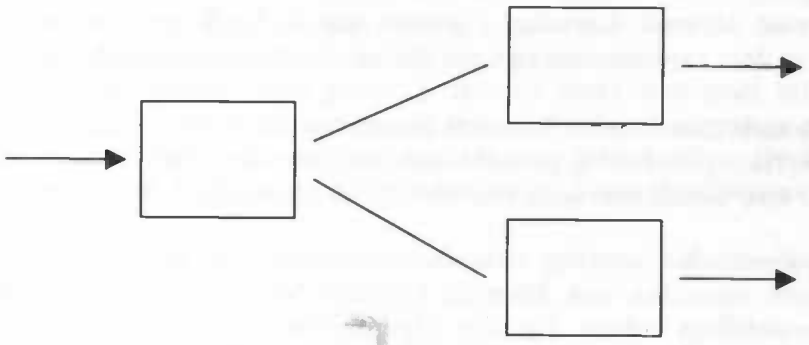
Alternate Learning

as a more stable method for learning modular neural networks.



RuG
Department of Mathematics and Computing Science

Martijn Kuiken
May 2000

Samenvatting

Modulaire neurale netwerken zijn vaak moeilijk te trainen indien er de standaard back propagatie methode wordt gebruikt. Doordat de structuur van modulaire netwerken danig is veranderd ten opzichte van een 'normaal' neuraal netwerk (er bestaat niet langer een volledige verbondenheid tussen de aangrenzende lagen), is het niet ongewoon dat het leerproces van zulk een modulair netwerk instabiel verloopt. Technieken om deze modulaire structuren toch te trainen bestaan veelal uit 'kunstmatige' oplossingen, zoals bijvoorbeeld het nog niet aanpassen van bepaalde submodules gedurende een gedeelte van het leerproces of het leren met zeer lage leersnelheid (wat een stabiliserend effect heeft). Voor elk probleem (modulair netwerk) moet dit echter worden onderzocht; welke module moet wanneer en hoe lang worden vastgehouden etc. Een kant en klare methode welke zonder voorkennis te hebben van het desbetreffende modulaire netwerk, het leerproces tot een succesvol eind brengt, is in dit opzicht gewild.

Het vermoeden bestaat dat door het ontbreken van volledige verbondenheid tussen de lagen in combinatie met het feit dat een gegenereerde fout door het gehele netwerk wordt terug gepropageerd en zodoende alle modules aanpast, er een te grote correctie in het modulaire netwerk plaatsheeft met als resultaat een nog grotere fout. De voorgestelde leermethode (Alternate Learning), welke in dit verslag besproken wordt, lost dit probleem op door niet het hele modulaire netwerk in één keer aan te passen maar alleen geselecteerde submodules.

Het onderzoek naar Alternate Learning is gedaan aan de hand van een tweetal experimenten. Gedurende elk van deze experimenten zijn een drietal selectiemethoden gebruikt om submodules te selecteren voor het leerproces. Deze Alternate Learning experimenten zijn vervolgens vergeleken met de resultaten verkregen door het leren met de standaard methode (alle submodules trainen). De beoordelings criteria welke hierbij gebruikt zijn omvatten de stabiliteit gedurende een run, de stabiliteit tussen verschillende runs (ook wel robuustheid genoemd) en de absolute fout.

Hoewel de selectiemethoden onderling verschillend presteren, laten de resultaten over het algemeen zien dat modulaire netwerken met Alternate Learning beter presteren als gekeken wordt naar bovenstaande beoordelings criteria. Uit deze experimenten is vervolgens gebleken dat Alternate Learning een veelbelovend alternatief is en daarom nog meer onderzoek vereist.

Abstract

When using the standard error backpropagation algorithm, modular neural networks are often very difficult to train. Because of change in the structure of a modular network (there is no longer full connectivity between adjacent layers) compared to that of a 'normal' neural network, one should not be surprised to see an instabile learning process. Techniques to still be able to train modular structures tend to be 'artificial' solutions, like for example not training certain submodules for a period of time during the training process or simply by using a very low learning rate (which has a stabilizing effect). However one needs to do research for every problem (modular network); which submodule must when be fixed and for how long etc. One would rather have a ready-made solution which brings the trainings process, without having knowledge about the modular network, to a successful ending.

Because of the loss of full connectivity between the layers together with the fact that a generated error is propagated back throughout the whole network and therefore adjusting all modules, it is thought there is too much correction in the modular network resulting in even bigger errors. The suggested learning method (Alternate Learning), being discussed in this paper, solves this problem by adapting only selected modules instead of the whole modular network.

Two experiments have been done to test Alternate Learning. During each of these experiments three different selection procedures have been used to select submodules for adapting. These Alternate Learning experiments have been compared with results gained from training according to the 'standard' method (training all submodules). Performance criteria consist out of stability during a run, stability among several runs (robustness) and the absolute error.

Although selection procedures among performed differently, the overall results showed that when using the above mentioned performance criteria Alternate Learning performed better. These experiments showed that Alternate Learning is a promising alternative and that therefore more research has to be done.

Contents

Chapter 1 Modular neural networks related learning problems

In the last couple of years a lot of progress has been achieved concerning artificial intelligence. Several techniques have been developed to tackle problems which before could not be solved at all or not as good as using conventional methods. One such technique is simulating the human brain, the so-called neural networks.

Neural networks earn their existence by learning about a problem and thus adapting to the problem space, generally resulting in better performance after a certain period of time. Although all this imitating-the-human-brain–by-learning-and-adjusting-to-the-problem-area may sound fantastic, there still are a lot of problems to overcome. For one the imitation of the human brain is a very poor/simple one. Opposite to the human brain neural networks are relatively speaking usually not that complex. More complex problems are often tackled with the divide and conquer method: split the problem in several smaller problems and solve each of these problems seperately [6]. With neural networks this is often done by learning/training a neural network with one certain aspect or feature of the problem. By training other neural networks with other remaining features and then combining all these networks in some way, one final result is obtained for the overall problem. In the past years lots of techniques and ideas have been lauched to solve the problems introduced by this modular approach. How do we split the problem? In how many parts do we split it? Since we want one answer only, how do we combine all the networks in order to get that one answer for the problem? These are important issues since for example not all features of a problem are by definition equally important and therefore not every network should be taken equally into account when forming the final answer.

One way of splitting up the problem is for example bagging [1]; training several networks with only a small part of the total data set and combining these networks again with for example majority voting. With this method, networks are created which perform very good on a small part of the dataset and worse on the rest.

Majoritity voting is a very simple method for combining networks again and is therefore not only related to bagging [2]. Whenever a classifier is build consisting of several networks the easiest way of generating one answer is a majority vote. Of course there are variations possible on how the vote should be done; like mentioned before it's possible to make during the vote one network more important than the other.

When creating a function approximator it's usually better to use something like a weighted sum of the outputs instead of majority voting [3], [4]. It is of course also possible to use another neural network to combine the results of all the networks [5]. This is somewhat more complicated depending on the technique used to split into several networks. Variations on these techniques like stacking, ensembles, tree structures, etc are all very close connected to the methods mentioned above. All have one goal: to split the problem because one neural network alone can't do the job properly.

All solutions mentioned above have one striking similarity; several networks are trained (each with it's own feature according to a certain modular view) after which a combiner in some form is created. This marks *the end of the learning process* and the modular network is supposed to be performing according to what is expected of it. Although this approach usually works (depending on the problem) there is still room for improvement. It's to be expected that problems solved with neural networks will only get more complex and more difficult in the future. In order to handle these problems the modular neural

networks[1] will also need to be more complex or become larger (consisting of more networks). As a result, tying all those networks together in the right way will become more difficult, especially the finetuning; there are so many parameters to change. It would be much easier if the modular network could finetune itself, in other words: learn from it's errors.

Sofar hardly any modular neural networks have the ability to learn. Some claim to have created a learning modular network but looking closely at these networks often reveals that a different definition of a modular network is used then done in this paper so far. In this paper a modular network is defined as a network build from other smaller networks which on their turn could function as standalone networks if needed. Each of these smaller networks has a specific function and therefore capable of extracting one feature from the problemspace.

There is a very good reason why hardly no modular networks have the capability to learn. Experiments show that the conventional method of training a neural network does not work on a modular network without extra help. One has to start with a very low learning rate and slowly increasing it; but even then it's a very tricky and unsure proces. The modular network easily gets instable during the training proces; sometimes it even loses its information obtained by pre-training it's networks seperately. But even when it does work it's not clear why it worked; there's no clear indication what effect all the parameters of the process have. We therefore need a new learning concept when we want to train a modular network. Let us first look at why the conventional learning method does not work with modular networks.

In a normal neural network a neuron in a certain layer is connected to all the neurons in the adjacent layers (figure 1.1). These cross connections make sure that, during the

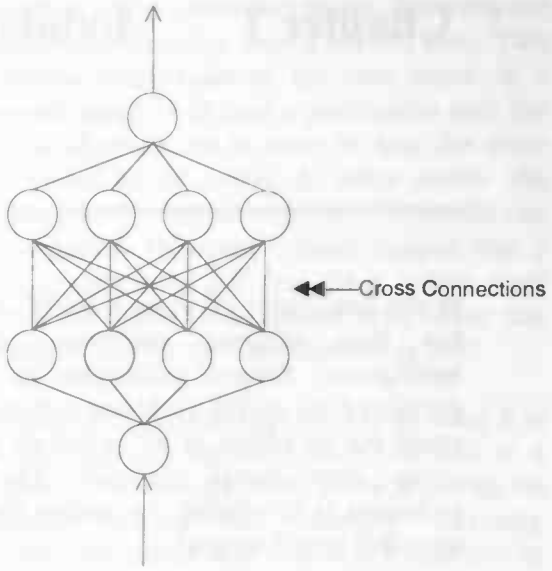


Figure 1.1 *Standard MLP with maximum cross connectivity*

learning phase when the error is propagated back, it's propagated through all existing paths. This way the error in a certain neuron won't become too large because it is most likely corrected in a certain degree by one or more of the other neurons (paths) it's connected to.

Looking at the modular build network mentioned before we see that a lot of cross connections are missing. Everytime an output

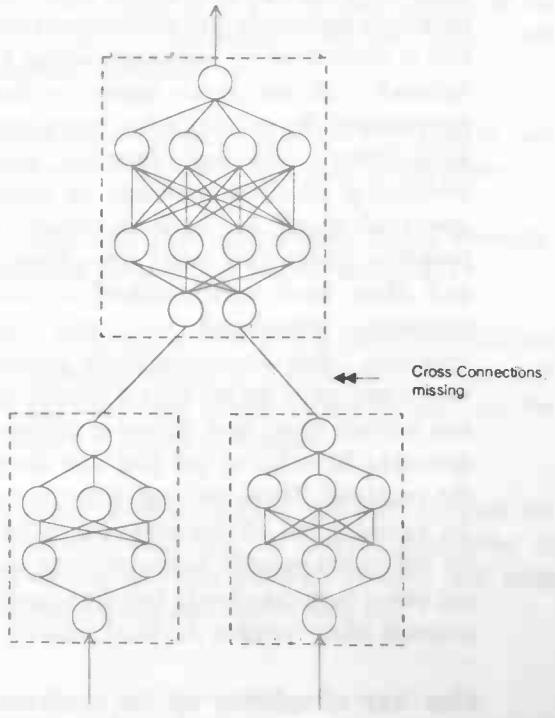


Figure 1.2 *Modular network demonstrating the loss of cross connections*

1 Throughout this paper we will use the term *modular network* when we refer to the *total* network. When the term *network* is used without the adjective modular then we refer to the network(s) being used to build the actual modular network.

of a network is connected to the input of another network we only have one connection and no cross connections (figure 1.2). It is very well possible that this is the cause of the instability when training a modular network; a very large error in a neuron cannot be corrected by other paths because there simply aren't any other paths.

Imagine a MLP with a variable number of neurons in the hidden layer (figure 1.3). Training this network with the same targets as inputs will show how well this network performs on learning to copy it inputs to its outputs. By repeatedly doing this experiment but everytime with a different number of neurons in the third (hidden) layer, will show how much influence removing the cross connections has. When there's only one neuron left in the third layer, an almost similar situation has been created as in a modular network between the output of a network and the connected input of the next network. This experiment shows that constricting such a layer can indeed lead to instability.

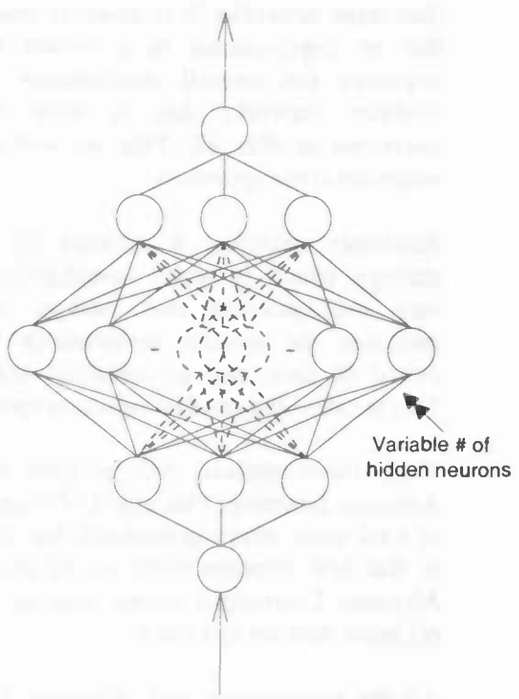


Figure 1.3 *Network showing the loss of cross connectivity by changing the number of hidden neurons per experiment*

1.1 Alternate Learning

So looking at and thus training such a modular network as one network is not working. We apparently need something to cancel the effect of the loss of cross connections. One way of dealing with this loss of cross connections is somehow training all the networks seperately but this time taking the environment into account. Here environment has the meaning of the rest of the modular network. Instead of adapting the whole network at once, we could decide to only train a part of it. Throughout this research we will call this approach *Alternate Learning*.

Alternate Learning is a method in which one pattern out of the global patternlist is used to train a *specific* part of the modular network. One could compare it to training only selected neurons during a normal training process of a neural network. Only now the normal neural network is replaced by a modular neural network in which every network can be seen a single neuron. To make use of this alternate learning, new patterns have to be generated based on on the information of the rest of the network. With these new patterns the selected networks can be trained seperately.

There are of course a lot of variations possible on this concept and as till now it is not clear at all what influence each of these variations has on the learning performance of a modular network. In order to come to a new learning concept an answer must be found on what decides how many and which networks need to be trained in a modular structure?

Since a modular network is judged on basis of its performance, which is closely related to the generated error, it makes sense to somehow use this error to select one or more networks for training.

One of the most sensible strategies seems to be selecting the network which is responsible for the greater part of the total resulting error and thus improving the worst part in a modular network; we will call this the *maximum error approach*.

Training a network with the smallest error seems illogical since it would mean that one is trying to improve the performance of the network which is already performing better

than other networks. It is however imaginable that an improvement in a certain network improves the overall performance of the modular network; this is what we are interested in after all. This we will call *the minimum error approach.*

Randomly selecting a network is another strategy which is worth investigating. It is very well possible that random selection increases the overall performance after a period of time, due to statistical behaviour. This is called the *random selection approach.*

These three methods will be used to study Alternate Learning. One could of course think of a lot more selection methods, but since this is the first experimenting to be done with Alternate Learning it seems wise to not bite off more than we can chew.

All the experiments with Alternate Learning will be compared to the performance of modular networks using a 'normal' learning method, in other words: training all networks. One could argue that for the chosen experiments a single network would outperform the modular network anyway. However, this is not relevant since we are trying to find an answer whether or not Alternate Learning performs better than nowadays techniques when training *modular* networks. Sometimes it is just not possible to solve a problem with one single neural network and more networks are needed. And this is the point where Alternate Learning is needed then.

The results of Alternate Learning will be judged by three criteria:

- the generated error
- network stability
- effort using Alternate Learning

The error and the network stability are closely related. It speaks for itself that a small error or none at all is very much wanted. It can occur though that a network generates a small error most of the time but has its peaks as well. When these huge errors succeeded by periods of small errors occur too often we will call a network instable. Instability can be seen from two viewpoints. First we have the instability *during* a run[1] as described above. Secondly we recognize instability *among* several runs. This happens when most of the runs result in a smooth graph or at least a predictable one; the result of every run is more or less the same compared to the others. In other words: the experiment and more important the results can be repeated. However it could happen that a run is completly different (in a worse way) from the others. When such runs occur too often we call the network instable as well.

Instability is such an important issue since it is a reference for the confidence one has in a network. We will therefore focus primarily on stability during the Alternate Learning experiments. This is of course in proportion to the absolute error; a huge error generating but stable modular network is not acceptable.

The last criteria mentioned is the effort one has to put in while using Alternate Learning. It is obvious that this issue is again related to the two criteria mentioned earlier. If for example one has to take into account the network structures, when to change learning rates, how many networks need to be pretrained, many do's and don'ts etc, then just maybe all this extra energy is not worth investing while the extra profit gained (in the previous two criteria that is) is minor. Again like the first two criteria it is a trade off one has to make. The ultimate result would be no extra effort at all and major improvement in the error and stability parameters.

With these issues in mind we will discuss and answer the following question in this paper:

Is Alternate Learning a more stable learning method for modular neural networks?

The next chapter will give a short introduction to neural networks. It'll give some background information and mathematical support on the subject of neural networks.

Chapter 3 describes the software needed for Alternate Learning. A brief explanation is given on how this software is used and what its limitations and side-effects are.

[1] A run is a training process during a certain number of epochs.

In chapter 4 the experiments done with Alternate Learning are discussed. First the experiment setups are described, after which the results are discussed and compared to 'normal' training techniques.

The last chapter answers the question previously asked whether or not Alternate Learning is a more stable learning methode for modular neural networks. This is of course based on and in context to the experiments done with Alternate Learning. Finally some suggestions are given on futher research with Alternate Learning.

Chapter 2 Introduction to neural networks

Humans have a very good generisation ability. Judging whether a certain drawing has the characteristics of a square or a circle, seems to be almost no problem at all. Even when this sample is a mixture of both a square and a circle, humans can still categorize it (to a certain degree). Conventional digital computers on the other hand have great difficulty accomplishing this job. For a computer it is either a square or a circle, and making it understand that it can be a mixture of those two is a rather awkward job. With the introduction of artificial neural networks this job became easier to solve though.

Artificial neural networks are derived from the way the human brain works, which is a completly different way than found in a computer. The human brain is build out of neurons which are connected to eachother by various paths, the so-called synapses. Although these neurons are not very fast in computer terms (milliseconds compared to nanoseconds for a computer) the massive ammount in which they exist in the brain (estimated to be about 10 billion) together with the staggering number of interconnections (said to be 60 trillion) leave us humans with a brain many times more efficient and complex than any nowadays or future computer [7]. For example, the brain can regconize a familiar face in an unfamiliar scene in about 100-200 ms, whereas even the biggest and most powerfull computer would take days to accomplish a job of lesser complexity.

But how do these neurons and synapses actually work? It all starts with birth, after which the brain begins to build up it's own rulebase commonly known as "experience". During the first two years about 1 million synapses are formed per second. These synapses function as mediaters between the neurons; by converting electrical signals into chemical signals and back they transport information. All incoming signals (through the synapses) are then summed in the neuron after which by means of an activation function the neuron "decides" what kind of signal (or none at all) to pass on. Now by adapting the magnitude in which synapses transport signals and by forming new connections between neurons the brain can learn; it can adapt to be able to solve or handle new problems and decisions. Several neurons can be organized to

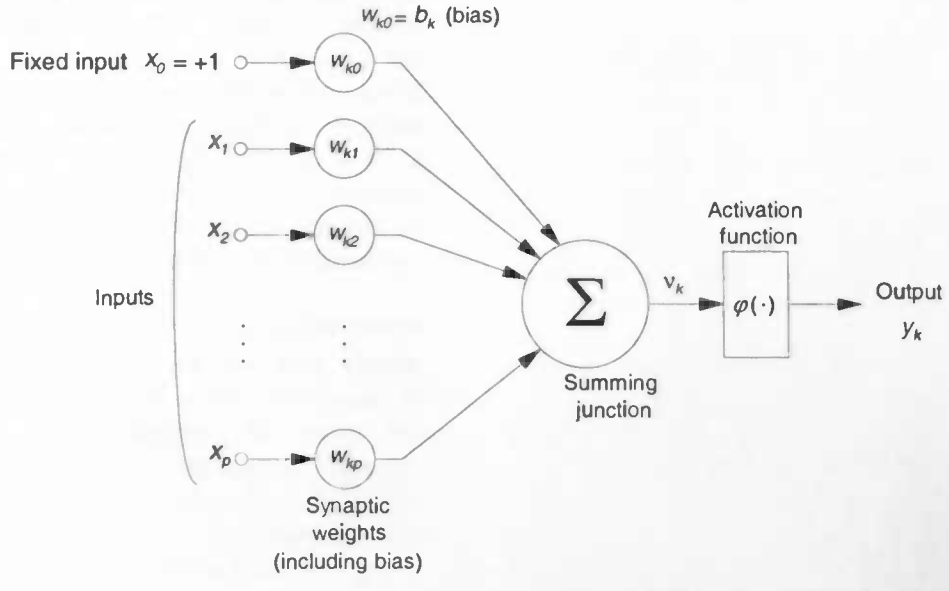


Figure 2.1 *Model of a neuron with the bias presented as an extra input*

form a structure capable of dealing with more complex problems. These structures can then be combined again to gain even more complexity.

The artificial neural networks (from here on called neural networks) act in more or less the same way as seen in the brain, though much more simple. The three characteristics found in the brain are also present in neural networks:

- *synapses* or *connecting links*, each of which has its own strenght, a so-called *weight*
- an *adder* for summing all the input signals received by the 'synapses' or inputs
- an *activation function* which defines the output of a neuron in terms of the activity level at its input

Besides these three characteristics there is also the *bias*. The bias is used as an offset to the activation function and can therefore also be looked at as a special kind of input (synapse). Figure 2.1 gives a schematic presentation of the model of a neuron which is also described by the mathematical formula:

$$y_k(n) = \varphi(v_k(n)) \qquad (2.1)$$

where $y_k(n)$ is the output of neuron k; $\varphi(\cdot)$ is the activation function and $v_k(n)$ is the activation level of neuron k which is defined by:

$$v_k(n) = \sum_p w_{kj}(n) x_j(n) \qquad (2.2)$$

where $w_{kj}(n)$ are the synaptic weights of neuron k and finally $x_j(n)$ is the output signal of the neuron j (which is an input signal for neuron k).

The activation function, denoted by $\varphi(\cdot)$, comes in various shapes and sizes. Among them is the sigmoid function (figure 2.2), which is by far the most common activation function used in the field of artifical neural networks. Because of its smoothness and asymptotic properties, it is able to project large internal values (inside the neuron) onto manageable output values.

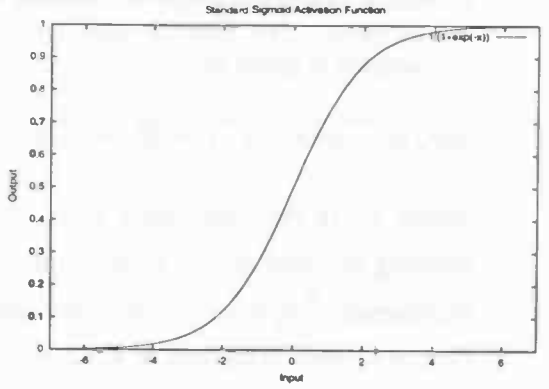


Figure 2.2 *Standard sigmoid activation or transfer function*

Connecting several of the neurons described above to eachother results in a neural structure or neural network. The most common form of structures is the Multi Layer Perceptron, or in short MLP. It is usually (however not necessarily) build out of three layers: an input layer, a hidden layer and an output layer (figure 2.3). Normally these layers are fully connected which means that every neuron is connected to all the neurons in the adjacent layers. Characteristic for the input layer is that its neurons only have an activation function and no weights.

When a neural network computes a certain output with a given input vector, it is hardly ever the correct output. Usually there is a difference between the target (desired output) and the calculated output. This is called the error of the neuron (or network). This error can be used to adapt the weights in such a way that in case the same input vector is ever

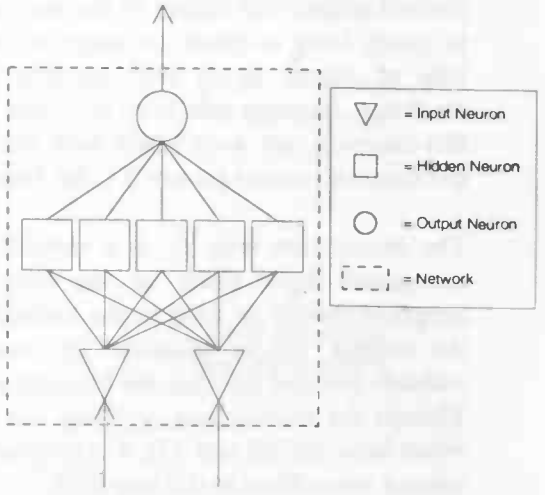


Figure 2.3 *Presentation of a standard MLP with three fully connected layers*

presented again, it'll result in a smaller error at the output(s). The formula used for adapting the weights is given by:

$$\Delta w_{jk}(n) = \alpha \Delta w_{jk}(n-1) + \eta \delta_j(n) y_k(n) \quad (2.3)$$

where α is the momentum term, η is the learning rate and $\delta_j(n)$ is the local gradient. Futhermore, $y_k(n)$ denotes the output of neuron k which is defined by eq. 2.1

The local gradient $\delta_j(n)$ for an output neuron j is defined by:

$$\begin{aligned} \delta_j(n) &= \varphi'(v_j(n))e(n) \quad (2.4) \\ &= y_j(n)[1 - y_j(n)][d_j(n) - y_j(n)] \end{aligned}$$

where $d_j(n)$ denotes the desired output (or target) of neuron j and $v_j(n)$ is given by eq. 2.2.

Finally the local gradient $\delta_j(n)$ for a hidden neuron j is defined by:

$$\begin{aligned} \delta_j(n) &= \varphi'(v_j(n)) \sum_i \delta_i(n) w_{ij}(n) \quad (2.5) \\ &= y_j(n)[1 - y_j(n)] \sum_i \delta_i(n) w_{ij}(n) \end{aligned}$$

The learning rate η determines how fast the weights adapt and is typically a value between 0.0 and 1.0. It should be noted that large learning rates usually cause instabile behaviour; instead of slowly moving to the desired output, the output of the neuron tends to jump from adaption to adaption. A good rule of thumb is to start training with a mediocre learning rate (say 0.7) and change this learning rate somewhere near the end of the training process (to say 0.1) for finetuning.

The momentum term α is a variable which determines how much of the last weight adaption should be used in the calculation of the current weight adaption. By using this variable one can stabilize the learning process. Though the momenterm typically can have a value between 0.0 and 1.0, it is recommended using a value close to 0.0 (say 0.2).

Since all layers are fully connected it is easy to see that the error at the output of the network is distributed and therefore affecting all weights in the network.

Because the error at the outputs is propagated back through all neurons where all weights are adapted, this kind of learning is called backpropagation. Backpropagation is a form of supervised learning; a method in which the neural network receives information from the outside world on how well it is performing. Contrary to supervised learning there are the less used unsupervised learning algorithms in which the neural network receives no help from the outside world and has to do its job by itself.

In this paper all experiments are based on the concept of the above explained neural networks, to be more specific: the MLP. When looking at a *modular* neural network however, it is clear this is nothing more than a neural network with:

- more layers
- no full connectivity between the layers

These two factors make the learning process of a modular neural network more difficult as explained in chapter one.

Chapter 3 Program functioning

In order to be able to handle modular structures a new program had to be written. This program would have to solve two main problems.

The first problem is that Interact[1] can only handle one network a time; several networks can be loaded into the memory but only one of these networks is active at a certain time. So when training a modular network (consisting of several networks) this program would need to switch active networks during the process.

The second problem is based one the fact that there is only one global patternlist available for training all the seperate networks. This means that this program to be written needed to somehow extract several patternlists out of this global patternlist.

3.1 General overview

The written program starts with an initialisation file (figure 3.1). In this initialisation file information about the global network is stored; for instance to which input an output of a certain network is connected and vice versa. These network may or may not be pre-trained but they must exist. Trying to load not existing networks will make the program halt. Besides this information also a few parameters concerning the training process are stored in the initialisation file such as the learn rate of each network, the momentum term, the method of Alternate Learning and the global training and testing patternlists. All this information is read by the program and used to build the modular network and then train it.

