ISSN 0280–5316
ISRN LUTFD2/TFRT--5476--SE

Classification of System Dynamics Using Neural Networks

Johan Eker
Stefan Vlachos

Department of Automatic Control
Lund Institute of Technology
August 1993

Department of Automatic Control
Lund Institute of Technology
P.O. Box 118
S-221 00 Lund Sweden

Document name
MASTER THESIS

Date of issue
August 1993

Document Number
ISRN LUTFD2/TFRT--5476--SE

Author(s)
Johan Eker and Stefan Vlachos

Supervisor
Professor Karl Johan Åström

Sponsoring organisation
NUTEK 92-04014P

Title and subtitle
Classification of System Dynamics Using Neural Networks

Abstract

The thesis is a part of a project which aims at developing highly autonomous controllers. The task of rough classification of system responses is considered. Neural networks are trained to classify system dynamics from step- and impulse responses. Two types of networks are discussed, the backpropagation net and Kohonen's self-organizing feature map. The theory behind these algorithms are presented. A method for normalization of the inputs in time and space is given. This is essential for robust classification. Different net configurations, training methods, and noise sensitivity are investigated. It is shown that the networks can be used as classifiers. Rules of thumb for choosing net structure and parameters are given.

Key words
Autonomous controller, Neural networks, System dynamics, Classification, Transient responses, Backpropagation, Kohonen.

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title
0280–5316

ISBN

Language
English

Number of pages
53

Recipient's notes

Security classification

Contents

1. Introduction

This work is a part of a project which aims at developing controllers that are highly autonomous, in the sense of being self-governing. Intelligent controllers is a closely related concept. It is then desirable to let the controller itself perform most of the work that now is done by process operators and engineers. This includes tuning and adaptation of parameters, diagnosis and supervision. The dynamics of the controlled process are then naturally of great interest. Alas, they are not always known entirely, or sometimes not known at all. It is also possible that the dynamics vary over time when a process goes through different phases. If a model of the process exists, then automatic classification of the real process can be used to compare model with reality and thus be used for diagnosis and fault detection. A classification procedure will be used to obtain a rough categorization of the dynamics before applying more detailed analytical techniques.

Also, when discussing processes with unknown dynamics, who has more knowledge of the process than the controller that is controlling the process? If the controller is able to classify each step- or impulse response then this information could be used when another similar process is implemented. It would also be possible for the process operator to ask the controller how it was performing: "Well, today I am a bit oscillating but yesterday I was monotonous", which could be valuable information. It might also be possible to examine load disturbances and noise. Knowledge about the nature of a disturbance would make it easier to take the correct measures to minimize the effects of the disturbance. Having this built into a controller would among other things vouch for better control of the process and a broader base for fault diagnosis. In this thesis, we will only examine classification of system dynamics, but with the knowledge gained from doing this, it will probably not be hard to make similar kinds of classifications for disturbances.

The purpose of the thesis is to investigate if neural networks can be used to roughly classify dynamical systems. There are several typical categories such as monotonous, oscillating, and non-minimum phase. It is of course of great interest to find out if a system is stable or unstable, but this can pretty easily be done without the use of neural networks. We have therefore limited our studies to systems that are known to be stable.

The problem of classification is basically one of pattern recognition. A human can look at a pattern, for example a step response from some process, and easily be able to recognize it as, for example, an oscillating system. Even if the signal is noisy, or parts of it is missing, it is still no big deal for the trained eye to recognize the basic characteristics of a system from the plot of the step response. Good tolerance of noise, the ability to generalize, and the ability to make do even when some pieces of information are missing, are some of the typical properties of neural networks. Neural networks have been used in such diverse applications as pattern- and speech recognition, data compression, solving optimization problems (including the traveling salesman problem) and robot control.

There are primarily two kinds of neural networks, depending on the way they are trained: nets trained with supervised- and unsupervised learning. Supervised learning means presenting the net with inputs and the corresponding desired outputs. Unsupervised learning means presenting the net with inputs, and letting it learn by itself. We have examined one version of each type, the backpropagation net with supervised learning and the Kohonen net with unsu-

pervised. Both types have been trained and examined on essentially the same kinds of systems, but with some variations to examine the specific properties of each net.

We have examined step and impulse responses for some typical systems. We have primarily been interested in finding out if system classification is possible, and if one type of response is easier to classify than the other. As we shall see, rough classification is possible with both step and impulse responses though it works best with the latter. Sensitivity to noise has of course been investigated, and found to be low. The sampled information containing the step- and impulse responses can be treated in some different ways before being presented to the network, that is it can be normalized. This has also been examined.

2. Neural Networks

Computers of today are indeed very powerful machines, that can make millions of arithmetic operations every second. Any computer can present an almost unlimited amount of decimals for π in a short time, a feat that would put a human being firmly into the Guinness Book of Records. But when it comes to recognizing your own mother, or even her handwriting, the traditional computer falls far behind the most moderately intelligent human. Recognizing images and patterns is a difficult task because they contain a lot of information, and it often requires that we can separate important information from unimportant. Two images of the same person are seldom or never exactly the same. Different light, background, photographic angle and facial hair configuration can make the same person look rather different. Yet a human is usually able to tell if it is the same person in the pictures. Our brains work in mysterious ways, but we know that there are a lot of interconnected neurons that work in parallel in a way that no computer does.

An artificial neural network is an attempt at modeling the way we believe the human brain works, and can be implemented in software on a computer or in hardware. There exists different kinds of artificial neural networks, each emphasizing different aspects of how we think the brain learns and works. Neural nets have successfully been used for a lot of different tasks but each task has been very specific in some sense, for example recognizing handwritten letters or "understanding" speech. We are limited to using rather few neurons compared to the number of neurons in the brain, due to storage and memory limitations. It probably would not be possible to build anything approaching the human brain anyway, but for simpler tasks such as classifying system dynamics, neural networks can be useful.

Before a neural network can be put to work, we have to train it, just like a human has to be trained before being able to perform a certain task. The network is usually trained on a set of examples that are somehow representative of the problem to be solved. This is usually called learning. The net is then supposed to be able to come up with an answer when presented with an input, even if that specific input was not in the collection of examples. Basically there are two types of learning: supervised and unsupervised.

Supervised learning is sometimes called learning with a teacher. The net is presented with inputs and the corresponding desired outputs. The output from the net is calculated and the difference between the obtained and the desired output is computed. The learning then takes place on the basis of this

error. Learning with a teacher is, for example, like learning the words of a foreign language. You sit with the vocabulary and try to learn how to spell the French word for "thank you" and try "mercy", which you see is wrong when you check the glossary, and correct yourself to "merci". Learning has taken place.

Unsupervised learning has no teacher. A set of examples are presented to a network which then figures out a way to classify them. This is much like how humans learn from experience. We might for example watch a Sylvester Stallone movie with all of its mindless violence and non-existing plot, and then a similar Chuck Norris movie. We will certainly consider them as being almost the same film, along with a host of similar movies that we have wasted our youth upon. If we then try to broaden our minds and decide to see Fried Green Tomatoes, we will notice that it has very little in common with the two earlier films. We have, without anyone telling us how, classified the films into two different categories: the excessive and entertaining violence category, and the intelligently sensitive category. Learning how to categorize the films has occurred by just watching them and seeing similarities and differences.

In this section we will describe the two methods, the backpropagation net and the Kohonen net, we used to classify transient responses. The algorithms will be motivated mathematically as well as explained verbally. A discussion about the need for normalizing the vectors in the Kohonen net will be given.

Supervised Learning

Supervised learning is the original neural network training method and it means presenting the net with an input and a desired corresponding output. Usually you use a whole set of input–output pairs and train the net until you get the desired result. There are many different kinds of networks that use supervised learning but in this report we are going to concentrate on the backpropagation network. The algorithm behind this network was independently formalized by [Werbos, 1974], [Parker, 1986], and [McClelland and Rumelhart, 1986] and is based upon an earlier method called the delta rule. The backpropagation algorithm is also sometimes referred to as the generalized delta rule. The backpropagation network (BPN) has been found useful in many areas, and there are existing applications in the fields of pattern recognition and data compression using the algorithm. A 2-layer backpropagation net is shown in figure 2. The net actually has three layers; an input layer, a hidden layer and an output layer but the input layer is usually not considered as a part of the neural net, hence the name 2-layer network.

What tasks can a BPN be taught to do? It is usually said that the BPN is good at generalization but will not extrapolate very well. Generalization is the net's ability to, given several input vectors belonging to the same class, key off significant similarities in the input vectors and thus learn to classify vectors that it was not trained on. For example a BPN can easily be trained to perform addition of integers. To do this we start with a training phase where we present input–output pairs, like $1 + 2 = 3$ and $3 + 4 = 7$, to the net. After a number of training cycles the net will be able to without errors sum up the integer pairs from the training batch. If we now use set of inputs, other than the ones used in the training phase and apply it to the net, we find that it will correctly add the sum. This ability to work with previously unknown input data is called generalization . But what happens if we present the net with two integer, whose sum will be larger than any of the sums in the training batch, or if you instead of adding two integers try to add two real numbers?

This will unfortunately not work neither for the case where the sum is large, nor for the case where we use real numbers, and this is a result of the BPN not extrapolating very well.

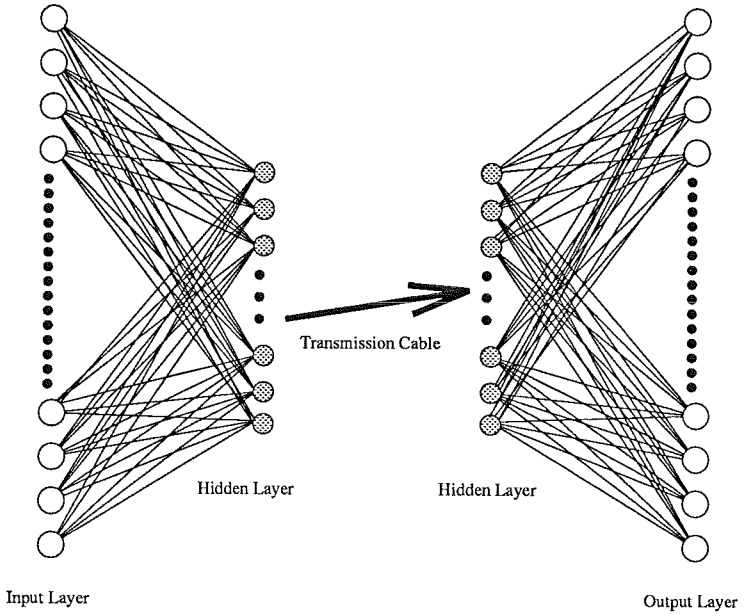

Figure 1. A backpropagation net designed for data compression. To the right the input data is compressed to the network's internal data representation which is held in the hidden layer neurons. The compressed data is transmitted to a receiving network that will decompress it.

The backpropagation networks can also, as mentioned earlier be used for data compression. When doing this you design a net where the input– and output layers are of the same size and where the hidden layer is considerably smaller, like in figure 1. The compression rate is given by the ratio between the size of the input layer and the size of the hidden layer. To manage a task like this the net must be trained with data similar to the one it is going to operate on. For example if we would like to compress a Whitney Houston song we would have to train the net with a couple of her latest hit songs and let the net build an internal representation of the most common beats and howlings, which are actually not that many. A net like this would for example probably not work very well if you tried to use it to compress Beethoven's fifth symphony. The net has been trained to recognize features common in Whitney Houston songs and would try find those characteristics in the classical masterpiece. Since these features are not very typical for Beethoven the resulting compressed data would not be sufficient to reconstruct the symphony.

Backpropagation

The backpropagation algorithm gives a method for changing the weights in any feedforward net. To train the net you use a set of input-output pairs. In the first phase you apply the input to the input neurons and it propagates through each layer until an output is generated. In the second phase you compare the output with the desired result and an error value is computed. The error is now propagated backwards through the net adjusting the weights by an amount proportional to the total error. This two phase cycle is repeated until the error value has reached an acceptable level. You will find a more

comprehensive description of the algorithm in [Freeman and Skapura, 1991] or [Hertz *et al.*, 1992].

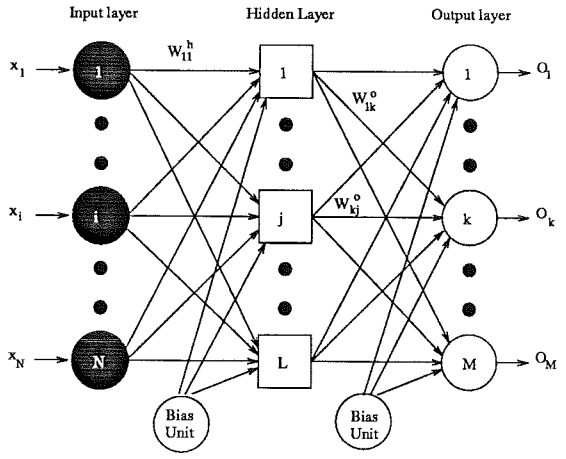

Figure 2. A two-layer backpropagation net.

An input vector, $x = (x_1, \ldots, x_N)^T$ N is the number of input neurons, is applied to the input layer of the network. The input propagates to the hidden layer and the net input to the jth hidden unit is

$$I_j^h = \sum_{i=1}^{N} w_{ji}^h x_i + b_j^h$$

where w_{ij}^h is the weight on the connection from the i input unit, and b_j^h is the bias term. The bias term can be considered as the input from a unit with the constant value of 1 and the weight b_j^h. The output from the node is then

$$O_j^h = f_j^h(I_j^h)$$

The superscript 'h' stands for hidden layer unit. f is the transfer function for the neuron, usually a sigmoid function such as $f(x) = 1/(1 + e^{-x})$, which is depicted in figure 3. Another typical sigmoid is $f(x) = \arctan(x)$. We will get

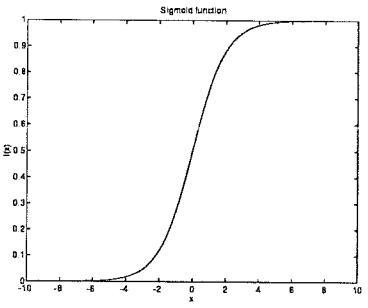

Figure 3. The sigmoid function $f(x) = 1/(1 + e^{-x})$. The name sigmoid comes from the s-shape of the function.

similar equations for the output layer

$$I_k^o = \sum_{j=1}^{L} w_{kj}^o O_j^h + b_k^o$$

$$O_k^o = f_j^h(I_k^o),$$

where L is the number of neurons in the hidden layer. The subscript 'o' stands for output layer units. The error of the output is calculated as

$$E = \frac{1}{2}\sum_{k=1}^{M}(y_k - O_k^o)^2$$

and

$$\frac{\partial E}{\partial w_{kj}^o} = -(y_k - O_k^o)\frac{\partial f_k^o}{\partial I_k^o}\frac{\partial I_k^o}{\partial w_{kj}^o} = -(y_k - O_k^o)f_k'^o(I_k^o)O_j^h$$

where $y = (y_1, \ldots, y_M)^T$ is the desired output vector and M is the number of output neurons. We wish to adjust the weights in the direction of the negative gradient of the error E. This method is called the gradient-descent technique. Thus the weights on the output layer are updated according to

$$w_{kj}^o(t+1) = w_{kj}^o(t) + \Delta w_{kj}^o$$

where

$$\Delta w_{kj}^o = -l_r\frac{\partial E}{\partial w_{kj}} = l_r(y_k - O_k^o)f_k'^o(I_k^o)O_j^h$$

The factor l_r is called the learning rate.

To be able to calculate the weight changes for the hidden layer we rewrite the error function E.

$$E = \frac{1}{2}\sum_k(y_k - O_k^o)^2 = \frac{1}{2}\sum_k(y_k - f_k^o(\sum_j w_{kj}^o O_j^h + b_k^o))^2$$

and

$$\frac{\partial E}{\partial w_{ji}^h} = -\sum_k(y_k - O_k^o)\frac{\partial O_k^o}{\partial I_k^o}\frac{\partial I_k^o}{\partial O_j^h}\frac{\partial O_j^h}{\partial I_j^h}\frac{\partial I_j^h}{\partial w_{ji}^h} = -\sum_k(y_k - O_k^o)f_k'^o(I_k^o)w_{kj}^o f_j'^h(I_j^h)x_i$$

Again we adjust the weight for the hidden layer in proportion to the negative gradient of the error E. The update equation for the hidden layer weight is thus:

$$w_{ji}^h(t+1) = w_{ji}^h(t) + \Delta w_{ji}^h$$

where

$$\Delta w_{ji}^h = -l_r\sum_k(y_k - O_k^o)f_k'^o(I_k^o)w_{kj}^o f_j'^h(I_j^h)x_i$$

Changing the weights in the hidden layer is the last step in the second phase of the backpropagation cycle.

Improved Backpropagation

To further improve the backprogagation algorithm we use a technique called momentum. With momentum you keep the weight changes moving in the same direction and prevents the network from converging to local minimum. This is done by adding a fraction of the previous weight change when calculating the new weight w_{kj}. The update equation now becomes

$$w_{kj}(t+1) = w_{kj}(t) + \Delta w_{kj} + \alpha\Delta w_{kj}(t)$$

The momentum parameter α must be between 0 and 1.

Adaptive parameters is another improvement. To decrease the learning time we use an adaptive learning rate. The idea is to check if the weight update did exceed the old error by more than a predefined value. If it did, the process is moving in too large steps and the learning rate must be reduced. It is also possible that the learning rate is increased if there has been several steps in a row that have decreased the error function.

Unsupervised Learning

The supervised learning approach described above uses sets of inputs and corresponding desired outputs when training a neural network. This is something like learning the vocabulary of a foreign language: "merci" means "thank you", "au revoir" means "good bye", and so on. However, if we consider the way we learn a lot of other things, like when we learned to speak as little children, this was done without a teacher. A lot of examples were presented to us in the form of other people talking to us or to each other. Eventually we understood the meaning of the sounds, and after some training we could even produce some intelligent speech ourselves. This is unsupervised learning. When training a neural network with an unsupervised learning technique, we present a selection of examples to a network which then organizes itself according to an adaptation algorithm.

There are several unsupervised learning methods. We will use Kohonen's network model. The resulting network will be referred to as a Self-Organizing Feature Map. In this section we will describe the model, discuss the need for normalization of the input vectors, and present a simple example. Self-Organizing Feature Maps have been used for speech recognition (Kohonen's neural phonetic type writer [Kohonen, 1988]), robot control, solving the traveling salesman problem, and much more.

Self-Organizing Maps

The name Self-Organizing Maps (SOM) comes from the nets' ability to classify and organize input data by themselves. When the net is trained, its weight matrix will form a surface in weight space that has adapted the shapes of the training vectors. A common approach to unsupervised learning is competitive learning. In competitive learning only one output unit can be active at a time. When an input vector has been applied to the network the output vector is calculated. The output unit with the greatest value wins the competition and becomes active. The winning unit is the one whose weight vector most closely matches the input vector and it is only this unit which is allowed to learn. The learning process means adjusting the winning unit's weight vector towards the input vector. The amount with which the weights are changed is set by a factor called learning rate.

