowErrors, 1);
  ObservationStart ();

}

void softmax(  long nr_outputs, out_St *gate_out )
{
  status_St              s;
  out_St                 sum;
  get_Sneuron_status_t   ns;
  neuron_Sid_t           neuron1;
  long                   i;

  sum=0.0;
  get_Sid_neuron(  get_Soption_first,(neuron_Sid_t)0, output_Slist,NULL,
                   &neuron1,&s);
  get_Sstatus_neuron( neuron1, &ns, &s );
  if( ns.sum <-80.0) ns.sum=-80;
  if( ns.sum >80.0) ns.sum=80;
  gate_out[0] = ns.sum;
  sum+=exp(gate_out[0]);
  for(i=1;i<nr_outputs;i++){
    get_Sid_neuron(  get_Soption_larger,neuron1, output_Slist,NULL,
                     &neuron1,&s);
    get_Sstatus_neuron( neuron1, &ns, &s );
    if( ns.sum <-80.0) ns.sum=-80;
    if( ns.sum >80.0) ns.sum=80;
    gate_out[i] = ns.sum;
    sum+=exp(gate_out[i]);
  }
  get_Sid_neuron(  get_Soption_first,(neuron_Sid_t)0,output_Slist ,NULL,
                   &neuron1,&s);
  gate_out[0]= (exp(gate_out[0])/sum);
  set_Sout_random( neuron1, 0L, out_Smix, out_Smax, gate_out[0],1.0, (random_St)2, &s);
  for(i=1;i<nr_outputs;i++){
    get_Sid_neuron(  get_Soption_larger,neuron1, output_Slist,NULL,
                     &neuron1,&s);
    gate_out[i]=exp(gate_out[i])/sum;
    set_Sout_random( neuron1, 0L, out_Smix, out_Smax, gate_out[i],1.0, (random_St)2, &s);
  }
}

void calc_SExpectation_step(   pattern_Slist_id_t list_id,
                               long               nr_experts,
                               status_St          *status)
{
  status_St              s;
  pattern_Slist_info_t   info;
  out_St                 inputs[INTERACT_MAX_PATTERN_INPUTS];
  out_St                   targets[INTERACT_MAX_PATTERN_TARGETS];
  out_St                   outputs[MAX_EXPERTS][INTERACT_MAX_PATTERN_TARGETS];
  out_St                   gate_outputs[MAX_EXPERTS];
  long                     nr_inputs,nr_outputs, nr_targets, nr_patterns;
  long                     pattern_id,expert_nr,i,j;
  double                   res1, res2, res3;
  pattern_Slist_get_info( list_id, &info, &s );
  nr_patterns = info.nr_patterns;
  for( pattern_id=1;pattern_id<=nr_patterns; pattern_id++){
    pattern_Slist_get_pattern(  list_id, pattern_id, inputs,&nr_inputs,
                                targets, &nr_targets, &s);
    /* calculate the response of the gate network */
    net_Sset_working_net( gate_id , &s);
    set_Sinput( inputs, nr_inputs, &s);
    calc_Sstart_evalu( 0L, hidden_Slist, &s);
    calc_Sstart_evalu( 0L, output_Slist, &s );
    softmax( nr_experts, gate_outputs);
    /* calculate the responses of the expert networks */
    for( expert_nr=0; expert_nr < nr_experts;expert_nr++){
      net_Sset_working_net( expert_id[expert_nr],&s);
      set_Sinput( inputs, nr_inputs, &s);
      calc_Sstart_evalu( 0L, hidden_Slist, &s);
      calc_Sstart_evalu( 0L, output_Slist, &s );
      get_Soutput_group( output_Slist, outputs[expert_nr],&nr_outputs, &s);
    }
    /* calculate the posterior probabilities h */
    for( expert_nr=0;expert_nr<nr_experts;expert_nr++){
      res1=0.0;
      for(j=0;j<nr_targets;j++)
        res1+=(targets[j]-outputs[expert_nr][j])*(targets[j]-outputs[expert_nr][j]);
       res1=exp(-(res1)/(2.0*expert_Variance[expert_nr]));
       res1*=1.0/(pow( sqrt(2.0*M_PI),nr_targets) *
                  pow( sqrt(expert_Variance[expert_nr]),nr_targets) );
      res1*=gate_outputs[expert_nr];
      res3=0.0;
      for(i=0;i<nr_experts;i++){
        res2=0.0;
        for(j=0;j<nr_targets;j++)
          res2+=(targets[j]-outputs[i][j])*(targets[j]-outputs[i][j]);
        res2=exp(-(res2)/(2.0*expert_Variance[i]));
        res2*= 1.0/(pow( sqrt(2.0*M_PI),nr_targets) *
                pow( sqrt(expert_Variance[i]),nr_targets) );
        res2*=fabs(gate_outputs[i]);
        res3+=res2;
      }
      res3=res1/res3;
      posterior[pattern_id-1][expert_nr]=res3;
    } /* posterior calculated */
  }
}