To train the modular network, the program first needs to create a training and/or testing patternlist for each network. It is important to understand that in order to look at the modular

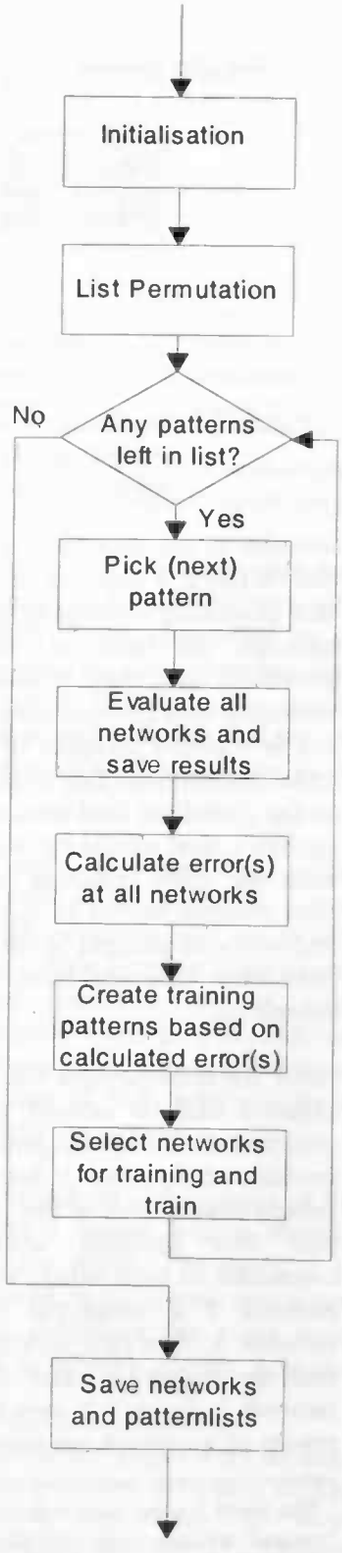


Figure 3.1 *Flowchart of main functioning of the used program*

[1] Interact is the program used at the RUG (Department of Computer Science) for training and analyzing neural networks.

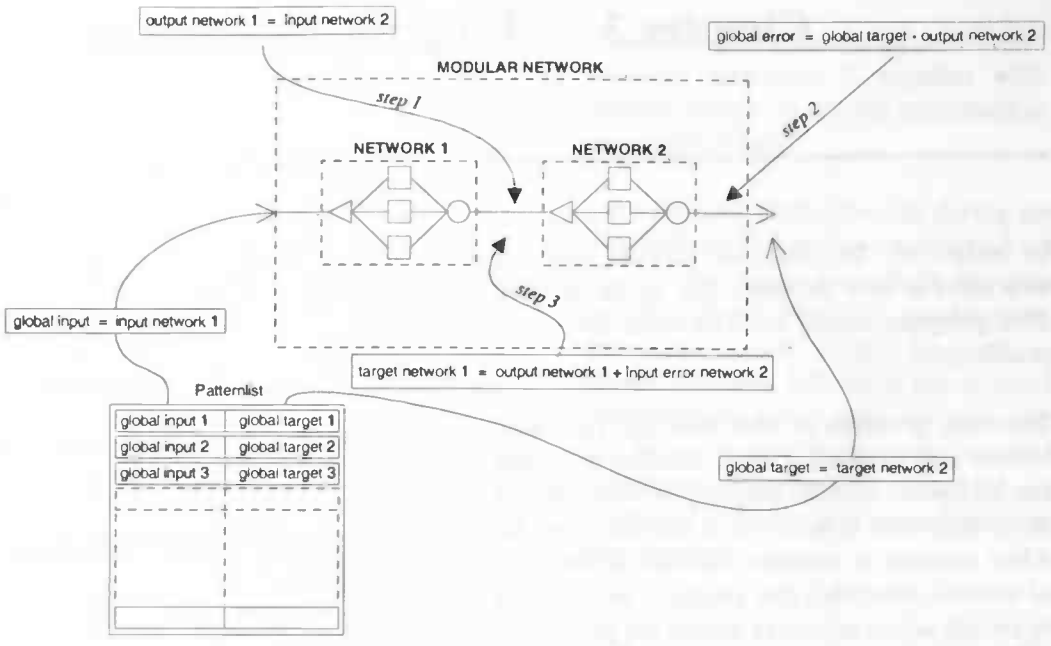


Figure 3.2 *Creation of the new input-target patterns by use of the global patternlist; evaluation of all networks (step 1), calculating error(s) through backpropagation of the global error (step 2), creating target patterns with evaluated outputs and backpropagated error(s) (step 3).*

network as *one* network it is necessary to also train it like if it were *one* network. This means that generating a complete patternlist for each network on basis of the *whole* global patternlist and then training each of these networks with this complete patternlist would not be a proper imitation of a normal training cylus. Rather than first evaluating all patterns in the global list (and then training), only one pattern a time should be used to evaluate and train the entire[1] modular network. Then the next patterns should be used to do the same and when all patterns in the global patternlist have been used one learn cycle or epoch is completed.

After the configuration file has been used to create a modular network the program takes one pattern out of the global patternlist and evaluates this pattern. It does this by using the information stored in the configuration file and thus knowing what networks are connected to each other. When the input of network 2 is connected to the output of network 1, then first network 1 is evaluated and the output is used for evaluation of network 2 (figure 3.2; step 1). The evaluation results of a network are stored so they can be used whenever an output is connected to other networks as well. It is of course possible that the inputs of network 2 are connected to outputs of several seperate networks. This means that all these seperate networks need to be evaluated before network 2 can be evaluated. When all the networks in the modular network are evaluated with this one global pattern then an error is calculated at the output(s) using the ouput value(s) of the evaluation and the target value(s) of the global patternlist (figure 3.2; step 2). This error is then backpropagated through the last network to its inputs. One can look at this error from different viewpoints which can be a rather important issue since it is likely that one would want to improve (train) the network *responsible* for the error. The first viewpoint is that the network through which the error was backpropagated is responsible for this error at its inputs. This would not be fair though. It would mean that no matter how many networks preceed this last network the error is always generated by the last network. An alternative viewpoint is that the error(s) is generated by all preceding networks. In the latter case the error generated at the global output has to be used to make new targets for all the networks. This is the reason why the error is propagated back through the network(s). A new target can be created for a network using the previously evaluated value

[1] The word '*entire*' here refers to what would be a 'normal' training cycle. In Alternate Learning not all networks are trained but only a few selected ones.

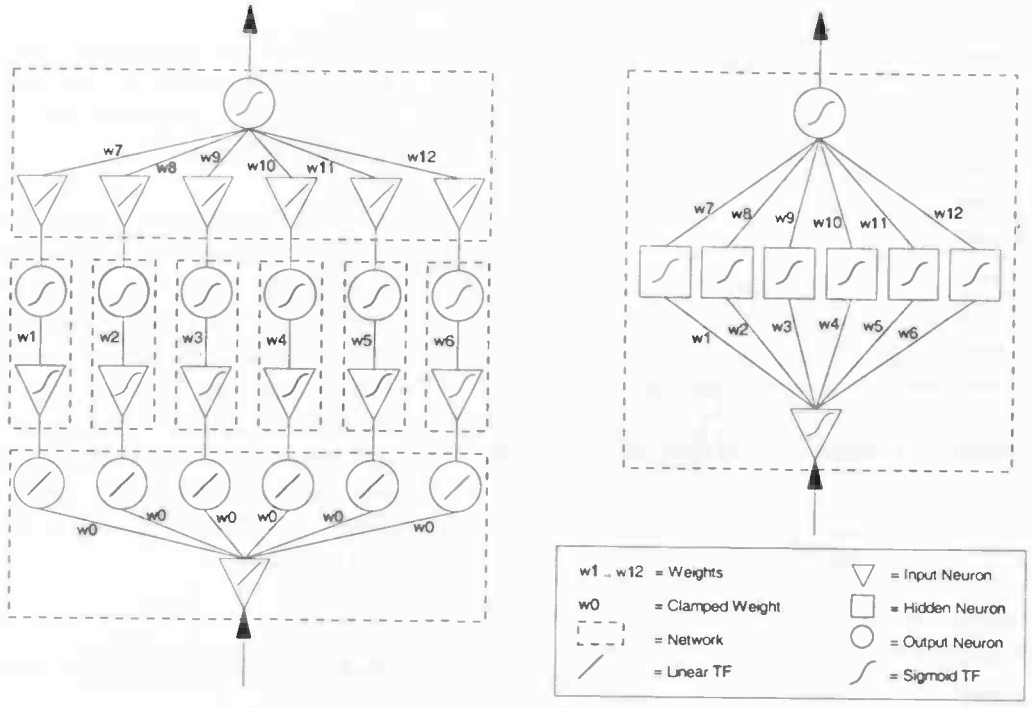


Figure 3.3 *Experiment setup used for testing integrity of the program. The state of both the modular and the normal network must stay identical to eachother in order for the program to pass this test.*

at the output(s) of this network and the backpropagated error at the inputs of the succeeding network (figure 3.2; step 3). This is done for all networks and it will result in input-target patternlists (obviously containing only one pattern) for each network. When all the patternlists have been created the training process starts. Depending on the choice of Alternate Learning, only selected networks will be adapted in order to achieve a better performing modular neural network. After using this first pattern out of the global patternlist a second pattern will be chosen and the whole process of creating new input-target patternlists and then training selected networks with them, starts again. When all patterns in the global patternlist have been used for training, one epoch has been completed and a performance measurement of some kind is used to review the modular network.

3.2 Correctness of the program

Prior to all the experimenting one important question remains:
Does this program function like it's expected to behave on basis of the describtion given above?

In other words: does it deal with modular networks like a single 'normal' network? It is rather difficult if not impossible to give a 100% guarantee in every possible situation which can occur. However, it is possible to test this tool in such a degree that we can rely on the correctness of this program in normal situations[1].

To test the proper working of the software a normal network is compared to a modular network. Both networks will try to learn to approximate a sinus function using exactly the same global patternlist. If the Alternate Learning program works properly and both the initial networks have the same weights and biases, the state of both networks should be the same after a learn cycle, .

The normal neural network consists of 1 input, 5 hidden and 1 output neuron (figure 3.3; network on the right). The modular neural network is based on the normal neural network but with this difference that every neuron is replaced by a network (figure 3.3; modular

[1] Since the goal of this project is not the writing of a software tool but rather the testing of Alternate Learning as a replacement for nowadays techniques, no effort has been put in dealing with 'not normal' situations like feedback in modular networks etc.

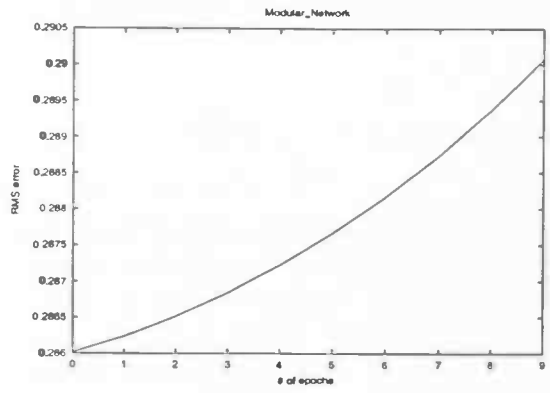


Figure 3.4a *Output curve of the modular network*

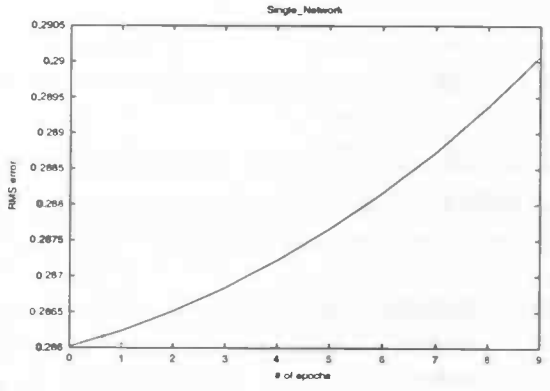


Figure 3.4b *Output curve of the standard network; the same input vectors were used as for figure 3.4a*

network on the left). Now by chosing the proper networks and weights a modular network is created which should learn in exactly the same way (generate the same output values for a certain input value) as a normal neural network would. The initial state (weights and bias) of the normal neural network and the modular network need to be the same of course as well as the learning speed and momentum term in both situations. Figure 3.3 shows the total experiment setup.

One issue is different in this learn process though. Normally a patternlist used for training a network is permuted to make sure the network really learns to generalize instead of memorizing values. This option is cancelled in this experiment though, since it is highly unlikely that the permutation in both situations (the normal and the modular neural network) would result in the same patternlist. If the patterns are not used in the same order in both situations then the resulting networks will not be *exactly* alike.

Specifications concerning this test can be found in Appendix A, together with a short manual on how to use the initialisation file and the program itself

After training both networks during one epoch it turned out that both networks had adapted in exactly the same way. Both the biases and weights stayed identical, resulting in exactly the same output values for both networks (figure 3.4a and 3.4b show the results for both experiments). It can therefore be assumed that the program works properly for the coming experiments concerning Alternate Learning. There are however also some limitations using this program.

3.3 Limitations and side-effects

The limitations of the Alternate Learning program can be divided into two classes. First there are the limitations based on the limitations of Interact. Secondly some limitations were introduced during the writing of the program. Most of these last limitations are only there because it simplyfied the creation of the program; fixing these limitations would be time consuming while it would not open new areas for study on the behaviour of Alternate Learning.

Since the internal structure of building new patternlists is based on the concept of Interact patternlists, Interacts hardcoded maximal number of opened patternlists is a limit in the program. Internally the program uses 4 patternlists for every network in the modular network, storing the following patterns:

- Input-Output (used during evaluation)
- Input-Target (used during training)
- Error (used during creation of the Input-Target patternlist)
- RMS-Error (only updated after completion of an epoch)

At the time of writing of this paper Interact would not allow more than 100 patternlists to be opened at the same time. This means that not more than 25 networks can be used to build a modular network. However to generate a new training patternlist for a network the program needs to open as much patternlists as there are inputs plus outputs in this network.

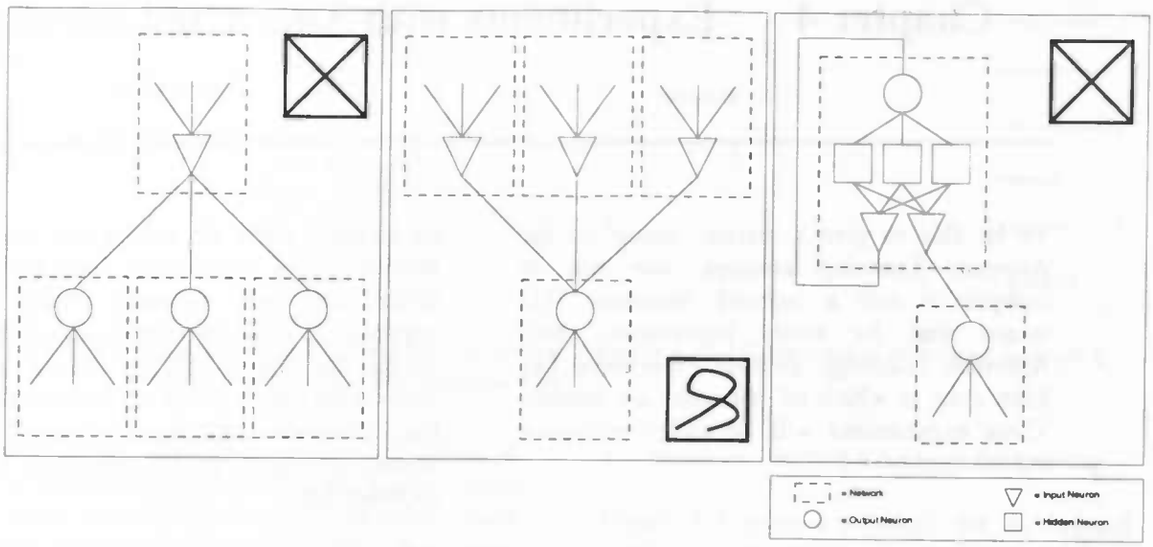


Figure 3.5 *(Im)possible network configurations; left: several outputs connected to one input (not possible!), middle: one output connected to several inputs (possible!), right: connection between input(s) and output(s) of the same network (not possible!).*

Because of this and also because there are several global patternlists (global train and test patternlists for example) opened during running time a good rule of thumb is not to use more than 20 networks. Since it is a rule of thumb it is possible this causes problems and for such cases some sort of garbage collection for patternlists has been build in. Although it does not automatically remove not in-use patternlists, it does generate the warning:
Patternlist with id: # is already in use!!
This means that the maximum of to be opened patternlists has been reached while the program still needs to open more patternlists.

It is not allowed to connect an input of a network to more than *one* output of another network (figure 3.5; first configuration). In case of several outputs connected to one single input the proper working of the program is not guaranteed. However this does not mean it is not allowed to tie a single output to more than one input (figure 3.5; the second configuration). The latter case is a perfect legitimate configuration.

It is also not allowed to use feedback. In other words: connect the output of a network to its own input(s) (figure 3.5; last configuration). This action will most certainly result in not specified behaviour.

The program expects a 100% correct initialisation file. No checking or warnings are given when an incorrect initialisation file is used. This is not so much as a limitation but more of a warning. The initialisation file is mainly a describtion of the configuration of the modular network and therefore no comment is allowed in the file itself. Appendix A gives the syntax to be used when writing the initialisation file along with a detailed instruction manual of the program.
The only side effect of this program for Alternate Learning worth mentioning is immediately the programs weakest point: using several networks to build a modular network causes the program to become slow. It usually pays to think a little longer about an experiment setup to minimize the number of networks (or to be more precise: minimize the total number of neurons in the modular network). To give an indication:

Using a Pentium Celeron 333 MHz to train three networks in a modular network[1], one run of a 1000 epochs took about one hour.

[1] Sinus experiment discussed in chapter 3 in which the first and last network consisted of one input and one output neuron and the network in the middle of one input, five hidden and one output neurons.

Chapter 4 Experiments with Alternate Learning

To be able to give a certain 'rating' to the Alternate Learning concept, one has to compare it with a 'normal' situation. This means that for every experiment using Alternate Learning, another experiment has been done in which *all* networks are trained. These experiments will be called *reference experiments*.

4.1 General experiment setup

Certain experiment parameters are the same for all experiments discussed in this chapter:

- Number of epochs
- Number of runs
- Number of epochs and learn parameters for pre-training a network

An epoch is defined as a completed learn cycle. This means that every pattern in a patternlist has been used once to adapt the (modular) network. To prevent the network from specifically memorizing the values in a patternlist, all patterns are permuted at the start of the epoch. During *one thousand epochs* a modular network is trained and after this last epoch *one* run is completed.

After ten of these runs one experiment has been finished. Due to the issue of network stability more runs would be welcome, but since these experiments are very time consuming the choice for ten runs has been made.

Although the learn rate changed between different experiments, the momentum term never changed: during all experiments (Alternate Learning and reference experiments) the momentum term had a value of 0.0. The reason for this is inherent to the concept of Alternate Learning itself. The momentum term is used to gain extra stability by, when adapting weights, also taking into account the previous error (generated with the previous pattern). Alternate Learning however is based on the idea that a certain network will be adapted while the others stay unchanged. Introducing a momentum term here would mean that even networks which are not supposed to adapt, do adapt a little (depending on the the value of the momentum term). In order to be able to make an honest comparison the reference experiments use the same momentum term as the Alternate Learning experiments.

Whenever a pre-trained network is needed in a modular structure, this network is trained during 250 epochs with a learn rate of 0.7 and a momentum term of 0.2.

Parameter values used in *all* experiments

# of epochs per run:	1000
# of runs per experiment:	10
momentum term:	0.0
# of epochs for pre-training:	250
learn rate for pre-training:	0.7
momentum term for pre-training:	0.2

4.1.1 Sinus experiment

Since no research on Alternate Learning has been done sofar, this first experiment is meant to gain some general knowledge on and get acquainted with the topic of Alternate Learning. One could look at this experiment as an opportunity to get some feeling on how to handle the concept of Alternate Learning.

4.1.2 Experiment setup

A modular neural network consisting of three networks will be trained with 150 patterns describing a standard sinus function. As shown in figure 4.2 the networks are connected in series. Network B has been pre-trained with the same 150 patterns as used for the modular network and both network A as the network C have been random initialised (weights and biases). In the ideal situation one

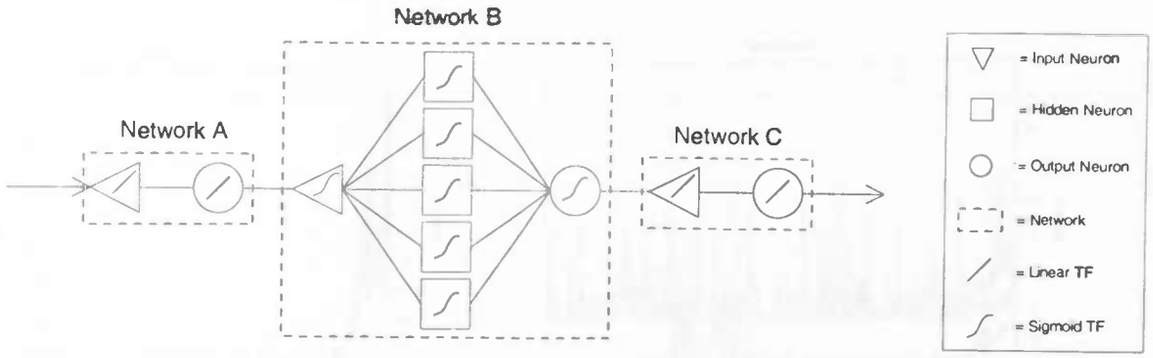


Figure 4.2 *Configuration used in the Sinus experiment; Network A and C are random initialised, network B has been pre-trained to fit a sinus function.*

would expect that since network B alone is already capable of producing the wanted sinus, that the other two network would learn to only pass on the value given to them. It is more likely though, that the pre-trained network will lose its pre-gained knowledge because of the values generated by the other two not yet trained networks.

To get some information about the influence of the learning rate when training modular neural networks, every experiment will be done four times, each with a different learning rate. The learning rates used are 0.7, 0.5, 0.3 and 0.1. Like mentioned in the beginning of this chapter, the momentum term for every experiment is set to 0.0.

The Alternate Learning experiments are divided into three different approaches:

- Training the network with the largest error
- Training the network with the smallest error
- Random selecting a network for training

Figure 4.3 gives a schedule for all created configurations now and also the numbers given to them by which they will be referred.

4.1.3 Results sinus experiment

Reference experiment

To be able to make the comparison between Alternate Learning and normal learning the reference experiment was done firstly. Because both the input and the output use a linear transfer function, one would expect that the pre-gained knowledge in the network in the middle is mostly preserved, at least at the start of the experiment. The reason for this is easy to understand. Since the pre-trained network is trained with exactly the same patterns as will be used for the modular network, the best way to preserve the gained knowledge is to train the middle network with

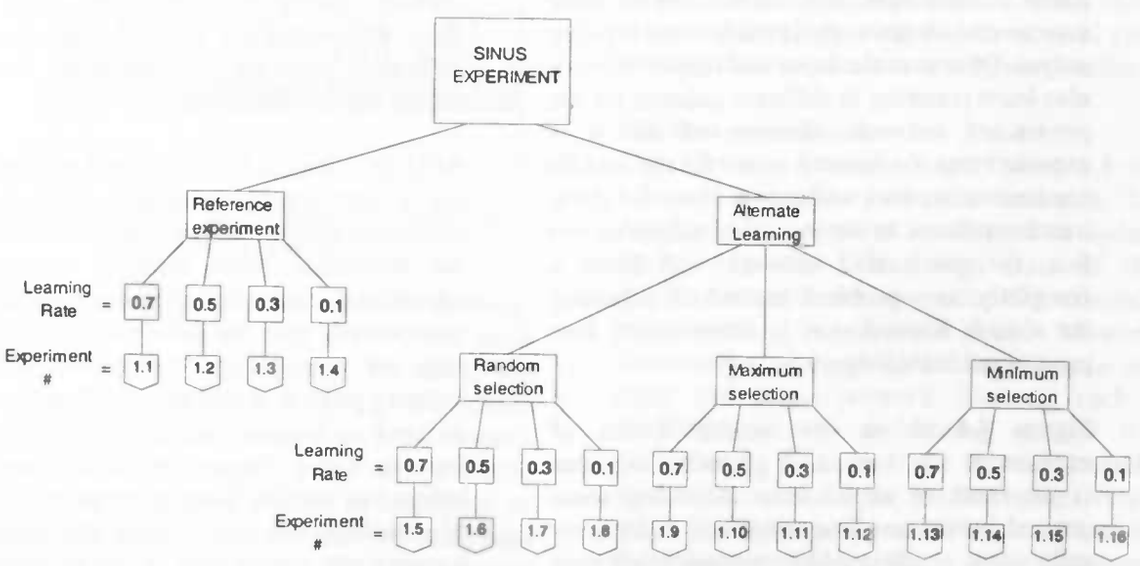


Figure 4.3 *Schedule for the Sinus experiment.*

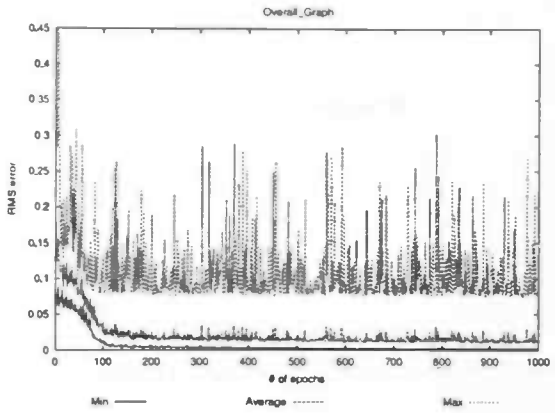


Figure 4.4 *Overall graph of experiment 1.1; type: reference; learning rate: 0.7*

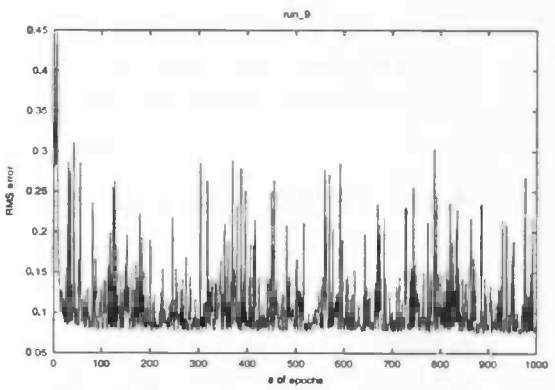


Figure 4.5 *Run 9 of reference experiment 1.1: no improvement and instable behaviour*

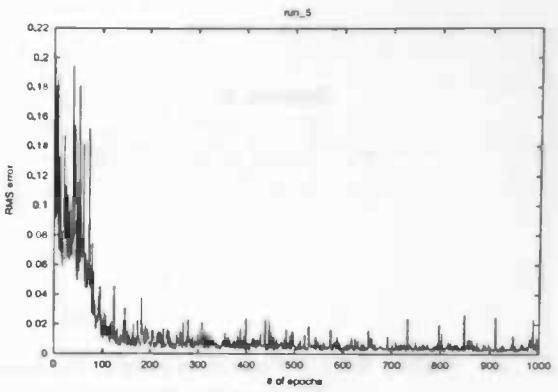


Figure 4.6 *Run 5 of reference experiment 1.1: improving but instable behaviour*

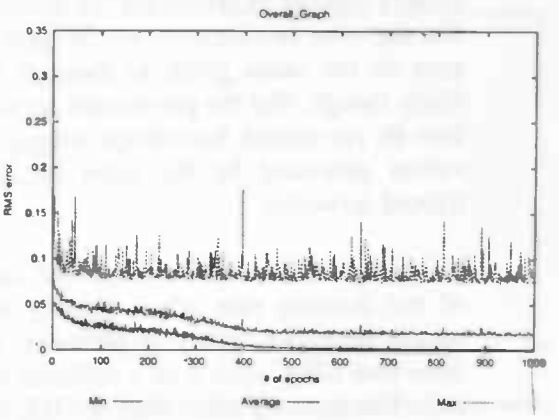


Figure 4.7 *Overall graph of reference experiment 1.3 with a learning rate of 0.3*

the same or almost the same values during the modular training phase. How can this be achieved? By using an input and an output network which do not modify the patterns by such a magnitude that the pre-trained network is confronted with new patterns outside the pre-trained space. This is the reason why *linear* transfer functions are used: to be more sure an one to one copy is made from input to output. Of course the input and output network also learn resulting in different patterns for the pre-trained network. Because of this it is expected that the learned space for the middle pre-trained network will move. However if the training process in the input and output go too fast, the pre-trained network will learn a completly new problem instead of adjusting the already learned one, in other words: lose pre-trained knowledge.

Figure 4.4 shows the overall results of experiment 1.1. An overall graph in this paper is the result of all ten runs indicating some general behaviour. The minimum RMS error reflects the smallest RMS error out of *all* runs for a certain epoch. The maximum RMS error is likewise but now for the largest error in an epoch. The average line shows, as expected, the average RMS error of all runs for an epoch. This overall figure is no reflection of an average, minumum or maximum individual run! It should be looked at as an indication for stability among runs only; the closer these three different lines are the more the same individual runs are and therefore the more stabile the experiment is.