A feature map is a SOM which is capable of not just organizing and classifying data but it is also able to map *nearby input patterns to nearby output units*, e.g. two similar input vectors P_1 and P_2 will be mapped to units located close to each other. If we let P_1 and P_2 get closer and closer the output will eventually coincide. This property is called *topology preservation.* The feature map can be of one or two dimensions depending on the relationship between the input vectors. A one-dimensional feature map is simply a linear array of neurons. A two-dimensional feature map is shown if figure 4. The connection geometry between the neurons can vary. In figure 4 the neurons are arranged in a square formation. Another type of two-dimensional arrangement

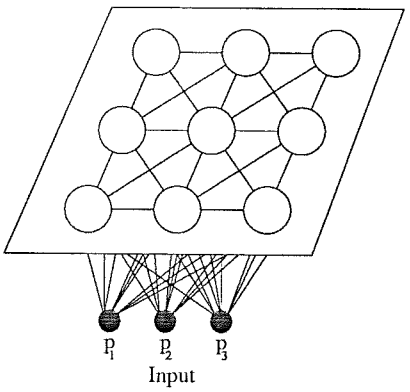

Figure 4. A two-dimensional feature map where the output neurons are arranged in a grid. Similar input patterns will be mapped to nearby output neurons.

is a hexagonal geometry.

With plain competitive learning the net will not be able to perform feature mapping. To do this we need to add a neighbourhood function. The neighbourhood function gives the neighbours of the winning unit. We use this to adjust not just the winning unit's weight vector but also the neighbouring units'. The neighbourhood size decreases over time as the training progresses. In figure 5, the neighbourhoods of a two-dimensional square feature map and a one-dimensional linear feature map are shown. In the following when we examine unsupervised classification, we will work with the feature maps, which will be referred to as self-organizing feature maps or SOFMs.

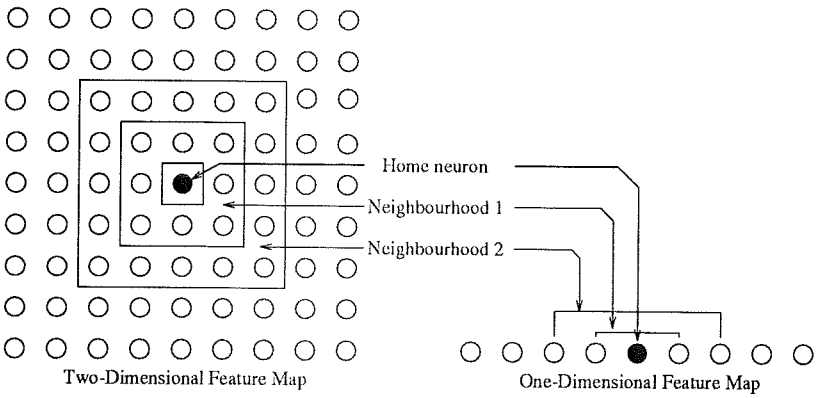

Figure 5. The output layers from a two-dimensional (left) and a one-dimensional (right) feature map, where neighbourhoods are shown. The home neuron is the winning neuron.

Creation of the Self-Organizing Feature Map

The SOFM consists of a number of units (neurons) which are all connected to the input. Each neuron receives the same input vector, $P = (p_1, p_2, \ldots, p_n)^T$, where p_i is a scalar. When training a net, we first choose a vector from a batch of supplied training vectors. This is usually done in a random manner, and the probability of choosing a training vector P is given by a probability density $\rho(P)$. For simplicity, one can have the same probability of choosing each vector in a batch and then put several identical vectors in the batch if we want the probability of choosing this particular vector to be greater than average. This vector is processed by the neurons to form an output. The simplest way of

processing is forming a linear function of the scalar inputs, giving the output from the ith neuron:

$$o_i = \sum_{j=1}^{n} w_{ij} p_j \tag{1}$$

Here w_{ij} is the weight from the jth scalar input to the neuron. It is these weights that are updated during training. Writing $w_i = (w_{i1}, w_{i2}, \ldots, w_{in})^T$, we see that (1) is nothing less than the inner (scalar) product $w_i^T P$. w_i is the weight vector of the ith neuron. We thus compare w_i with P, and we wish to find the neuron with the weight vector that is closest to the input vector. Comparing the neurons is done by forming a weight matrix, W with the different neurons' weight vectors as rows. We then take the matrix product WP which gives an output vector, $O = (o_1, o_2, \ldots, o_m)^T$ where the ith element is the inner product of w_i with the input P, and m is the number of neurons. The competition then consists of finding the largest element in O. This is schematically illustrated in figure 6. The necessity of normalization is discussed below.

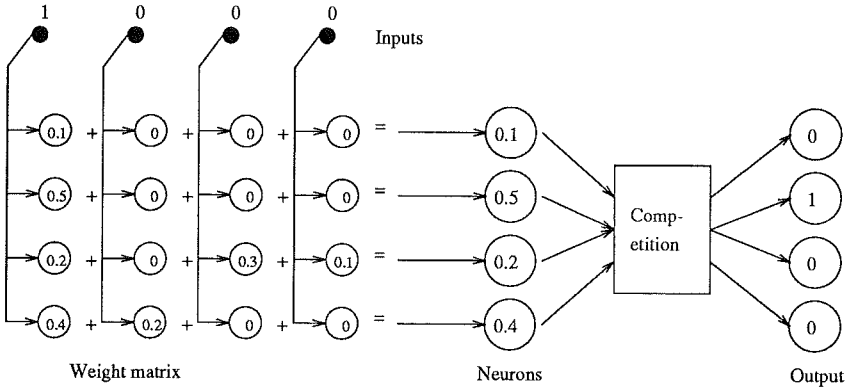

Figure 6. An illustration of a self organizing map with four inputs and four neurons. The inputs are compared with each neuron's weight vector through scalar multiplication, giving the neurons their value. The competition then assigns a one to the neuron with the largest value and zero to the others.

After finding the winning neuron we want to apply an adaptation process that makes the weights of each neuron converge to such values that every neuron becomes sensitive to a particular kind of input. The adaptation equation for the winning neuron can have the form

$$\frac{dw_{ij}}{dt} = \alpha(t)\{o_i(t)p_j(t) - \gamma[o_i(t)]w_{ij}\} \tag{2}$$

[Kohonen, 1989], where $\alpha(t)$ is the learning rate, which is a decreasing function of time to guarantee convergence of the weights to a unique limit. $o_i(t)$ is as we saw the output from the ith neuron. $\gamma(\cdot)$ is the lateral interaction between the neurons. In the brain -which we try to mimic- the amount interaction has the form of a Mexican hat as in figure 7. With the vocabulary used above, the neurons that are close to the winning are strongly excited, that is, updated with nearly the same amount as the winning. Neurons a little further away are inhibited: the lateral interaction is negative. Neurons far away from the winning have a small positive interaction.

Next we turn to the problem of preserving the topology of the input patterns. This means mapping similar input patterns to nearby output units.

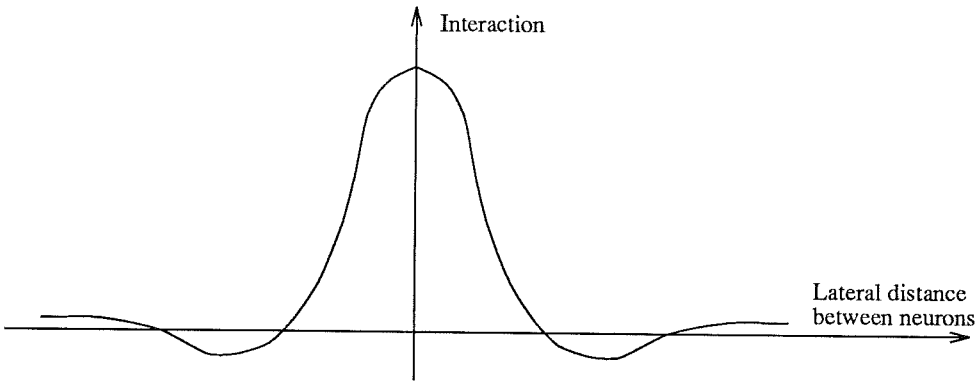

Figure 7. The lateral interaction of the neurons in the brain. The winning neuron is in the middle.

Similar here means that they look alike in some way, not that they are located next to each other in a batch of training vectors. Two stable and monotonous step responses with different rise times should for example be mapped to neurons close to each other, whereas an oscillating step response should be mapped to a neuron that is further away from the stable monotonous ones. This is done by updating not only the winning neuron c, but also the neurons within a neigbourhood N_c, of a certain radius from c. If we simplify (2) by taking $o_i = 1$ for all neurons inside N_c and $o_i = 0$ for those outside, and $\gamma(0) = 0, \gamma(1) = 1$ equation (2) will have the following simple form:

$$\frac{dw_{ij}}{dt} = \alpha(t)\{p_j(t) - w_{ij}(t)\} \qquad i \in N_c \tag{3}$$

which is more like a top-hat like function, as in figure 8. The neurons outside

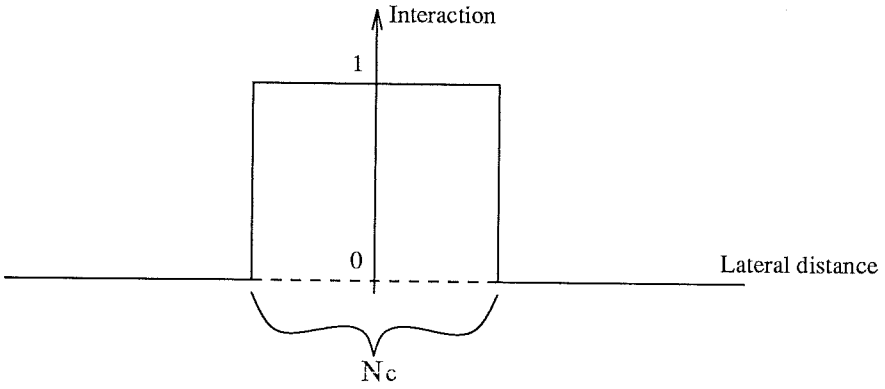

Figure 8. The lateral interaction of the neurons in our simplified model. The winning neuron is in the middle.

of N_c will not have their weights changed. The shape of the neighbourhood is of course dependent on the dimensionality of the SOFM. In this work we will only study one-dimensional feature maps, so N_c in (3) will be linear as seen to the right in figure 5.
To summarize the updating procedure in discrete-time:

1. Choose a random vector from a batch of training vectors.
2. Find the weight vector $w_i(k)$ that matches the input P(k) the best.

3. Update the weights:

$$\begin{aligned} w_i(k+1) &= w_i(k) + \alpha(k)\{P(k) - w_i(k)\} \quad \text{for } i \in N_c \\ w_i(k+1) &= w_i(k) \qquad\qquad\qquad\qquad\quad \text{otherwise.} \end{aligned} \tag{4}$$

This will be referred to as **Kohonen's algorithm.**

What we do when we us Kohonen's algorithm is map the n-dimensional input vectors onto the one- or two- dimensional neural network via the weight vectors. The mapping

$$\Phi_w : V \mapsto A, \qquad P \in V \mapsto \Phi_w(P) \in A$$

where $\Phi_w(P)$ is defined as finding the neuron that wins the competition, which can be formalized as

$$\|\Phi_w(P) - P\| = \min_{i \in A} \|w_i - P\| \tag{5}$$

is illustrated in two dimensions in figure 9, along with a schematical illustration of Kohonen's algorithm. Finding the winner in the way described by (5) versus through comparing scalar products is discussed in the section about normalization on page 15. The mapping $\Phi_w(P)$ is a neural map from the in-

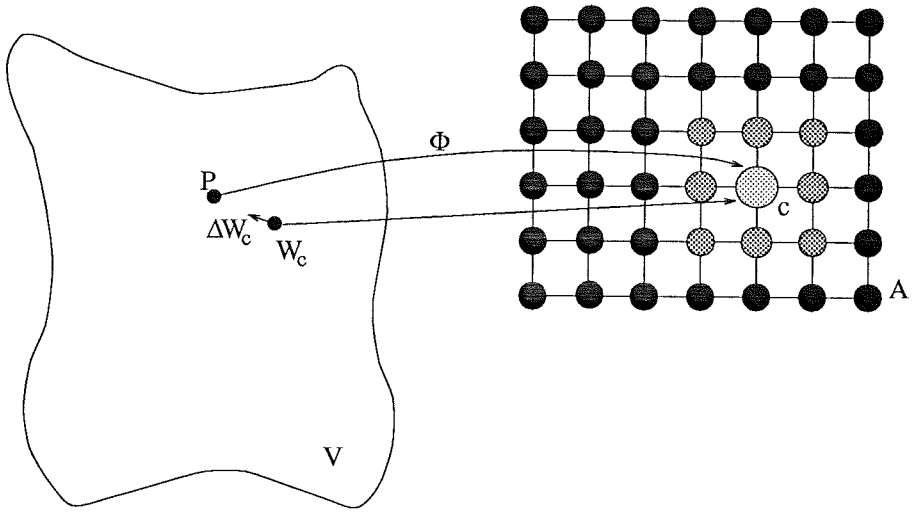

Figure 9. The updating of the weight vectors in Kohonen's algorithm. The winning neuron c and its closest neighbours have their weight vectors adjusted towards the training vector P. Only w_c is shown. The magnitude of the adjustment Δw_c is determined by the learning rate $\alpha(k)$.

put space V to the neuron lattice A. The adaptation algorithm, in words, takes the following steps (see figure 9). Choose an input vector P from the n-dimensional manifold V (step 1). Find the winning neuron c in the neuron lattice A (step 2). Update the winning neuron and its neighbours' weight vectors, that is shift w_c towards P with the amount Δw_c which is determined by the learning rate (step 3).

The neighbourhood N_c is a function of time, as is the learning rate α. Exactly what functions can best be determined by experimenting. Linearly decreasing functions seem to work well. It is useful to realize that the formation of the map consists of two stages: the initial formation of the correct order, and the convergence of the weights to their final values. The convergence of

the weight values may take several times as many steps (even up to 100 times as many) as the initial formation. Therefor the size of N_c should decrease more rapidly than the value of α, and reach its minimum of 1 well before the end of the training. α should reach zero at the last training cycle. The initial size of the neighbourhood is best set to cover a large part of the net. α is typically set to 0.01 or smaller to start with. The weight matrix is initialized with small random numbers.

Do we know what will happen with a net if we apply the algorithm in (4)? Kohonen states that every neuron will become maximally sensitive to one particular training input P. He also proves in the case of a one-dimensional input that as $k \rightarrow \infty$, the weight values w_i will become ordered, and that once they are ordered they will remain so. The fraction of neurons that become sensitive to a certain kind of input, approximates the relative occurrence of this kind of pattern in the training batch. So if you for example have many monotonous vectors in the training batch, there will be many monotonous weight vectors. This ability to map statistical distributions, however, works well only with large networks [McCord Nelson and Illingworth, 1991].

The interested reader will find that the training procedure is very well described in [Ritter *et al.*, 1992]. The book contains some elucidating illustrations on how training vectors are mapped to the weight space. It also describes a few examples on interesting applications of Kohonen nets. [Hecht-Nielsen, 1989] is intended as a text book in introductory neurocomputing and contains material on both supervised and unsupervised learning, as well as suggestions on implementation. Another book that is more application- and implementation oriented is [Freeman and Skapura, 1991].

Normalization of the Input Vectors

The need for spatial normalization is obvious when you consider exactly what the feature map does in training and operation. The input vector is multiplied with the weight matrix, W. This means taking the scalar product between the rows in W with the input vector. The input is then classified as corresponding to the row which gave the largest scalar product - this row wins the competition. Now, there are two ways for this scalar product to become large: either the row in W has large elements, or the row is nearly parallel to the input vector. We are interested in finding the row that is most parallel to the input, because this means that the elements in the two vectors are very much alike, and thus the corresponding patterns are similar.

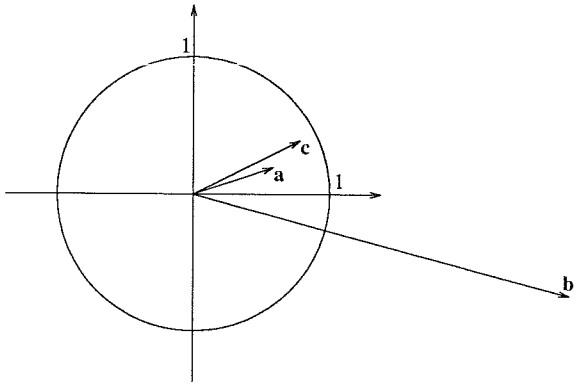

Figure 10. Three vectors that are not normalized.

What happens if the vectors are not of the same length? Consider the

two-dimensional vectors in figure 10. Let **a** and **b** be the vectors used to train the net. **b** has larger elements than **a** which is shown by its greater length. The initial weights in W are chosen as small random numbers. When training the net, the winning unit's weight vector is updated to look like the input. In the beginning of the training the neighbours of the winning unit will also be updated. When one row in W has won with **b** as input a couple of times, its weights will have become so large that this row will always win. Thus the net will only be able to give one output, namely the one corresponding to **b**. When applying **c** to the net trained with **a** and **b**, the net will tell you that the input was of type **b** even though **a** is much more parallel to **c**. With this discussion in mind, it is natural to perform all training and evaluation with vectors of the same (Euclidean) length. If nothing else is stated, all input vectors are scaled to unit length, i.e. a vector **a** is divided by $\sqrt{a^T a}$.

The process of finding the winning neuron is often described as finding the weight vector that minimizes the distance to the input vector, that is finding the index c for which

$$\|P(k) - w_c(k)\| = \min_i \{\|P(k) - w_i(k)\|\} \tag{6}$$

This procedure does not require that the vectors are normalized. However, if we are to use a matrix representation of the weight vectors as in the procedure described above, we need to normalize all concerned vectors to unit length. The procedure of taking the inner product and finding the greatest value is equivalent with (6) if the vectors are normalized.

A simple net

A simple example is used to illustrate unsupervised learning. Consider a one-dimensional feature map with 12 output neurons. In figure 11 the four training vectors $P_j = (p_{j1}, \ldots, p_{j100})^T$, $j = 1 \ldots 4$, are shown. The output from each

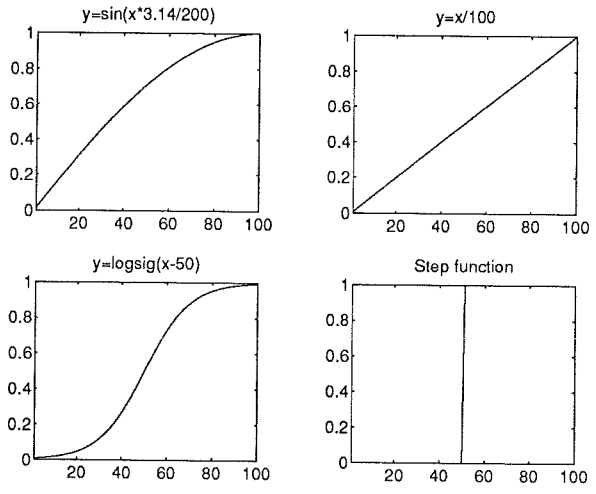

Figure 11. The training vectors for the example net before normalization.

neuron is calculated as

$$o_i = w_i P$$

where w_i is the weight vector of the ith neuron, $w_i = (w_{i1}, w_{i2}, \ldots, w_{i100})^T$, $i = 1 \ldots 12$. The output from the net thus becomes

$$O = WP$$

where W is the weight matrix, which has each neuron's weight vector as a row:

$$W = \begin{pmatrix} w_{11} & w_{12} & \dots & w_{1n} \\ w_{21} & w_{22} & \dots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{22} & \dots & w_{mn} \end{pmatrix}$$

and O is the output vector

$$O = \begin{pmatrix} o_1 \\ o_2 \\ \vdots \\ o_m \end{pmatrix}$$

with $n = 100$ and $m = 12$. In figure 12 the weight matrix is shown at its initial state and after some different number of training cycles. The weight vectors are ordered along the y-axis and the index of each element is on the x-axis, and the value on the z-axis. We can clearly see how each weight vector gradually adapts the values of the training vectors, and how the vectors become ordered in a certain manner. Note however that the ordering differs for each figure. This is explained below.