void calc_SMaximization_step(  pattern_Slist_id_t list_id,
                               long               nr_experts,
                               status_St          *status)
{
  status_St                 s;
  pattern_Slist_info_t      info;
  out_St                    inputs[INTERACT_MAX_PATTERN_INPUTS];
  out_St                    targets[INTERACT_MAX_PATTERN_TARGETS];
  out_St                    outputs[MAX_EXPERTS][INTERACT_MAX_PATTERN_TARGETS];
  out_St                    gate_outputs[MAX_EXPERTS],
                             gate_targets[INTERACT_MAX_PATTERN_TARGETS];
long                         nr_inputs,nr_outputs, nr_targets, nr_patterns;
long                         pattern_id,expert_nr,j,i;
double                       sumnom,sumdenom,res1;
pattern_Slist_get_info( list_id, &info, &s );
nr_patterns = info.nr_patterns;

/* update the experts variances */
for( expert_nr=0; expert_nr < nr_experts;expert_nr++){
   sumnom=0.0;sumdenom=0.0;
   for( pattern_id=1;pattern_id<=nr_patterns; pattern_id++){
      pattern_Slist_get_pattern(   list_id, pattern_id, inputs,&nr_inputs,
                                   targets, &nr_targets, &s);

      /* calculate the response of the gate network */
      net_Sset_working_net( gate_id , &s);
      set_Sinput( inputs, nr_inputs, &s);
      calc_Sstart_evalu( 0L, hidden_Slist, &s);
      calc_Sstart_evalu( 0L, output_Slist, &s );
      softmax( nr_experts, gate_outputs);
      /* calculate the responses of the expert network */
      net_Sset_working_net( expert_id[expert_nr],&s);
      set_Sinput( inputs, nr_inputs, &s);
      calc_Sstart_evalu( 0L, hidden_Slist, &s);
      calc_Sstart_evalu( 0L, output_Slist, &s );
      get_Soutput_group( output_Slist, outputs[expert_nr],&nr_outputs, &s);

      res1=0.0;
      for(j=0;j<nr_targets;j++)
         res1+=(targets[j]-outputs[expert_nr][j])*(targets[j]-outputs[expert_nr][j]);
      res1*=posterior[pattern_id-1][expert_nr];
      sumnom+=res1;
      sumdenom+=posterior[pattern_id-1][expert_nr];
   }
   sumdenom*=nr_targets;
   expert_Variance[expert_nr]=sumnom/sumdenom;
   if( expert_Variance[expert_nr] <0.0001 ) expert_Variance[expert_nr]=0.0001;
} /* end calculation of the new variances */