As clearly seen in figure 4.4, experiment 1.1 is not a very stable experiment. Although the minimum RMS error is no reason for concern, the maximum RMS error is showing an unpredictable line. Zooming in on the different runs unveils that the differences between the runs are indeed huge. Some runs show no learning process at all (figure 4.5) while others do tend to improve but only by an irregular learning curve (figure 4.6). Like mentioned before this instable behaviour can be the result of a too hasty learning process. Decreasing the learning rate should then, if this assumption is

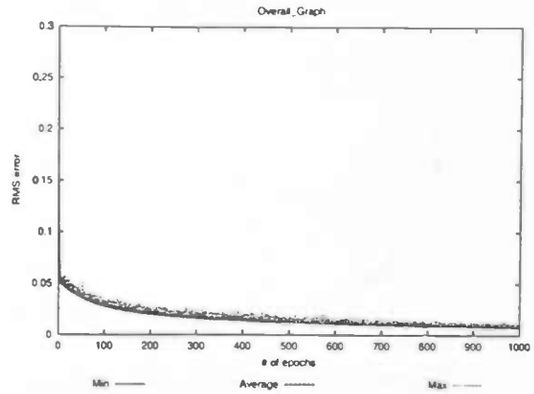


Figure 4.8 *Overall graph of reference experiment 1.4 using a learning rate of 0.1*

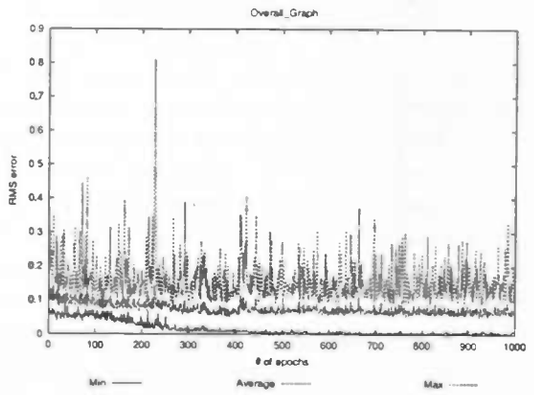


Figure 4.9 *Overall graph of A.L. (random) experiment 1.5; learning rate: 0.7*

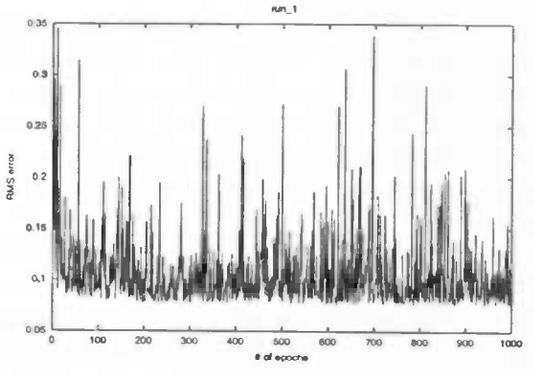


Figure 4.10 *Run 1 of reference experiment 1.5: instable behaviour*

correct, result in more stable learning. Although experiments 1.2 and 1.3 do show improvement in stability while decreasing the learning rate, the instability is still there. Figure 4.7 for example shows the overall graph for experiment 1.3, in which it is clearly visible that the lines are not as capricious as seen in experiment 1.1. Still, the maximum RMS error is not nearly alike the average

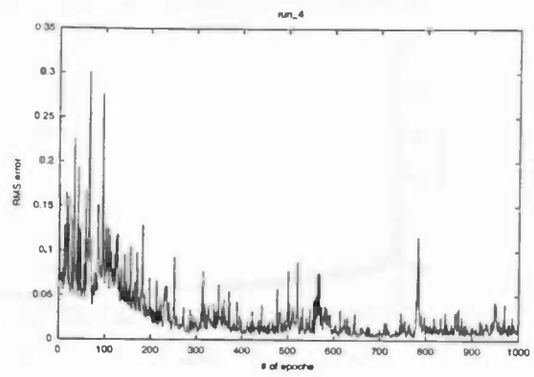


Figure 4.11 *Run 4 of reference experiment 1.5: improving but instable behaviour*

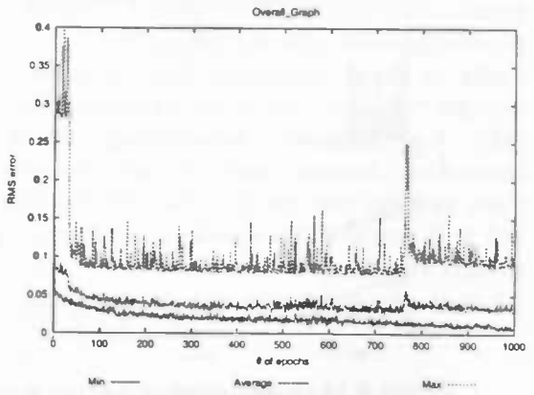


Figure 4.12 *Overall graph of A.L. (random) experiment 1.7; learning rate: 0.3*

value (in magnitude and in 'shape'), suggesting large differences between the seperate runs again.

Only with a small learning rate of 0.1, the configuration behaves more stable. The three curves in figure 4.8 (experiment 1.4) are close to one another indicating that the several runs forming this experiment are very much alike.

Random experiment

The Alternate Learning experiments 1.5 till 1.8 use a random selection methode for deciding which network is allowed to update during a pattern. The initial state of the modular network is same as for the reference experiments, except for the random initialisation of the weights and biases in the input and output network. Training such a configuration with a learning rate of 0.7 (experiment 1.5) resulted in an instable modular network. Figure 4.9 shows the overall graph for this experiment and again as seen in the reference experiments, it is the maximum RMS error graph which is irregular and

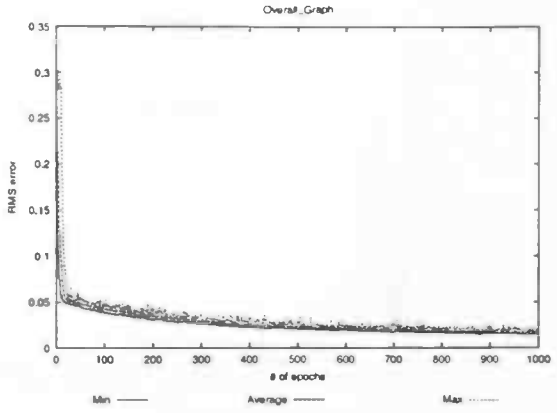


Figure 4.13 *Overall graph of A.L. (random) experiment 1.8; learning rate: 0.1*

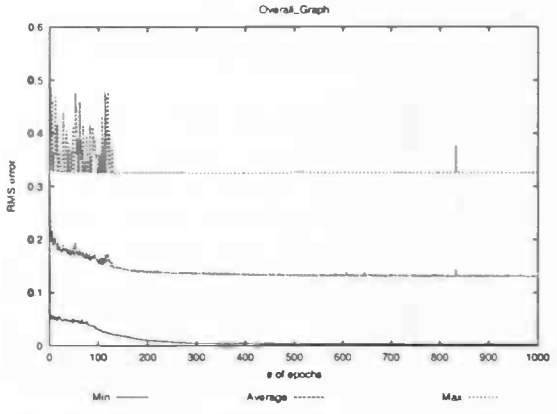


Figure 4.14 *Overall graph of A.L. (maximum) experiment 1.9; learning rate: 0.7*

unpredictable. The runs themself reveal behaviour which is seen in the reference experiments as well: no learning at all (figure 4.10) or some learning but through an irregular learning curve (figure 4.11).

Experiment 1.6 and 1.7 are not much different from the reference experiments with comparable learning rates. The overall graph of experiment 1.7 given in figure 4.12 is much alike the graph seen for reference experiment 1.3. Again the experiment with the smallest learning rate (experiment 1.8) has the best stability of all: the three curves are almost identical to eachother (figure 4.13). When looking at the runs seperately it is clearly visible that each run is more or less the same when compared to the overall graph. Apparently a small learning rate has a good influence on the stability of the learning process in a modular network.

Maximum experiment

The Alternate Learning experiments 1.9 till 1.12 are based on the concept in which the

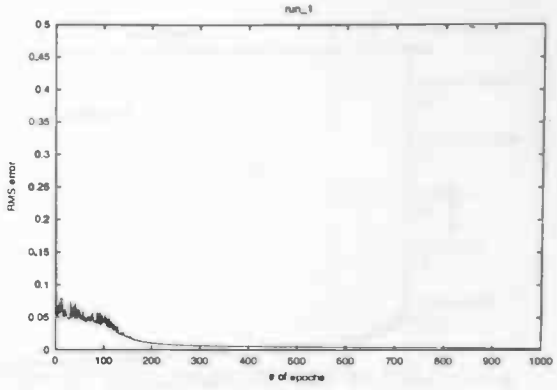


Figure 4.15 *Run 1 of A.L. (maximum) experiment 1.9: a stable and learning process*

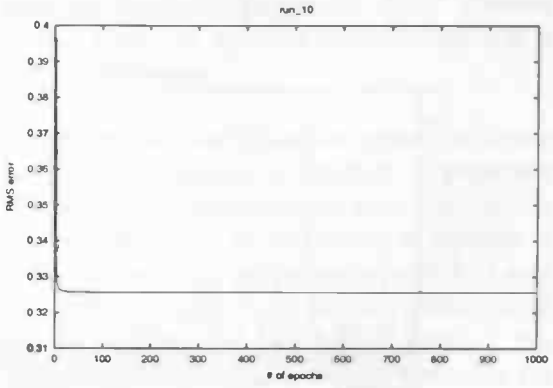


Figure 4.16 *Run 10 of A.L. (maximum) experiment 1.9: a stable but NOT learning process*

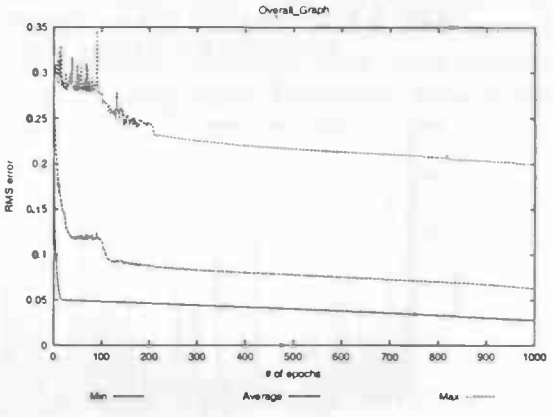


Figure 4.17 *Overall graph of A.L. (maximum) experiment 1.12; learning rate: 0.1*

network with the largest error for a certain pattern will be trained. The experiments carried out using this concept show some very extreme results, both in a positive and a negative way. Starting with experiment 1.9 in which a learning rate of 0.7 is used, some very 'straight' graphs were produced (figure 4.14). It is remarkable that the maximum and the minumum curve in this overall graph are almost solely caused by two single runs. This immediatly reveils the problem in this experiment: very large differences between the

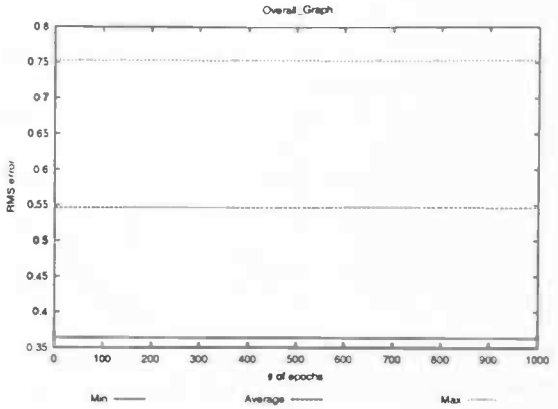


Figure 4.18 *Overall graph of A.L. (minimum) experiment 1.13; learning rate: 0.7*

runs as seen in for example run 1 (figure 4.15) and run 10 (figure 4.16).
Experiments 1.10 and 1.11 show almost exactly the same characteristics even though the learning rate has been decreased to 0.3 in the latter experiment.
The last experiment with the maximum error concept (figure 4.17) is somewhat more stable in the way that all runs show a steady improvement. Some of the runs start at a high RMS error value but even then they improve steadily, very slow though. Since all runs still show improvement when the end of the runs come in sight, it seems that a thousand epochs is not enough for this learning rate.

Minimum experiment
Experiments 1.13 to 1.16 use the minimum error of all networks to select a network for training. The idea behind this concept is that by improving the already best network, the overall performance should also improve.

The results of the minimum RMS error experiments can be called rather dull. Without exception they all show straight horizontal graphs (figure 4.18). Here it does not matter whether an experiment is carried out with a learning rate of 0.7 or if it is done with a learning rate of 0.1: characteristic shapes and magnitudes of the curves are all the same. Some runs do show a decreasing RMS error during the first 5 epochs or so, but after this promising start they show no progress any futher (figure 4.19).

The reason for this behaviour must be looked for in the fact that training the already best network does not automatically mean the overall performance increases *continuesly*.

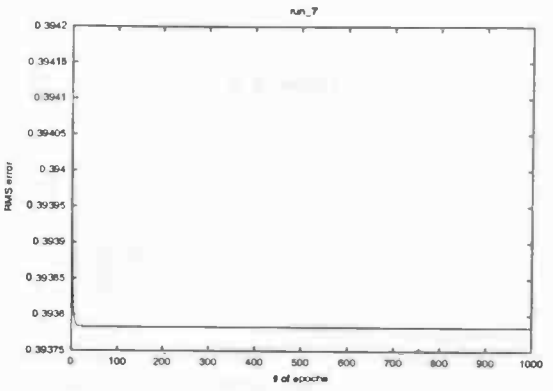


Figure 4.19 *Run 7 of A.L. (minimum) experiment 1.15: a not improving process*

Figure 4.19 does show improvement during the first few epochs, but after those epochs the modular network apparently needs to adjust other networks than the best to improve overall performance. Because no other networks than the best network are trained added to the fact that the best network can't get much better after a while (those first few epochs), the increase in overall performance halts: no more adjustments to the modular network are made.

4.1.4 Preliminary conclusion

Although more testing has to be carried out, some general conclusions can already be made.

First and most obviously, the minimum error approach does not work as it is defined before and will therefore not be used throughout the rest of this paper anymore. This does not mean the minimum error approach should not be considered at all anymore. For example one could think about an adjusted minimum error approach in which the 'other' networks (all networks except for the one performing best) are also trained with a fraction of the learning rate used for training the best network. Because this paper focusses primarly on testing the suggested approaches, no effort will be put in discovering better mutations of the minimum error approach.

Another conclusion which can be made is that none of the tried learning methods is really convincing in terms of stability and RMS error, unless a very small learning rate (0.1) is used. This is not really a surprise since it is common knowledge that using a small

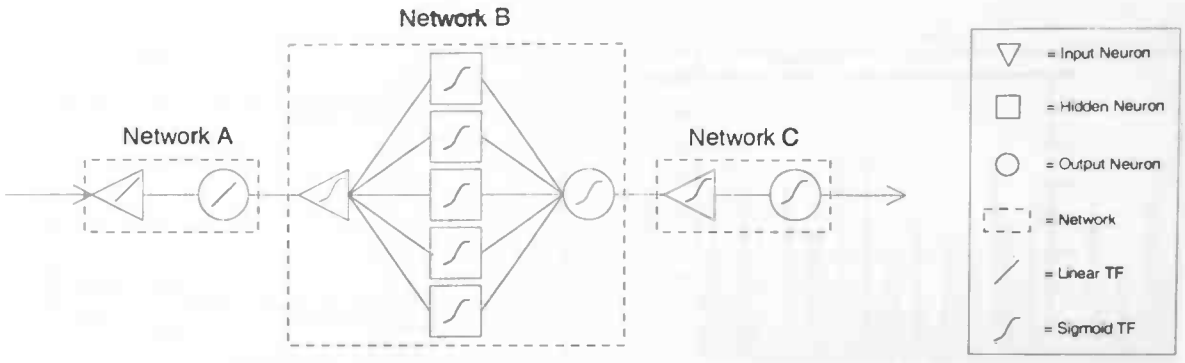


Figure 4.20 *Modular network configuration used for the additional sinus experiment; this configuration is the same as used for the (first) sinus experiment, except for the transfer functions in network C*

learning rate guarantees a more 'smooth' learning process. The disadvantage however is that it takes more time to learn to network(s).

One could wonder though why there seems to be such a 'sharp' border in the range of learning rates; a border from where on the learning process seems to be more stabile. To examine whether this behaviour is somehow related to the structure of the modular network, some more experiments where carried out using a different output network. Compared to the previous experiments the output network used a sigmoid transfer function instead of a linear transfer function, as can be seen in figure 4.20. The idea behind this experiment is that a linear transfer function at an output neuron results in an instabile learning process for large learning rates, while the sigmoid transfer function is more stabilizing.

Based on the configuration given in figure 4.20 a new experiment schedule is presented in figure 4.21. The minimum error approach is left out here, since no difference in stability is to be expected; the problems encountered with minumum error approach are not related to the transfer function of the output network but rather to the method of selecting networks for adapting, as explained before. Different from the previous experiments is also that only learning rates of 0.7 and 0.5 are used. The reason for this is choice is that, as a general rule, smaller learning rates automatically result in more stable behaviour. Now if the

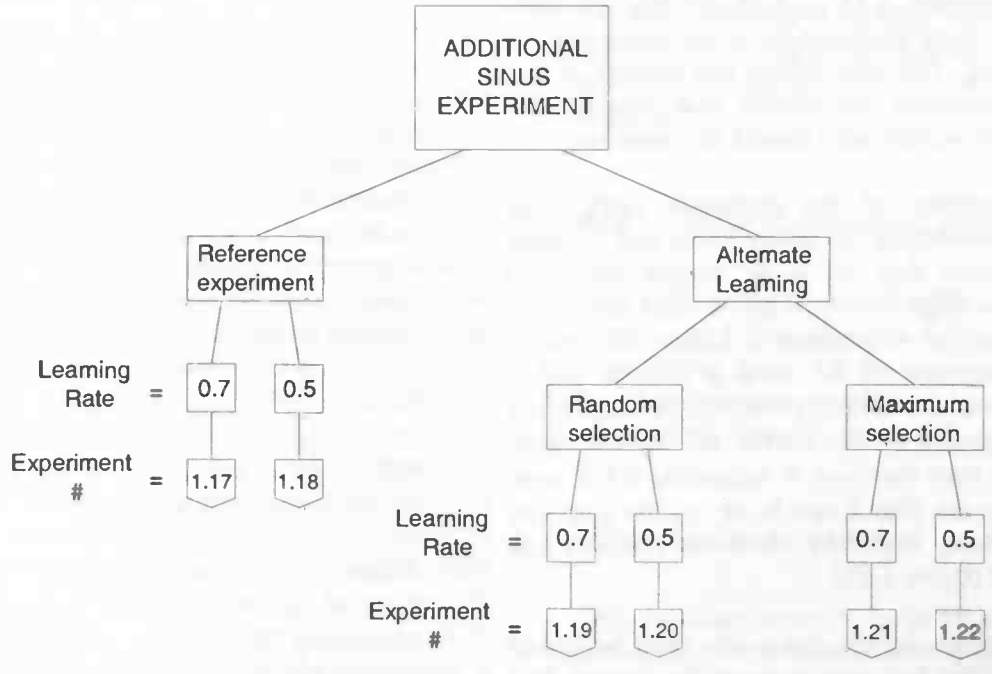


Figure 4.21 *Schedule for the Additional sinus experiment*

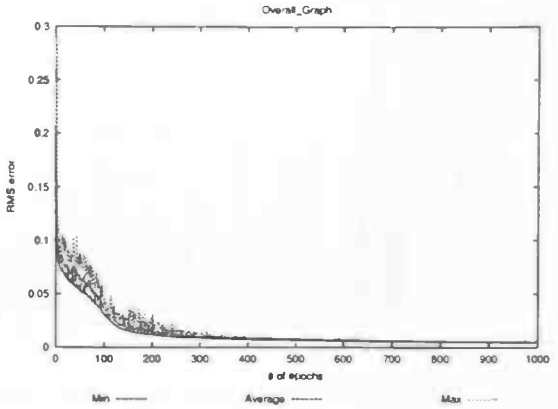


Figure 4.22 *Overall graph of reference experiment 1.17; learning rate: 0.7*

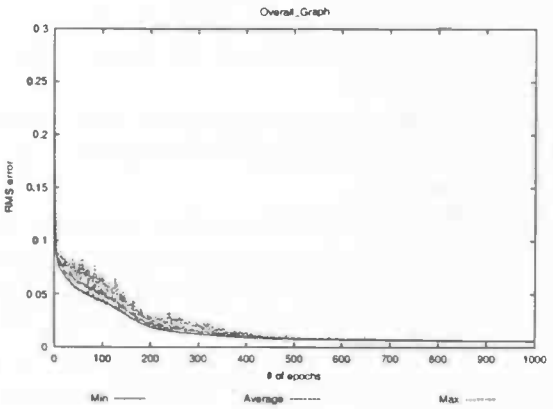


Figure 4.23 *Overall graph of reference experiment 1.18; learning rate: 0.5*

assumption that a sigmoid transfer function in an output neuron stabilizes the learning process then there is no reason to believe that a smaller learning rate would somehow causes more instability again.

Reference experiment

The reference experiments 1.17 and 1.18 immediatly show major improvement in stability as seen in figure 4.22 respectively 4.23. These overall graphs show a convergency in the minumum, average and maximum curve after about 300 (figure 4.22) and 500 epochs (figure 4.23). As concluded before this is a measure for stability *among* the several runs. Looking at the runs themself indeed shows a similar behaviour throughout these experiments (appendix ??). The stability *during* the runs compared to those in the previous reference experiments (experiments 1.1 and 1.2) becomes better as well. A more smooth curve is seen especially after the 300 and the 500 epochs.

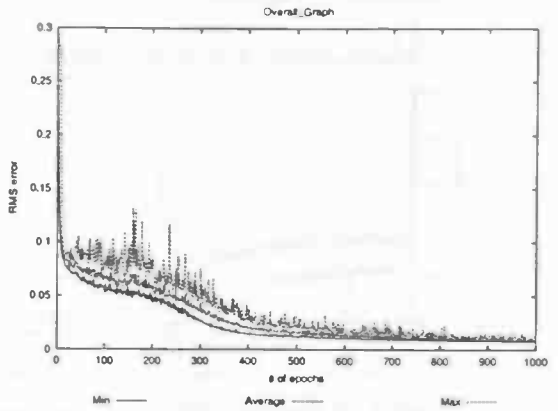


Figure 4.24 *Overall graph of A.L. (random) experiment 1.19; learning rate: 0.7*

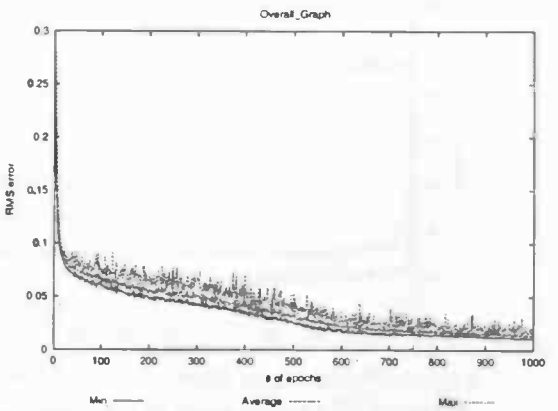


Figure 4.25 *Overall graph of A.L. (random) experiment 1.20; learning rate: 0.5*

Sofar the changed output network definitely brought more stability in these reference experiments. The big question of course is whether this change in the network structure also has its (positive) effect(s) on the learning process using the Alternate Learning method?

Random experiment

Looking at the overall graphs of experiment 1.19 and 1.20 in figure 4.24 and 4.25, this question can partially be confirmed. Although the graphs do show a major improvement in stability again, it doesn't seem to be 'as major' as seen in the reference experiments 1.17 and 1.18. The curves show less convergency. However, one should remind that the modular network in the reference experiments had three times more adjustments as it had in this Alternate Learning experiment: now only one network is adapted each pattern instead of all three networks. The convercency of the curves in the reference experiment reached its maximum after 300 epochs (figure 4.22) which is after about the same amount of

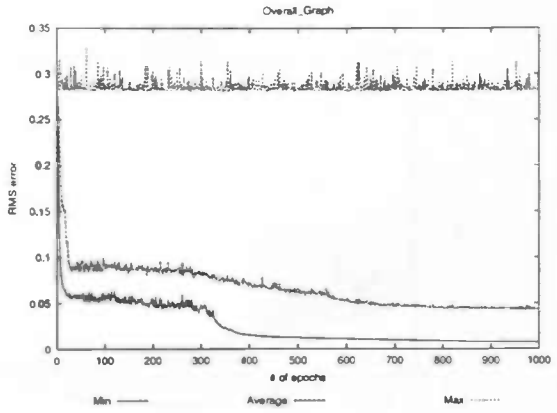


Figure 4.26 *Overall graph of A.L. (maximum) experiment 1.21; learning rate: 0.7*

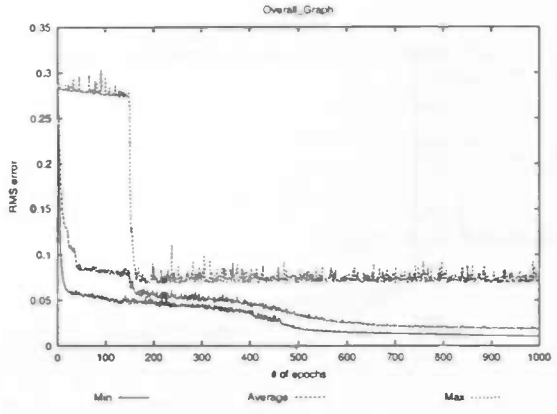


Figure 4.27 *Overall graph of A.L. (maximum) experiment 1.22; learning rate: 0.5*

adjustments as at the end of experiment 1.19 (figure 4.24). This is a one to three ratio and therefore supporting the theory explained above.

So instead of using the number of epochs as a timebasis but the rather the number of *adjustments* made to the modular network, will show that the random selection approach is not outperformed by the normal learning method in terms of stability. However, since it is not performing much better either one could wonder if the maximum error approach is able to do so?

Maximum experiment

Experiments 1.21 and 1.22 show no improvement compared to the previous experiments 1.17 to 1.20. The differences between the seperate runs are significantly larger as can be clearly seen in the corresponding overall plots (figure 4.26 respectively 4.27). Most of the runs show a more or less common learning curve, but some runs differ so much from this common

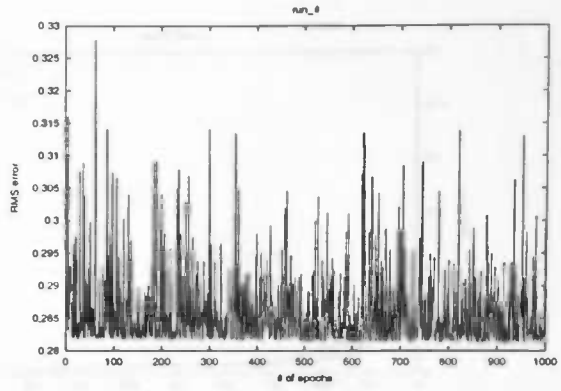


Figure 4.28 *Run 4 of A.L. (maximum) experiment 1.21: responsible for the maximum error in the overall plot*

learning curve that they are solely responsible for the maximum error in the overall plot (experiment 1.21; run 4 in figure 4.28).

Based on these additional experiments one important preliminary conclusion can now be made:

The structure of a modular network can be of such an influence on the learning process that the method of learning this modular network becomes less important. This does not mean the learning method does not matter at all, but rather that a 'wrong' configuration has such a negative impact that neither a normal learning method nor Alternate Learning, as suggested in this paper, is really making a difference anymore.

To test Alternate Learning futher, another experiment is done by training a modular network to approximate a more complex function.

4.2 Natural exponent experiment

This experiment is carried out to confirm the results and conclusions of the sinus experiment. Some of the conclusions made based on the sinus experiment are used in the setup of this experiment.