The initial learning rate was 0.15, which is why we only need 500 cycles to find the approximate shape of the weight vectors. As discussed above, a significantly lower learning rate should be used to get really useful nets, but what we have here is sufficient for example purposes. Also, the training vectors are rather distinct from each other. The training was restarted for every simulation, which is why the nets are not ordered in exactly the same way after different amounts of training. The ordering process goes on for a longer time the more training cycles that are used, so the net trained for 500 cycles is more thoroughly ordered. Moreover, there is no essential difference if the step is furthest to the left or to the right. The main thing is the internal order between the different vectors.

3. Classification with Neural Nets

In this section we will present the common factors for the supervised learning and the unsupervised learning approach when classifying transient responses of dynamical systems with neural networks. The different systems used to train and test the nets will be defined and illustrated, along with a description of how they were generated. Noise is always present to some extent when measuring signals in real life, so we are interested in how the nets perform on noisy responses. Here we will describe how the noise was generated and applied. To make systems that have essentially the same properties look more alike, we need to perform some kind of normalization. For time normalization we used a method based on the average residence time, which is also described in this section. To complete the normalization we scale the vectors to unit length.

The Example Systems

We used nine different step responses to train the nets. These are shown in figure 13 and will be referred to as $P_1 \dots P_9$, or as the training steps. We also trained nets using impulse responses, that is, the time derivative of the step

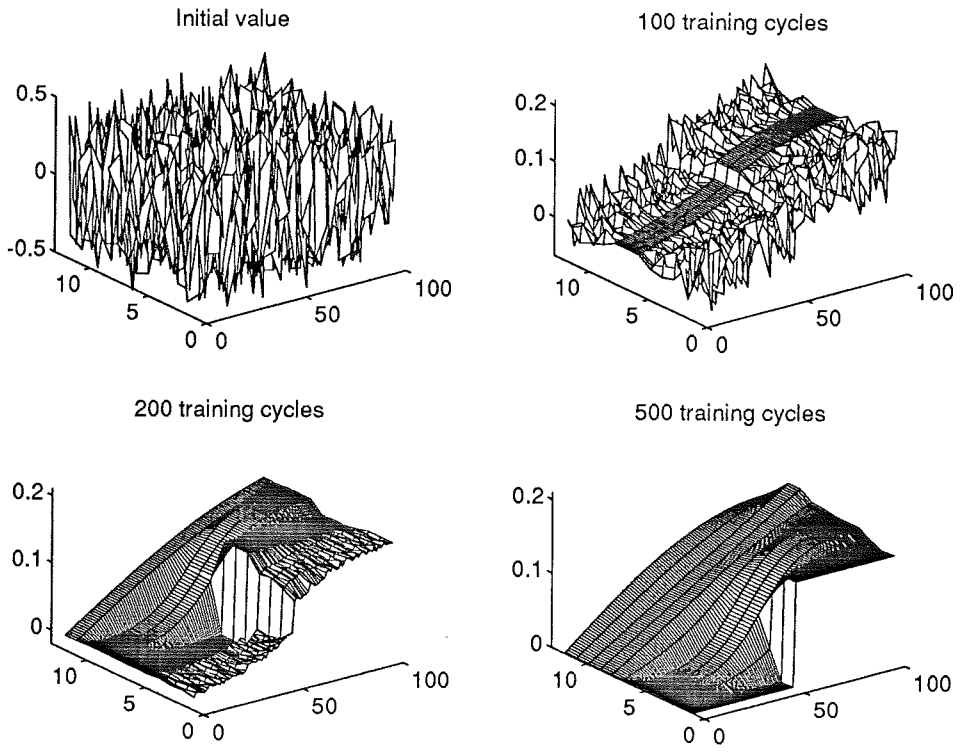

Figure 12. The weight matrix for the example net. On the x-axis are the sample, i.e. the inputs, the neuron number is on the y-axis, and the weight values w_{ij} on the z-axis. For a two dimensional feature map it is much more difficult to visually understand the weight matrix.

responses. These were approximated with forward differences of the step responses. The impulse responses are illustrated in figure 14, and will be referred to as $\dot{P}_1 \ldots \dot{P}_9$.

The steps were generated using the `step` function in the Control System Toolbox for MATLAB. The time scale is 0 to t seconds with sampling points every $t/100$ seconds giving vectors with 101 elements.
$P_1 \ldots P_3$ have the transfer function

$$G(s) = \frac{1}{(s+1)^n}$$

with $n = 1, 4, 8$ and $t = 5, 20, 40$ seconds.
$P_4 \ldots P_5$ have the transfer function

$$G(s) = \frac{1}{s^2 + 2\zeta\omega + \omega^2}$$

with $\omega = 1$ for both transfer functions and $\zeta = 0.5, 0.1$, $t = 20, 50$ seconds.
P_6 has $t = 20$ seconds and

$$G(s) = \frac{1-s}{(s+1)^3}$$

P_7 has $t = 20$ seconds and

$$G(s) = \frac{e^{-2s}}{s+1}$$

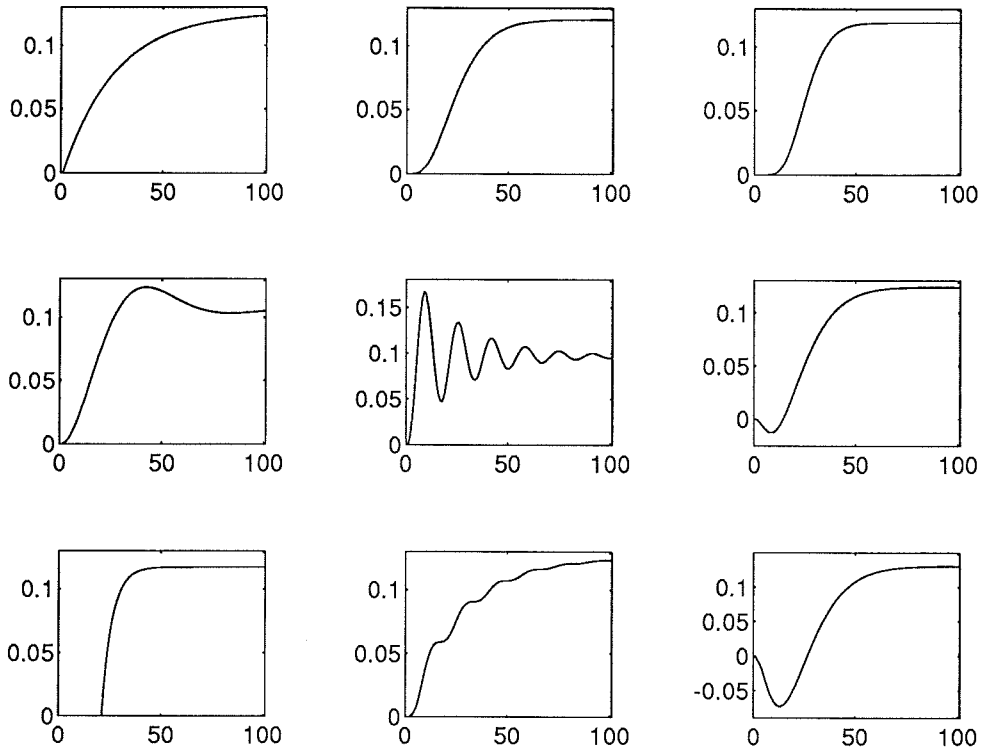

Figure 13. The normalized step responses used to train the nets. The steps will be referred to as P_1 (top left), P_2 (top middle) ... P_9 (bottom right).

P_8 has $t = 50$ seconds and

$$G(s) = \frac{1}{(s + 0.1)(s^2 + 0.2s + 01)}$$

Finally, P_9 has $t = 30$ seconds and

$$G(s) = \frac{1 - 3s}{(s + 1)^3}$$

What is interesting to find out is how the nets can classify these responses. Among the nine step responses in figure 13 we are interested in identifying four to five classes:

1. Monotonous: $P_1 \ldots P_3$.
2. Essentially monotonous: P_4 and P_8.
3. Oscillating: P_5.
4. Inverse response (right half-plane zeros): P_6 and P_9.
5. Time delay: P_7.

We say four to five classes because we do not consider it a big error if the essentially monotonous responses P_4 and P_8 are mistaken for monotonous responses. The small inverse response P_6 might also be hard to identify, but we use it to test how well the nets are able to perform. We will use the same classification for the impulse responses.

To evaluate the nets' ability to identify unknown signals we used a different batch of step- and impulse responses. These were of the same kind as the ones above, but with varying parameters. We generated a total of 31 different responses. A representative selection of the step responses are depicted in

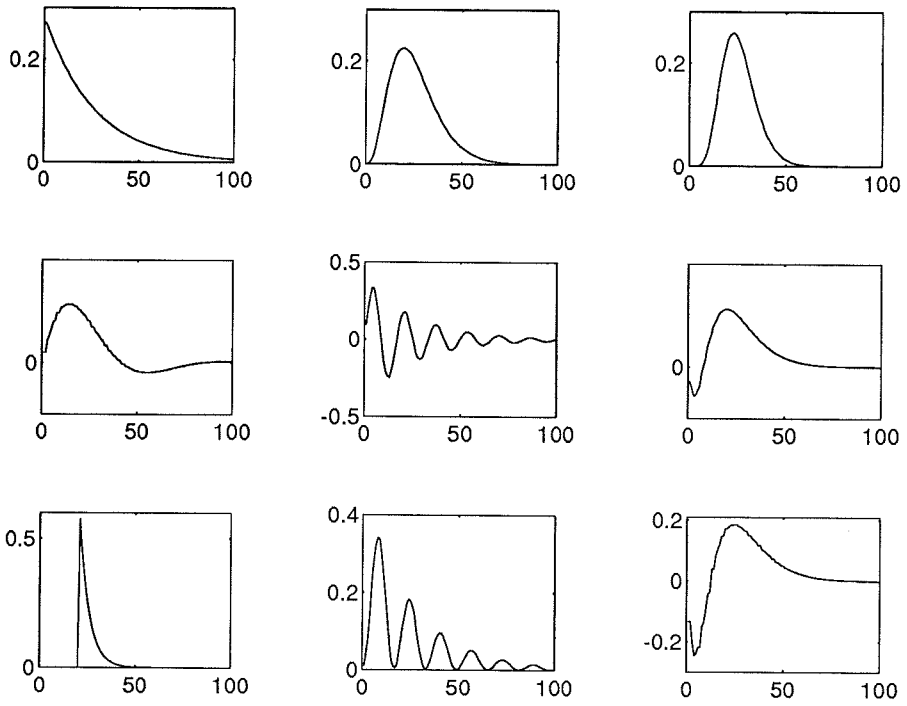

Figure 14. The normalized impulse responses used to train the nets. They will be referred to as $\dot{P}_1$ (top left), $\dot{P}_2$ (top middle) $\ldots \dot{P}_9$ (bottom right). The raggedness that some responses show are due to the numerical differentiation.

figure 15, and the corresponding impulse responses in figure 16. They will be referred to as $Q_1 \ldots Q_9$ and $\dot{Q}_1 \ldots \dot{Q}_9$, or as the test responses. The systems have the following transfer functions:

$$G(s) = \frac{1}{(s+0.5)^n}$$

with $n = 1, 4, 8$, for $Q_1 \ldots Q_3$;

$$G(s) = \frac{1}{s^2 + 2\zeta\omega + \omega^2}$$

with $\omega = 1$ and $\zeta = 0.75, 0.25, 0.1$, for $Q_4 \ldots Q_6$;

$$G(s) = \frac{e^{-1.5s}}{s+1}$$

for Q_7;

$$G(s) = \frac{1}{(s+0.08)(s^2+0.15s+1)}$$

for Q_8; and last but not least

$$G(s) = \frac{1-2s}{(s+1)^3}$$

for Q_9.

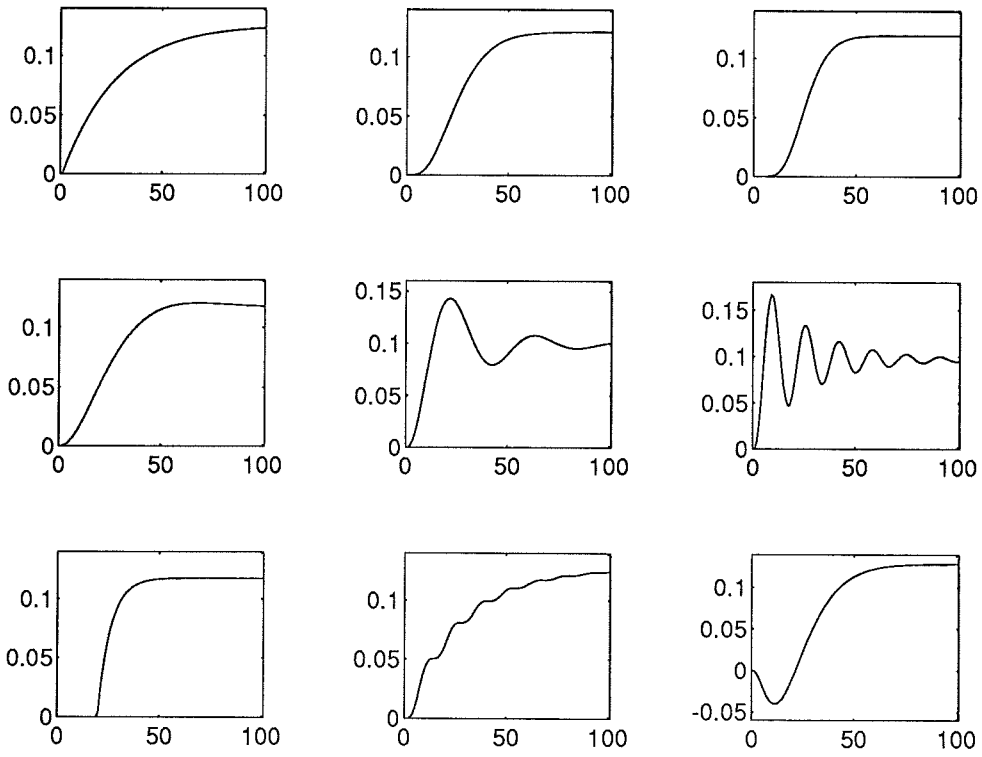

Figure 15. The normalized step responses, $Q_1 \ldots Q_9$, used to test the nets.

Normalization in Time

It seems like a good idea to make systems that have essentially the same

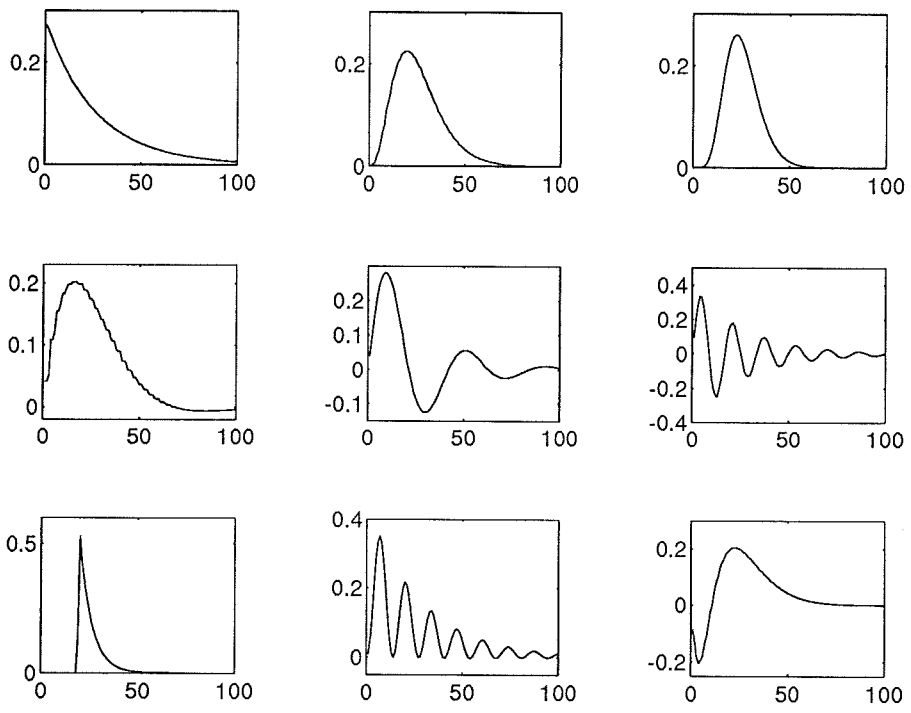

Figure 16. The normalized impulse responses, $\dot{Q}_1 \ldots \dot{Q}_9$ used to test the nets.

characteristics, look as similar as possible. If all first order systems look the same, you only need to train the net on one such system. A normalization like this can be done using the average residence time, T_{ar}, which is defined as

$$T_{ar} = \frac{\int_0^\infty th(t)dt}{\int_0^\infty h(t)dt}$$

where $h(t)$ is the impulse response, i.e. the derivative of the step response.

The definition of T_{ar} implies that $h(t)$ tend to zero, which in turn means that the step response must be stable. It is fairly easy to sort out unstable responses so this is a reasonable condition. During simulation, $h(t)$ was approximated with forward differences of the step response, and the integrals approximated with sums.

T_{ar} is calculated for a step response which is then cut off after nT_{ar}, where $n = 4$ was found to be a suitable value. To get a vector the same size as the original, linear interpolation is used on the remaining elements. Care was taken when generating the step responses not to sample them too long so that too much information would be lost when normalizing.

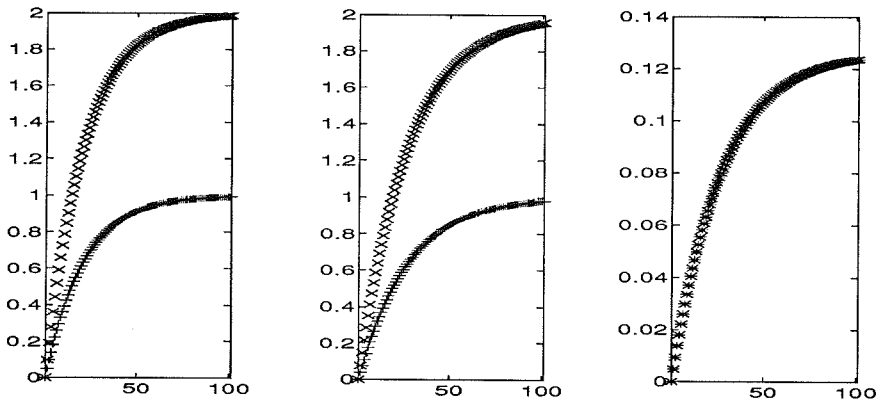

Figure 17. Two different first order systems. $1/(s+1)$ is plotted with '+', $1/(s+0.5)$ plotted with 'x'. Left: The step responses before normalization. Middle: After time normalization. Right: After time- and space normalization.

Noise

Neural networks have the property of being able to classify noisy signals. We examined the noise tolerance on nets trained both on vectors to which noise had been added, and vectors unaffected by noise. In the following when noise is discussed, it is white Gaussian noise with zero mean and a standard deviation of n percent of $max(v) - min(v)$, where v is the unnormalized vector to which the noise is added. That is, if $max(v) - min(v) = a$, then the noise will be $N(0, a/100)$. A statement like "ten percent noise was added" thus means that $n = 10$. Two step responses with 10% noise added are shown in figure 18.