/* update weights of the experts */
for( expert_nr=0; expert_nr < nr_experts;expert_nr++){
      net_Sset_working_net( expert_id[expert_nr],&s);
      for( pattern_id=1;pattern_id<=nr_patterns; pattern_id++){
         pattern_Slist_get_pattern(   list_id, pattern_id, inputs,&nr_inputs,
                                      targets, &nr_targets, &s);
         set_Sinput( inputs, nr_inputs, &s );
         calc_Sstart_evalu( 0L, hidden_Slist, &s);
         calc_Sstart_evalu( 0L, output_Slist, &s );
         set_Starget( targets, nr_targets, &s );

      change_Slearn(   error_back_St, learn_rate*
                       posterior[pattern_id-1][expert_nr],
                       momentum, 0.0, 0.0, 0.0, &s );
      calc_Sstart_learn( 0L, output_Slist, &s );
      change_Slearn( error_back_St, learn_rate, momentum, 0.0, 0.0, 0.0, &s );
      calc_Sstart_learn( 0L, hidden_Slist, &s );
      }
  } /* update weights of the gate */
  net_Sset_working_net( gate_id,&s);
  change_Slearn( error_back_St, learn_rate*0.5,momentum*0.5, 0.0, 0.0, 0.0, &s );
  for( pattern_id=1;pattern_id<=nr_patterns; pattern_id++){
    pattern_Slist_get_pattern(   list_id, pattern_id, inputs,&nr_inputs,
                                 targets, &nr_targets, &s);
        set_Sinput( inputs, nr_inputs, &s );
        calc_Sstart_evalu( 0L, hidden_Slist, &s);
        calc_Sstart_evalu( 0L, output_Slist, &s );
        softmax( nr_experts, gate_outputs);
        for(i=0;i<nr_experts;i++){
          if( fabs((posterior[pattern_id-1][i]-gate_outputs[i]))>0.0001 )
            gate_targets[i]=   (posterior[pattern_id-1][i]-gate_outputs[i]) /
                               (gate_outputs[i]*(1.0-gate_outputs[i])) +
                               gate_outputs[i];
          else
            gate_targets[i]=posterior[pattern_id-1][i];
        }
        set_Starget( gate_targets, nr_experts, &s );
        if( s.all != status_Sok ) printf("\nTarget for Gate not set !!");
        calc_Sstart_learn( 0L, output_Slist, &s );
        calc_Sstart_learn( 0L, hidden_Slist, &s ); } for(j=0;j<nr_experts;j++){
        printf("\nNew Variance: %2.6f",expert_Variance[j]);
    }
}

void gen_Stest_epoch(   pattern_Slist_id_t   list_id,
                        long                 nr_experts,
                        int n, float         *error,
                        status_St            *status)
{
  status_St                 s;
  pattern_Slist_info_t      info;
  out_St                    inputs[INTERACT_MAX_PATTERN_INPUTS];
  out_St                    targets[INTERACT_MAX_PATTERN_TARGETS];
  out_St                    outputs[MAX_EXPERTS][INTERACT_MAX_PATTERN_TARGETS];
  double                    gen_outputs[INTERACT_MAX_PATTERN_TARGETS];
  out_St                    gate_outputs[MAX_EXPERTS];
  long                      nr_inputs,nr_outputs, nr_targets, nr_patterns;
  long                      pattern_id,expert_nr,i,j;
  FILE *fp;

  pattern_Slist_get_info( list_id, &info, &s );
  nr_patterns = info.nr_patterns;
  *error=0.0;
  fp=fopen("output.txt","w+");
  if( fp!=NULL){
    fprintf(fp,"Epoch: %d\n",n);
    for( pattern_id=1;pattern_id<=nr_patterns; pattern_id++){
      pattern_Slist_get_pattern(   list_id, pattern_id, inputs,&nr_inputs,
                                   targets, &nr_targets, &s);

        /* calculate the response of the gate network */
        net_Sset_working_net( gate_id , &s);
        set_Sinput( inputs, nr_inputs, &s);
```

```
            calc_Sstart_evalu( 0L, hidden_Slist, &s);
            calc_Sstart_evalu( 0L, output_Slist, &s );
            softmax( nr_experts, gate_outputs );