4.2.1 Experiment setup

This modular network is build out of three networks and is trained with a patternlist of 625 patterns. The modular network is

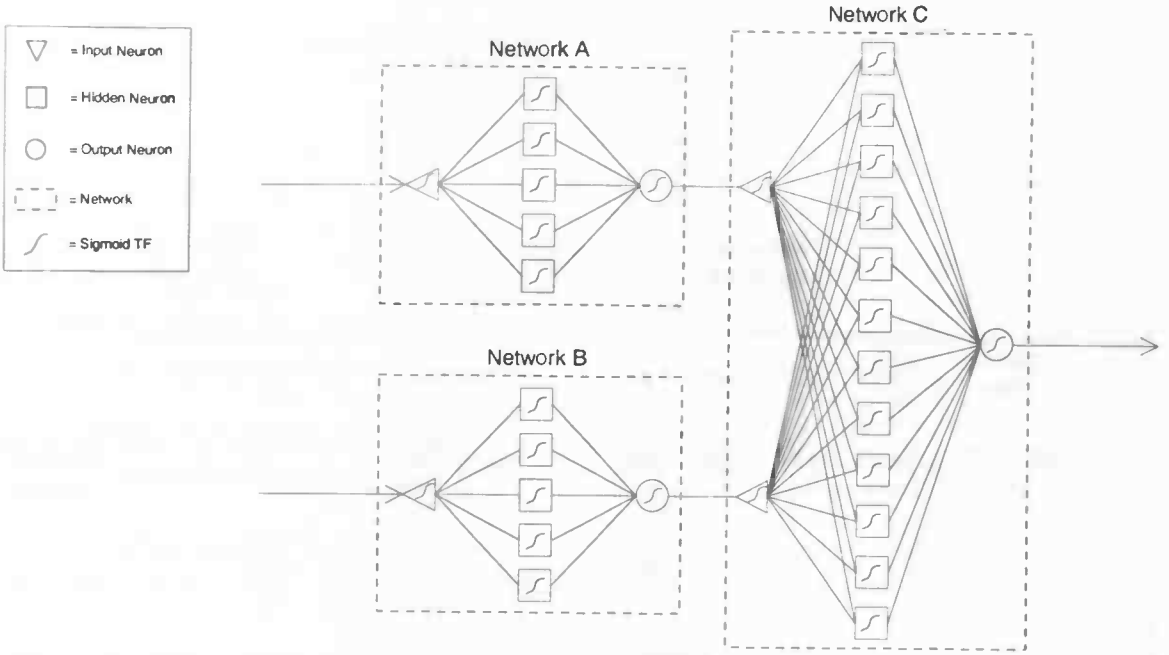


Figure 4.29 *Modular network configuration used for the natural exponent (second) experiment; network A and B have been pre- trained to fit a quadratic function*

supposed to approximate the function f described by:

$$f(x, y) = e^{-8.0*(x^2+y^2)} \qquad (4.1)$$

with

$$-1 < x < 1$$
$$-1 < y < 1$$

The configuration given in figure 4.29 shows two networks A and B, and one combining network C. Both networks A an B are pre-trained with a different patternlist than the one used for the modular network. Network A is trained with 25 patterns to be able to approximate:

$$k(x) = x^2 \qquad (4.2)$$

and network B to approximate:

$$p(y) = y^2 \qquad (4.3)$$

The combining network C, which weights and biases are random initialised, is expected to learn the function:

$$t(x, y) = e^{-8.0*(k(x)+p(y))} \qquad (4.4)$$

which is obviously equal to (4.1).

For pre-training both input networks the same parameters are used as given in section 4.1.

Like in the previous experiment is every experiment carried out in quadruplicate; each time with a different learning rate. The same learning rates are used as before namely 0.7, 0.5, 0.3 and 0.1.

As a result of the sinus experiment, the Alternate Learning variant in which the network with the smallest error is trained (the minimum error approach) is left out in this experiment. The reason for this decision is given in section 4.1.4. Having said this the experiment schedule looks like figure 4.30.

Patterlist characteristics:	
network A and network B	
# of patterns	25
input range	[-1 .. 1]
modular network	
# of patterns	625
input range	[-1 .. 1]

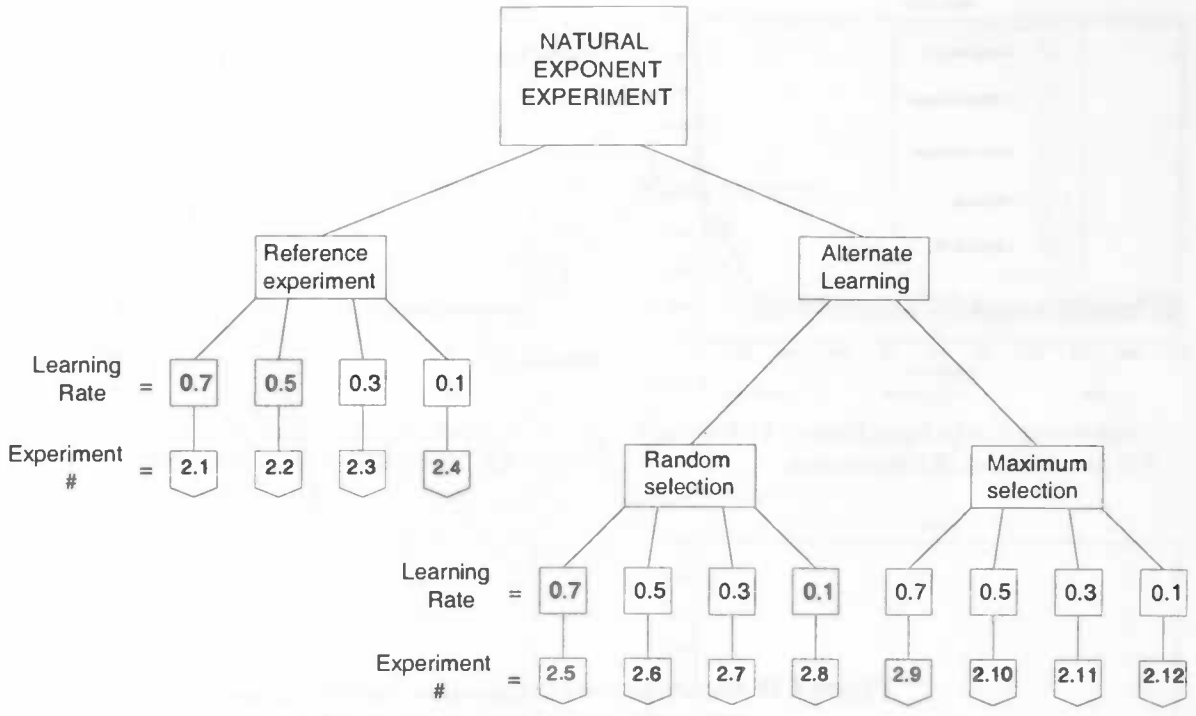


Figure 4.30 *Schedule for the second experiment*

4.2.2 Results second experiment

Reference experiment

After pre-training both input networks one would expect that training the whole modular network would result in a combining network fitting function (4.4). However since *all* networks (at the same time or selected after eachother) adapt during the learning process, it is more likely that both pre-trained networks shift their learned space, from which the combining network needs to fit some other function in order for the modular network to fit function (4.1).

Figure 4.31 shows the overall graph for reference experiment 2.1. Compared to the sinus reference experiment a clear more or less stable learning curve can be seen. All three curves in the overall plot are close to one another indicating identical runs. Looking at some of the individual runs confirms this conclusion. Figure 4.32 and 4.33 show run 1 and run 4 of this experiment. When the 'noise' in these curves is left out, both graphs show the same learning curve. This 'noise' is, like explained before, also a sort of unwanted instabile behaviour. The overall plot reveils that although the minumum and average errors

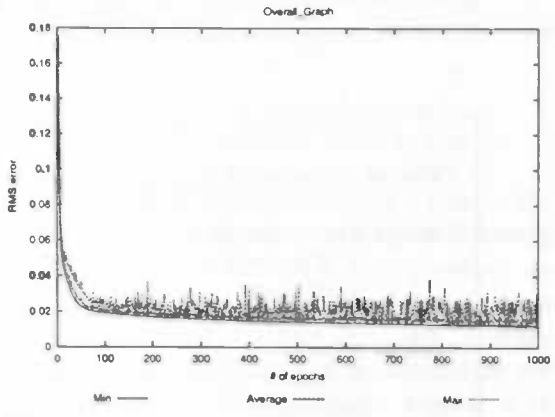


Figure 4.31 *Overall graph of ref. experiment 2.1 using a learning rate of 0.7*

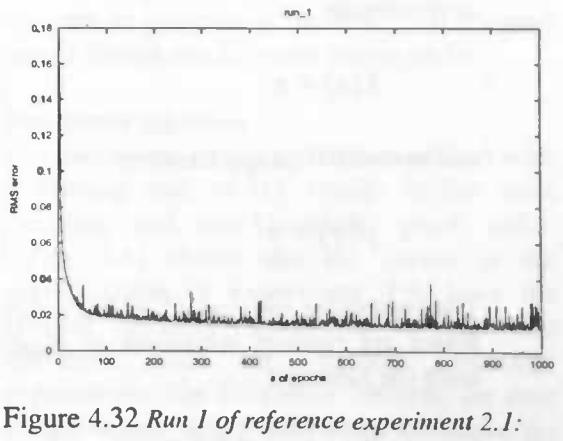


Figure 4.32 *Run 1 of reference experiment 2.1: steady learning curve, little instability*

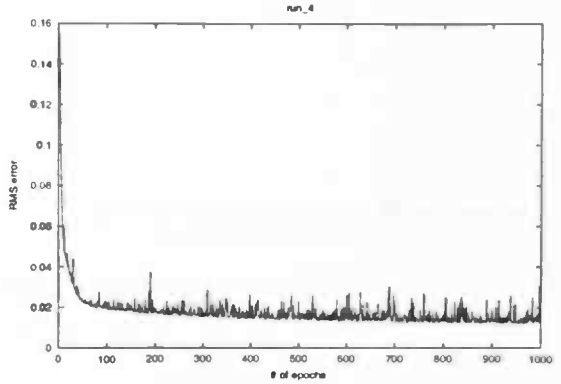


Figure 4.33 *Run 4 of reference experiment 2.1: identical graph as in run 1 (fig 4.32)*

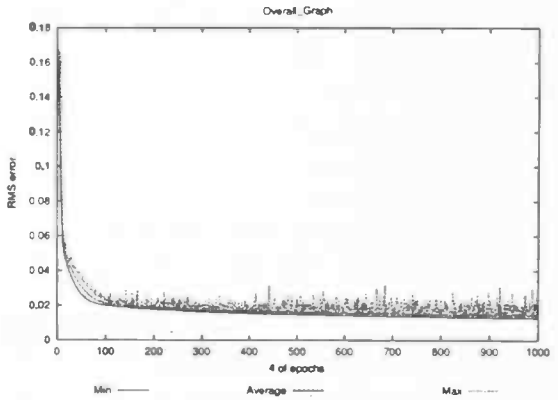


Figure 4.34 *Overall graph of ref. experiment 2.2 using a learning rate of 0.5*

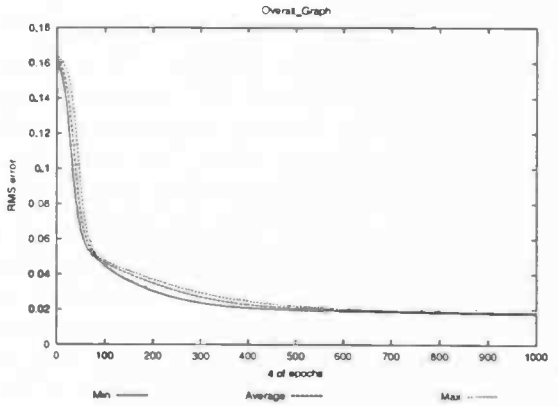


Figure 4.35 *Overall graph of ref. experiment 2.4 using a learning rate of 0.1*

tend to decrease during time, the maximum error increases by such a magnitude as what gained with the minimum or average error. Because of this behaviour the instability becomes more dominant.

Decreasing the learning rate in experiment 2.2 to 2.4, does help in removing this instabilty. In figure 4.34 the overall graph of experiment 2.2 shows less noise on top of the average curve

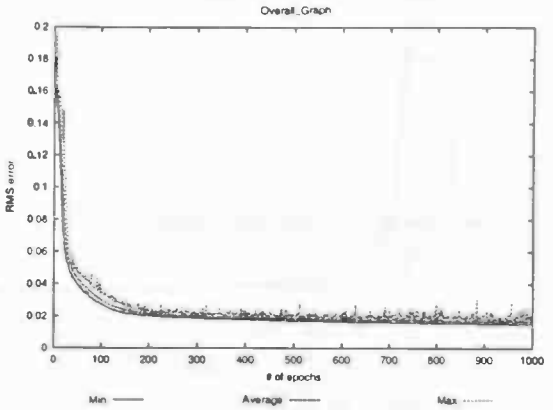


Figure 4.36 *Overall graph of A.L. (random) experiment 2.5; learning rate: 0.7*

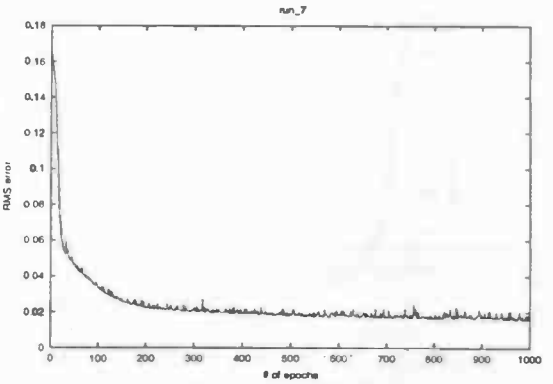


Figure 4.37 *Run 7 of A.L. (random) experiment 2.5: more smooth than experiment 2.1*

than seen in experiment 2.1. Although this decrease in learning rate does increase stability both during runs and among runs, it is remarkable that it doesn't effect the average learning curve. It seems like the modular network does not learn slower nor worse than when using a greater learning rate. It is not till a learning rate of 0.1 is used before a slower learning curve is seen (experiment 2.4; figure 4.35). Experiment 2.4 shows the most stable learning curve together with the slowest learning curve, resulting in a slightly higher RMS error after a 1000 epochs.

Random selection

The overall graph for experiment 2.5 in figure 4.36 shows a more stable learning process than which can be seen with the comparable reference experiment. Although the absolute value of the average curve is a bit larger than seen at the reference experiment, the noise on top of this average curve is significantly smaller with this random selection method. It is noteworthy that when looking at individual runs (run 7 in figure 4.37 for example), the

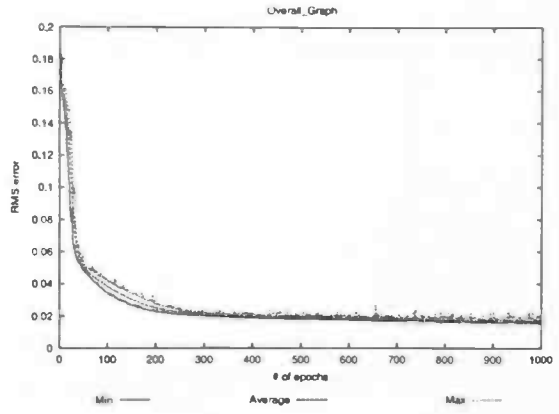


Figure 4.38 *Overall graph of A.L. (random) experiment 2.6; learning rate: 0.5*

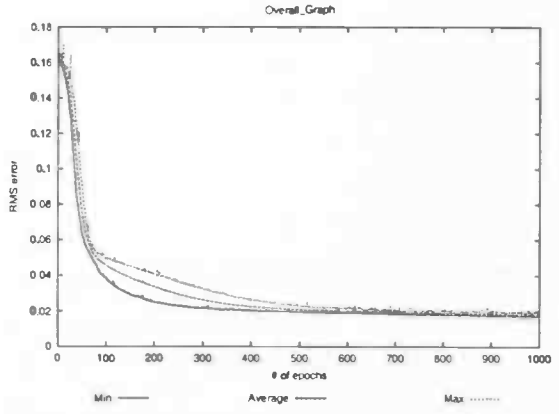


Figure 4.39 *Overall graph of A.L. (random) experiment 2.7; learning rate: 0.3*

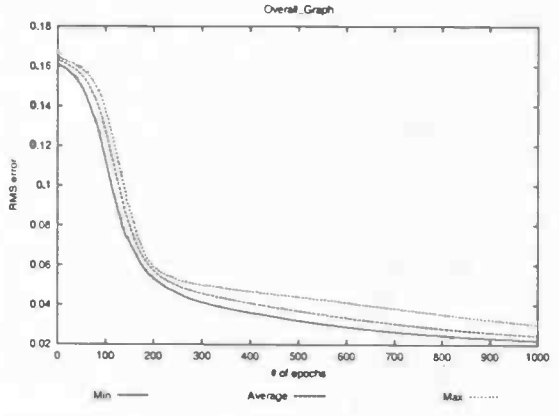


Figure 4.40 *Overall graph of A.L. (random) experiment 2.8; learning rate: 0.1*

graphs are more smooth, less spiky, than in the reference experiment. Decreasing the learning rate with this method of learning shows an even greater improvement. Again, just as seen in the reference experiment, the average learning curve appears to be uneffected by the decrease, unless the stability! Figure 4.38 shows the overall plot for experiment 2.6 in which the noise is reduced to such a level that

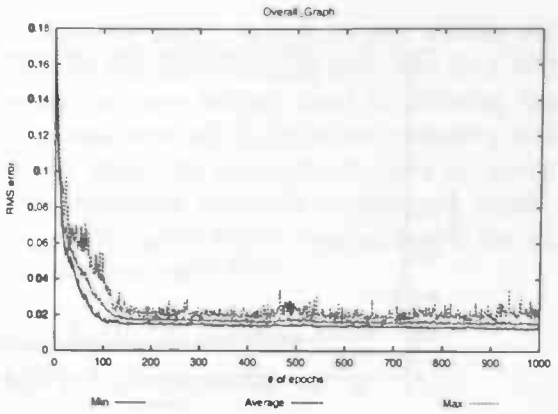


Figure 4.41 *Overall graph of A.L. (maximum) experiment 2.9; learning rate: 0.7*

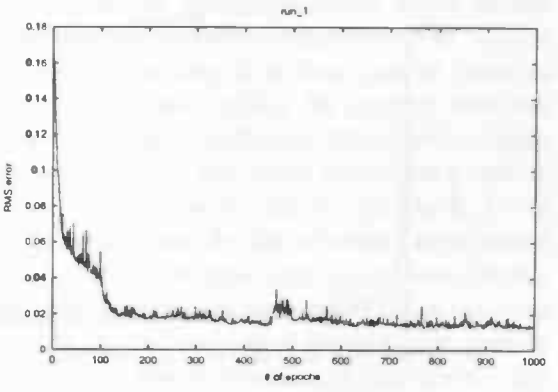


Figure 4.42 *Run 1 of A.L. (maximum) experiment 2.9: a learning but somewhat instable process*

it becomes difficult to distinguish the three different curves. Training with a learning rate of 0.3 (experiment 2.7) increases the stability even more, but also slows down the learning process (figure 4.39). The last experiment, using a learning rate of 0.1, reveals that a 1000 epochs are not enough when training with such a small learning rate. The overall plot of this experiment (figure 4.40) points best out too few adaptions were made for the modular network to perform at its best. The learning curves though are the most stabile so far.

Maximum selection

The maximum error approach combined with a learning rate of 0.7 results in the most irregular, and thus instabile, graph sofar. Figure 4.41 shows that the curves in the overall graph of experiment 2.9, have the greatest deviation compared to both the reference and the random selection experiments. The difference between the runs is significantly larger than when looking at the other experiments, as can be seen in figure 4.42 and 4.44, showing run 1 repectively run 4

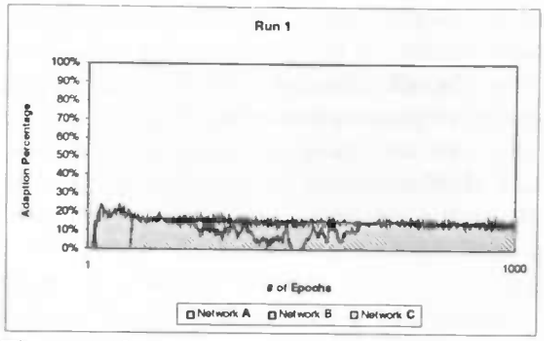


Figure 4.43 *Adaption percentage plot derived from figure 4.42*

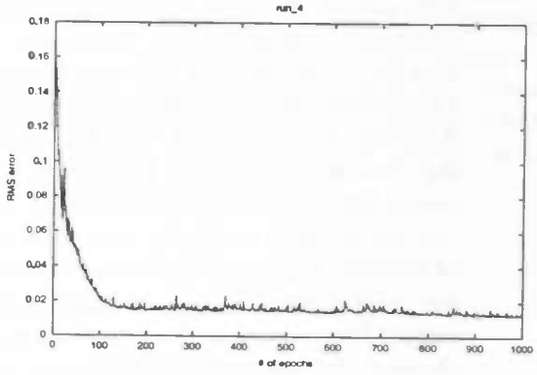


Figure 4.44 *Run 4 of A.L. (maximum) experiment 2.9: a somewhat different learning curve than seen with run 1 (fig. 4.42)*

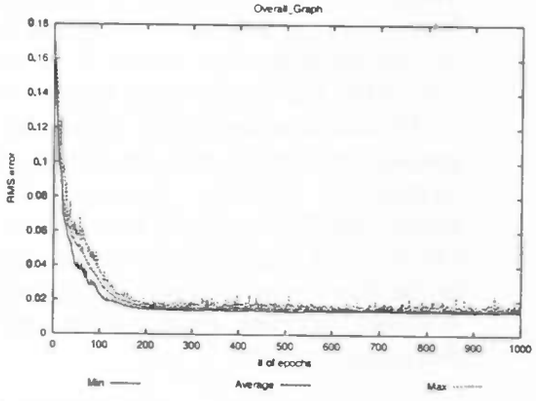


Figure 4.45 *Overall graph of A.L. (maximum) experiment 2.10; learning rate: 0.5*

of experiment 2.9. Since the decision of what network to train is based on the network generating the largest error, it is interesting to see what part each network has in the total adaption time. Figure 4.43 gives some insight on this matter. As expected, the untrained combining network (network C) gets all the adaption time during the first few epochs. What is surprising though, is that after about a 100 epochs the percentage of adaptions used for network C stays at a stable 80%. The other 20 % is divided between the other two pre-

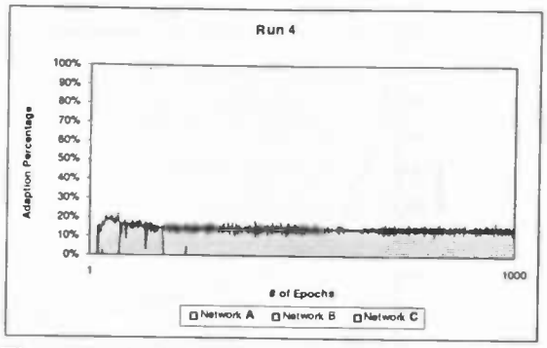


Figure 4.46 *Adaption percentage plot derived from figure 4.44*

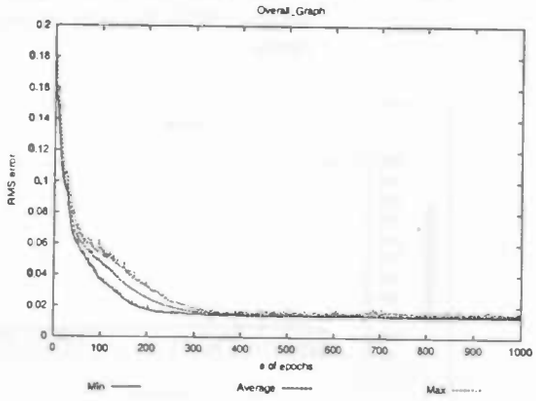


Figure 4.47 *Overall graph of A.L. (maximum) experiment 2.11; learning rate: 0.3*

trained networks. Apparently the untrained network can not adapt in such an extent that it performs just as good as both pre-trained networks.

Decreasing the learning rate in this experiment has a pleasant effect. More stability is gained as can be seen in the overall plot of experiment 2.10 (figure 4.45). Where the three curves showed such a deviation to one another when training with a 0.7 learning rate, with 0.5 as the learning rate they are almost as close to one another as what was seen in the random selection experiment 2.4. Apparently the learning rate has more influence when using the maximum error approach than when using random selection or training all networks. This decrease in learning rate however has no effect on the proportion of adaptions between the different networks (figure 4.46). As seen with experiment 2.9, it is the untrained network C which is taking a steady 80% of the training cycles, leaving the other 20% to be divided between the other two pre-trained networks. Although figure 4.46 reflects only the adaption ratio for run 4 of experiment 2.10, it is representative for the entire experiment 2.10.

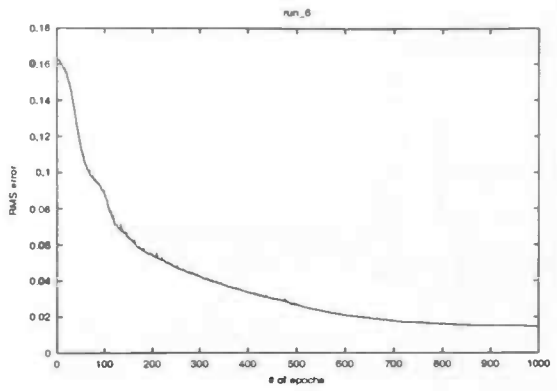


Figure 4.48 *Run 6 of A.L. (maximum) experiment 2.12: a steady but slow learning curve*

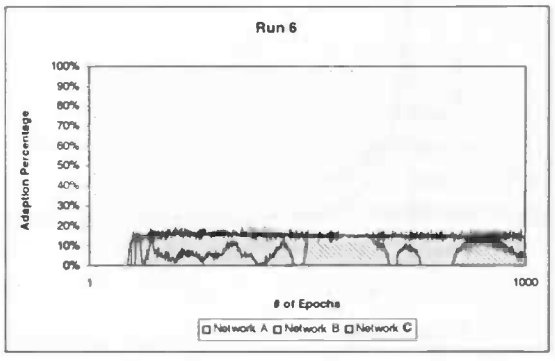


Figure 4.49 *Adaption percentage plot derived from figure 4.48*

Decreasing the learning rate even futher to 0.3 results in a slightly slower learning process but also a more stabile one. The three curves in the overall plot of experiment 2.11 (figure 4.47) show an even better performing modular network than seen with the random selection method (experiment 2.7). Both the speed of learning and the value of the RMS error show improvement when compared to experiment 2.7. Compared with the reference experiment however, a slower learning curve but a better RMS error (smaller error) is seen. In figure 4.48 (experiment 2.12) the effect of a small learning rate is clearly demonstrated. Here the network selected for training is adapted with a learning rate of 0.1. This results in a slow but stabile learning process. But unlike experiment 2.8 where a 1000 epochs were not enough to come to the maximum performing modular network, by using maximum selection here it seems like a 1000 epochs are just enough to get the best performing network. Looking at an adaption ratio plot in figure 4.49 (experiment 2.12, run 6) reveils more or less the same characteristics as seen in figure 4.43 and 4.46 (experiment 2.9 respectively 2.10), though clearly visible the effect of the slow learning rate. The first 100 epochs are now entirely used for training the un-trained network C, while the remaining 900 epochs have the same 80/20 ratio as every other maximum approach experiment. Again, the plot in figure 4.49 is representative for all runs in experiment 2.12.

4.3 Conclusions

The results of these two experiments with Alternate Learning are promising. When looking at the general learning curve of the experiments the first conclusion would be that Alternate Learning is at least just as good as the conventional method for training modular neural networks. Although Alternate Learning did not result in satisfying behaviour in terms of stability, during the sinus experiment, it was not outperformed by the reference experiment either. The additional sinus experiment clearly showed that the instabile behaviour was not caused by the method of learning but merely by a wrong network configuration. By changing the network configuration (a sigmoid tranfer function instead of a linear transfer function in the output network) the learning process became more stable. It is therefore justified to note that a wrong modular network configuration unfortunately affects the whole learning process to such an extent, that the method of learning becomes of minor importance. Unfortunately because one would like to use a learning method less dependent on the way the modular network is configured. Alternate Learning does however not solve this problem either.

The natural exponent experiment clearly showed that the Alternate Learning method *does* has its postive effect on the learning process. Especially the random selection approach performed significantly better for the greater learning rates (0.7), while the maximum selection approach did as well but only for the smaller learning rates (0.5 and smaller). While stability during the runs did indeed improve, was it not at the expense of the absolute error. In all runs the absolute error was after a 1000 epochs just as small or smaller than seen in the reference experiment.

When using Alternate Learning one has to think of a selection method to decide what network to train. This is merely the only extra effort one has to put in when using Alternate Learning. When more research on this topic has been done though, it becomes likely that depending on the modular network configuration used, one can choose from a range of selection methods and thus saving time and energy in developping one themself.

On basis of these experiments it is justified to conclude that, when using the proper modular network configuration, Alternate Learning (and in special the random selection approach) results in more stabile and robust behaviour during the learning process while the absolute performance (i.e. error) stays unchanged or becomes better.

Chapter 5 Conclusions and recommendations

The goal of this paper is, as mentioned and explained in chapter one, to investigate and answer the question whether or not Alternate Learning is a more stable learning method. Chapter one also provided three different issues to be used for answering this question, namely:

- the generated error
- network stability
- effort using Alternate Learning

Based on the experiments described in this paper and the three topics above, the answer to the main question can be answer with: ***"Yes, Alternate Learning is a more stable learning method for modular networks"***. There are some foot-notes to this answer though.

5.1 Conclusions

The sinus experiment shows that the structure of the modular network can be of such influence, that the choice of learning becomes of minor importance. This is clearly visible when comparing reference experiment 1.1 and the random approach in experiment 1.5 for example (figure 4.4 and figure 4.9). These experiments behave in likewise manner, even though the average error seems to be slightly less in the experiment 1.1, which is an instable behaviour. All sinus experiments, both reference and Alternate Learning, seem to be suffering from the same cause. This cause turned out to be the *structure* of the modular network.