When a derivative is approximated by forward difference (e.g. when calculating the impulse response) noise is added *after differentiation*. This is because differentiating a noisy signal gives an even noisier result. There are good ways to deal with this problem so we chose this simple way when simulating since we are interested in neural networks and not in filter design.

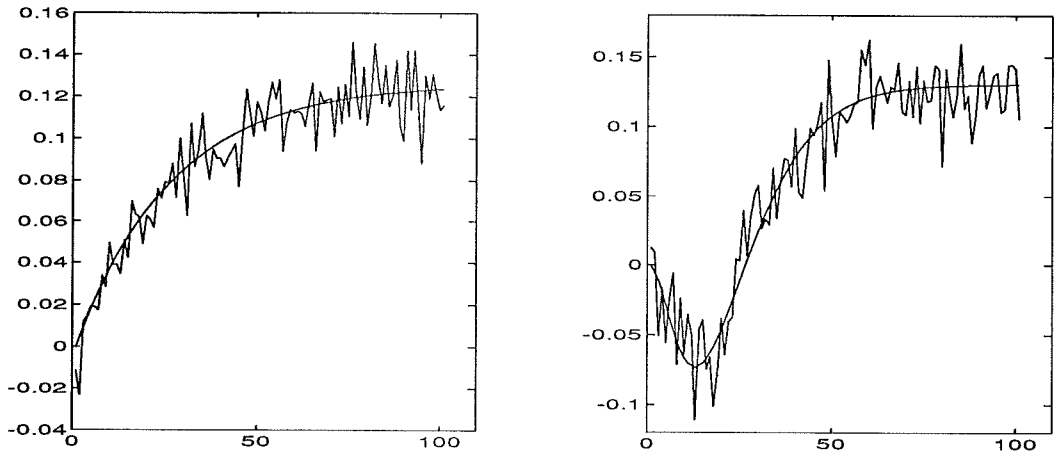

Figure 18. Two step responses with 10% noise added. P_1 to the left, P_9 to the right. The solid line in the middle of the noise is the step response without noise.

Summary

When we generate transient responses, things are done in the following order: generate a step response using the `step` function, normalize in time using the procedure described above, differentiate using forward differences if we want an impulse response, reduce the number of samples by for example picking out every fifth sample, add noise if desired, and normalize vectors to unit length. This is illustrated for complete clarity in figure 19.

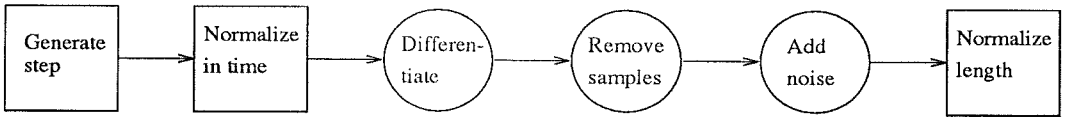

Figure 19. How transient responses are generated. Actions described inside squares are always performed, those inside circles are optional.

4. Classification Using 2-Layer BPN

In this section we will discuss how backpropagation nets can be used to classify transient responses, i.e. step- and impulse responses. The goal is to find nets that are able to classify systems as monotonous, oscillating, essentially monotonous, inverse response systems or as systems with time delay. We will first investigate how the nets behave when trained on the nine vector training batch $P_1 \ldots P_9$ and later on a larger test batch with twenty-seven different vectors. Different training methods will be used and we will see how these affect the performance of the nets. An interesting question is how good the nets are at classifying transient responses they were not trained on. Another question is how much noise the nets tolerate without doing incorrect classifications. An important parameter when designing the net is the number of neurons in the hidden layer and later in this sections we will show how the net size affects the performance of the net.

The Training

When designing training algorithms our goal is to find nets that converge fast, that can recognize the training vectors with or without noise, and that have

good generalization capability i.e. the ability to correctly classify vectors that they were not trained on. The training was done using the function `trainbpx` from MATLAB's Neural Network toolbox [Beale and Demuth, 1992]. We used three different methods to train the backpropagation nets;

- Method A, which is the most straightforward method, just means training the net with the training vector batch.
- Method B. The net is trained for N cycles. The training time is divided into three phases. First training $\frac{2N}{5}$ cycles without noise, then training the following $\frac{1N}{5}$ cycles with noise applied to the input vectors and at last training the remaining cycles without noise.
- Method C. This an another method to train with noise. A training batch that consists of two sets of the original training batch is created. To one of the sets is then noise added. This means training both with and without noise during all training cycles.

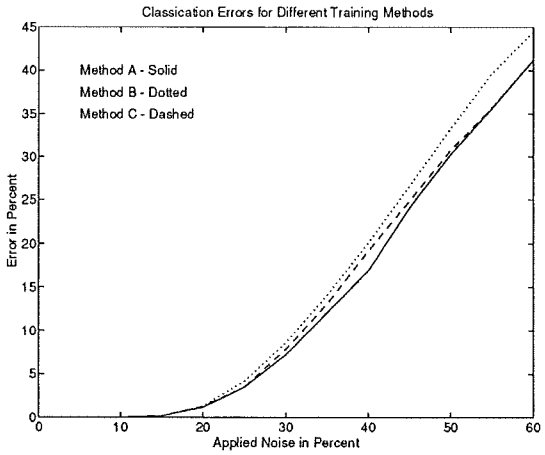

Figure 20. Performance of nets trained with three different methods. The solid line shows the performance of the net trained without noise.

In the literature it is often said that if the net is going to operate on noisy input you should also be trained with noise. It is also claimed in [Freeman and Skapura, 1991] that a net trained with noise will converge faster than one that is trained without. However in our simulations we have discovered neither better performance nor faster convergence when training with noise. A large test batch with three groups of twenty nets were trained for 10 000 cycles using different training methods for the three groups. All nets were tested on the training vector batch with noise applied to it. The noise ranged from 0 to 60 percent in steps of 5 percent. The result can be seen in figure 20. Method A is slightly better than the two methods where noisy training input were used. With twenty percent noise applied the net trained with method A can classify 99.84 percent of the inputs correctly.

Network Structure and Training Parameters

There are a number of different parameters to consider when training a back-propagation net. We have investigated how the performance was affected by varying some of them while others were kept fixed. The simulations were concentrated on 2-layer nets with bias neurons and a logarithmic sigmoid transfer function. The initial values for the weights between the input layer and the hidden layer is set using the `nwlog` function in the Neural Network Toolbox,

which implements a method described in [Nguyen and B.Widrow, 1990]. The weights connecting the hidden layer and the output layer are initialized to random values between -0.1 and 0.1. We discovered that nets with larger initial values in the output matrix converged slower.

Parameters used during training are the learning rate, l_r and the *momentum*. The learning rate is adaptive and the parameters for controlling it are

- lr_{inc} -the multiplier used to increase the learning rate.
- lr_{dec} -the multiplier used to decrease the learning rate.
- *Error-ratio* -the maximum ratio of new and old error allowed without rejecting new values of weights and matrices. The *Error-ratio* can also be used only with the *momentum*.

We found that a suitable value for the learning rate is 0.01, with $lr_{inc} = 1.02$ and $lr_{dec} = 0.98$. With those parameters there is a good chance that the net will converge but it will proberably take some time. For faster convergence but with a severe risk that the net will get stuck in a local minimum you can you use $lr = 0.15$, $lr_{inc} = 1.05$ $lr_{dec} = 0.7$. With our training batch we got better results for nets with more than about 18 neurons in the hidden layer using the later lr-parameter configuration. A good value for the *momentum* is 0.95, and 1.04 for the *Error-ratio*. In figure 21 it is shown how the parameters vary over time for a net with twenty neurons in the hidden layer, trained with the step response vectors $P_1 \ldots P_9$. Figure 22 shows the same net trained with the impulse responses $\dot{P}_1 \ldots \dot{P}_9$ and it is clear that the convergence speed is much higher than for the net trained on step responses.

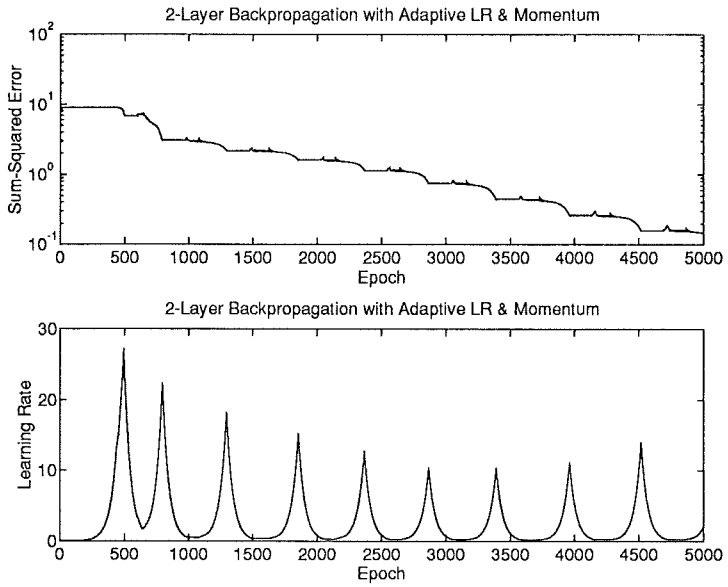

Figure 21. A net with twenty neurons in the hidden layer trained on step responses for 5000 cycles with adaptive parameters.

In our simulation we discovered that when working with considerably smaller nets, i.e. 5-10 hidden layer neurons, it was preferable to use a small l_r, approximately 0.01. Those nets converged rather quickly and performed well. At larger net-configurations those parameters were not sufficient anymore and this lead to more and more nets that did not converge. This problem can be solved through tuning the parameters more carefully.

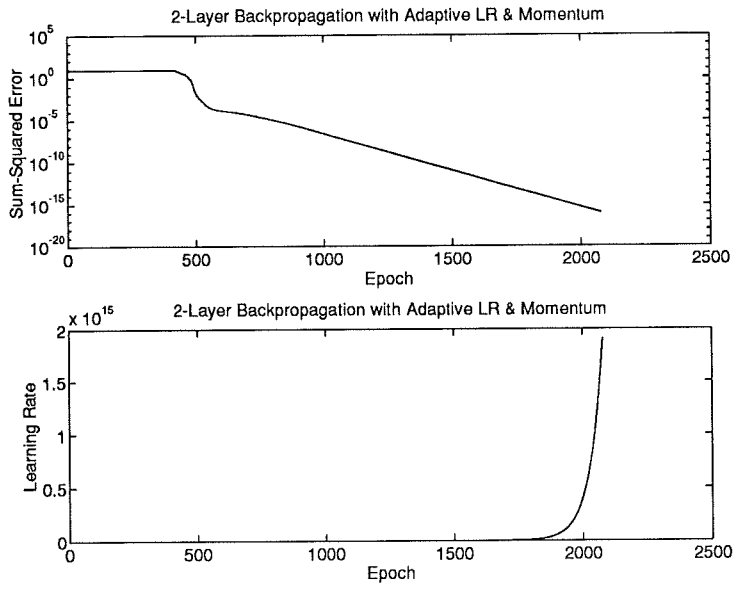

Figure 22. A net with twenty neurons in the hidden layer trained on impulse responses for about 2000 cycles with adaptive parameters. Notice the extremely small sum-squared-error.

Reducing the Number of Output Neurons

The desired output when training the net is called the target vector. There exists one unique target vector for every input training vector $P_1 \ldots P_9$. We used two different types of target vectors: binary output and single output, see figure 23. Binary output simply means that the output neurons are interpreted as bits in a binary number. The bits are set as follows

$$BinaryOutput_k = \begin{cases} 0, & O_k < 0.5 \\ 1, & O_k \geq 0.5, \end{cases} \tag{7}$$

where $k = 1, \ldots, M$, M is the number of output neurons, and O_k is the output from neuron k. The leftmost neuron is interpreted as the least significant bit. In figure 23 (top) the binary representaion 1001 of 9 is illustrated. Single output means that every output neuron is given a unique number and only one neuron is active at a time. A neuron with an output value of 1 is defined as active and similarly is an inactive neuron defined as a neuron with an output value of 0. An example of a single output representing the number 9 is shown in figure 23 (bottom). The advantage of using binary output is that you reduce the number of free parameters, i.e. the weights on the connections between the layers, quite drastically and this can be very important when the nets get larger. Take for example a net with twenty-seven hidden layer neurons and the same number of input-output pairs. If we used single output target vectors this will give us 616 more free parameters than if we used binary output. Equation (7) shows the weak spot of the binary output method which is its sensitivity to noise. In the case of using single output target vectors, the output neuron with the highest value is set to one and all the others are set to zero, which leads to a net with higher tolerance to noise.

In our simulations with the training sets $P_1, \ldots, P_9$ and $\dot{P}_1, \ldots, \dot{P}_9$ we have used both methods and we found that with single output target vectors you get a smaller final-sum-square-error and better performance with or without noise, but when the net size grows it gets harder and harder to make the net

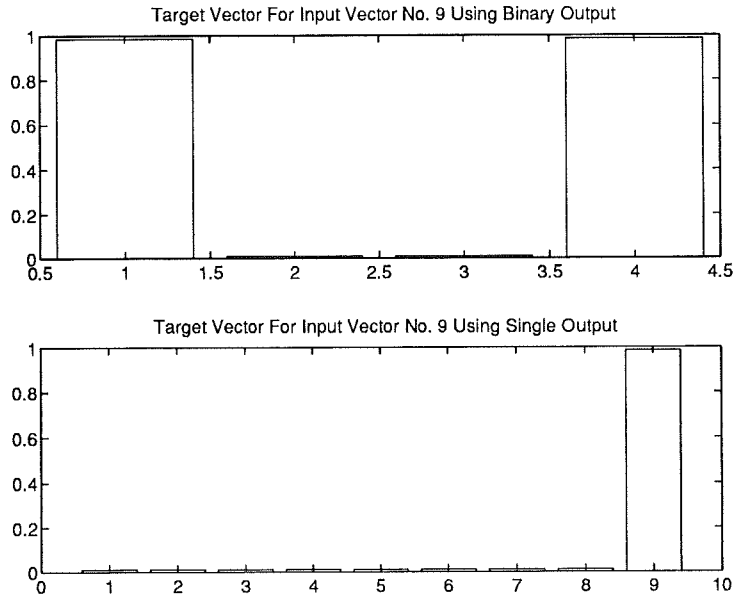

Figure 23. The two types of target vectors: Binary output (top) and single output(bottom). Both target vectors represent the output nine. Least significant bit for the binary output is the bar to the left in the figure.

converge. At more than thirty neurons in the hidden layer we are forced to use binary output to get a working net.

The Size of the Net

When designing a net our goal is to get as good performance as possible and at the same time use as few neurons as possible. It is possible to train a net with only two neurons to recognize the training vectors $\dot{P}_1 \ldots \dot{P}_9$ but the net will be sensitive to noise and not very good at generalization. We tried to find the optimal net configuration for the training vectors $\dot{P}_1 \ldots \dot{P}_9$ and to do that we used a large test batch where we trained ten nets for every hidden layer configuration from two to forty neurons. To get the best possible performance we used single output target vectors. Each net was trained 10 000 cycles and later tested 500 cycles on the training vectors, with 15 percent noise applied. The result is shown in figure 24.

It seems to be sufficient with only about eight neurons, which is close to the number of training vectors. We first did the simulations with both single output and binary output target vectors and though the former performed much better the latter did not have any problem to converge even when the net size was increased to over fifty hidden layer neurons. This was a big problem when using single output target vectors, since the nets rarely convergered when the number of hidden layer neurons exceeded thirty. The best-net curve reaches below 0.5 percent classification error at only seven neurons in the hidden layer.

Computational Complexity

When deciding the net size one important consideration is the complexity of the calculations required to train and later to run the net. First the number of free parameters is very important. If you have too few the net is not able to learn and if you have too many the net is likely not to converge or to converge to a local minimum. For a 2-layer backpropagation net the number of free

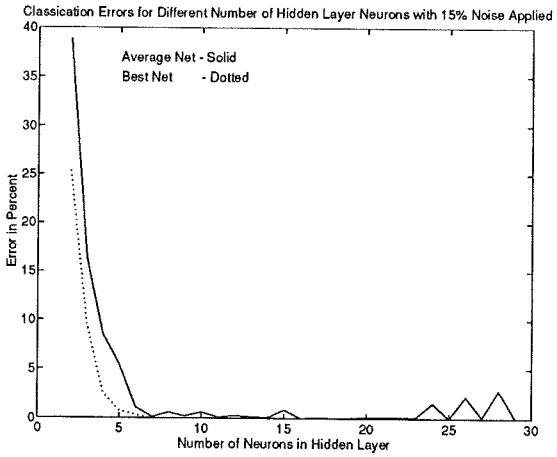

Figure 24. The classification errors for nets with different numbers of neurons in the hidden layer. The solid line shows the how the average net bahaves after traing. We also show with the dotted line how the best net, for every configuraion performed.

parameters are

$$f = (N+1) * L + (L+1) * M,$$

where N is the number of neurons in the input layer, L is the number of hidden layer neurons and M is the number of neurons in the output layer. The extra ones come from the bias unit. The net is depicted in figure 2. This means that if we add one more neuron to the hidden layer, the number of free parameters will increase with $N + 1 + M$. In the net trained with impulse responses we had $N = 100$ and $M = 9$ this will give 110 new parameters for every new neuron.

During training the weights are updated every cycle. The update equation for the output layer weights is

$$w_{kj}^o(t+1) = w_{kj}^o(t) + \Delta w_{kj}^o$$

where

$$\Delta w_{kj}^o = -l_r \frac{\partial E}{\partial w_{kj}} = l_r(y_k - O_k^o) f_k'^o(I_k^o) O_j^h$$

and for the hidden layer weights

$$w_{ji}^h(t+1) = w_{ji}^h(t) + \Delta w_{ji}^h$$

where

$$\Delta w_{ji}^h = -l_r \sum_k (y_k - O_k^o) f_k'^o(I_k^o) w_{kj}^o f_j'^h(I_j^h) x_i.$$

Thus for every cycle we have to calculate Δw_{kj}^o and Δw_{ji}^h, where $i = 1, ..., N$, $j = 1, ..., L$, and $k = 1, ..., M$. The sigmoid function used in our simulation is

$$f(x) = \frac{1}{1+e^{-x}} \quad (3\ flops)$$

$$f'(x) = \frac{e^{-x}}{(1+e^{-x})^2} \quad (7\ flops).$$

To the right of the functions you can see the number of floating point operations (flops) required for MATLAB to calculate the function values. The

training cycle for a backpropagation net has two phases. First every vector in the training batch is applied to the net and the total sum-square-error is calculated. In the second phase the weights are adjusted to minimize the error. The output is calculated as

$$O = f_M(W^o \cdot f_M(W^h \cdot P_i + b^h) + b^o),$$

where W^o is a $M \times L$ matrix, W^h is a $N \times L$ matrix and the function f_M is just a matrix version of the sigmoid function above. Taking the scalar product of two vectors with the lengths L requires $2L$ floating point operations. This means that if we have a training batch P containing S vectors it will take $S\{M(2L+8)+L(2N+8)\}$ flops to calculate the output for the entire batch P.