            /* calculate the responses of the expert networks */
            for( expert_nr=0; expert_nr < nr_experts;expert_nr++){
              net_Sset_working_net( expert_id[expert_nr],&s);
              set_Sinput( inputs, nr_inputs, &s);
              calc_Sstart_evalu( 0L, hidden_Slist, &s);
              calc_Sstart_evalu( 0L, output_Slist, &s );
              get_Soutput_group( output_Slist, outputs[expert_nr],&nr_outputs, &s);
            }
            for(j=0;j<nr_targets;j++)
                gen_outputs[j]=0.0;
            for(j=0;j<nr_targets;j++)
              for(i=0;i<nr_experts;i++)
                gen_outputs[j]+=(double) (outputs[i][j]*gate_outputs[i]);
            for(j=0;j<nr_targets;j++){
              (*error)+=(targets[j]-gen_outputs[j])*(targets[j]-gen_outputs[j]);
              fprintf(fp,"%f %f ",targets[j],gen_outputs[j] );
            }
            for(i=0;i<nr_experts;i++){
                for(j=0;j<nr_targets;j++)
                fprintf(fp,"%f ",outputs[i][j]);
            }
            for(j=0;j<nr_experts;j++)
              fprintf(fp,"%f ",(float)gate_outputs[j]);
            fprintf(fp,"\n");
          }
          *error=*error/nr_patterns;
          fprintf(fp,"\n");
          fclose(fp);
        }else{
          *error=1.0;
        }
    }

    void main( int argc, char *argv[] )
    {
      status_St           status;
      pattern_Slist_id_t patterns, trn_patterns, tst_patterns;
      long                nr_experts, nr_exphiddens,nr_gatehiddens;
      float error=0.0;
      int  i,n;
      char str[30];
      FILE *fp_log;
      interact_Sinit( init_mode_Sno_network, "", &status );

      if( argc==5 ){
        patterns = MyLoadPatterns( argv[1] );
        MySplitPatterns( patterns, &trn_patterns, &tst_patterns );
        nr_experts = atoi( argv[2] );
        nr_exphiddens = atoi( argv[3] );
        nr_gatehiddens= atoi( argv[4] );
        printf("\n#expert: %i",nr_experts);
        printf("\nCreating Gated Experts Network, Please Wait...");
    gen_Screate_net( patterns, nr_experts, nr_exphiddens,nr_gatehiddens, &status);
    net_Sset_working_net( gate_id , &status);
    observation_Sdisable( all_Sobservations,&status);
    for(i=0;i<nr_experts;i++){
        sprintf(str,"Expert %2i",(i+1) );
    }
    getchar();
    fp_log=fopen("log.txt","w+");
    fclose(fp_log);
    n=0;
    printf("\nLearnspeed: %1.2f Learnmomentum: %1.2f",learn_rate, momentum);
    for(;;){
      printf("\n#%i  : E-step",n);
      calc_SExpectation_step(trn_patterns, nr_experts, &status );
      printf(" -> M-step");
      calc_SMaximization_step( trn_patterns, nr_experts, &status );
      if( n % 2 == 0 && n!=0 ) {
        gen_Stest_epoch( tst_patterns, nr_experts, n,&error, &status  );
        printf("\nError: %f",error);
        learn_rate-=0.02;
        momentum+=0.02;
        if( learn_rate<=0.1) learn_rate=0.1;
        if( momentum>=0.3 ) momentum=0.3;
        printf("\nLearnspeed: %1.2f Learnmomentum: %1.2f",learn_rate, momentum);
        fp_log=fopen("log.txt","a");
        fprintf( fp_log,"#%d Learnspeed: %f Momentum: %f Error: %f",n,
                  learn_rate,momentum,error);
        for(i=0;i<nr_experts;i++)
          fprintf(fp_log," %f ",expert_Variance[i] );
        fprintf(fp_log,"\n");
        fclose(fp_log);
      }
      n++;
    }
    fclose(fp_log);
    printf("\nReady...");
  }else{
    printf(   "\nUsage: expert <pdbf-file> <nr-of-experts> <expert-hiddens>
                <gate-hiddens>\n");
  }
  getchar();
  interact_Sterminate( init_mode_Sstop, &status );
} /* end main */
```