The additional sinus experiments show that by changing the transfer function of the output network, the behaviour of the modular network becomes more stable and performs better as well in terms of the absolute error (see table 5.1 for a 'consumer' reference table). Here the random approach behaves in the same manner as does the reference experiments: stable with a small absolute error. Although it may look like the random approach is slower in learning compared to the reference experiments (figure 4.24 and figure 4.22), looking at these experiments with the total amount of adaptions carried out on the modular network as a time basis, will make clear that the random approach is no slower in learning than the reference experiment. The learning rate has no extra impact on the stability of the modular networks; not more than a normal decrease in learning rate has. The maximum approach behaves less stable and has a higher absolute error. Decreasing the learning rate here has more impact than seen in the other additional sinus experiments, but

Additional sinus experiment 1.17-1.22		Stability *among* runs		Stability *during* runs		RMS error	
		Random approach	Maximum approach	Random approach	Maximum approach	Random approach	Maximum approach
Refrence experiment	Learning rate = 0.7	0	--	0	-	0	-
	Learning rate = 0.5	0	-	0	-	-	-

Table 5.1 *Results of the additional sinus experiment. In the table the performance of Alternate learning is compared to the reference experiments. – means AL performs less on this issue than the reference experiments, 0 no major difference, + AL outperforms reference and ++ AL outperforms reference significantly (the number of epochs is used as time basis!).*

Natural exponent experiment 2.1-2.12		Stability *among* runs		Stability *during* runs		RMS error	
		Random approach	Maximum approach	Random approach	Maximum approach	Random approach	Maximum approach
Refrence experiment	Learning rate = 0.7	++	-	+	+	0	0
	Learning rate = 0.5	+	+	++	+	0	0
	Learning rate = 0.3	0	+	+	0	0	0
	Learning rate = 0.1	-	0	0	0	-	+

Table 5.2 *Results of the natural exponent experiment. In the table the performance of Alternate learning is compared to the reference experiments. – means AL performs less on this issue than the reference experiments, 0 no major difference, + AL outperforms reference and ++ AL outperforms reference significantly(the number of epochs is used as time basis!).*

not enough to make it a really stable experiment.
Because of the results with the minumum approach (in the sinus experiment) it is not recommended to use this method for training modular networks. Using the minimum appraoch usually ends up in a situation where the modular network does not adapt itself anymore. This is exactly the reason why it is not used in the second experiment anymore.

The natural exponent experiment proved that, when using the proper modular network configuration, Alternate Learning does have a positive effect on stability *and* the absolute error (see table 5.2). Both the random and the maximum approach (latter only for smaller learning rates) performed better than the reference experiment. Stability during the runs is significantly better; plots of the RMS error are more smooth than seen with reference runs. Stability among the runs, or robustness, is slightly better; though this is of course closely related to the stability *during* the runs, there are no runs reported to be completly differing from the others nor during the reference experiments nor during the Alternate Learning experiments (both random and maximum approach).

Like mentioned in the previous chapter, not much extra effort has to be put in before one can use Alternate Learning. Yes, one has to put time into writing software capable of dealing the Alternate Learning approach, but this is a one time matter. When such a piece of software is available there are only two disadvantages related to Alternate Learning which are both solvable:

- The selection method has to be chosen
- Timeperiod needed for learning is longer

When more is known about Alternate Learning, it is to be assumed that more already researched selection methods are available, thus saving time.

The software used for the experiments in this paper, is only developed to be able to deal with modular structures. Not much time is put in making the software more efficient. Surely a lot of this software can be made more efficient resulting in shorter learning periods.

5.2 Recommendations

It is clear that this paper merely reports about primature experiments and that therefore more research has to be done before Alternate Learning can really become successful. Though primature, these first experiments with Alternate Learning are certainly encouraging and a lot can be expected of it in future when more is known about the impact of the learning algorithm on modular networks in general. Topics worth investigating futher are:

- What knowledge can be derived from the plots on the percentage of adaptions (figure 4.43, 4.46, 4.49)? In this paper the modular network decides *every* pattern what network to adapt. It is also possible to decide to intervene in this selection process when a certain behaviour is

detected. This detection can for example be based on the adaption plots mentioned before.

- Can the minimum approach be used when it is sligtly changed? In this paper the minimum approach was defined as adapting the network with the smallest error. This means that whenever a network reaches a state where it doesn't generate an error anymore, it isn't adapted either (see equation 2.3). This leaves the modular network in a deadlock: the network selected for adapting does not adapt because it is as good as it can get and the networks 'desperate' for adaption are not selected for adapting. It is worth investigating what will happen when the not selected network are too trained, but only with a very small learning rate; this should then prevent the deadlock.
- What effect has Alternate Learning on modular neural networks used for classification purposes? Sofar Alternate Learning is only used for training approximation modular networks. Since classification is a completly different topic it is interesting to see what influence Alternate Learning has on such problems.
- How well does Alternate Learning perform when using only random initialised networks in the modular structure instead of pre-trained modules? By pre-training networks, one pushes the modular network at forehand into a certain direction. Does Alternate Learning need such a push into a direction or can it deal with completly randomized modular networks? If so, it would make the learning process of modular structures considerably easier.

Appendix A Software Manual

Syntax of initialisation file:

```
<# of networks>
<filename train patternlist> <filename test patternlist>
<filename network A> <# of inputs network A> <# of outputs network A>
<input #1 of network A> <network B> <output #c of network B>
      .
      .
<input #d of network A> <network E> <output #a of network E>
<output #1 of network A> <network C> <input #d of network C>
      .
      .
<output #b of network A> <network D> <input #b of network D>
<randomize boolean> <learnrule for network A> <learnrate variables>
      .
      .
      .
<filename network D> <# of inputs network D> <# of outputs network D>
<input #1 of network D> <network C> <output #b of network C>
      .
      .
<input #b of network D> <network E> <output #b of network E>
<output #1 of network D> <network B> <input #a of network B>
      .
      .
<output #b of network D> <network B> <input #b of network B>
<randomize boolean> <learnrule for network D> <learnrate variables>
```

- Whenever an input or output of a network, is an input or output of the modular network then the network number needs to be `-1` and the in- or output number needs to be negative as well, followed by the corresponding global in- or output number.
  So if a certain network A has an input 2 which is also a modular network input number 3 then the initialisation file would have this line in it:
  `2 -1 -3`
- The order in which the networks are described is relevant, since it determines which number the network gets. When a an in- output of a certain network is connected to this network it must be refered to by the assigned number. The first network described is refered to by the number 1. Every network following, gets a number of one point higher compared to the previous one.
- The `<randomize boolean>` is one of two values: `y` or `n`. When it's `y` then the corresponding network is random initialised (weights and biases), when it's `n` then the network is used as loaded from disk.
- `<learnrule>` can be one of the following rules, each with its specific number of variables:

  normal <learnrate> <momentum>
  *Use this rule when **all** networks within the modular network need to be adapted.*

  randnet <learnrate A> <momentum A> <learnrate B> <momentum B>
  *Use this rule when, for adapting, **a** network needs to be **randomly** selected.*

Both variables containing A are applied to the selected network and variables containing B are applied to the unselected network(s)

minerror <learnrate A> <momentum A> <learnrate B> <momentum B>

Use this rule when the network with the ***smallest error*** *needs to be adapted. Both variables containing A are applied to the selected network and variables containing B are applied to the unselected network(s)*

maxerror <learnrate A> <momentum A> <learnrate B> <momentum B>

Use this rule when the network with the ***largest error*** *needs to be adapted. Both variables containing A are applied to the selected network and variables containing B are applied to the unselected network(s)*

Example:

Suppose one wants to train the modular network visualised in figure A.1. The specifications for this experiment are given by:

- Network A is stored on disk as "networkA.net".
- Network B is stored on disk as "networkB.net".
- Network C is stored on disk as "networkC.net".
- The patternlist used for training is stored on disk as "trainlist.pdbf".
- The patternlist used for testing is the same as the one used for training, namely "trainlist.pdbf"
- For every pattern a network needs to be randomly selected which will then be trained with a learning rate of 0.7 and a momentum term of 0.0; the remaining unselected networks must be trained with a learning rate of 0.2 and a momentum term of 0.0.
- Network A and B are pre-trained, so they are not supposed to be randomly initialised.
- Before the training process starts, network C needs to be randomly initialised

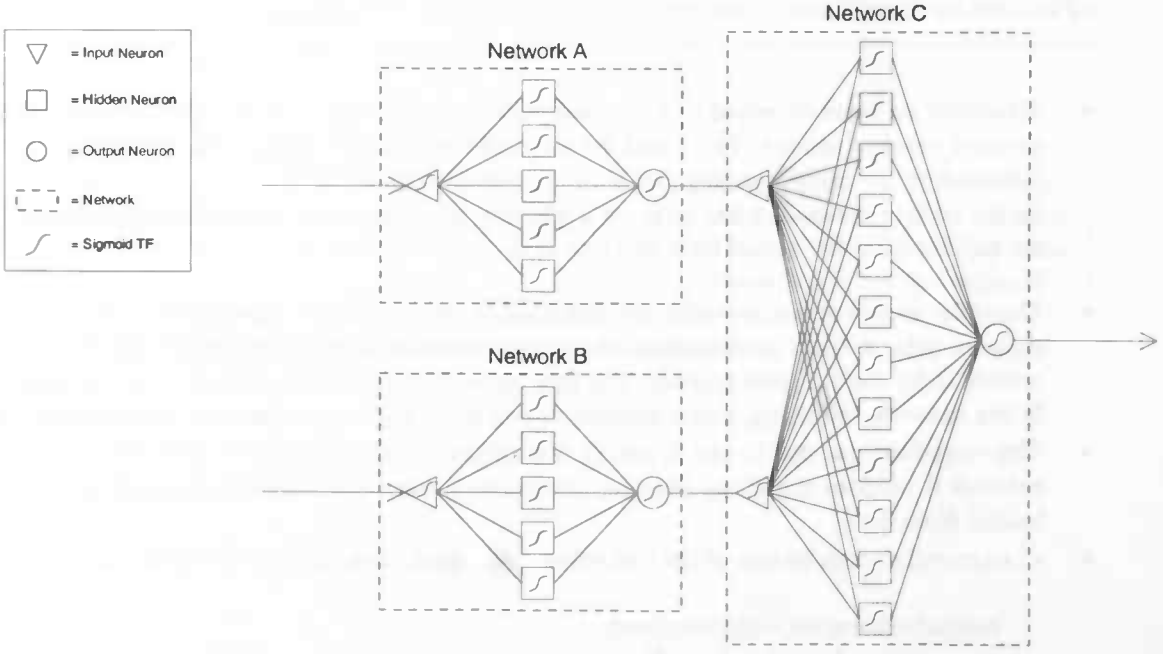


Figure A.1 *Modular network used for demonstrating the use of the initialisation file.*

To do this experiment the initialisation file needs to be like this:

```
3
trainlist trainlist
networkA.net 1 1
1 -1 -1
1 3 1
n randnet 0.7 0.0 0.2 0.0
networkB.net 1 1
1 -1 -2
1 3 2
n randnet 0.7 0.0 0.2 0.0
networkC.net 2 1
1 1 1
2 2 1
1 -1 -1
y randnet 0.7 0.0 0.2 0.0
```

Program command line options

The following command is used to start the program:

```
interact -x lrn_sinpow -e<#of epochs> -s <initialisation filename>
```

The `-s` is optional, since it is responsible for *saving* all internally used patternlists. It is not recommend to use this option since it will slow down the whole training process considerably. Both the `-e` option and the `<initialisation filename>` are required; without these parameters the program won't run.

Example:

```
interact -x lrn_sinpow -e100 -s ini_testfile.txt
```

Specifications for testing the program

The setups shown in figure A.2 are used to test the software. Both the modular and the normal network are trained with the same *unpermuted* patternlist (stored on disk as "sinus150.pdbf"). This list contains 150 patterns describing one cycle of a normal sinusfunction (domain: 0 - 2π). Network specifications for this test can be found in table A.1 and A.2. Both setups are trained with a learning rate of 0.7 and a momentum term of 0.0.

NORMAL NEURAL NETWORK	# of neurons	Bias(es)	Weight(s)	Transfer function
Input layer	1	0.0	-	sigmoid
Hidden layer	6	0.0	w1 .. w6	sigmoid
Output layer	1	0.0	w7 .. w12	sigmoid

Table A.1 *Normal neural network specification used for testing software*

MODULAR NETWORK	# of input neurons	# of output neurons	Bias(es)	Weight(s)	Transfer func. input neuron(s)	Transfer func. output neuron(s)
Network 1	1	6	0.0	clamped!	linear	linear
Network 2	1	1	0.0	w1	sigmoid	sigmoid
Network 3	1	1	0.0	w2	sigmoid	sigmoid
Network 4	1	1	0.0	w3	sigmoid	sigmoid
Network 5	1	1	0.0	w4	sigmoid	sigmoid
Network 6	1	1	0.0	w5	sigmoid	sigmoid
Network 7	1	1	0.0	w6	sigmoid	sigmoid
Network 8	6	1	0.0	w7 .. w12	linear	sigmoid

Table A.2 *Modular network specification used for testing software*

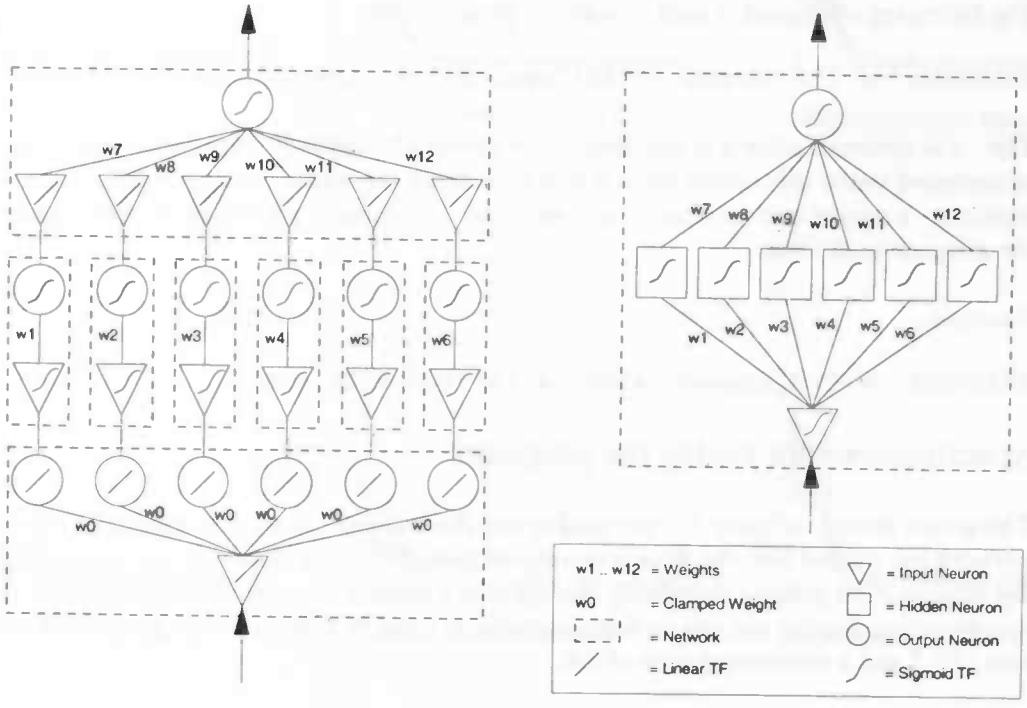
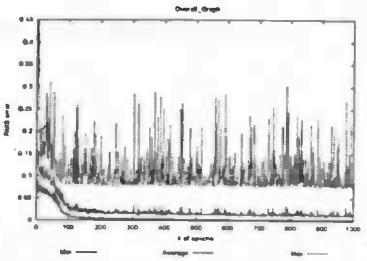


Figure A.2 *Experiment setup used for testing integrity of the program*

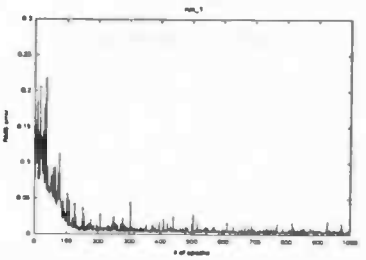
Appendix B Experiment results

Experiment 1.1

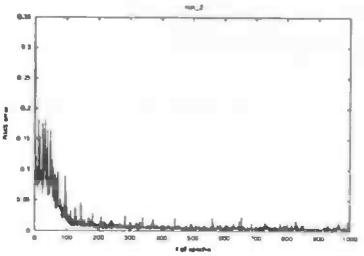
Sinus reference experiment
Learning rate = 0.7



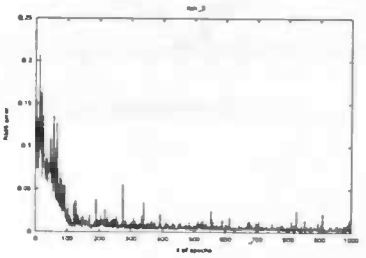
Experiment 1.1; Overall



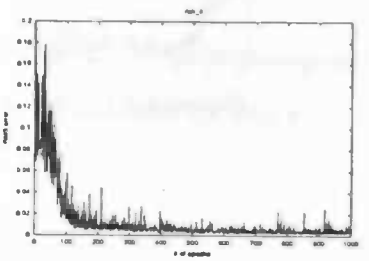
Experiment 1.1; Run 1



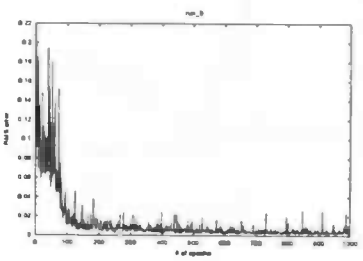
Experiment 1.1; Run 2



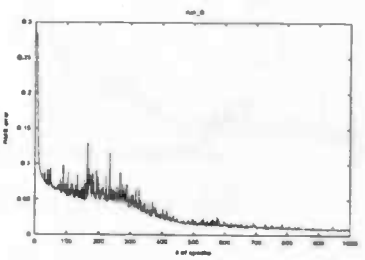
Experiment 1.1; Run 3



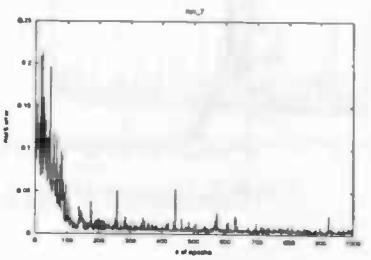
Experiment 1.1; Run 4



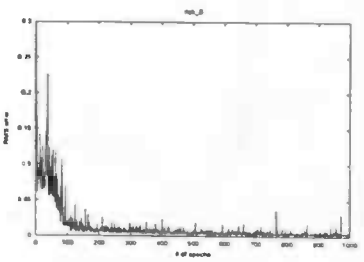
Experiment 1.1; Run 5



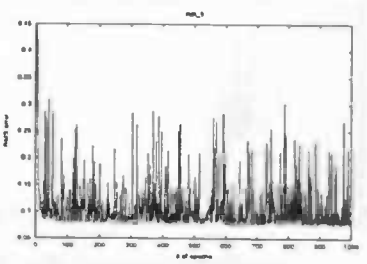
Experiment 1.1; Run 6



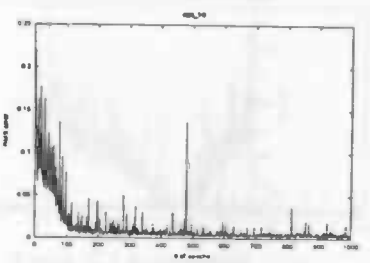
Experiment 1.1; Run 7



Experiment 1.1; Run 8



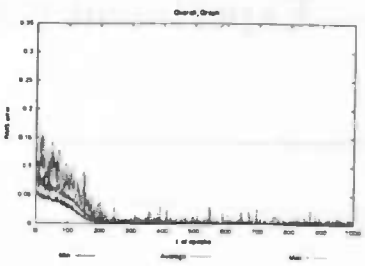
Experiment 1.1; Run 9



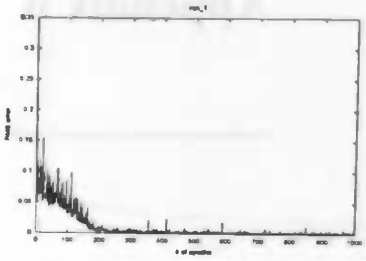
Experiment 1.1; Run 10

Experiment 1.2

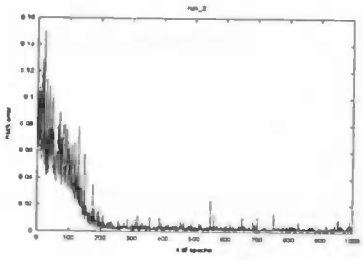
Sinus reference experiment
Learning rate = 0.5



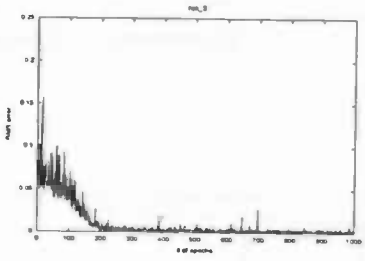
Experiment 1.2; Overall



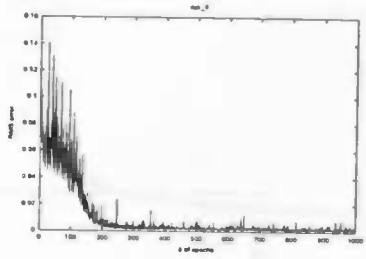
Experiment 1.2; Run 1



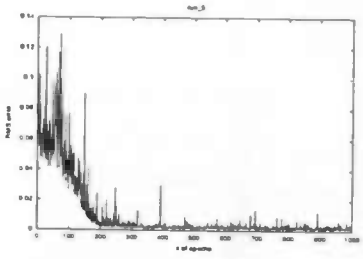
Experiment 1.2; Run 2



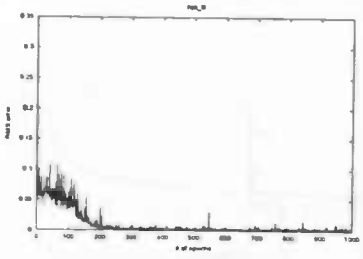
Experiment 1.2; Run 3



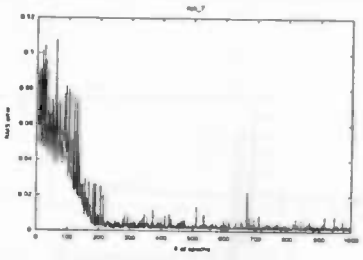
Experiment 1.2; Run 4



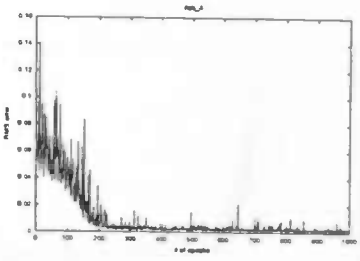
Experiment 1.2; Run 5



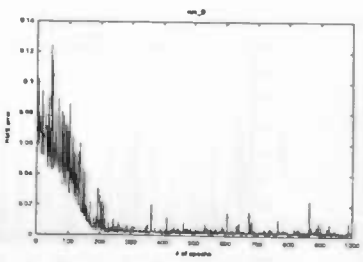
Experiment 1.2; Run 6



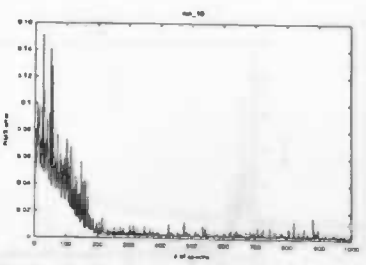
Experiment 1.2; Run 7



Experiment 1.2; Run 8



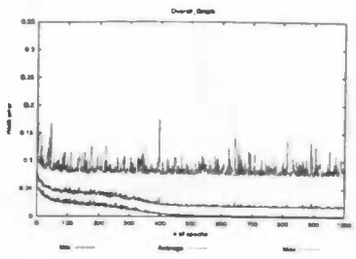
Experiment 1.2; Run 9



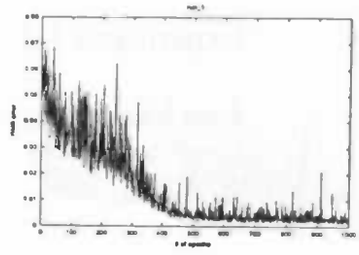
Experiment 1.2; Run 10

Experiment 1.3

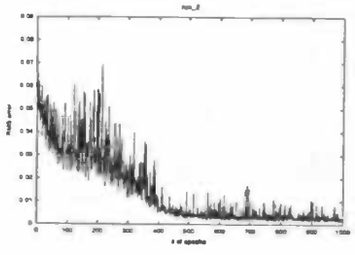
Sinus reference experiment
Learning rate = 0.3



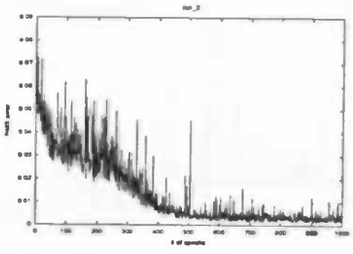
Experiment 1.3; Overall



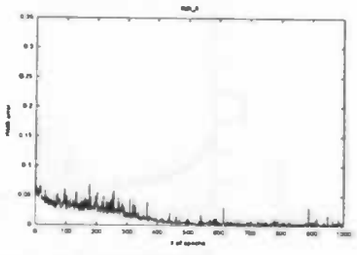
Experiment 1.3; Run 1



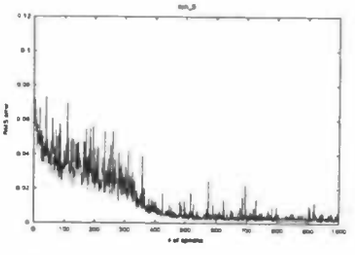
Experiment 1.3; Run 2



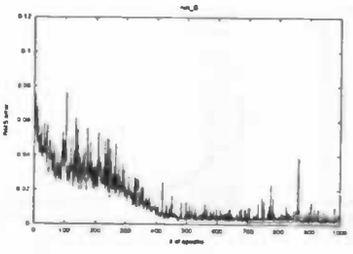
Experiment 1.3; Run 3



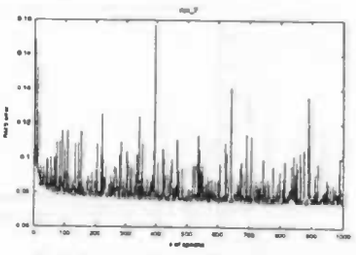
Experiment 1.3; Run 4



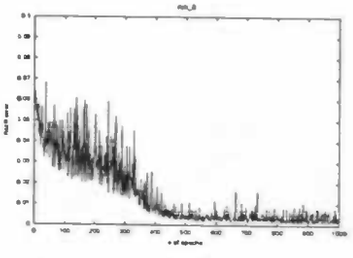
Experiment 1.3; Run 5



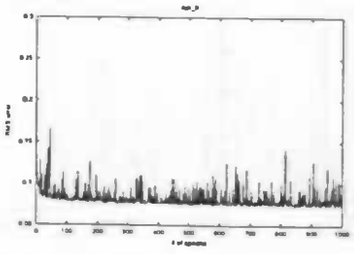
Experiment 1.3; Run 6



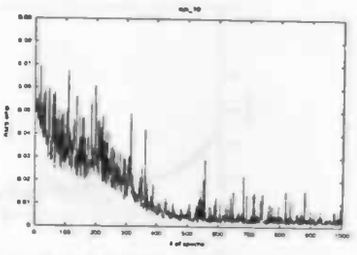
Experiment 1.3; Run 7



Experiment 1.3; Run 8



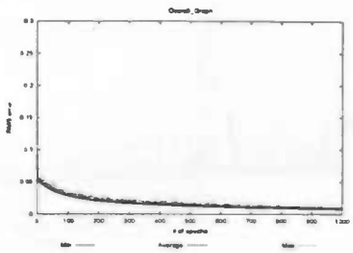
Experiment 1.3; Run 9



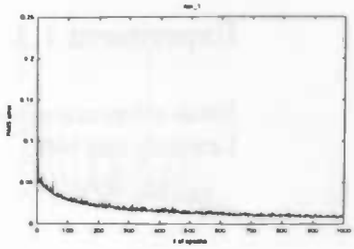
Experiment 1.3; Run 10

Experiment 1.4

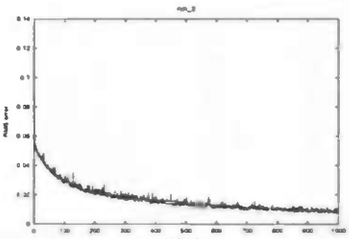
Sinus reference experiment
Learning rate = 0.1