The next step is to backpropagate the error and adjust the weights in the two layers. Calculating the new weights takes $(L+1)M(L+12)$ operations for the output layer and $(N+1)L(1+M(L+N+21))$ for the hidden layer. All in all this makes a total of

$$flops(L, M, N) = \tag{8}$$

$$S\{M(2L+8)+L(2N+8)\}+(L+1)M(L+12)+(N+1)L(1+M)(L+N+21))$$

per training cycle. It is usually sufficient to choose $L = S$ and if we use a single output target vector we will get $M = S$. For the case $N \gg S$ we can now write equation (8)

$$flops(S, N) = NS^3 + (N^2 + 25N)S^2 + (N^2 + 22N)S \tag{9}$$

The number of training cycles will also depend on the complexity and the number of training vectors in the training batch. The more training vectors the more cycles are required.

For an ordinary net with nine hidden layer neurons and nine output neurons trained with the vectors $\dot{P}_1 \ldots \dot{P}_9$, the number of cycles needed is about 1500. With $N = 100$ and $S = L = M = 9$ we can use equation(9). This gives us

$$1500 \cdot flops(9, 100) =$$

$$1500 \cdot (100 \cdot 9^3 + (100^2 + 25 \cdot 100) \cdot 9^2 + (100^2 + 22 \cdot 100) \cdot 9 = 1.79 \cdot 10^9$$

needed floating point operations to train the the net. This is a large number of operations and it is always wise to estimate the training time before you start running.

Results for Transient Responses

A backpropagation net can easily learn to recognize all nine training vectors $P_1 \ldots P_9$ and their differentiated counterparts $\dot{P}_1 \ldots \dot{P}_9$. With the proper parameter setting you can get a net with anything from 2 up to 40 neurons in the hidden layer to converge and thus manage the task of recognizing the training vectors. The number of training cycles required to get a net with sufficiently small sum-squared-error was about 5000 for the step responses and less than 2000 for the impulse responses. We have found that having the same number of neurons in the hidden layer as the number of training vectors will work out very well. Earlier in this section it is shown that the number of neurons in the

hidden layer will not affect the performance with noisy inputs, as long as the number exceeds a certain limit. This limit is in our case, with nine input vectors, approximately nine hidden layer neurons. When designing a net to test the sensitivity to noise we chose to use twenty neurons in the hidden layer and Single-Output target vectors. We created two batches: one using $P_1 \ldots P_9$ and the other using $\dot{P}_1 \ldots \dot{P}_9$ as training vectors. The first thing that strikes you

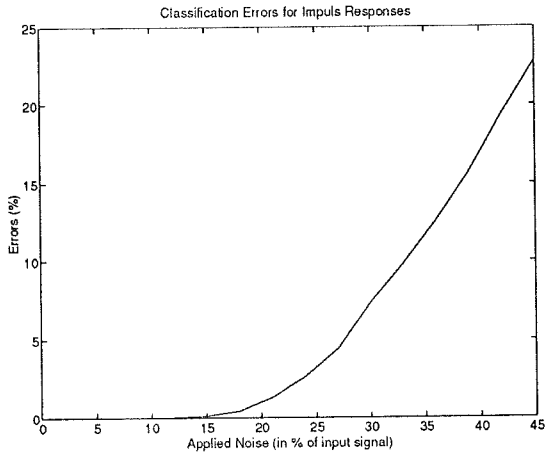

Figure 25. Classification errors for a net trained on impulse responses. The net has 100 input neurons, 20 hidden layer neurons and 9 output neurons. With 20 percent noise on the input signal the error is only 0.11 percent.

when comparing nets that have been trained on the two training sets is the overall higher performance when using the differentiated set. They converge much quicker and usually get a much lower final sum-squared-error, i.e. for our training batch we will get a final error at around 10^{-10}. Having a final error this small automatically gives the net good noise tolerance. In figure 25 it is shown how such a net performs when working on noisy input. We get 100 percent correct classification with up to 12 percent noise. The average error when having twenty percent noise applied to the vectors $\dot{P}_1 \ldots \dot{P}_9$ is only 0.4 percent and that must be considered as a very good result. The vectors $\dot{P}_2$, $\dot{P}_4$, $\dot{P}_5$ and $\dot{P}_9$ with twenty percent noise are shown in figure 26.

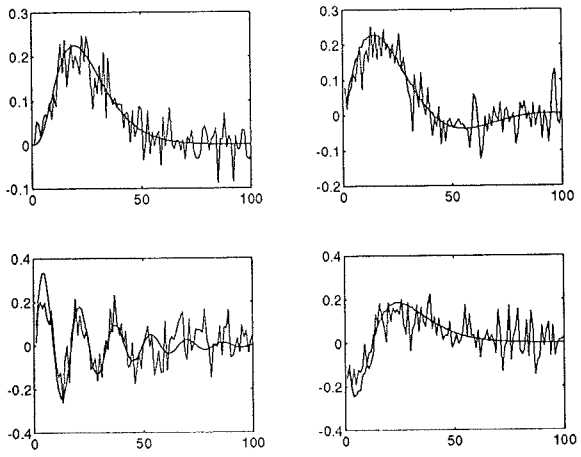

Figure 26. The differentiated training vectors $\dot{P}_2$, $\dot{P}_4$, $\dot{P}_5$ and $\dot{P}_9$ with twenty percent noise on top.

The nets trained to work on step responses converged slower and gave

a final sum-squared-error at around 0.1, which is much larger than for the impulse responses. Not very surprisingly this lead to greater noise sensitivity. In figure 27 you can see the performance of such a net when tested with noisy inputs. When we apply more than five percent noise to the training vectors P_1 ... P_9 the classification error will be larger than zero. Another major drawback when using step responses compared to using impulse responses is that it is much more difficult to set the training parameters for the latter. If the parameters are to large the nets are not very likely to converge and if they are to small the convergence rate will be very slow.

All in all we can see that nets using impulse responses are easier and faster to train and also perform better than nets using step responses.

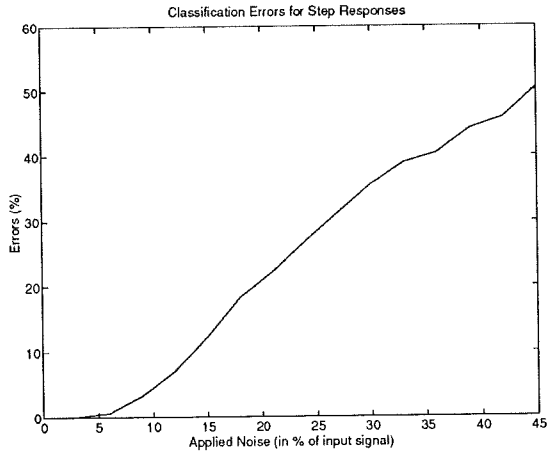

Figure 27. Classification errors for a net trained on step responses. The net has 101 input neurons, 20 hidden layer neurons and 9 output neurons.

Testing on Unknown Inputs

The real power of neural networks lies in their ability to operate on inputs that they were not trained on. During the training phase you prepare the net to work with a certain kind of data and the net will learn to key off the inputs' significant features. In previous sections we have shown how different training methods and training parameters affect the performance of a backpropagation net and with all this we shall now design a new net and see how it handles previously unknown inputs. We will use two nets, both with twenty neurons in the hidden layer, and we will train them without any noise. The first net will be trained to work with step responses and the second to work with impulse responses. The number of training cycles will be individually adjusted to get the best possible nets. A description of the test batches $Q_1, \ldots, Q_9$ and $\dot{Q}_1, \ldots, \dot{Q}_9$ is given in section 3. To avoid statistical fluctuations we have trained and tested about twenty nets of each category. Typical classifications for nets trained on step responses are:

- Q_1 (Monotonous 1st order) classed as P_1 (Monotonous 1st order)
- Q_2 (Monotonous 4th order) classed as P_2 (Monotonous 4th order)
- Q_3 (Monotonous 8th order) classed as P_2 or P_3 (Monotonous 4th order)
- Q_4 (Essentially monotonous) classed as P_1 (Monotonous)
- Q_5 (Oscillating) classed as P_4 (Oscillating)
- Q_6 (Oscillating) classed as P_5 (Oscillating)

- Q_7 (Delay) classed as P_7 (Delay)
- Q_8 (Essentially monotonous) classed as P_1 (Monotonous)
- Q_9 (Inverse response) classed as P_9 (Inverse response)

This is undoubtedly a very good result. The net is capable of separating the five different classes mentioned in the introduction. The results for the net trained on impulse responses are shown below, and it makes about the same classification as the step response net. When further comparing the two nets it is found that the impulse response net performs slightly better than the step response net when tested with noisy input. Even with only ten percent noise on top of the inputs the nets will make a different classification. They usually identify the correct system type, but further information, like the order of a monotonous system, will be lost. For example a system that without noise is classified as monotonous of the 1st order is with noise instead classified as monotonous of the 4th order. Typical classifications for nets trained on impulse responses are:

- $\dot{Q}_1$ (Monotonous 1st order) classed as $\dot{P}_1$ (Monotonous 1st order)
- $\dot{Q}_2$ (Monotonous 4th order) classed as $\dot{P}_2$ (Monotonous 4th order)
- $\dot{Q}_3$ (Monotonous 8th order) classed as $\dot{P}_3$ (Monotonous 8th order)
- $\dot{Q}_4$ (Essentially monotonous) classed as $\dot{P}_2$ (Monotonous)
- $\dot{Q}_5$ (Oscillating) classed as $\dot{P}_8$ (Essentially monotonous)
- $\dot{Q}_6$ (Oscillating) classed as $\dot{P}_5$ (Oscillating)
- $\dot{Q}_7$ (Delay) classed as $\dot{P}_7$ (Delay)
- $\dot{Q}_8$ (Essentially monotonous) classed as $\dot{P}_1$ (Monotonous)
- $\dot{Q}_9$ (Inverse response) classed as $\dot{P}_9$ (Inverse response)

Training on a Larger Batch

To further investigate how the backpropagation nets can be used we will in this section train them with a larger training set $R_1 \ldots R_{27}$. We will show that it is possible for the net to make a much more detailed classification than has been done previously. Our goal now is to use the nets, not just to classify the system as for example monotonous, but also in this case to give information about the order of the system. The step responses in the new training set come from the following systems:

- Monotonous $R_1 \ldots R_9$

$$G(s) = \frac{1}{(s+0.75)^n},$$

  where $n = 1, \ldots, 9$.

- Oscillating $P_{10} \ldots P_{17}$

$$G(s) = \frac{1}{s^2 + 2\zeta\omega + \omega^2},$$

  where $\omega_i = 1$ for $i = 10, \ldots, 14$ and $\omega_i = 0.5$ for $i = 15, \ldots, 17$ and $\zeta_i = 0.25, 0.2, 0.15, 0.1, 0.05, 0.25, 0.2, 0.15$.

- Essentially monotonous P_{18}

$$G(s) = \frac{1}{(s+0.1)(s^2 + 0.2s + 01)}$$

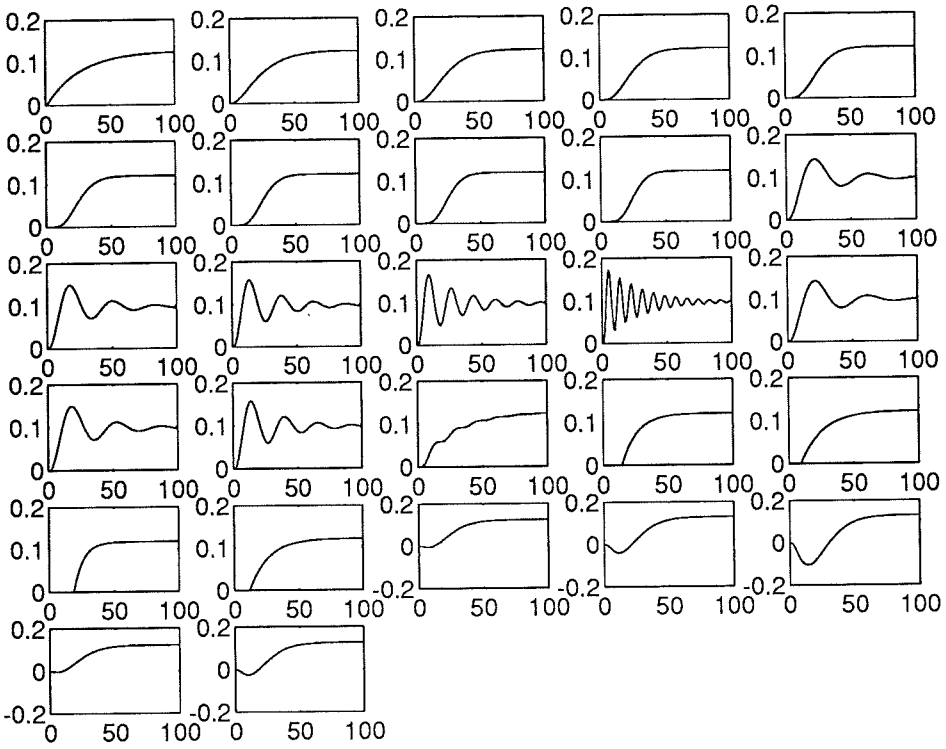

Figure 28. The large training batch R_1–R_{27}.

- Time Delay $R_{19} \ldots R_{22}$

$$G(s) = \frac{e^{-ts}}{s+a},$$

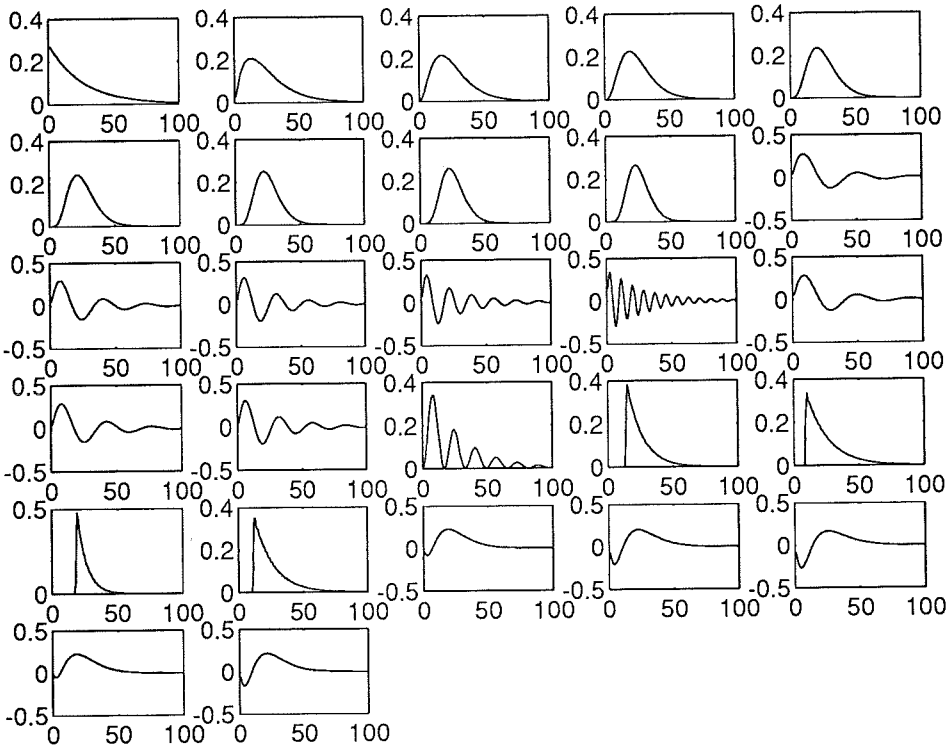

Figure 29. The large training batch $\dot{R}_1$–$\dot{R}_{27}$.

where $(a_i, t_i) = (0.75, 1.5), (1.0, 0.5), (2.5, 1.0), (1.5, 0.5)$

- Inverse Response $R_{23} \ldots R_{27}$

$$G(s) = \frac{1 - as}{(1+s)^3}$$

where $a = 0.75$, 2, 4, 0.5, 1.5.

Nets were trained both directly on the training vectors and on their differentiated counterparts. While nets trained on the latter converged without any problems nets trained on the step responses hardly converged at all. Hence, we will only discuss the nets trained on the impulse responses. The number of cycles needed to get final-sum-square error of the order 10^{-11} usually was about 25 000. To get the same error with the smaller training batch $\dot{P}_1 \ldots \dot{P}_9$ only about 1500 training cycles are needed. To test the nets' ability to classify vectors that it was not trained on, we used a test batch with impulse responses similar to $Q_1, \ldots, Q_9$ but generated with different parameters to prevent the vectors in the test and in the training batch from being to similar. The result was a net that was able to give information about the order of a monotonous system and a approximate value of the delay for a system with a time delay. The test systems had the following transfer functions:

- $Q2_1, \ldots, Q2_3$ Monotonous

$$G(s) = \frac{1}{(s+0.5)^n},$$

where n=1,4,8.

- $Q2_4, \ldots, Q2_6$ Oscillating

$$G(s) = \frac{1}{s^2 + 2\zeta\omega + \omega^2},$$

where $(\omega, \zeta) = (1.0, 0.75), (1.0, 0.5), (0.75, 0.25)$.

- Time delay

$$G(s) = \frac{e^{-1.5s}}{s+1}.$$

- Essentially monotonous

$$G(s) = \frac{1}{(s+0.08)(s^2+0.15s+1)}.$$

- Inverse response

$$G(s) = \frac{1-2s}{(s+1)^3}.$$

We applied those test vectors to a net trained with impulse responses and the results are presented below. The net is capable of deciding the order of the first three monotonous impulse responses which is in line with observations made on other test batches, where the net correctly recognized monotonous inputs with order from 1 to 9. With the oscillating impulse responses the test vectors are classed a vectors with the same appearance but without giving any direct information about ω and ζ. The last three vectors $\dot{Q}2_7, \ldots, \dot{Q}2_9$ were all mapped to correct categories. The classification of $\dot{Q}2_7$ even gave the right time delay for the system.

The next step is to add some noise to the inputs and see how this affects the net's generalization capability. When doing this we find that information such as the value of a time delay or the order of a system is soon distorted by the applied noise. Not very surprisingly the conclusion is that the more noise there is on top of your input the less information you get about the system. Typical classifications for nets trained on $\dot{R}_1, \ldots, \dot{R}_9$ are:

- $\dot{Q}2_1$ classed as $\dot{R}_1$ (Monotonous 1st order)
- $\dot{Q}2_2$ classed as $\dot{R}_4$ (Monotonous 4th order)
- $\dot{Q}2_3$ classed as $\dot{R}_8$ (Monotonous 8th order)
- $\dot{Q}2_4$ classed as $\dot{R}_2$ (Monotonous 2nd order)
- $\dot{Q}2_5$ classed as $\dot{R}_{13}$ (Oscillating $(\omega, \zeta) = (1.0, 0.1)$)
- $\dot{Q}2_6$ classed as $\dot{R}_{15}$ (Oscillating $(\omega, \zeta) = (0.5, 0.05)$)
- $\dot{Q}2_7$ classed as $\dot{R}_{21}$ (Time Delay $(a, t) = (2.5, 1.0)$)
- $\dot{Q}2_8$ classed as $\dot{R}_{18}$ (Essentially monotonous)
- $\dot{Q}2_9$ classed as $\dot{R}_{27}$ (Inverse response $a = 1.5$)

Conclusions and Practical Considerations

The backpropagation network seems to be well suited for use as a classifier of system dynamics. We have investigated nets trained on impulse responses and and nets trained on step responses and found the former superior in several aspects. They converged faster, were less sensitive to noise, and had a better ability to generalize. One rather tricky question is how to design and train a net to perform a certain task and we have in this section tried to give guidelines regarding this. One of the most important things about net design is to keep the number of neurons as few as possible to get good training results and real-time performance. One good way of doing this is to code the output as binary numbers. To train a net with impulse responses requires about 1500 training cycles.

We found that using an adaptive learning rate of 0.01 with $lr_{inc} = 1.02$, $lr_{dec} = 0.98$ worked well. A good value for the *Error-ratio* is 1.04, and 0.95 for the *momentum*.

It is clear that the more you want the net to know and recognize, the more sensitive it will become to noise. For example, the impulse responses from two monotonous systems of order two and three will become very much alike when you add some noise to the signal, even if it is just a few percent. But it is also obvious that it is possible to use the nets to get a lot more information about the system than just a rough classification.

5. Classification Using One-dimensional SOFMs

Classifying transient responses is basically a pattern recognition task. Self-Organizing Feature Maps have proven to be able to perform pattern recognition well, like for example Kohonen's phonetic typewriter [Kohonen, 1988]. In this section we will present our experiments with SOFMs and show how they can be used for classification of step- and impulse responses. Simulations were carried out to investigate how the nets' performance were affected by the number of neurons. We examined what resolution can be obtained, that is how many different classes of responses we can separate. How much computation is required? We are naturally interested in how sensitive to noise the nets are.

Nets trained on inputs that were not affected by noise were compared with nets trained on both noisy and unnoisy inputs. The sampling rate is usually of some importance to applications, and we investigated how the nets' performance varied with the number of samples in the input vectors. The SOFMs organize the training vectors in some way according to their mutual resemblance. Which are alike and which are not? We will present our findings, and try to explain the results. Finally, we will show two nets that we constructed when we took the results from all of the experimenting into account.

A 3D-plot of the normalized batch of training vectors is shown in figure 30. This will later be compared with 3D-plots of the weight matrices. The vectors were first normalized in time then in space. By normalization in space we mean scaling the vectors to unit length.

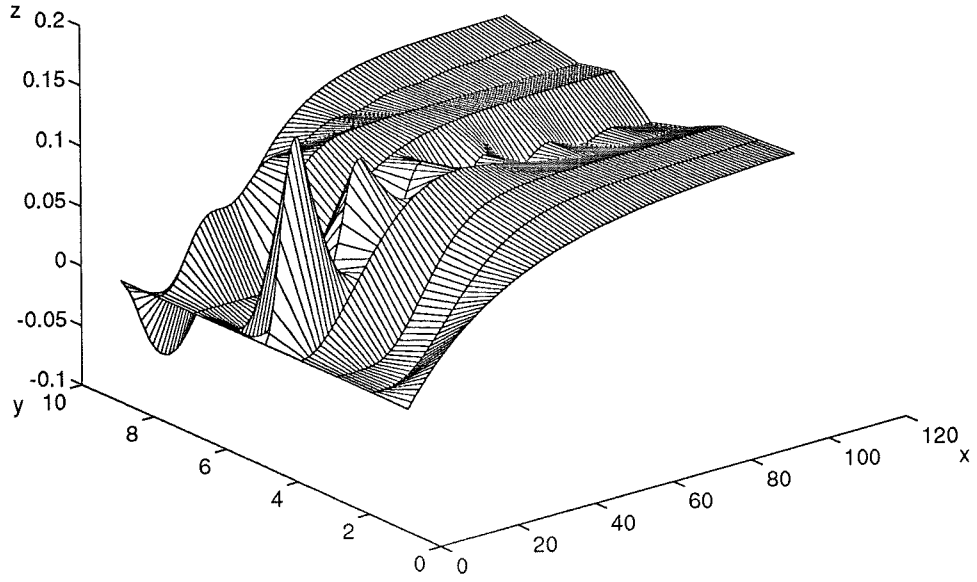

Figure 30. A 3D-map of the normalized training vectors. On the x-axis we have the samples (the inputs), the number of the training vector is on the y-axis, and the value of each vector component is on the z-axis.

The Training

The training was done with the function `trainfm` in the MATLAB Neural Network Toolbox [Beale and Demuth, 1992]. The function trains a competitive layer by randomly presenting an input vector from the supplied batch of input vectors. The neuron whose input is greatest and the neighbours within the current neighbourhood then has its weights updated with the Kohonen rule. During training with `trainfm` the number of neighbours affected by the winning neuron is decreased linearly to reach a minimum of affecting only the neurons next to the winning after one quarter of the training cycles. The learning rate also decreases linearly from a given initial value, to zero at the last training cycle. If nothing else is stated, the nets were trained for 100 000 cycles with an initial learning rate of 0.005, which were found to be good

parameters. As stated in section 2, we use a "top hat" approximation of the "Mexican hat" lateral interaction function.

After training was completed the nets were tested on both the training batch and a batch of vectors that it had not been trained on. When a net has been trained with unsupervised learning it is necessary to test it on the training batch to see how it classifies known inputs. The nets were also tested for noise sensitivity by adding Gaussian white noise to the vectors that were tested. Some nets were trained on vectors that were both affected and unaffected by noise. This was done by having a double batch of training vectors, one of which was noisy.

The choice of training vectors is important when creating a feature map. Vectors that are representative of the interesting classes we wish to find should naturally be used. But the type of responses the net is trained on is not the only important aspect. The amount of patterns that are similar must also be appropriate. The number of neurons that adapt to a certain pattern is proportional to the relative occurrence of that pattern in the training batch. Consider figure 31 where the weight vectors of a 20 neuron network trained on the vectors in figure 30 are shown. There is a smooth transition from the large inverse response at neurons 1 and 2 to the oscillating response at neurons 18, 19 and 20. Notice that the net orders the responses automatically. The ordering is easily understood, and clearly intuitive since we use a one-dimensional map, i.e. the neurons are arranged in a linear array. The net becomes ordered because not only the winning neuron has its weights updated, but also the neurons within a certain neighbourhood, as described in section 2. This is called topology preservation. Had any pattern been more frequent in the training batch, this would have been visible in the figure.

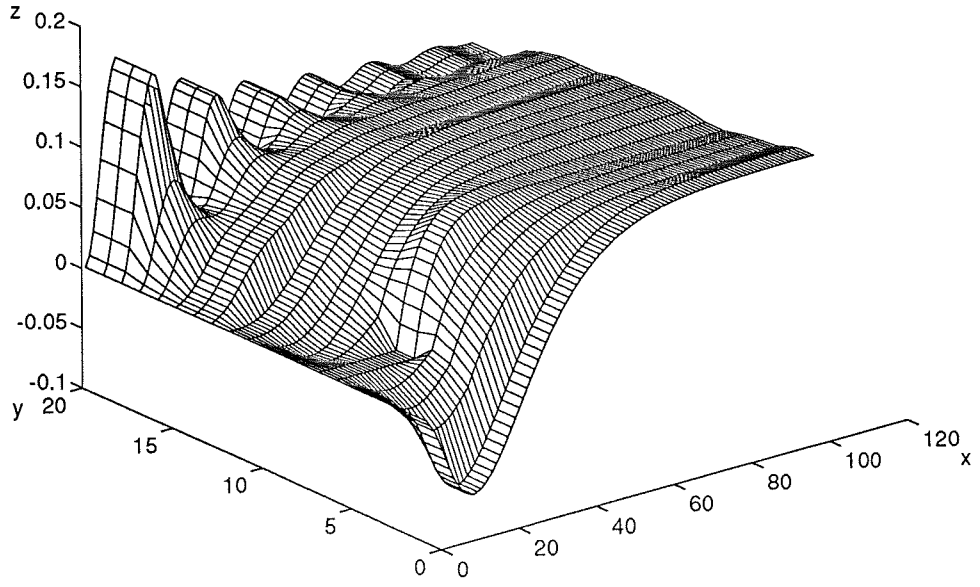

Figure 31. A 3D-map of of the weight matrix from a 20 neuron network, trained on the step responses $P_1 \ldots P_9$. The samples are on the x-axis, the neuron number on the y-axis, and the value of each weight w_{ij} on the z-axis.

Using the SOFM

Knowledge about the shape of the weight surface such as that in figure 31 can be of great help when evaluating the output from the net. A good way of finding out which neurons respond to which input is to apply the training vectors on the net and study the output of each neuron. An example of this is shown in figure 32. The training vector P_9 was applied to the 20 neuron net whose weight matrix is shown in figure 31. There we can see that neurons 1

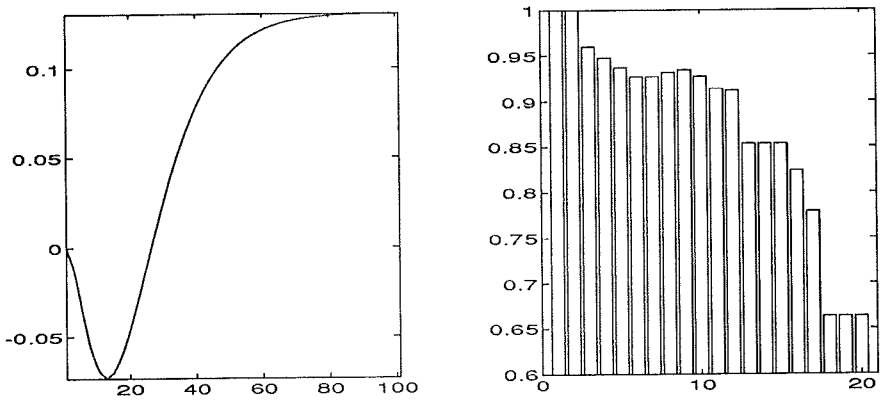

Figure 32. The outputs from the 20 neurons in the net depicted in the figure above (right) when P_9 (left) was applied.

and 2 both have values equal to one while the other neurons have significantly smaller values. The winning neuron is the one with the largest output value. If more than one neuron has the same output (as is the case here), the one with the lowest index is considered as the winner. When the winning neuron has been found, its value is set to 1 and the other neurons are set to 0. As we can see in figure 31 the weight vectors for neurons 1 and 2 look very much like P_9. Similar results are obtained for the other training vectors.

The Number of Neurons and the Number of Samples

Obviously we need at least as many neurons as the number of classes we are interested in identifying, since the number of neurons equals the number of different outputs we are able to receive. If we are interested in being able to distinguish between the first order system P_1 and the fourth order system P_2 we need more neurons than if we only want to classify them both as monotonous. Further, we are not able to decide what classes the SOFM will organize itself into. Only experiments can tell us how many different classes the nets will be able to distinguish between, and what amount of neurons is most appropriate.

Computational Complexity

The amount of computations performed naturally increases as the number of neurons and the number of samples increase. If the number of neurons is R and the number of samples is C, then the weight matrix W is R by C. One training cycle consists of multiplying W with an input vector P having C elements, finding the largest element in the resulting R by 1 vector, and then updating the winner and its neighbours. Taking the inner product of two vectors with C elements requires $2C$ floating point operations (flops), finding the largest element in a vector with R elements requires R flops, and updating

one weight vector requires $3C$ flops. The number of neurons that have their weights updated varies over the training cycles. If we assume that an average of N neurons have their weights updated each cycle then one training cycle requires $2CR + R + 3NC$ flops, that is the number of flops per training cycle increases linearly both with the number of neurons and the number of samples. Evaluating one input vector will require $2CR + R$ flops, which is a rather limited amount of computation. This means that once the net is trained, it is potentially very fast and could work in real time.

Experiments

Experiments were done both on the step responses and the impulse responses. A few testruns showed that nine neurons did not give any useful results for nets trained on step responses. Starting at twelve neurons experiments were performed on full training vectors, training vectors with every fifth sample, and vectors with every tenth sample. For vectors trained on impulse responses, testing began at nine neurons. After training a net it was tested for how many of the training vectors it could distinguish between. This is to find out which responses a net is able to recognize as well as to find out which responses are *similar to the net.*

Nets trained on every tenth sample seemed somewhat ambiguous, especially when tested on oscillating signals. This is probably because too much information is neglected and can be compared with too slow a sampling rate. We therefore concentrate on comparisons between nets trained on vectors with all of the 101 samples and vectors with every fifth sample.

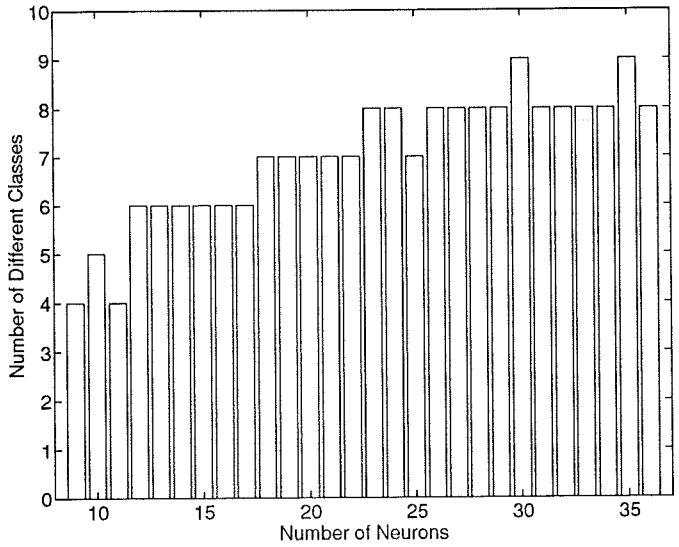

Figure 33. The number of different classes nets were able too distinguish between depending on the number of neurons in the net. The nets were trained on impulse responses with vectors containing every fifth sample.

The number of different classes that a net is able to distinguish between depends on a number of factors. The number of neurons is the most obvious factor to consider. This is illustrated in figure 33 where the number of different classes are shown as a function of the number of neurons. Nets with more neurons are evidently able to distinguish between more vectors. The number of training cycles is another factor of importance. We can for example see

that a net with twelve neurons was able to single out six different classes after training for 100 000 cycles. When training for fewer cycles, the nets were able to distinguish between fewer classes. Training for more and more cycles show that there is a limit for each amount of neurons. Tests were conducted from 20 000 to 200 000 training cycles. Also, nets with the same amount of neurons trained for the same amount of cycles were not always able to distinguish between the same same amount of classes. This occurred mainly when relatively few training cycles were used, which means that the nets do not have time to become fully ordered. Thus the final ordering depends more on the initial weight values, which are random.

Results for Step Responses

As mentioned above, the number of different classes that a net can distinguish between depend mainly on the number of neurons and the number of training cycles. As the number of different classes increase we are interested in which steps the nets are able to single out. Note that there is no difference if another step is singled out due to an increased number of neurons, or because the net was trained for a longer time.

The nets trained on step responses with every fifth sample were at their worst able to group the training batch $P_1 \ldots P_9$ into four different classes, and at best into eight different classes. The nets trained on full training vectors could at their worst find five different classes, and nine at their best. There was no difference between the nets trained on every fifth sample and nets trained on full vectors as to the order in which the different step responses emerged.

Four classes: P_5 and P_9 are singled out, $\{P_1, P_4, P_8\}$ are considered as one group (group A), and $\{P_2, P_3, P_6, P_7\}$ as another (group B).

Five classes: P_4 is now distinguished from group A.

Six classes: P_7 is separated from group B.

Seven classes: Two different possibilities are manifested; either P_1 can be distinguished from P_8 and group A is split up, or P_6 can be singled out from group B. The two cases seem equally likely.

Eight classes: Group A exists no more. Group B have two constellations: $\{P_3, P_6\}$ or $\{P_2, P_3\}$

Nine classes: Well, it's obvious, isn't it.

Nets trained on full vectors needed 14–15 neurons to find six classes, while nets trained on every fifth sample needed 16–17 neurons. To find seven classes full vector nets needed about 19–20 neurons, and nets trained on the smaller vectors needed 20–21. Full vector nets could find eight different classes with 21–22 neurons, when nets trained on every fifth sample needed 24–25. At 30 to 35 neurons the nets trained on full vectors could single out nine classes.

Results for Impulse Responses

Impulse responses seem to be easier to separate. This is clear from the fact that nets trained on impulse responses could find the same amount of classes as nets trained on step responses, with fewer neurons. Both nets trained on full vectors and on every fifth sample could find between four and nine different classes. The order in which the classes emerge was independent of the number of samples used.

Four classes: $\dot{P}_5$ is singled out. The rest are collected in three groups; $\{\dot{P}_1, \dot{P}_4, \dot{P}_8\}$ (group A), $\{\dot{P}_2, \dot{P}_3, \dot{P}_6\}$ (group B), and $\{\dot{P}_7, \dot{P}_9\}$ (group C).

Five classes: Either $\dot{P}_4$ is singled out from group A, or group C is split.

Six classes: A lot of different cases occur.

- Group A is reduced to $\{\dot{P}_1, \dot{P}_8\}$ and group C is split. Group B intact.
- group A is reduced to $\{\dot{P}_1, \dot{P}_4\}$, group C is split, and group B intact.
- Two new groups are formed: $\{\dot{P}_2, \dot{P}_4\}$, and $\{\dot{P}_3, \dot{P}_6, \dot{P}_9\}$, while the rest are uniquely identified.
- One new group is formed: $\{\dot{P}_1, \dot{P}_2, \dot{P}_6, \dot{P}_7\}$, the rest are identified.
- Another new group: $\{\dot{P}_2, \dot{P}_3, \dot{P}_6, \dot{P}_9\}$, identifying the rest uniquely.

Seven classes: Back to simplicity, only two different constellations. Either group A remains $\{\dot{P}_1, \dot{P}_8\}$ and group B is reduced to $\{\dot{P}_3, \dot{P}_6\}$, or group A is split up leaving group B $\{\dot{P}_2, \dot{P}_3, \dot{P}_6\}$. Group C is split up for good.

Eight classes: $\{\dot{P}_2, \dot{P}_3\}$ is the only remaining group.

To find five different classes, both kinds of net needed 9–10 neurons. Six classes were found with 11–12 neurons. Nets trained on full vectors could find seven classes with 14–15 neurons, while nets trained on every fifth neuron needed 17–18 neurons. Full vector nets found eight classes with 20–22 neurons, when nets trained on the smaller vectors needed 23–24 neurons. Both types needed around 30–33 neurons to find all nine different classes.

As we can see, the nets trained on impulse responses are able to distinguish between more different classes with fewer neurons, than nets trained on step responses. This is probably because the step response is the integral of the impulse response, and when we integrate we always lose some features. This means that step responses will look more alike than impulse responses. It is also clear that nets trained on vectors with every fifth sample need a few more neurons than nets trained on full vectors to find the same amount of classes. This can be due to the fact that the vectors containing fewer samples also contain less information. Those vectors have fewer distinguishing properties, so we need more neurons to separate them.

Noise

Feature maps were trained on sets with only unnoisy vectors and sets containing both noisy and unnoisy versions of the training batch $P_1 \ldots P_9$. This is easily done with the `trainfm` function, as it randomly choses one vector from the training batch and presents it to the net. When training nets on both noisy and unnoisy vectors, we only need to supply a training batch containing one set of each type.

When evaluating a net for noise sensitivity, each vector was presented 100 times to the net with a new noise series added every time. For the evaluation of a net, an unnoisy vector was presented to begin with. The output was then considered as the correct output when evaluating the output from the net when a noisy input vector was applied. The noise evaluation procedure was done with a MATLAB function `noisetest`, which applies noise up to a given level in five equidistant steps. It is a function that was written especially for our own purposes. The nets' sensitivity to noise depend basically upon four things:

- The type of transient response. The nets trained on impulse responses had a significantly lower error rate than those trained on step responses. As stated in the previous section, this is probably because impulse responses contain more distinguishing features than step responses and can therefore be more easily recognized.