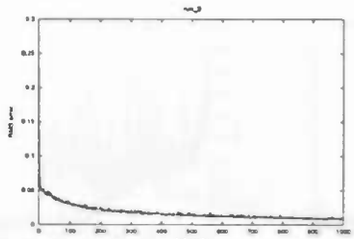
Experiment 1.4; Overall



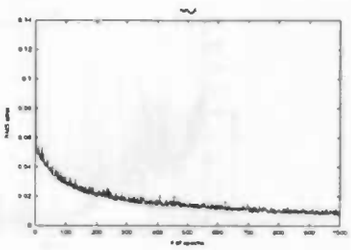
Experiment 1.4; Run 1



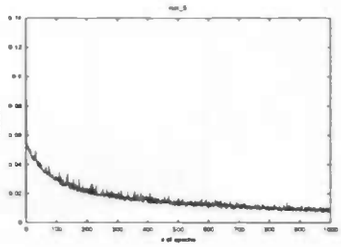
Experiment 1.4; Run 2



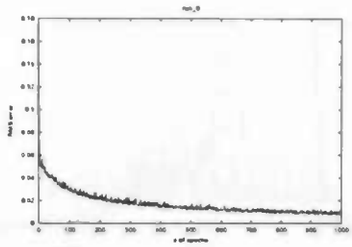
Experiment 1.4; Run 3



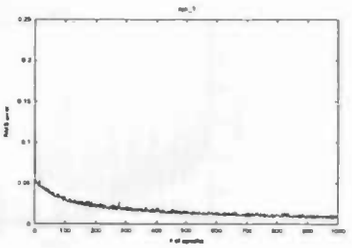
Experiment 1.4; Run 4



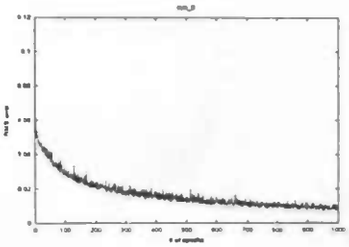
Experiment 1.4; Run 5



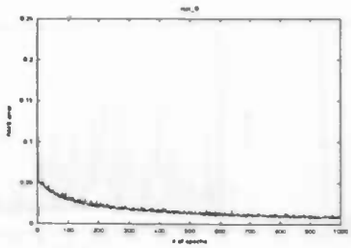
Experiment 1.4; Run 6



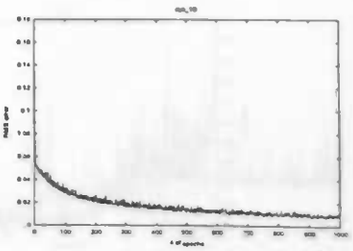
Experiment 1.4; Run 7



Experiment 1.4; Run 8



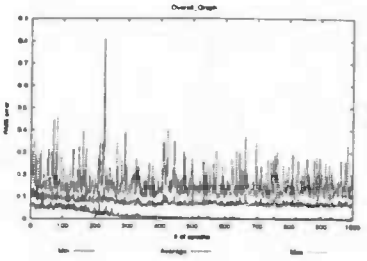
Experiment 1.4; Run 9



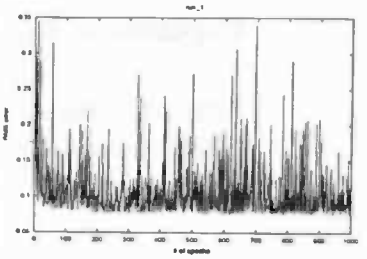
Experiment 1.4; Run 10

Experiment 1.5

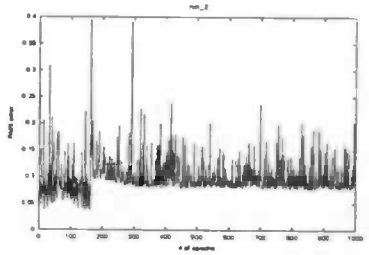
Sinus AL (random) experiment
Learning rate = 0.7



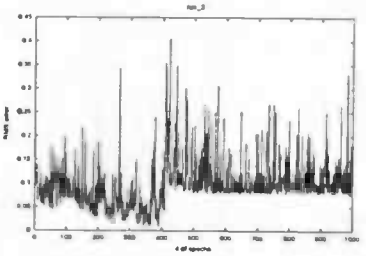
Experiment 1.5; Overall



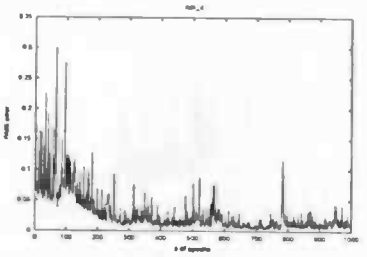
Experiment 1.5; Run 1



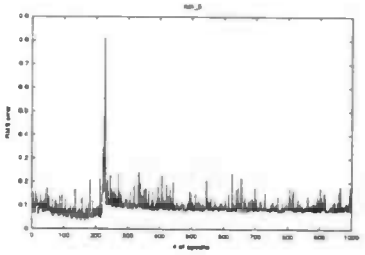
Experiment 1.5; Run 2



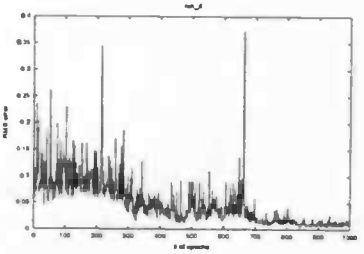
Experiment 1.5; Run 3



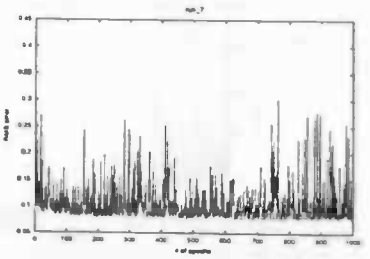
Experiment 1.5; Run 4



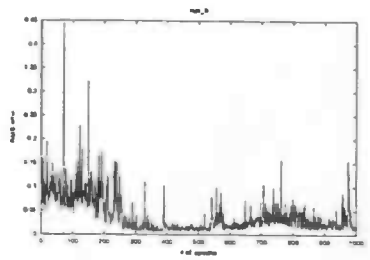
Experiment 1.5; Run 5



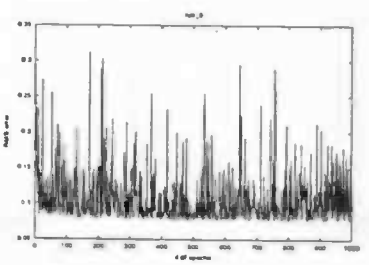
Experiment 1.5; Run 6



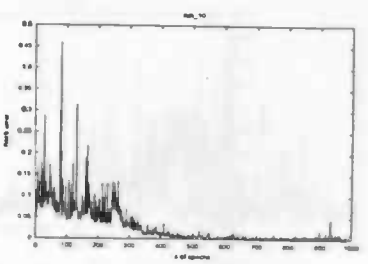
Experiment 1.5; Run 7



Experiment 1.5; Run 8



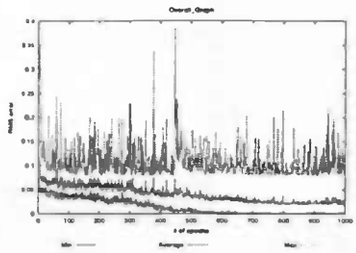
Experiment 1.5; Run 9



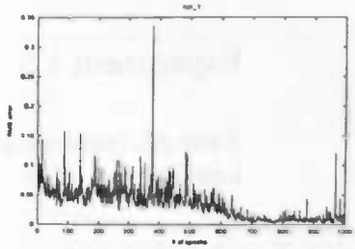
Experiment 1.5; Run 10

Experiment 1.6

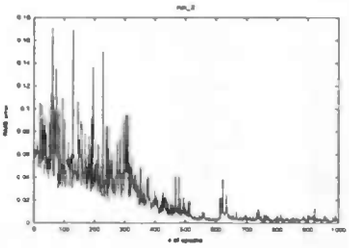
Sinus AL (random) experiment
Learning rate = 0.5



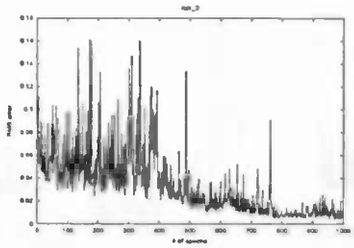
Experiment 1.6; Overall



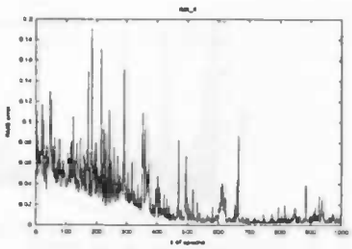
Experiment 1.6; Run 1



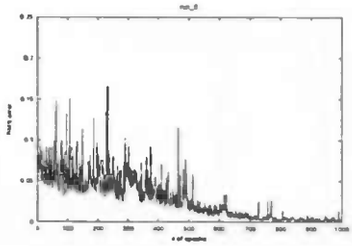
Experiment 1.6; Run 2



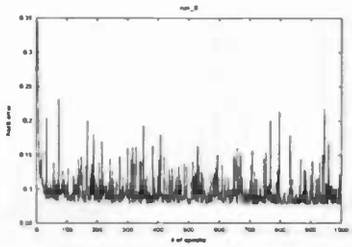
Experiment 1.6; Run 3



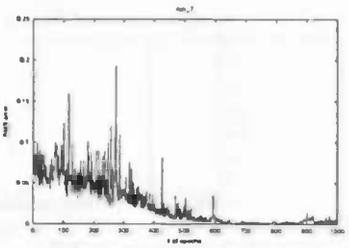
Experiment 1.6; Run 4



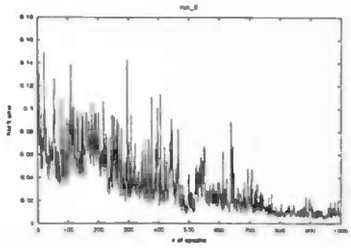
Experiment 1.6; Run 5



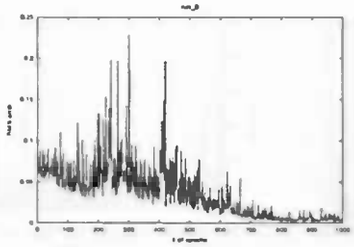
Experiment 1.6; Run 6



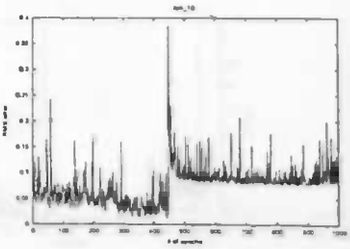
Experiment 1.6; Run 7



Experiment 1.6; Run 8



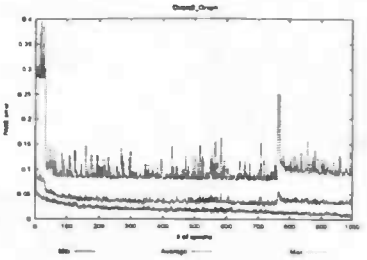
Experiment 1.6; Run 9



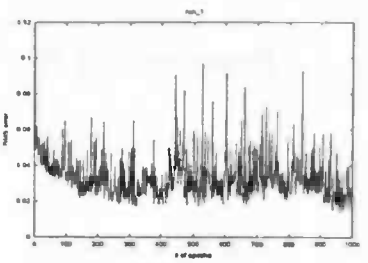
Experiment 1.6; Run 10

Experiment 1.7

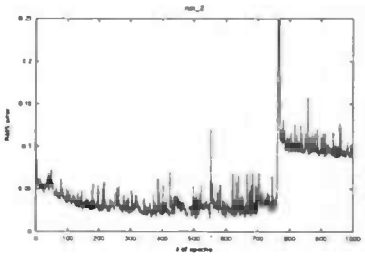
Sinus AL (random) experiment
Learning rate = 0.3



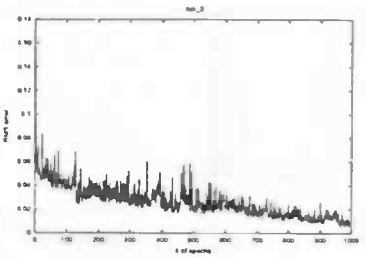
Experiment 1.7; Overall



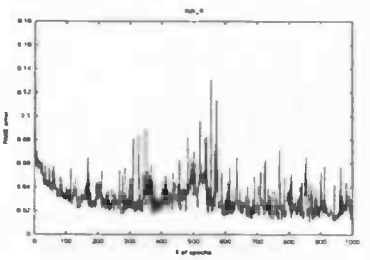
Experiment 1.7; Run 1



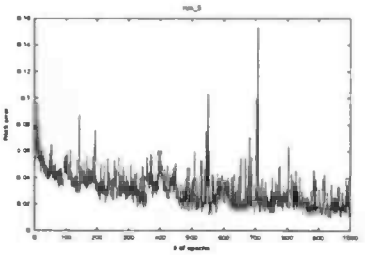
Experiment 1.7; Run 2



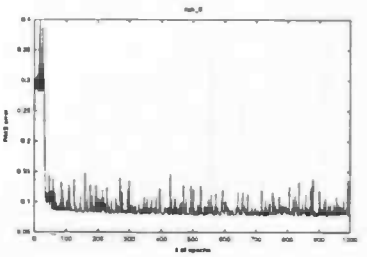
Experiment 1.7; Run 3



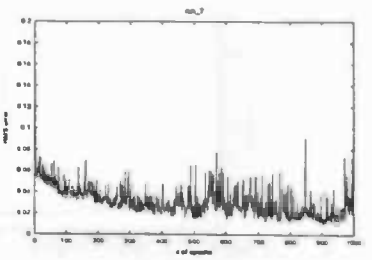
Experiment 1.7; Run 4



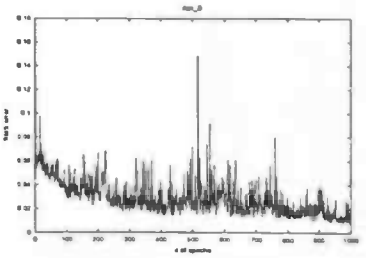
Experiment 1.7; Run 5



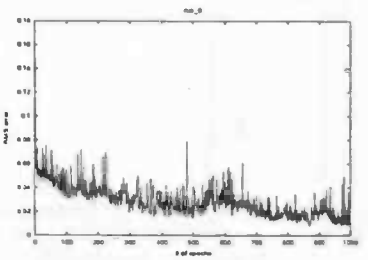
Experiment 1.7; Run 6



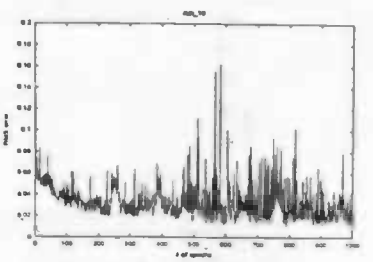
Experiment 1.7; Run 7



Experiment 1.7; Run 8



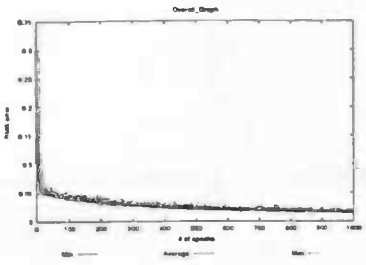
Experiment 1.7; Run 9



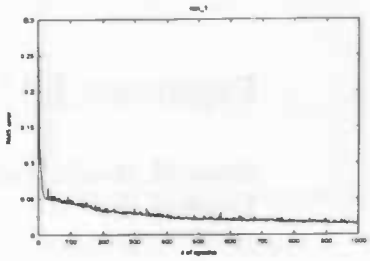
Experiment 1.7; Run 10

Experiment 1.8

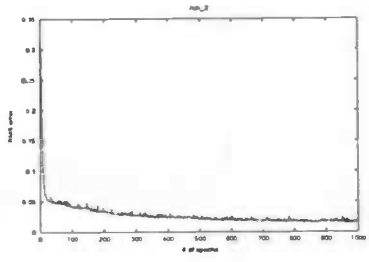
Sinus AL (random) experiment
Learning rate = 0.1



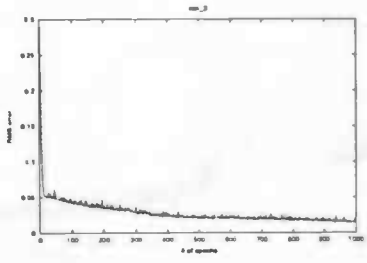
Experiment 1.8; Overall



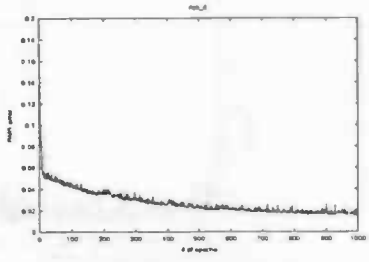
Experiment 1.8; Run 1



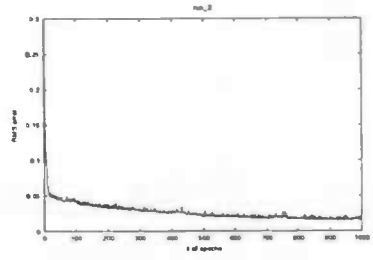
Experiment 1.8; Run 2



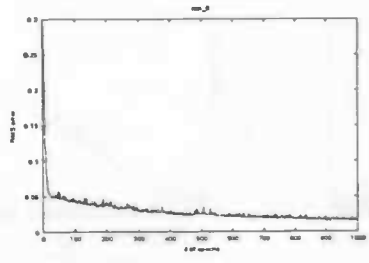
Experiment 1.8; Run 3



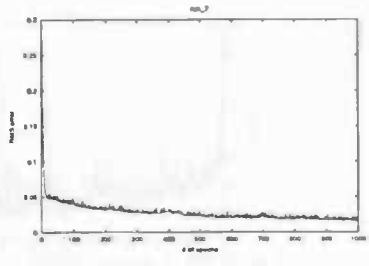
Experiment 1.8; Run 4



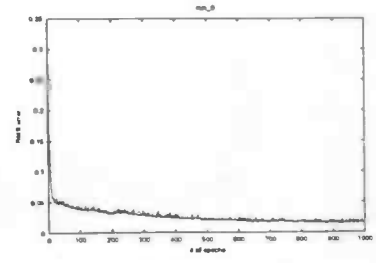
Experiment 1.8; Run 5



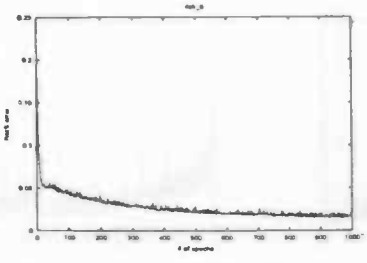
Experiment 1.8; Run 6



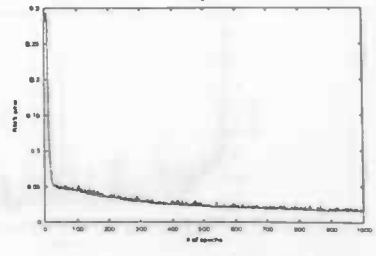
Experiment 1.8; Run 7



Experiment 1.8; Run 8



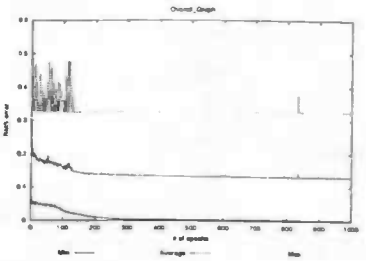
Experiment 1.8; Run



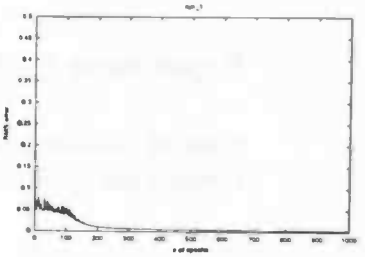
Experiment 1.8; Run 10

Experiment 1.9

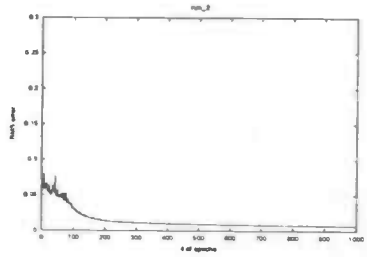
Sinus AL (maximum) experiment
Learning rate = 0.7



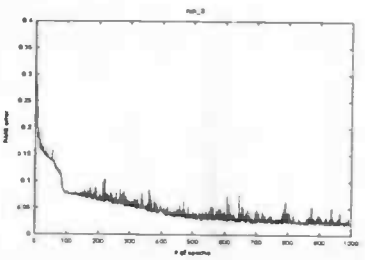
Experiment 1.9; Overall



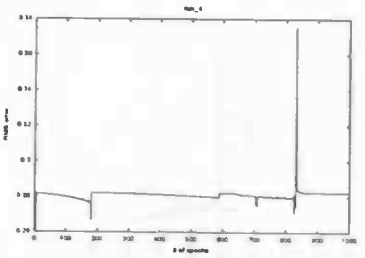
Experiment 1.9; Run 1



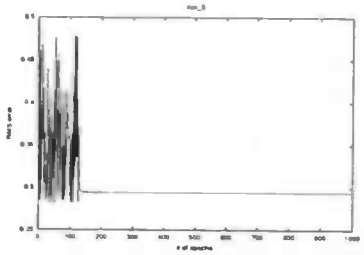
Experiment 1.9; Run 2



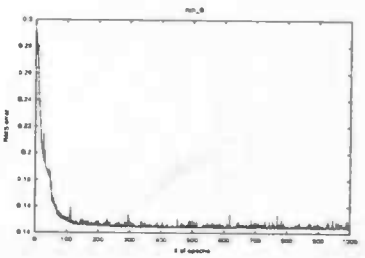
Experiment 1.9; Run 3



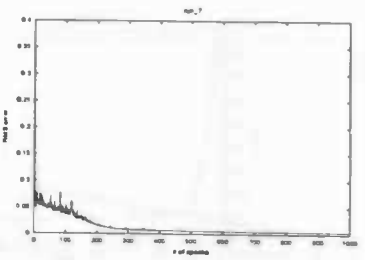
Experiment 1.9; Run 4



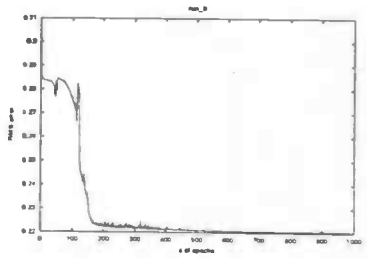
Experiment 1.9; Run 5



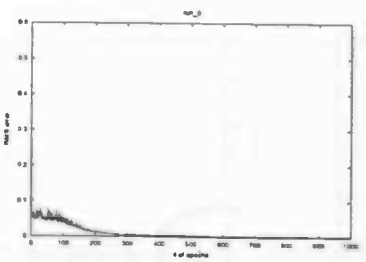
Experiment 1.9; Run 6



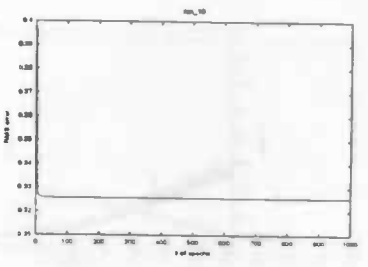
Experiment 1.9; Run 7



Experiment 1.9; Run 8



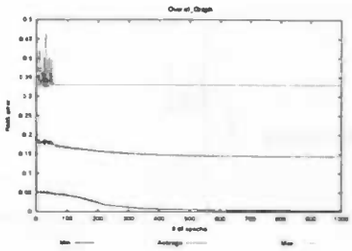
Experiment 1.9; Run 9



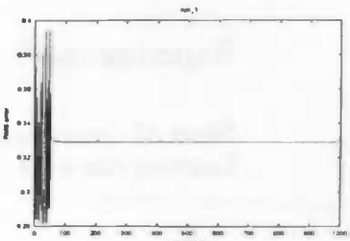
Experiment 1.9; Run 10

Experiment 1.10

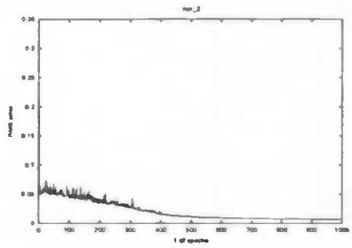
Sinus AL (maximum) experiment
Learning rate = 0.5



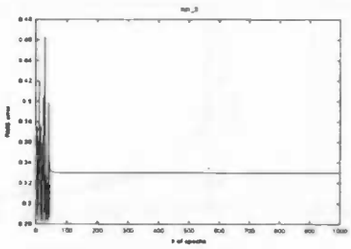
Experiment 1.10; Overall



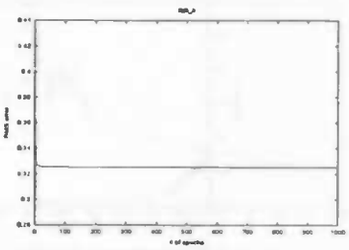
Experiment 1.10; Run 1



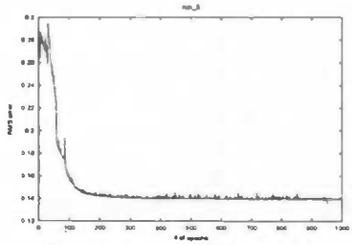
Experiment 1.10; Run 2



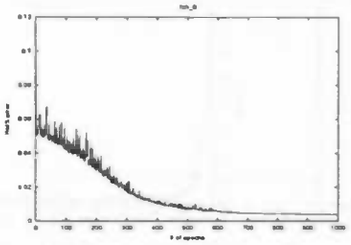
Experiment 1.10; Run 3



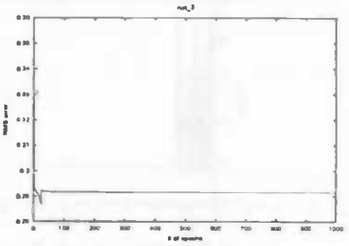
Experiment 1.10; Run 4



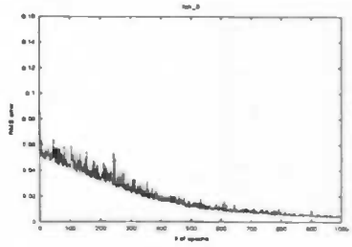
Experiment 1.10; Run 5



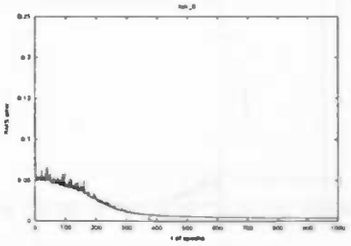
Experiment 1.10; Run 6



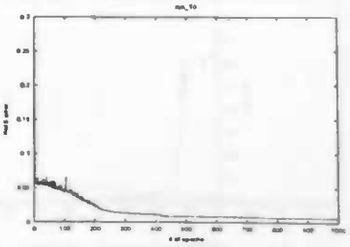
Experiment 1.10; Run 7



Experiment 1.10; Run 8



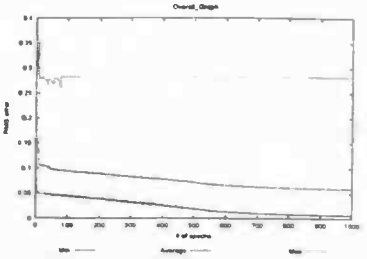
Experiment 1.10; Run 9



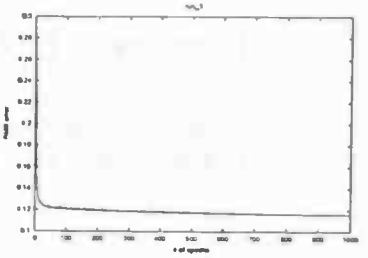
Experiment 1.10; Run 10

Experiment 1.11

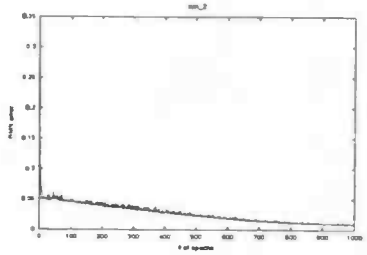
Sinus AL (maximum) experiment
Learning rate = 0.3