- The number of samples. Nets trained on full vectors had a decidedly lower error rate than those trained on every fifth sample. With fewer points to average out the effects of noise, this seems like a plausible effect of training with fewer samples.
- The number of classes the nets were able to distinguish between. Nets that could separate fewer classes had a lower error rate. If a net is able to distinguish between fewer classes, then there are fewer classes to make mistakes about.
- Training with or without noise. Nets trained without noise have a considerably lower error rate. This is discussed below.

The number of neurons did not seem to affect the noise sensitivity.

Nets were trained on vectors not affected by noise, on mixes of unnoisy vectors and vectors affected with 3% noise, and a mix including vectors with 7% noise added. Those nets will for brevity be referred to as nets trained on noisy vectors. One important observation from training with noise added, is that those nets can distinguish between fewer classes. Typically, nets trained on noisy vectors were able to separate one or two classes less than their unnoisy counterparts.

Also, nets trained on noisy vectors were more sensitive to noise. Nets trained on noisy vectors show a significantly higher error rate than their comparable unnoisy counterparts. By comparable we here mean nets that have approximately the same number of neurons, and can distinguish between the same amount of classes. Consider figure 34. Two nets with 16 neurons were trained on step responses with all of the 101 samples. They were trained on vectors without noise (net A), and vectors with 7% noise applied (net B). Both nets were able to single out six different classes. The input vectors were the nine training vectors, each being presented 100 times per vector and noise level to the nets. As we can see, net A has an overall higher rate of correct classification than net B.

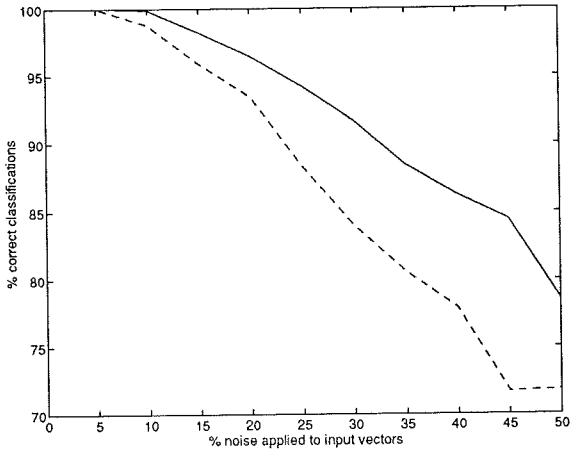

Figure 34. Percentage correct classifications as a function of the amount of noise applied to the input vectors, containing all of the 101 samples of the step responses $P_1 \ldots P_9$. Solid line for the net trained on vectors without noise, dashed line for net trained on vectors with 7% noise applied.

The effect of applying noise to training vectors can be seen in figure 35, where the weight vectors for the two nets described above are plotted. It is obvious which weight vectors belong to the net trained on noisy vectors.

Clearly, it must be more difficult for nets with weight vectors such as the ones depicted to the left, to distinguish between inputs. Further, the nets are trained on one noise sequence. No two noise sequences are alike, so training on one noise sequence does not make the corresponding weight vector more similar to the input vector when another sequence is applied when testing. Probably it is more *unlike* the noisy input vector. However, the noise has zero mean. So if we were to train on a lot of different noise sequences, that would be something like training without noise. As we showed above, those nets are considerably less sensitive to noisy inputs.

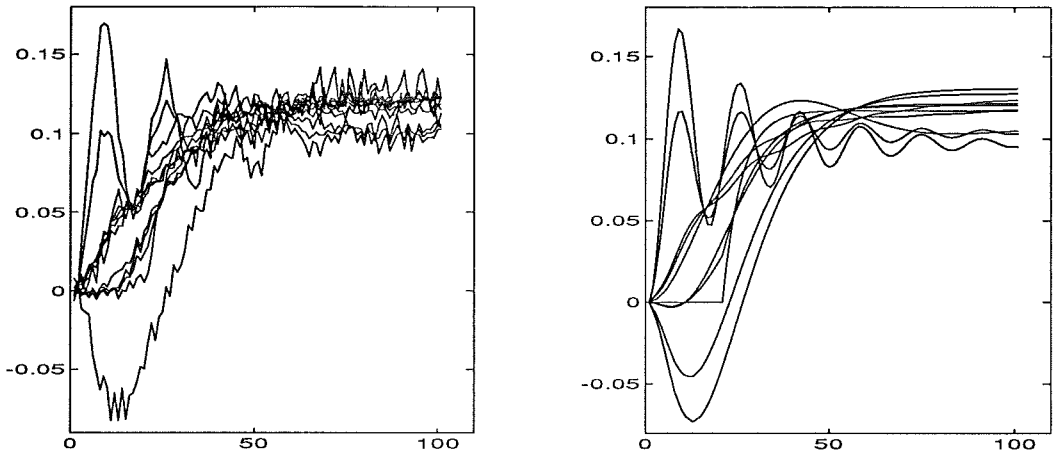

Figure 35. The weight vectors for two 16 neuron nets. To the left, a net trained with 7% noise; to the right a net trained without noise.Note how the weight vectors have adapted to the values of the training vectors.

As we have shown above, nets trained without noise are the least sensitive to noisy input vectors. There are also rather big differences between nets trained on step- and impulse responses, and between nets trained on full vectors versus nets trained on every fifth sample. This is shown in figure 36. There the noise sensitivity of four 16-neuron nets is illustrated, one each of the above described types. All of the nets were trained on noise free vectors. The

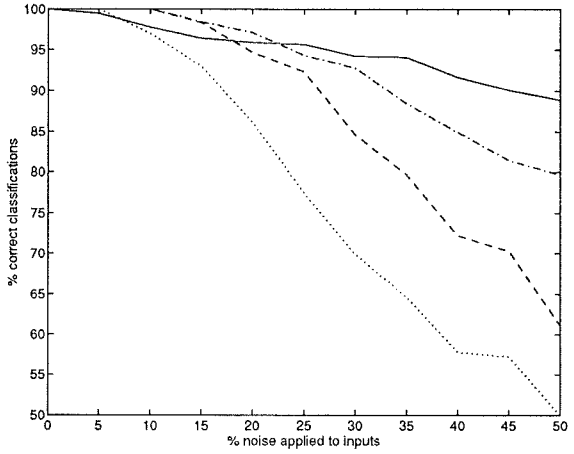

Figure 36. Percentage correct classifications as a function of applied noise. Solid line: Impulse response, full vectors; dashed line: Impulse response, every fifth sample; dash-dotted: Step response, full training vectors; dotted: Step responses, every fifth sample.

impulse response nets, trained on full vectors are generally much less affected by noise. They have between 80% and 92% correct classifications with 50% noise applied to input vectors, varying with the factors stated above.

Testing on Unknown Inputs

One of the main advantages of using neural networks is their ability to classify systems they have never seen before. The idea is to train the nets on typical examples, and then let the nets evaluate unknown signals. When we use nets trained with unsupervised learning we need a method to evaluate the output from the net. As the nets have organized themselves in some way, we cannot say for example that if neuron four won the competition, then it is an oscillating system. The way we solved this problem was to have an evaluation function to which we supplied the training batch along with the test responses together with the weight matrix of the net we wished to examine. The number of the winning neuron for each of the training vectors was computed, and then compared with the number of the winning neuron when an unknown signal was applied to the net. The unknown signal was classified as being of the same type as the training signal with the winning neuron closest to its own winning neuron. For example, if the inverse response Q_9 was applied to the net in figure 31, and neuron number one won the competition, then Q_9 would be classed as being of the same type as P_9, as this is the training vector to which neuron one corresponds. This is equivalent to providing a lookup table of classifications constructed after completed training.

The nets were tested on the step responses $Q_1 \ldots Q_9$ in figure 15, the corresponding impulse responses $\dot{Q}_1 \ldots \dot{Q}_9$ in figure 16, and on several other similar vectors where parameters were slightly changed. Results for $Q_1 \ldots Q_9$ and $\dot{Q}_1 \ldots \dot{Q}_9$ are representative of the entire collection of test vectors.

The results from these tests are similar for nets trained on step- and impulse responses. The winning neuron when a test vector is used as input is almost always the same as one of the winning neurons when one of the training vectors is used. The classification of the test vectors did not vary much for nets with more than 14 to 15 neurons. The number of different classes that the nets were able group the training vectors into made no difference when classifying the test vectors. The sensitivity to noise was not surprisingly *slightly* higher than the sensitivity when testing the nets with the training vectors. There was no difference between nets trained on every fifth sample and nets trained on full vectors. Typical classifications for nets trained on step responses are:

- Q_1 (Monotonous) classed as P_1 (Monotonous)
- Q_2 (Monotonous) classed as P_2 (Monotonous)
- Q_3 (Monotonous) classed as P_2 or P_3 (Monotonous)
- Q_4 (Essentially monotonous) classed as P_1 (Monotonous)
- Q_5 (Oscillating) classed as P_4 (Essentially monotonous)
- Q_6 (Oscillating) classed as P_5 (Oscillating)
- Q_7 (Delay) classed as P_7 (Delay)
- Q_8 (Essentially monotonous) classed as P_1 (Monotonous)
- Q_9 (Inverse response) classed as P_9 or sometimes as P_6 (Inverse response)

As we can see, the classifications are basically correct, although not entirely. Q_4 is really an oscillating step response, although very damped, so mistaking it for the monotonous P_1 is nothing to lose sleep over. Q_5 is as we can see a

moderately damped oscillating signal. It would perhaps have been more desirable to classify it as P_5 rather than P_4 which is even more damped. This could possibly be dealt with when choosing training vectors: it is probably useful to have a more smooth transition between very damped oscillating systems and weakly damped systems. The same could be said about the inverse responses P_6 and P_9. Q_8 is essentially monotonous, so classifying it as P_1 does not seem so bad. Typical classifications for nets trained on impulse responses are:

- $\dot{Q}_1$ (Monotonous) classed as $\dot{P}_1$ (Monotonous)
- $\dot{Q}_2$ (Monotonous) classed as $\dot{P}_2$ (Monotonous)
- $\dot{Q}_3$ (Monotonous) classed as $\dot{P}_2$ or $\dot{P}_3$ (Monotonous)
- $\dot{Q}_4$ (Essentially monotonous) classed as $\dot{P}_4$ or $\dot{P}_2$ (Essentially monotonous or monotonous)
- $\dot{Q}_5$ (Oscillating) classed as $\dot{P}_1$ (Monotonous)
- $\dot{Q}_6$ (Oscillating) classed as $\dot{P}_5$ (Oscillating)
- $\dot{Q}_7$ (Delay) classed as $\dot{P}_7$ (Delay)
- $\dot{Q}_8$ (Essentially monotonous) classed as $\dot{P}_8$ or $\dot{P}_1$ (Essentially monotonous or monotonous)
- $\dot{Q}_9$ (Inverse response) classed as $\dot{P}_9$ (Inverse response)

As we can see, these nets can more easily identify $\dot{Q}_4$ and $\dot{Q}_8$. On the other hand $\dot{Q}_5$ is wrongly classified as being monotonous. The same suggestions as those stated for step responses apply here too.

Two Concluding Nets

We will now present two nets that were created with all of the experiences gained during the work in mind. One net was trained on step responses and the other on impulse responses. Both nets were trained on full vectors, without noise applied, for 100 000 cycles with an initial learning rate of 0.005. The nets had 18 neurons, which seems to be a reasonable amount considering computational complexity versus resolution. It also happens to be twice the number of the training vectors. As training vectors, we used a modified selection of $P_1 \ldots P_9$ where we took into consideration the discussion about the oscillating- and inverse response cases above. The step- and impulse responses, which we will refer to as $T_1 \ldots T_9$ and $\dot{T}_1 \ldots \dot{T}_9$, are illustrated in figures 37 and 38. P_3 and P_6 have been replaced by one oscillating response that has a damping which lies between P_4 and P_5, and one inverse response which could be placed between P_6 and P_9. None of the new responses have the same transfer function as the responses in $Q_1 \ldots Q_9$, which were used as unknown inputs to evaluate the new nets.

When training was completed, the net trained on the step resopnses could single out all of the nine training steps. This was a little surprising, since we needed more than 30 neurons when training on the original $P_1 \ldots P_9$, and with 18 neurons the nets could typically find six to seven classes. One explanation can be that $T_1 \ldots T_9$ are a little more unlike each other than $P_1 \ldots P_9$ are. A 3-D plot of the weight vectors is shown in figure 39. When testing the net with the training vectors we learn how the net has ordered the step responses:

- $T_1 \mapsto$ neuron 11
- $T_2 \mapsto$ neuron 7
- $T_3 \mapsto$ neuron 13
- $T_4 \mapsto$ neuron 16

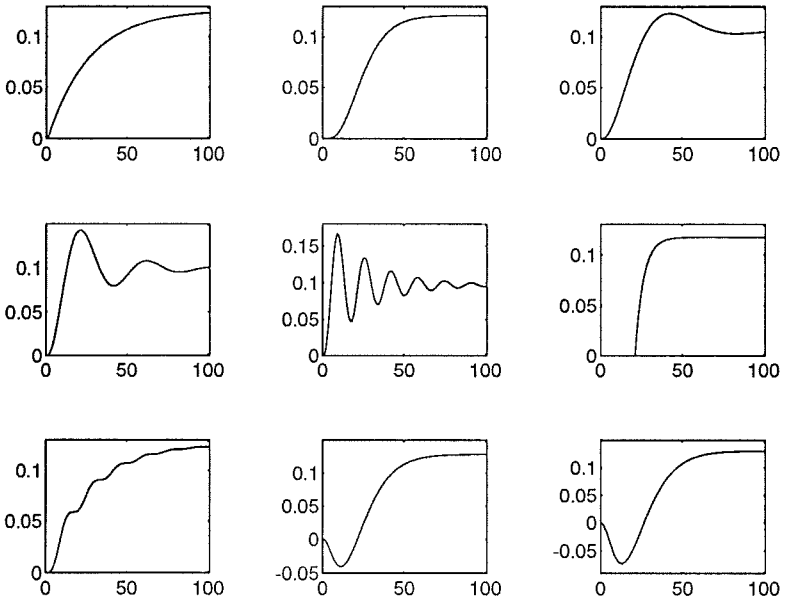

Figure 37. The new training batch of step responses, $T_1 \ldots T_9$.

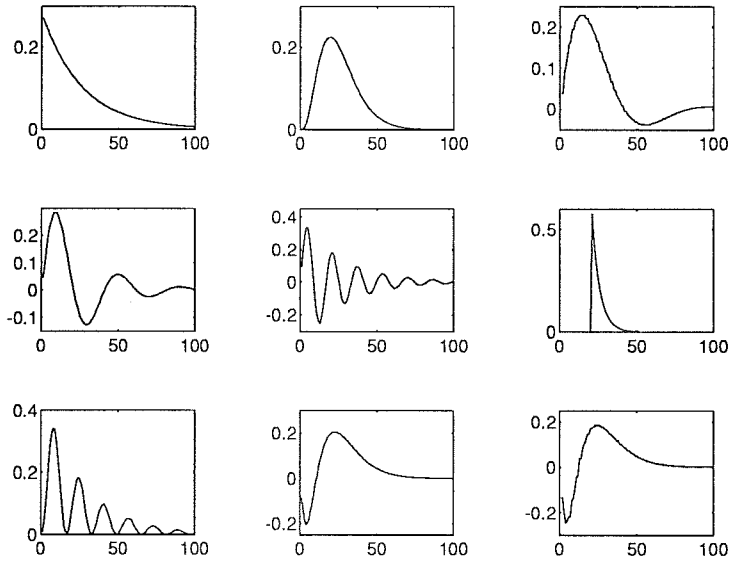

Figure 38. The training batch of impulse responses $\dot{T}_1 \ldots \dot{T}_9$.

- $T_5 \mapsto$ neuron 18
- $T_6 \mapsto$ neuron 5
- $T_7 \mapsto$ neuron 9
- $T_8 \mapsto$ neuron 3
- $T_9 \mapsto$ neuron 1

This tells us for example that T_8 and T_9 are very much alike, because nearby neurons win the competition when they are used as inputs. We can also deduce that T_5 and T_9 are the step responses that are the most different from each other; neurons on either side of the net win when they are applied.

The net trained on impulse responses could distinguish between seven different classes, which is the same amount as the 18-neuron net trained on

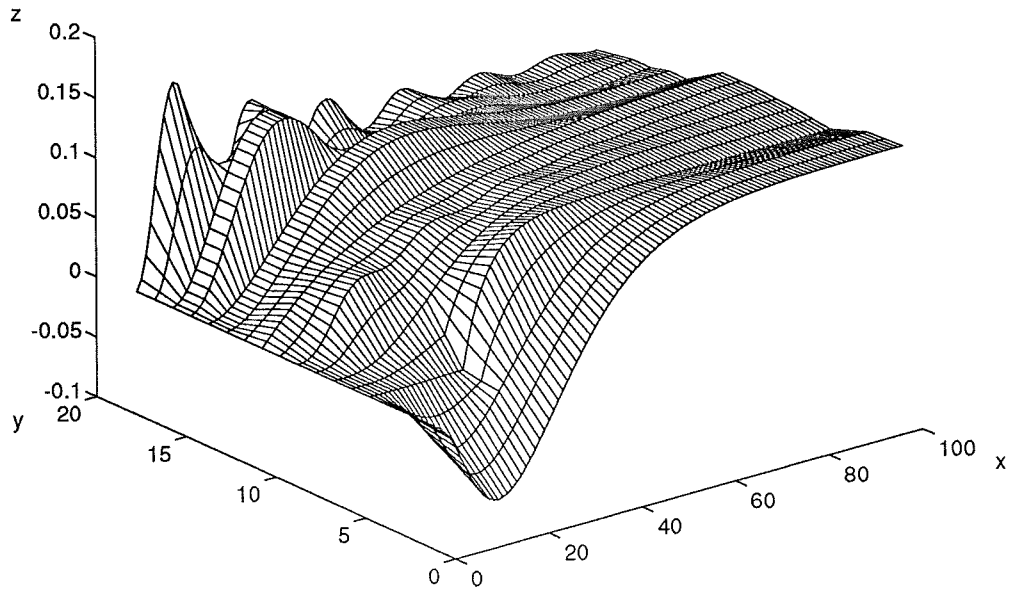

Figure 39. The weight vectors for the net trained on the step responses $T_1 \ldots T_9$. The samples are on the x-axis, the neuron number on the y-axis, and the value of each weight w_{ij} on the z-axis.

$\dot{P}_1 \ldots \dot{P}_9$ could. The weight vectors are illustrated in figure 40. The ordering of the impulse responses is not the same as that of the step responses:

- $\dot{T}_1 \mapsto$ neuron 17
- $\{\dot{T}_2, \dot{T}_3\} \mapsto$ neuron 5
- $\dot{T}_4 \mapsto$ neuron 15
- $\dot{T}_5 \mapsto$ neuron 10
- $\dot{T}_6 \mapsto$ neuron 1
- $\dot{T}_7 \mapsto$ neuron 18
- $\{\dot{T}_8, \dot{T}_9\} \mapsto$ neuron 3

Here we see that two pairs of impulse responses are so much alike that they are classed as the same, $\{\dot{T}_8, \dot{T}_9\}$ and $\{\dot{T}_2, \dot{T}_3\}$. The most different steps are $\dot{T}_6$ and $\dot{T}_7$.

One thing that may seem a little odd is that $\dot{T}_1$ and $\dot{T}_7$ appear to be very much alike. This is obviously true for the step responses, but does not at first glance seem to be the case with the impulse responses. However if they are both drawn in the same figure as in figure 41, we see that the amplitude of the oscillating $\dot{T}_7$ decreases with about the same rate as $\dot{T}_1$ decreases.