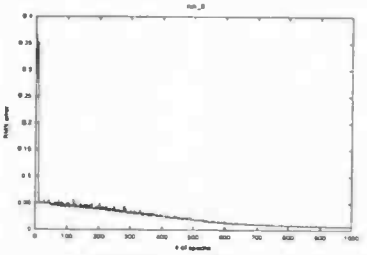
Experiment 1.11; Overall



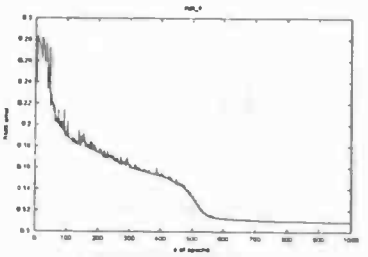
Experiment 1.11; Run 1



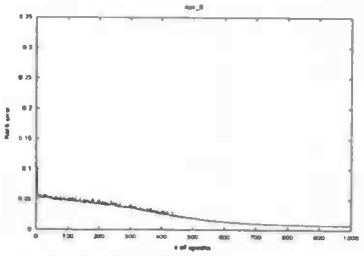
Experiment 1.11; Run 2



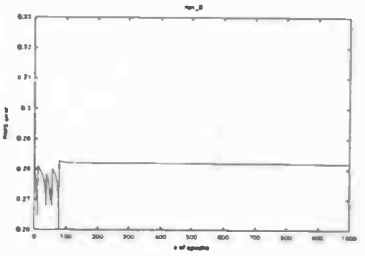
Experiment 1.11; Run 3



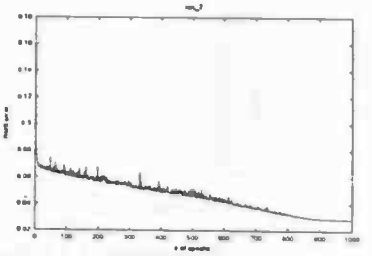
Experiment 1.11; Run 4



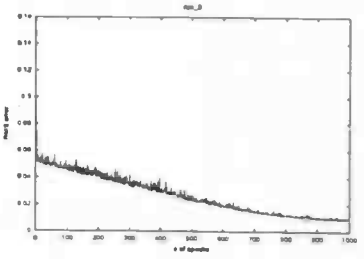
Experiment 1.11; Run 5



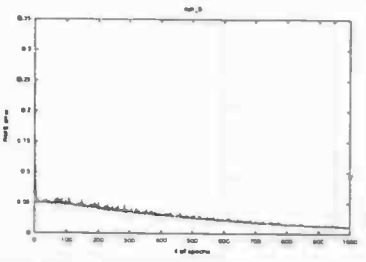
Experiment 1.11; Run 6



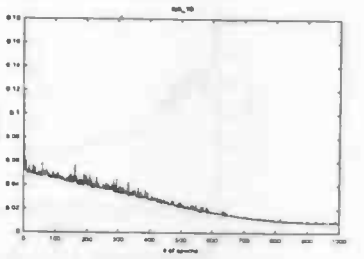
Experiment 1.11; Run 7



Experiment 1.11; Run 8



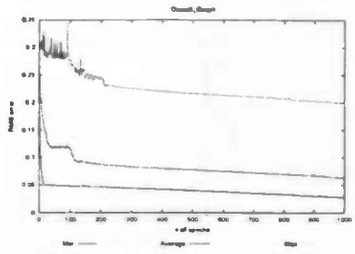
Experiment 1.11; Run 9



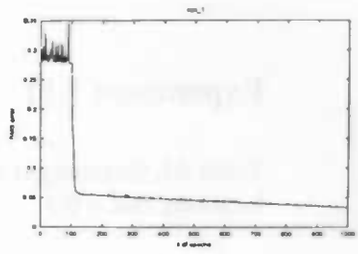
Experiment 1.11; Run 10

Experiment 1.12

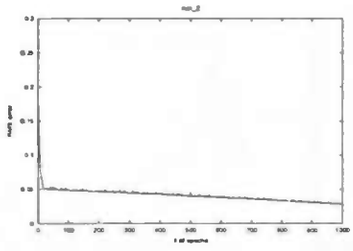
Sinus AL (maximum) experiment
Learning rate = 0.1



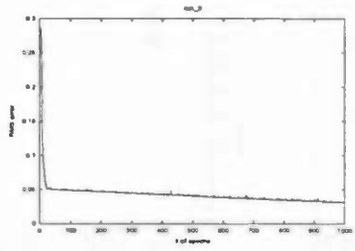
Experiment 1.12; Overall



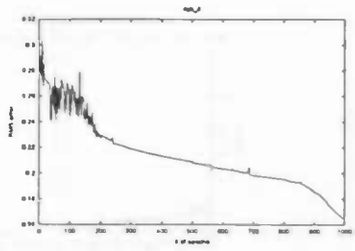
Experiment 1.12; Run 1



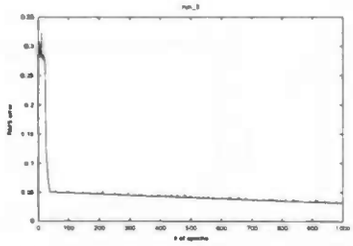
Experiment 1.12; Run 2



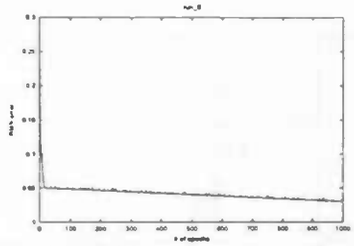
Experiment 1.12; Run 3



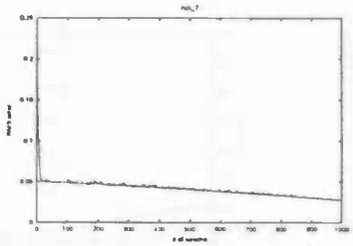
Experiment 1.12; Run 4



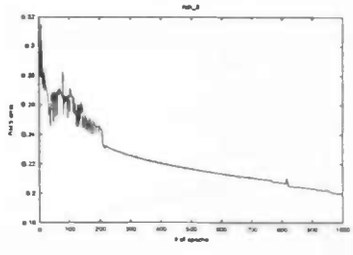
Experiment 1.12; Run 5



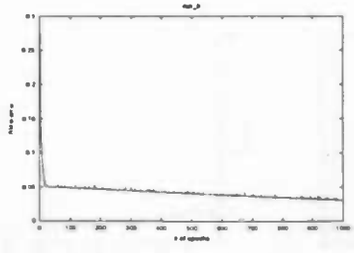
Experiment 1.12; Run 6



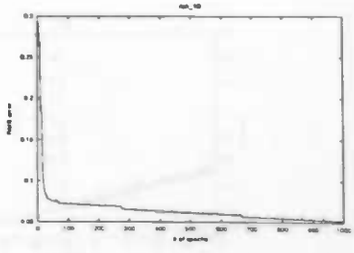
Experiment 1.12; Run 7



Experiment 1.12; Run 8



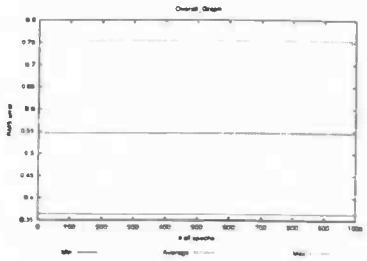
Experiment 1.12; Run 9



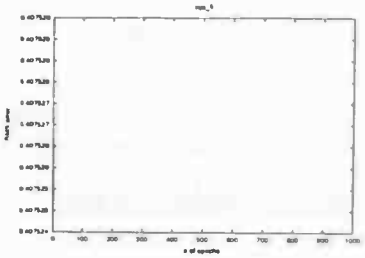
Experiment 1.12; Run 10

Experiment 1.13

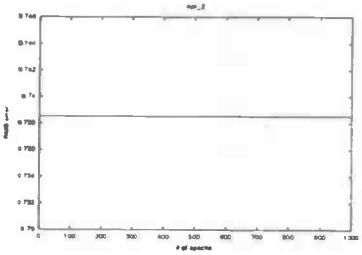
Sinus AL (minimum) experiment
Learning rate = 0.7



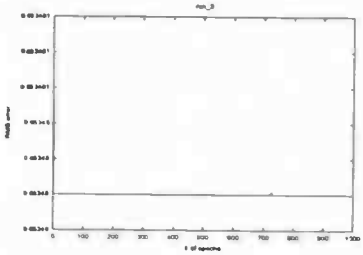
Experiment 1.13; Overall



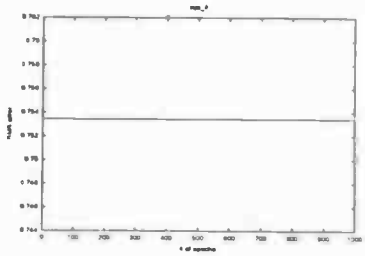
Experiment 1.13; Run 1



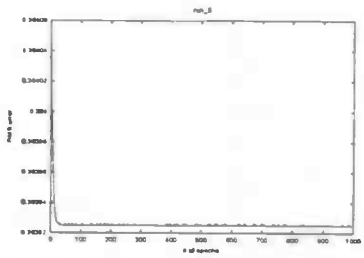
Experiment 1.13; Run 2



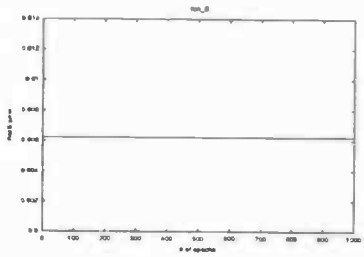
Experiment 1.13; Run 3



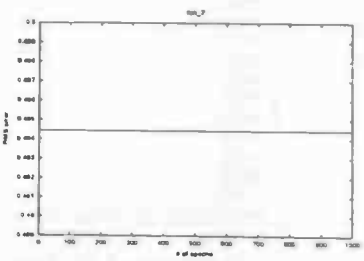
Experiment 1.13; Run 4



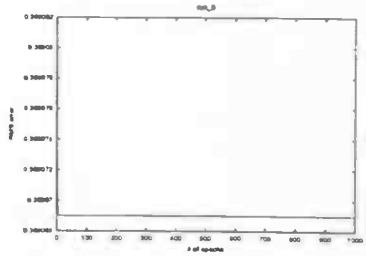
Experiment 1.13; Run 5



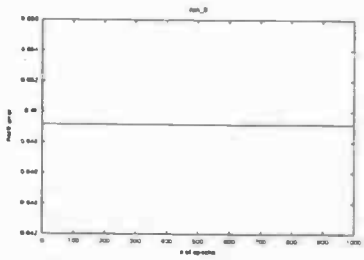
Experiment 1.13; Run 6



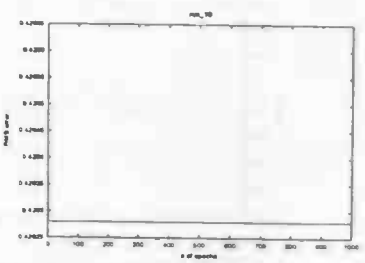
Experiment 1.13; Run 7



Experiment 1.13; Run 8



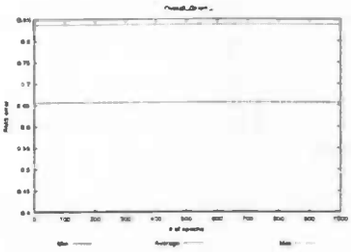
Experiment 1.13; Run 9



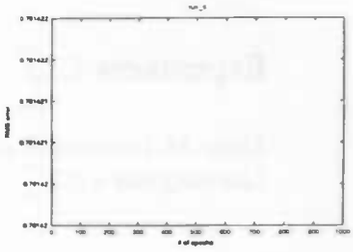
Experiment 1.13; Run 10

Experiment 1.14

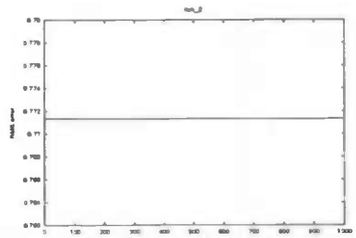
Sinus AL (minimum) experiment
Learning rate = 0.5



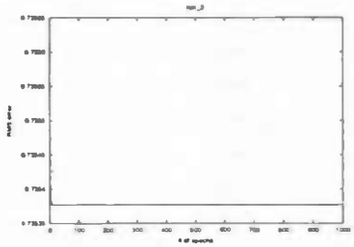
Experiment 1.14; Overall



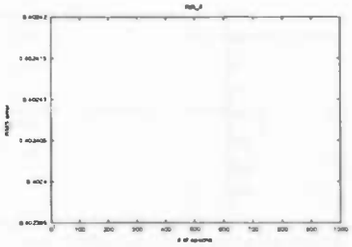
Experiment 1.14; Run 1



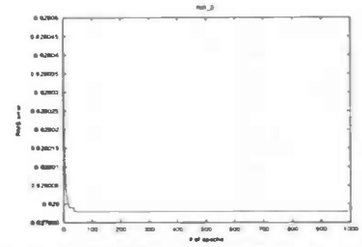
Experiment 1.14; Run 2



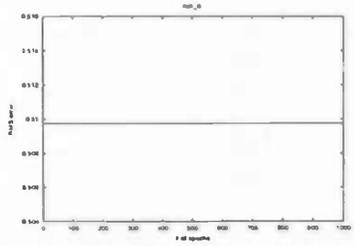
Experiment 1.14; Run 3



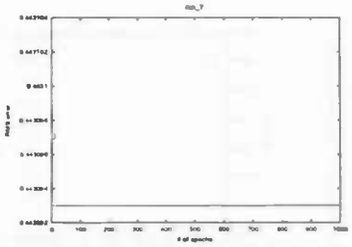
Experiment 1.14; Run 4



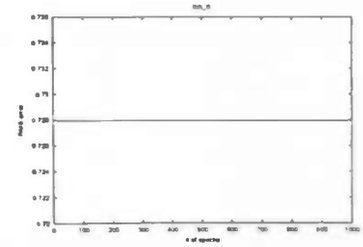
Experiment 1.14; Run 5



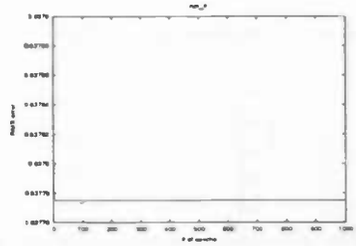
Experiment 1.14; Run 6



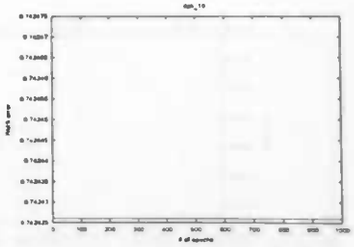
Experiment 1.14; Run 7



Experiment 1.14; Run 8



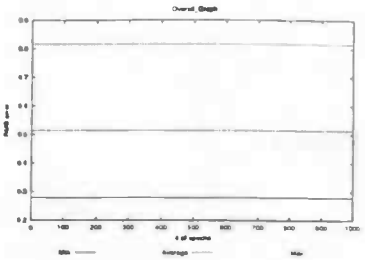
Experiment 1.14; Run 9



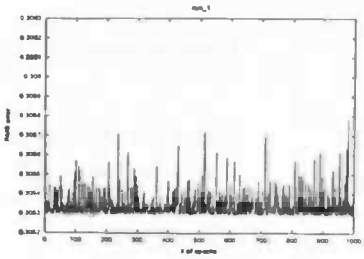
Experiment 1.14; Run 10

Experiment 1.15

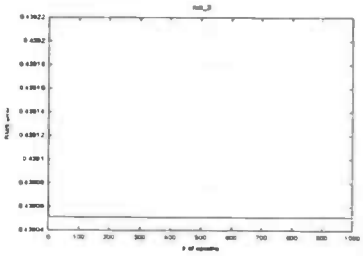
Sinus AL (minimum) experiment
Learning rate = 0.3



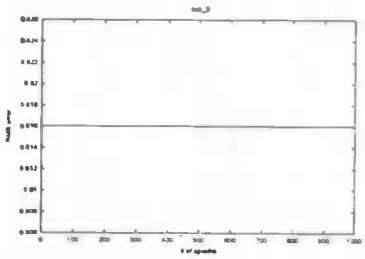
Experiment 1.15; Overall



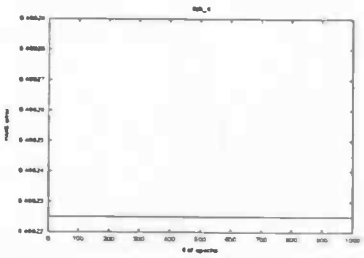
Experiment 1.15; Run 1



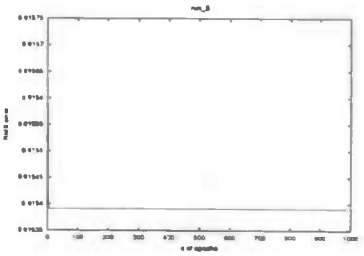
Experiment 1.15; Run 2



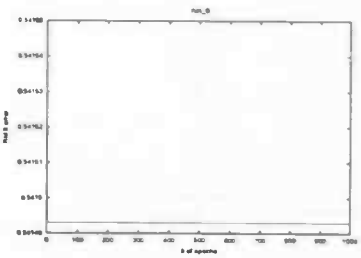
Experiment 1.15; Run 3



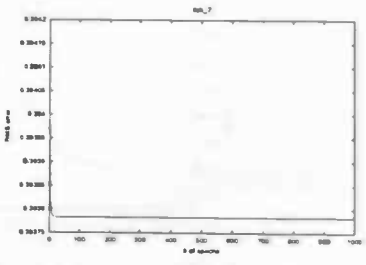
Experiment 1.15; Run 4



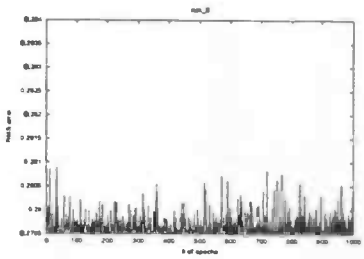
Experiment 1.15; Run 5



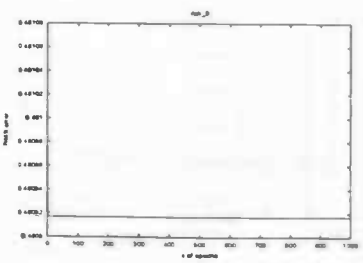
Experiment 1.15; Run 6



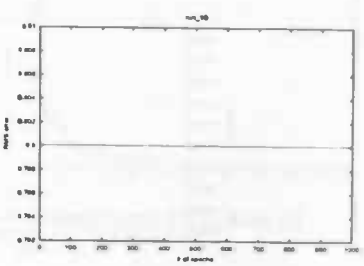
Experiment 1.15; Run 7



Experiment 1.15; Run 8



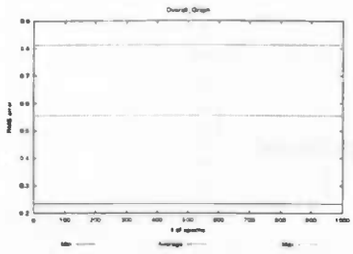
Experiment 1.15; Run 9



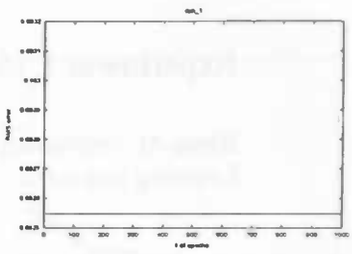
Experiment 1.15; Run 10

Experiment 1.16

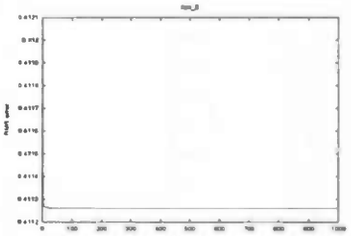
Sinus AL (minimum) experiment
Learning rate = 0.1



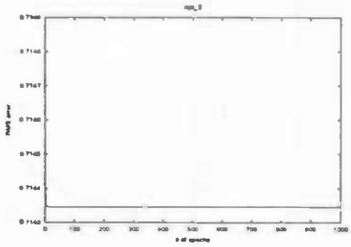
Experiment 1.16; Overall



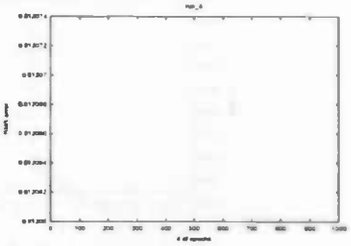
Experiment 1.16; Run 1



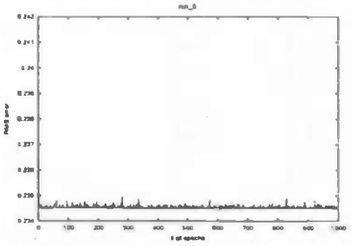
Experiment 1.16; Run 2



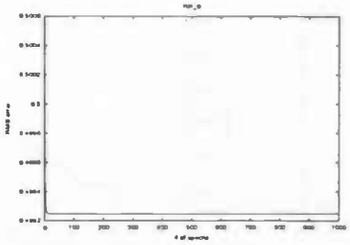
Experiment 1.16; Run 3



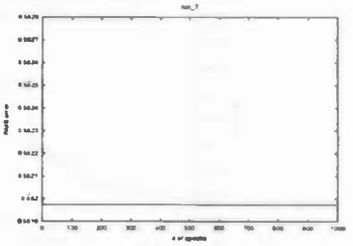
Experiment 1.16; Run 4



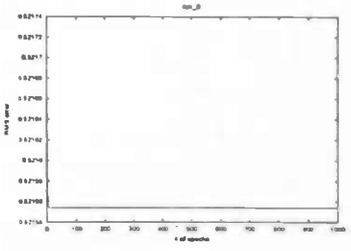
Experiment 1.16; Run 5



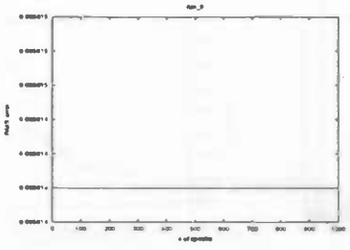
Experiment 1.16; Run 6



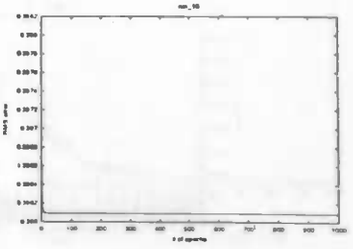
Experiment 1.16; Run 7



Experiment 1.16; Run 8



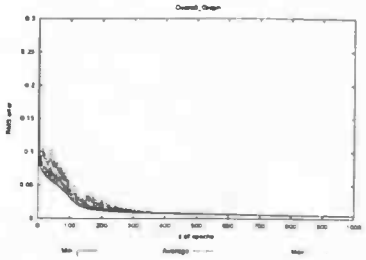
Experiment 1.16; Run 9



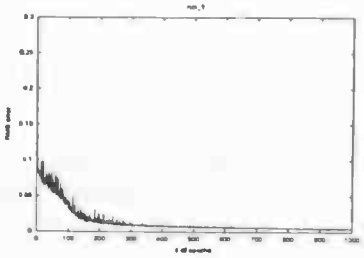
Experiment 1.16; Run 10

Experiment 1.17

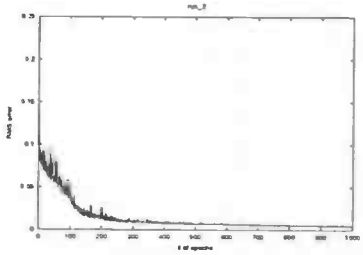
Additional sinus
reference experiment
Learning rate = 0.7



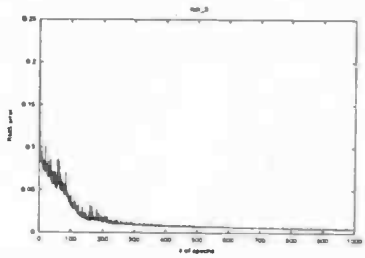
Experiment 1.17; Overall



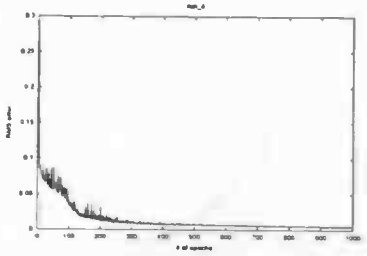
Experiment 1.17; Run 1



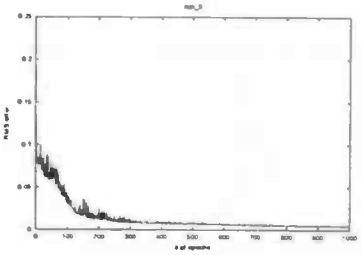
Experiment 1.17; Run 2



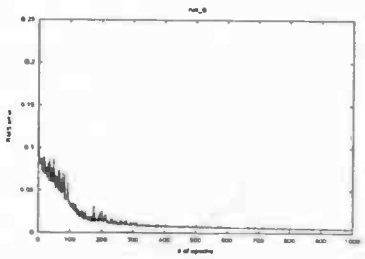
Experiment 1.17; Run 3



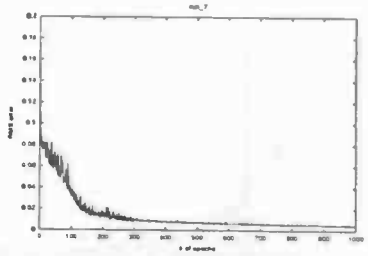
Experiment 1.17; Run 4



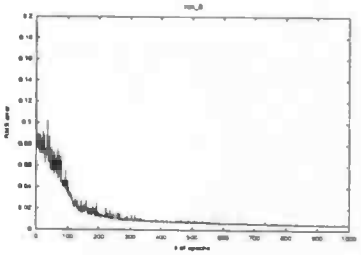
Experiment 1.17; Run 5



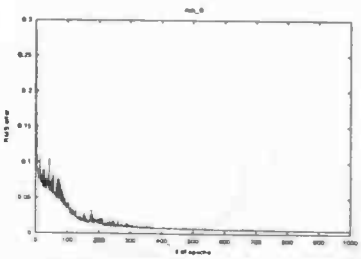
Experiment 1.17; Run 6



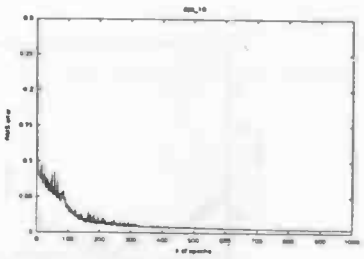
Experiment 1.17; Run 7



Experiment 1.17; Run 8



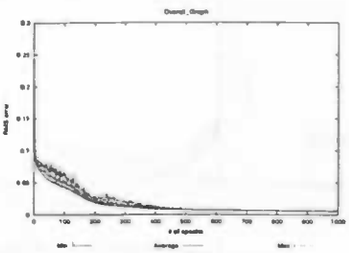
Experiment 1.17; Run 9



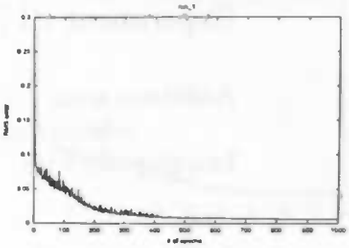
Experiment 1.17; Run 10

Experiment 1.18

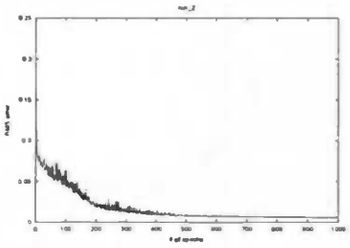
Additional sinus
reference experiment
Learning rate = 0.5



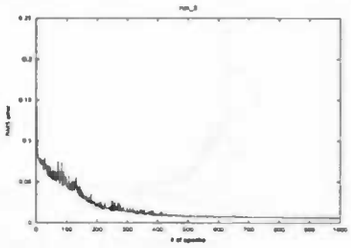
Experiment 1.18; Overall



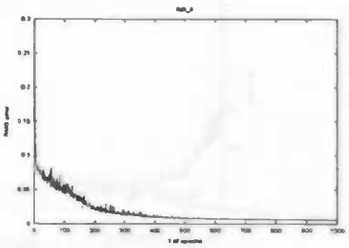
Experiment 1.18; Run 1



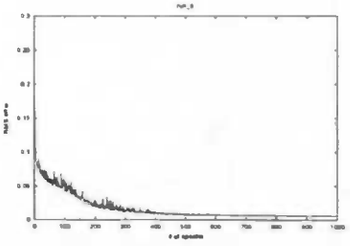
Experiment 1.18; Run 2



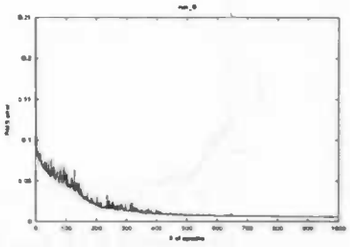
Experiment 1.18; Run 3



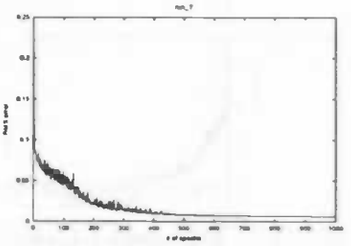
Experiment 1.18; Run 4



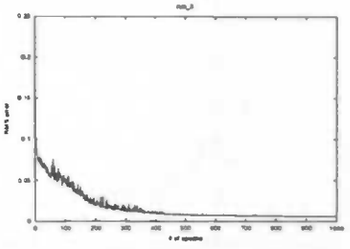
Experiment 1.18; Run 5



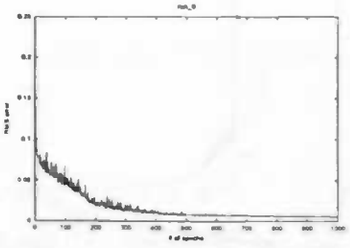
Experiment 1.18; Run 6



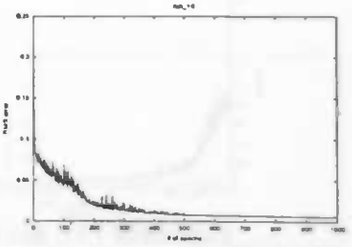
Experiment 1.18; Run 7



Experiment 1.18; Run 8



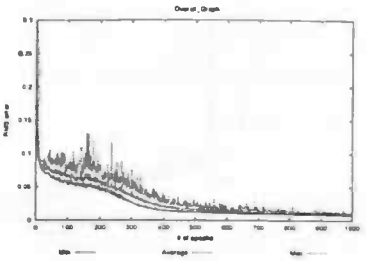
Experiment 1.18; Run 9



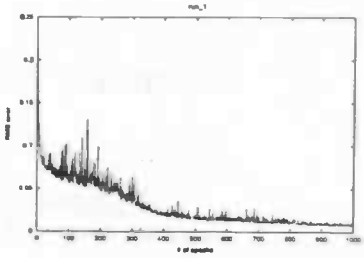
Experiment 1.18; Run 10

Experiment 1.19

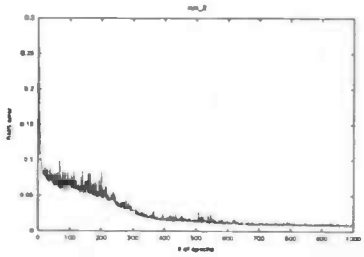
Additional sinus
AL (random) experiment
Learning rate = 0.7