The net trained on impulse responses was, just as we saw above, much less sensitive to noise. Note however that the net trained on step responses could distinguish between more classes than the impulse response net. As we showed, this is a factor to take into account when evaluating noise sensitivity. A comparison when the noisy training vectors were used as inputs is shown in figure 42. When evaluating the output from the nets, we provided a table with acceptable outputs for each input. If for example T_8 or T_9 were used as

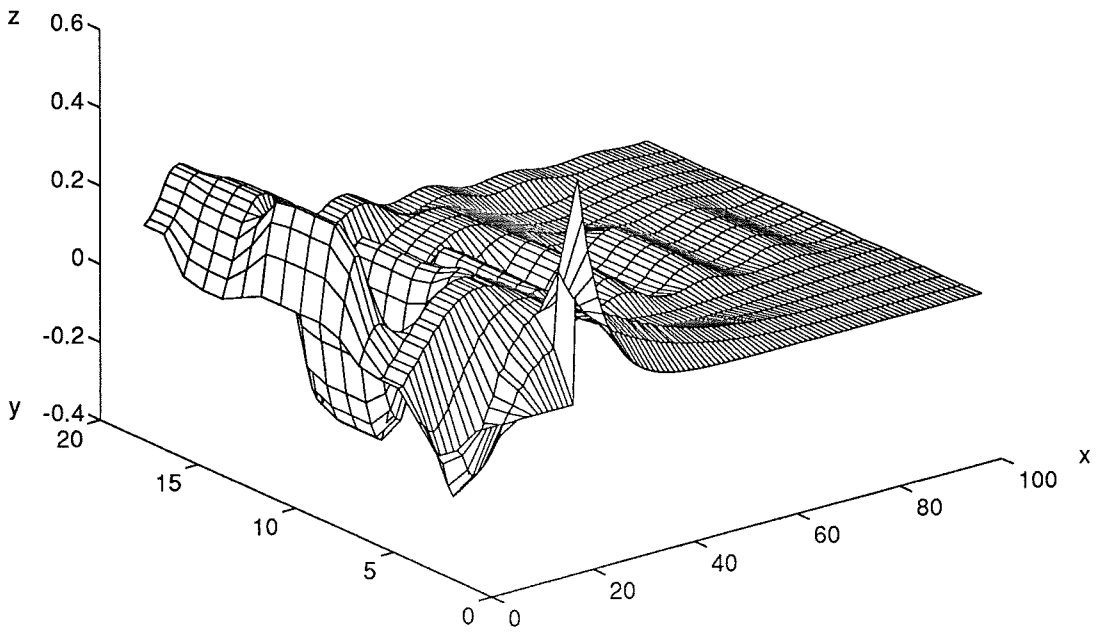

Figure 40. The weight vectors for the net trained on the impulse responses $\dot{T}_1 \ldots \dot{T}_9$.

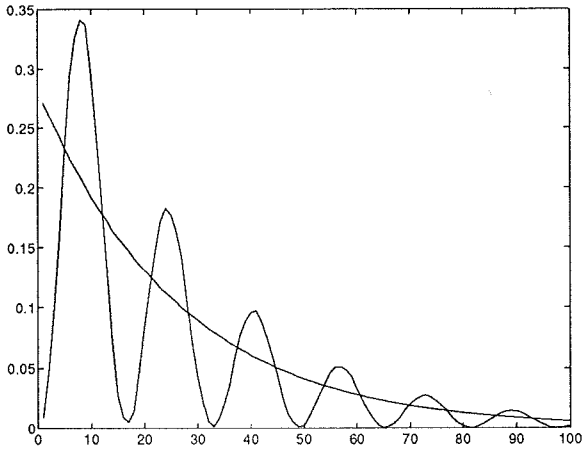

Figure 41. The impulse responses $\dot{T}_1$ and $\dot{T}_9$.

inputs, the output was considered correct if any of the neurons one through four had the greatest value. The neurons considered as correct winners for each input were decided by examining the weight matrices in figures 39 and 40, and comparing it with the output when unnoisy inputs were used.

Providing such a table naturally leads to a higher degree of correct classifications than what we saw in the previous noise section. However, it is a more realistc approach since this is how you would use it when operating in an application. The noise sensitivity of the step responses was varying to a high degree for the different steps. T_1, T_2 and T_7 were decidedly more sensitive, and had a correct classification rate of about 50% when applying 50% noise. The other steps had 100% correct classification up to, and including, 20% noise.

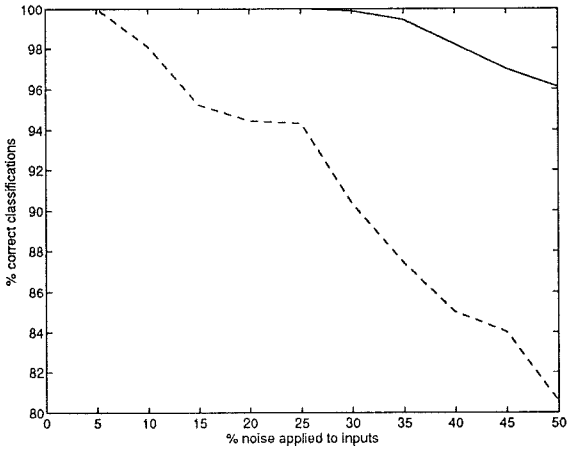

Figure 42. Correct classifications as a function of the amount of noise applied to the input vectors. Results for the net trained on impulse responses are drawn with a solid line, step responses with a dashed.

The impulse responses were as we can see very insensitive to noise.

Last, we tested the nets on the Q and $\dot{Q}$ batches of test vectors described previously. The nets were tested on unnoisy as well as noisy testvectors. The error rate was only somewhat higher than what we got when we tested on the training batch. See figure 43. Both nets classified the test vectors correctly. In

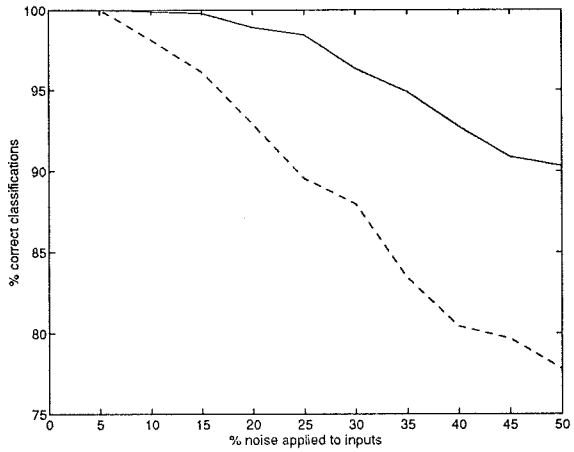

Figure 43. The percentage of correct classifications when the test batch Q with varying degrees of noise was used as input. Solid line for impulse responses, dashed for step responses.

fact, both nets classified all of the test vectors in the same way:

- Q_1 (Monotonous) classed as T_1 (Monotonous)
- Q_2 (Monotonous) classed as T_2 (Monotonous)
- Q_3 (Monotonous) classed as T_2 (Monotonous)
- Q_4 (Essentially monotonous) classed as T_2 (Monotonous)
- Q_5 (Oscillating) classed as T_4 (Oscillating)
- Q_6 (Oscillating) classed as T_5 (Oscillating)
- Q_7 (Delay) classed as T_6 (Delay)
- Q_8 (Essentially monotonous) classed as T_1 (Monotonous)
- Q_9 (Inverse response) classed as T_8 (Inverse response)

Conclusions

The original question which we set out to answer was if it is possible to classify system dynamics with neural networks. It rather early became clear that Kohonen nets (SOFMs) are able to perform this task. We have seen how the weight vectors of the nets beautifully adapt to the shapes of the training vectors, and found a theoretical ground to lean on concerning ordering and convergence of weight values [Kohonen, 1989].

When it had become clear that it is possible to use SOFMs for system classification, we proceeded to investigate some of the variables that may effect the nets' performance. We started out by only investigating step responses, for which the classifications " monotonous", "essentially monotonous", etc. are obvious from inspection. When contemplating how to make different systems look more different, we decided to differentiate the step responses, i.e. study the impulse responses too. Both step- and impulse responses were then investigated in parallel.

The first thing we looked into was how many neurons were necessary. Training on the original training vectors $P_1 \ldots P_9$ showed that it is possible to identify all nine of them, but that we had to use 30–35 neurons to do this. Nets trained on impulse responses needed 2–3 neurons less than nets trained on step responses to find the same number of different classes. When more care was taken to choose the training vectors, we could find all of the nine different training vectors using nets with a little more than half the number of neurons as were needed earlier. The number of samples used in the training vectors is of interest, since fewer samples means less calculations. We discovered that when training on vectors with fewer samples, we need more neurons to find the same number of classes as when training on vectors with more samples.

When applying noise to input vectors, it became evident that fewer neurons gave bad results. Nets trained on every fifth sample are considerably more sensitive to noise than nets trained on full vectors. The nets trained on impulse responses emerged as least noise sensitive. Training nets with noisy vectors showed that this is not something we want to do. Those nets were able to separate fewer classes, and they were more sensitive to noise.

As we evaluated the nets with unknown inputs, we discovered that the number of different classes they could distinguish between did not matter all that much. We wanted to classify the unknown inputs as one of the known training vectors, and this worked well. It also became clear that great care must be taken when chosing training vectors, and we must really be aware of which classifications we want to make. Nets trained on $P_1 \ldots P_9$ and on $\dot{P}_1 \ldots \dot{P}_9$ showed some differences when classifying the unknown inputs. The most severe case was that the nets trained on impulse responses mistook an oscillating test vector ($\dot{Q}_5$) as being monotonous ($\dot{P}_1$). When training nets on a more carefully selected batch, however, no such mistakes were made, and nets trained on both kinds of responses classified the unknown inputs in exactly the same ways.

To summarize, we can state that nets trained on impulse responses are considerably less sensitive to noise, than are nets trained on step responses. Also, the more samples that are used, the less sensitive are the nets. With carefully selected training vectors, the classification of unknown inputs will not differ depending on what kind of training vectors have been used.

6. Tools

The way neural networks organize themselves when trained is not completely understood. How to choose the best configuration of a network for a particular task can therefore not entirely be derived "analytically", so much of our work consisted of making computer simulations of different network configurations. In this section we will describe the hardware and software we used, and also give some hints on how we would consider implementing a neural net in software.

Hardware and Software

We did our work on Sun SPARC stations 1 and 2, running SunOS 4.1.2, connected to a SPARC Server 690 in a network at the Department of Automatic Control in Lund. These machines are pretty powerful, which is needed when training large nets. Training a 15 neuron SOFM for 100 000 cycles still required around 45 minutes of CPU time.

All of the simulations were done in MATLAB version 4.1, using the Neural Network Toolbox version 1.0c by Mark Beale and Howard Demuth [Beale and Demuth, 1992]. As we have described, training and using neural networks consists mainly of vector and matrix operations. This is exactly what MATLAB is designed for, which makes it an ideal tool for experimenting with different network configurations.

The Neural Network Toolbox implements training algorithms for different kinds of nets, seven kinds of neuron transfer functions, neighbourhood functions for the Kohonen nets, and other useful functions for creating and examining neural networks. After training, the nets are represented by one or more weight matrices, depending on the number of neuron layers. These weight matrices, created with the algorithms in the toolbox, were used for evaluating the corresponding net configuration. For evaluation of the nets, we wrote small MATLAB scripts to add noise, check classifications, evaluate known and unknown input vectors etc. The way MATLAB works makes it easy to evaluate large test batches of input vectors on many different nets, and thus gathering a lot of information about how the nets behave. Most of the figures in this report were generated in MATLAB, for example all diagrams and plots of matrices and vectors. Those figures that were not generated in MATLAB, were made using a drawing program called Xfig. This report was written using LaTeX.

Hints on Implementation

Implementation of a specific task in software would probably make training and operation of a network a lot more efficient. Though MATLAB has highly optimized vector and matrix operations, such things as loops take considerable time. One book that provides good background and programming suggestions for implementing neural networks is [Freeman and Skapura, 1991]. We would however like to give some of our points of view.

As we have stated several times, the nets we have studied are mainly represented by one or more weight matrices, and training them involves primarily vector and matrix operations. We therefore suggest that a set of functions for doing vector and matrix operations are implemented, preferably in an object oriented language such as C++. Classes for vector and matrix manipulations independent of the number of elements, like in MATLAB, would be useful. A layer class could be represented by an input vector, the weight matrix between

the layer's input and the layer itself, a user supplied transfer function for the current layer, and a vector representing the output from the layer.

A backpropagation net could then be a class consisting of the appropriate number of layers. This class should also contain functions for calculating the output from the whole net and for training it. The training needs functions for computing the error between the output from the net and the desired output, for backpropagating the error, and for updating the weights in each layer.

A Kohonen net is not so complicated as a backpropagation net. The net is represented with only one weight matrix. A SOFM class would then consist of one input vector, one output vector, and a weight matrix. The training needs a neighbourhood function, and a function for calculating the lateral interaction, i.e. some approximation of the Mexican hat function, and a function for appropriately updating the weights. A competition function to give the winning neuron should also be supplied.

After training, the nets have to be evaluated and provided with some lookup table that can interpret the output from the output from the net. Once trained and evaluated the nets should be ready to use. What we have described here are only brief suggestions on how we think a good implementation could be conceived. The literature includes some other suggestions and anyone interested in implementing a neural network should look into some different books.

7. Conclusions

This work has clearly shown that it is possible to perform rough classification of system dynamics using both the backpropagation net and the Kohonen net. Both types of nets are capable of identifying the different classes as desired, and their ability to generalize is good. The sensitivity to noise is low, especially for nets trained on impulse responses.

The backpropagation net and the Kohonen net appear equally well suited for the task of rough classification. The nets differ mainly in how we can control what they learn. The target vector used when training the backpropagation net gives us the authority to decide which features we consider important. A SOFM on the other hand, organizes itself during training, which yields an intuitive generalization ability, but it has perhaps not the same resolution as the backpropagation net. When in operation, the SOFM requires considerably less floating point operations than a backpropagation net, an important feature for real time applications.

It has become perfectly clear that nets trained on impulse responses perform better than nets trained on step responses. They are considerably less sensitive to noise, and show a slightly better ability to generalize. This applies to both the backpropagation net and the Kohonen net. To achieve a good ability to generalize, it is important to normalize the inputs. We normalize the inputs in time so that only the significant part of the signal is considered. Normalizing in space after normalizing in time makes similar inputs (e.g. inputs of the same order) look very much alike. Normalizing in space means scaling the vectors to unit length. Using input vectors of the same length is crucial, especially for the Kohonen net. When considering noise sensitivity, we found that the fewer samples used, the more sensitive were the nets. The nets should for best results be trained on inputs with no noise applied.

To sum up the results, we can say that both the backpropagation and the Kohonen net are able to perform system classification. Regarding the type

of input, there is no doubt that impulse responses with not too few samples should be used.

What's Next?

We have shown that it is possible to use neural nets to roughly classify system dynamics. One question that immediately arises is, can we use nets for finer classification? Is it possible to use neural networks to, for example, find the order of a system? We made some brief examinations on monotonous systems with $G(s) = 1/(s + 0.75)^n$, $n = 1 \ldots 9$. SOFMs trained on both step- and impulse responses could uniquely identify systems with $n = 1 \ldots 4$. BPNs could identify all of the nine responses, which illustrates their greater resolution. More detailed experiments could show exactly what information it is possible to extract from the transient responses. It might be possible to use several consecutive nets to get a more and more detailed classification of a system. As is usually the case when you study things with some care, each new discovery immediately resulted in more questions, but with limited time at hand we were not able to pursue all of the interesting sidetracks that we encountered.

8. Acknowledgements

We would like to thank our supervisor, Professor Karl Johan Åström for his inspiring guidance, and for keeping us on the right track. Eva Dagnegård has provided invaluable help with LaTeX. Charlotta Lindquist has proven her angle like patience during lunches when nothing but neural nets and LaTeXwas discussed. She also helped with the proof reading.

9. References

BEALE, M. and H. DEMUTH (1992): *Neural Network Toolbox User's Guide.* MathWorks, final draft edition.

CARLING, A. (1992): *Introducing Neural Networks.* Sigma Press.

DARPA (1988): *Neural Network Study*, chapter 8.4, pp. 93–95. AFCEA International Press. Study Director: Bernard Widrow.

FREEMAN, J. A. and D. M. SKAPURA (1991): *Neural Networks Algorithms, Applications and Programming Techniques.* Addison–Wesley.

HECHT-NIELSEN, R. (1989): *Neurocomputing.* Addison–Wesley.

HERTZ, J., A. KROGH, and R. G.PALMER (1992): *Introduction to the Theory of Neural Competition.* Addison–Wesley.

HINTON, G. E. (1992): "How neural networks learn from experience." *Scientific American*, **267:3**, pp. 104–109. Special Issue: Mind and Brain.

KOHONEN, T. (1988): "The 'neural' phonetic typewriter." In SÁNCHEZ-SINENCIO and LAU, Eds., *Artificial Neural Networks. Paradigms, Applications, and Hardware Implementations.* IEEE Press. IEEE Computer Magazine (March 1988).

KOHONEN, T. (1989): *Self–Organization and Associative Memory.* Springer–Verlag.

KOHONEN, T., G. BARNA, and R. CHRISLEY (1990): "Statistical pattern recognition with neural networks: Benchmarking studies." In *IEEE Conference on Neural Networks, San Diego*, volume I. IEEE.

KONG, S.-G. and B. KOSKO (1992): "Differential competitive learning for phoneme recognition." In KOSKO, Ed., *Neural Networks for Signal Processing*. Prentice Hall.

KUNG, S. and J. HWANG (1990): "An algebraic projection analysis for optimal hidden units size and learning rates in back–propagation learning." In *IEEE Conference on Neural Networks, San Diego*, volume I. IEEE.

MCCLELLAND, J. and D. RUMELHART (1986): *Parallel Distributed Processing, volumes 1 and 2*. MIT Press, Cambridge, MA.

MCCORD NELSON, M. and W. ILLINGWORTH (1991): *A Practical Guide to Neural Nets*. Addison–Wesley.

NASRABADI, N. M. and Y. FENG (1990): "Vector quantization of images based upon the kohonen self–organizing feature maps." In *IEEE Conference on Neural Networks, San Diego*, volume I. IEEE.

NGUYEN, D. and B.WIDROW (1990): "Improving the learning speed of 2-layer neural networks cy choosing initial values of the adaptive weights." In *International Joint Conference of Neural Networks*, volume 3, pp. 21–26.

OJA, E. and T. KOHONEN (1990): "The subspace learning algorithm as a formalism for pattern recognition and neural networks." In *IEEE Conference on Neural Networks, San Diego*, volume I. IEEE.

PARKER, D. (1986): "Technical report tr–47." Technical Report, Center for Computationl Research in Economics and Management Science, MIT, Cambridge, Ma.

RITTER, H., T. MARTINETZ, and K. SCHULTEN (1992): *Neural Computation and Self–Organizing Maps*. Addison–Wesley.

RITTER, H. and K. SCHULTEN (1990): "Kohonen's self–organizing maps: Exploring their computational capabilities." In *IEEE Conference on Neural Networks, San Diego*, volume I. IEEE.

VAN CAMP, D. (1992): "Neurons for computers." *Scientific American*, **267:3**, pp. 125–127. Special Issue: Mind and Brain.

WERBOS, P. (1974): *Beyond Regressions: New Tools for Prediction and Analysis in the Behavioral Sciences*. PhD thesis, Harvard, Cambridge, Ma.