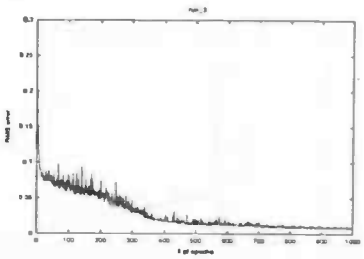
Experiment 1.19; Overall



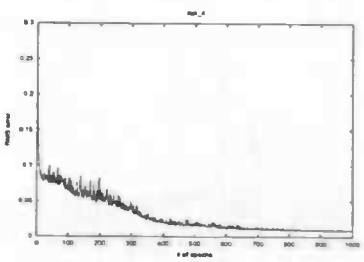
Experiment 1.19; Run 1



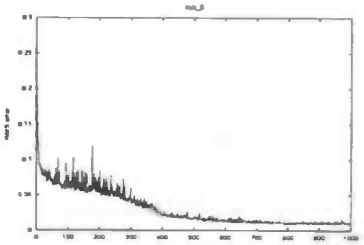
Experiment 1.19; Run 2



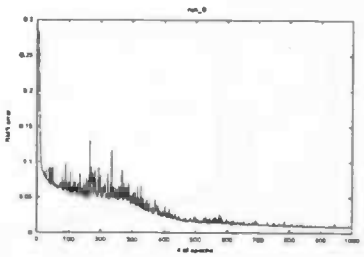
Experiment 1.19; Run 3



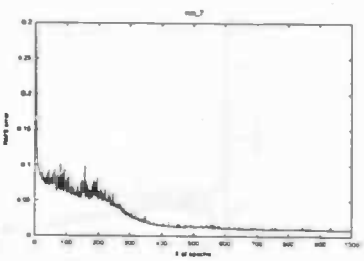
Experiment 1.19; Run 4



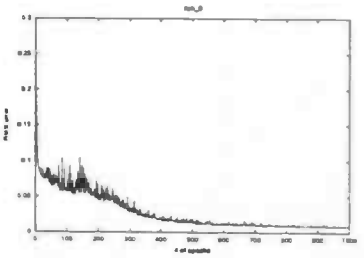
Experiment 1.19; Run 5



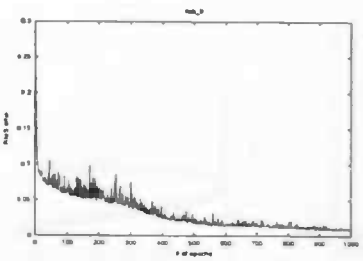
Experiment 1.19; Run 6



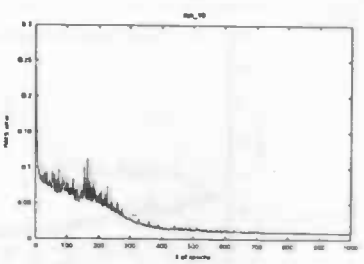
Experiment 1.19; Run 7



Experiment 1.19; Run 8



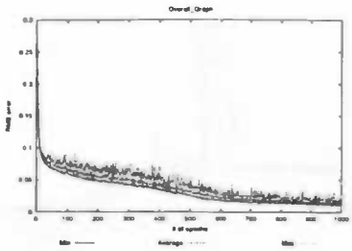
Experiment 1.19; Run 9



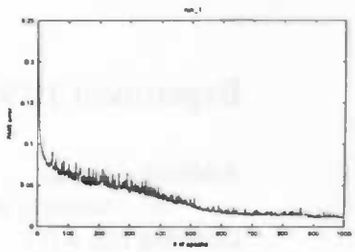
Experiment 1.19; Run 10

Experiment 1.20

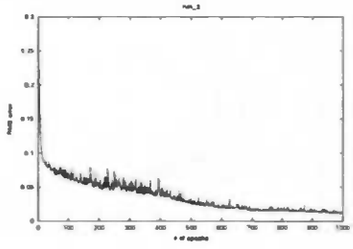
Additional sinus
AL (random) experiment
Learning rate = 0.5



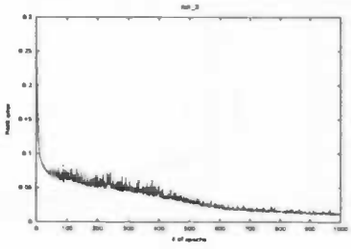
Experiment 1.20; Overall



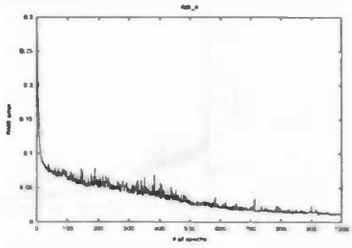
Experiment 1.20; Run 1



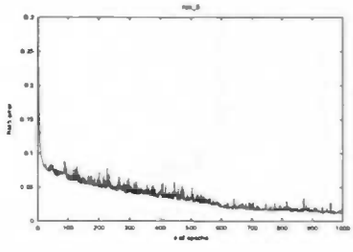
Experiment 1.20; Run 2



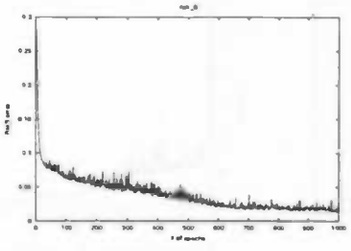
Experiment 1.20; Run 3



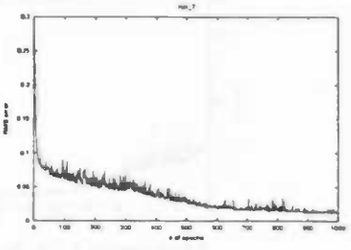
Experiment 1.20; Run 4



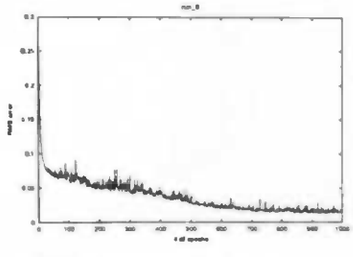
Experiment 1.20; Run 5



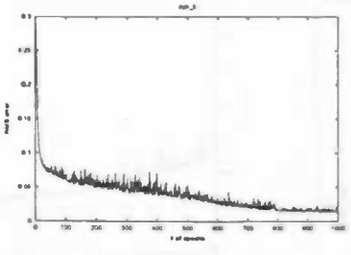
Experiment 1.20; Run 6



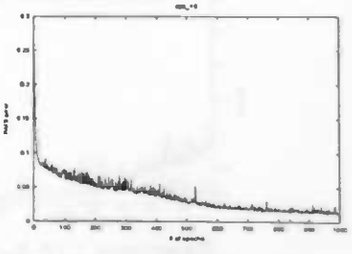
Experiment 1.20; Run 7



Experiment 1.20; Run 8



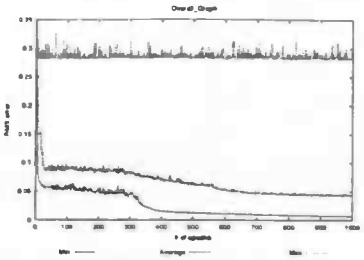
Experiment 1.20; Run 9



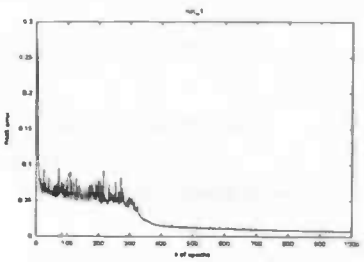
Experiment 1.20; Run 10

Experiment 1.21

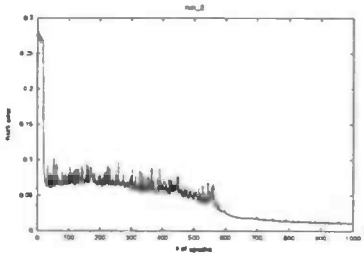
Additional sinus
AL (maximum) experiment
Learning rate = 0.7



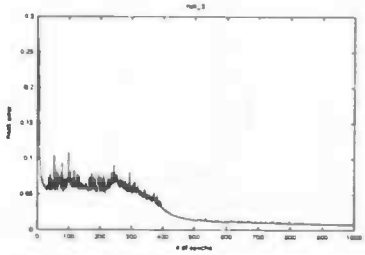
Experiment 1.21; Overall



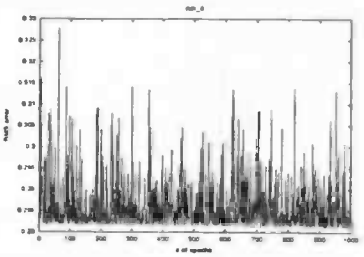
Experiment 1.21; Run 1



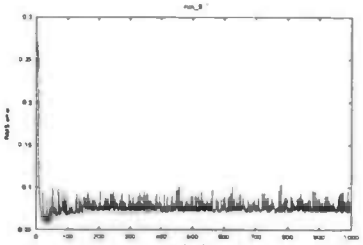
Experiment 1.21; Run 2



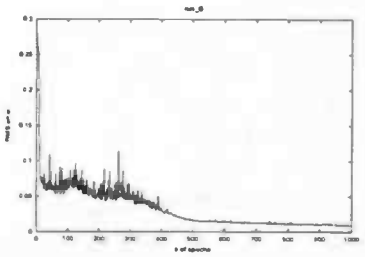
Experiment 1.21; Run 3



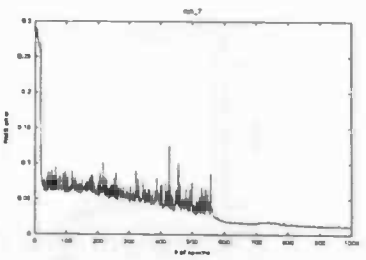
Experiment 1.21; Run 4



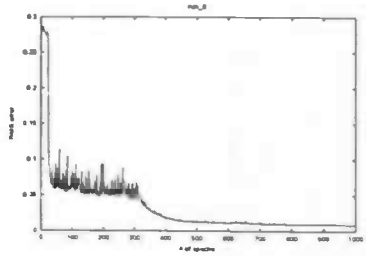
Experiment 1.21; Run 5



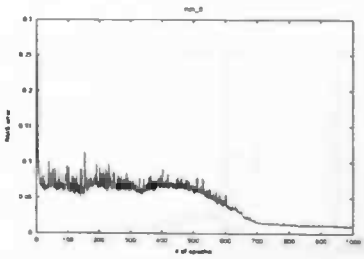
Experiment 1.21; Run 6



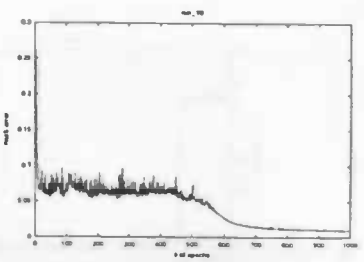
Experiment 1.21; Run 7



Experiment 1.21; Run 8



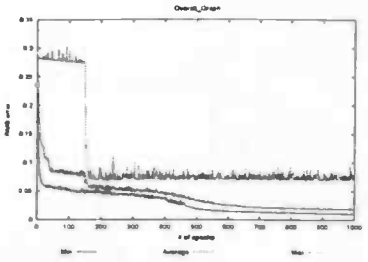
Experiment 1.21; Run 9



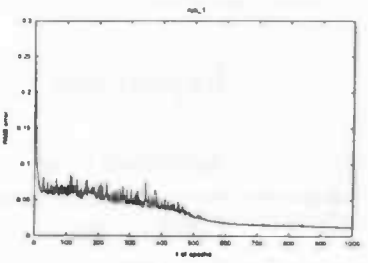
Experiment 1.21; Run 10

Experiment 1.22

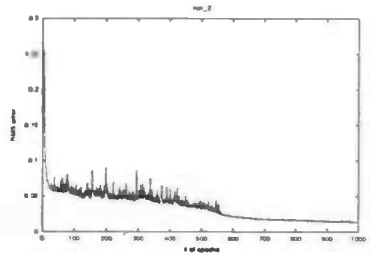
Additional sinus
AL (maximum) experiment
Learning rate = 0.5



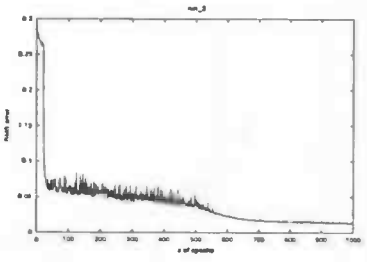
Experiment 1.22; Overall



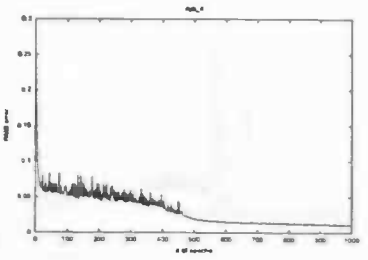
Experiment 1.22; Run 1



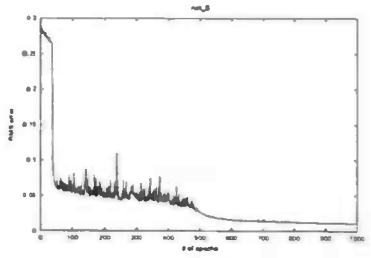
Experiment 1.22; Run 2



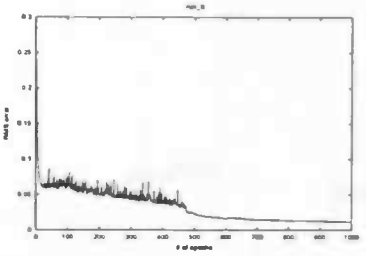
Experiment 1.22; Run 3



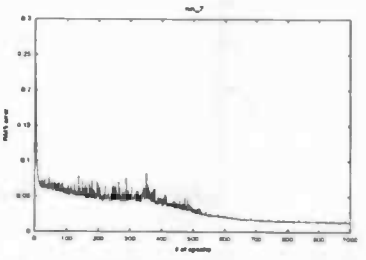
Experiment 1.22; Run 4



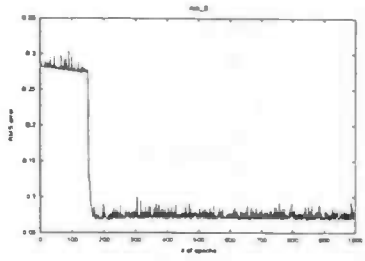
Experiment 1.22; Run 5



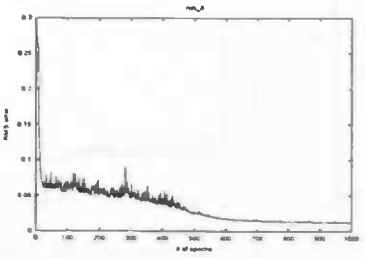
Experiment 1.22; Run 6



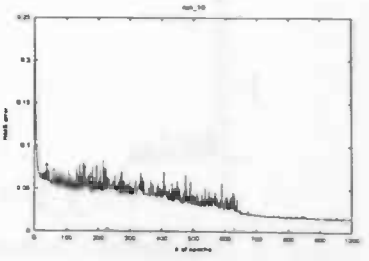
Experiment 1.22; Run 7



Experiment 1.22; Run 8



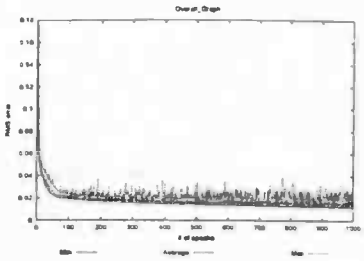
Experiment 1.22; Run 9



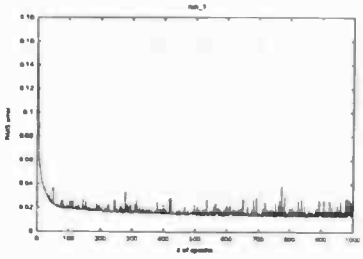
Experiment 1.22; Run 10

Experiment 2.1

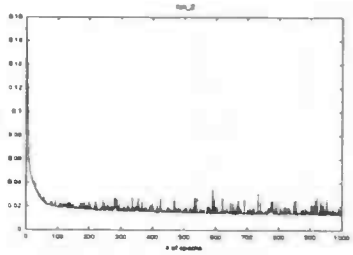
Natural exponent
reference experiment
Learning rate = 0.7



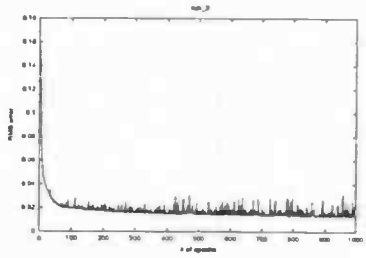
Experiment 2.1; Overall



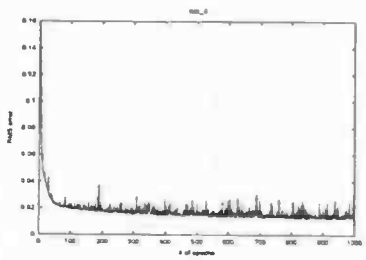
Experiment 2.1; Run 1



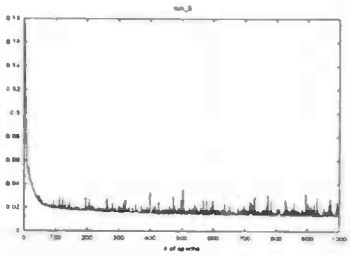
Experiment 2.1; Run 2



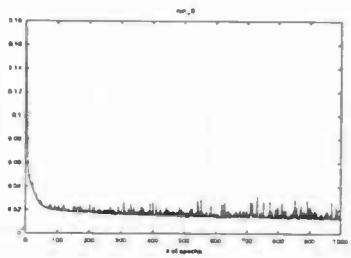
Experiment 2.1; Run 3



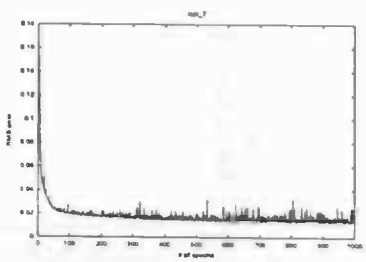
Experiment 2.1; Run 4



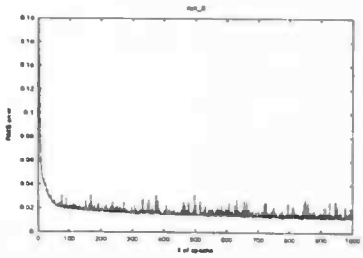
Experiment 2.1; Run 5



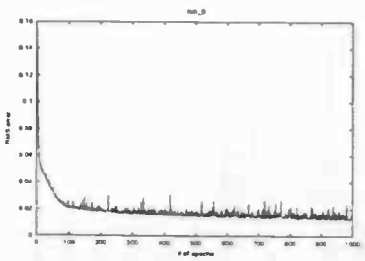
Experiment 2.1; Run 6



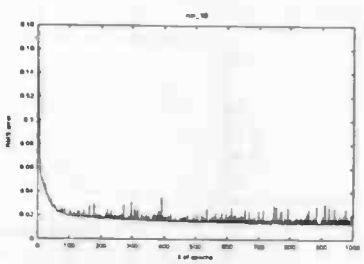
Experiment 2.1; Run 7



Experiment 2.1; Run 8



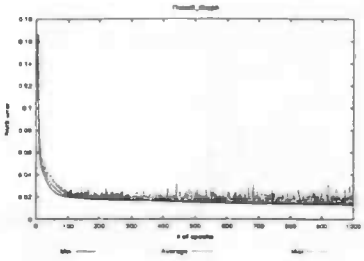
Experiment 2.1; Run 9



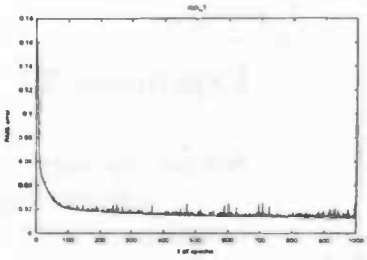
Experiment 2.1; Run 10

Experiment 2.2

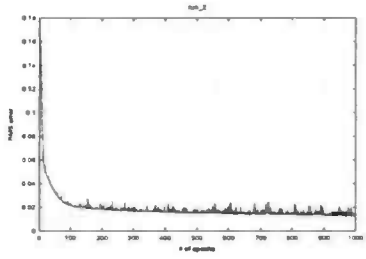
Natural exponent
reference experiment
Learning rate = 0.5



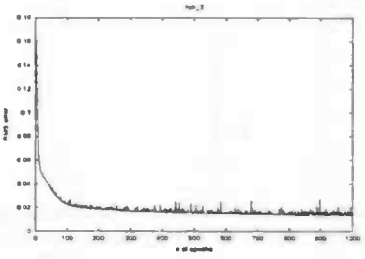
Experiment 2.2; Overall



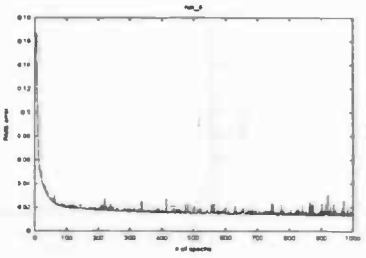
Experiment 2.2; Run 1



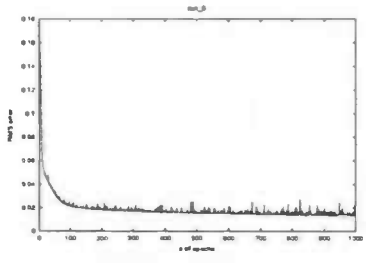
Experiment 2.2; Run 2



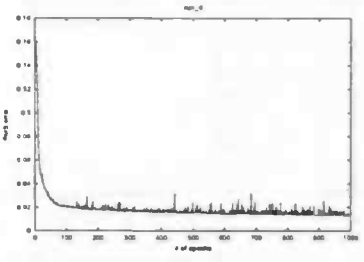
Experiment 2.2; Run 3



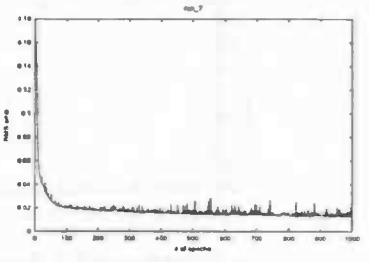
Experiment 2.2; Run 4



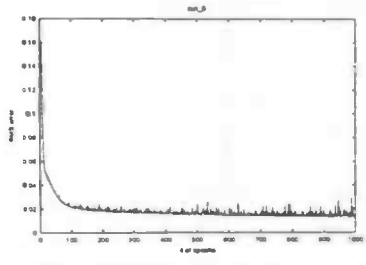
Experiment 2.2; Run 5



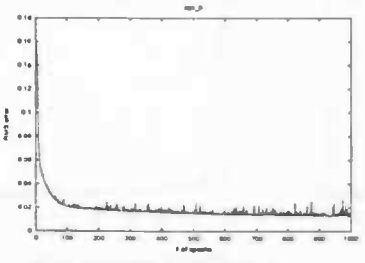
Experiment 2.2; Run 6



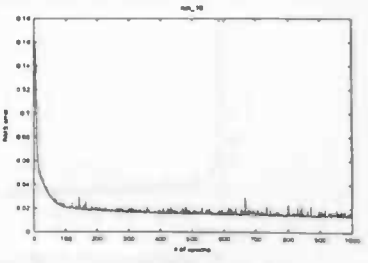
Experiment 2.2; Run 7



Experiment 2.2; Run 8



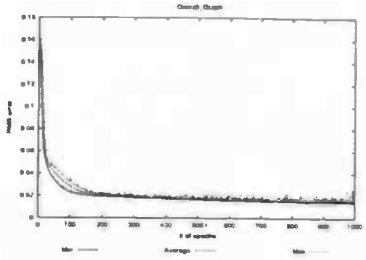
Experiment 2.2; Run 9



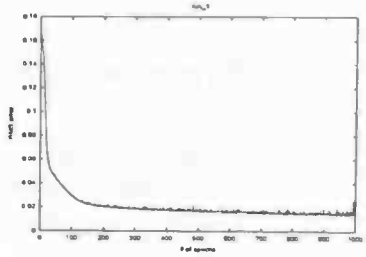
Experiment 2.2; Run 10

Experiment 2.3

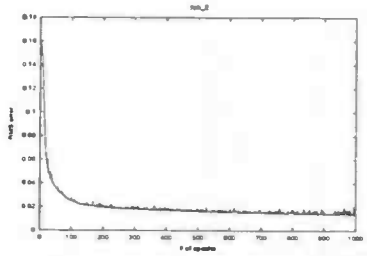
Natural exponent
reference experiment
Learning rate = 0.3



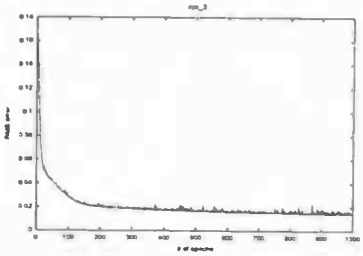
Experiment 2.3; Overall



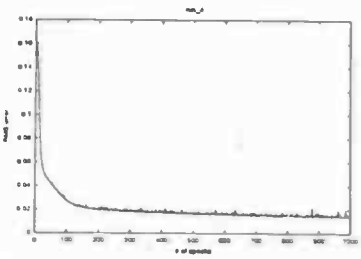
Experiment 2.3; Run 1



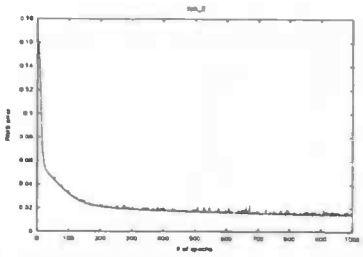
Experiment 2.3; Run 2



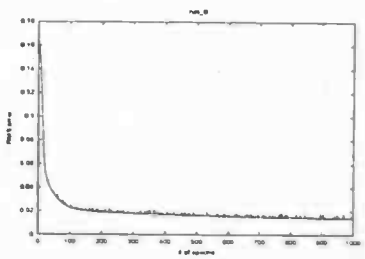
Experiment 2.3; Run 3



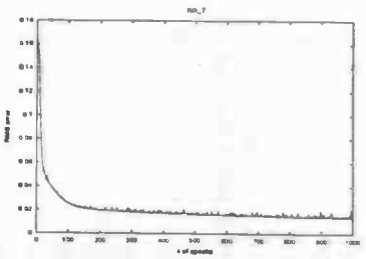
Experiment 2.3; Run 4



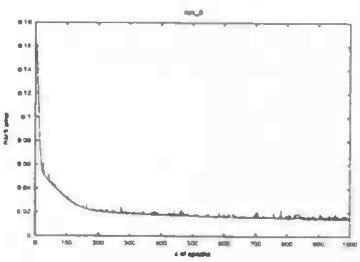
Experiment 2.3; Run 5



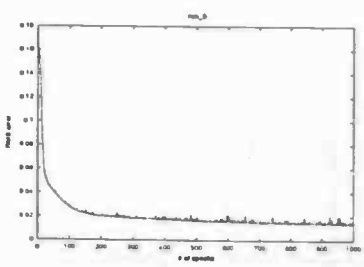
Experiment 2.3; Run 6



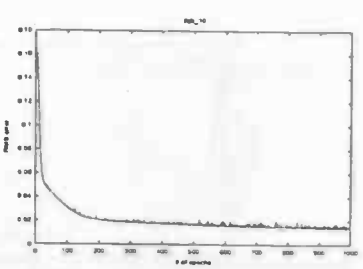
Experiment 2.3; Run 7



Experiment 2.3; Run 8



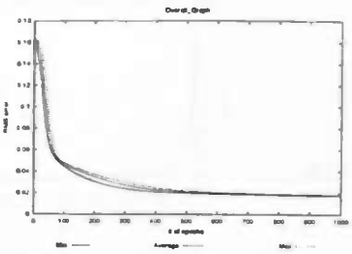
Experiment 2.3; Run 9



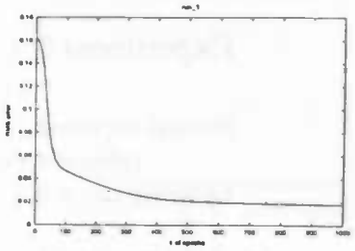
Experiment 2.3; Run 10

Experiment 2.4

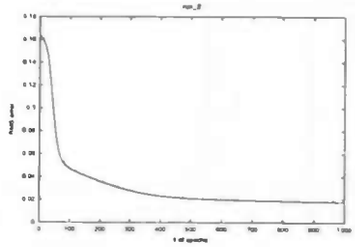
Natural exponent
reference experiment
Learning rate = 0.1



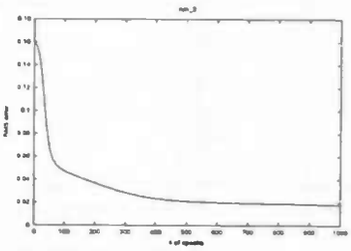
Experiment 2.4; Overall



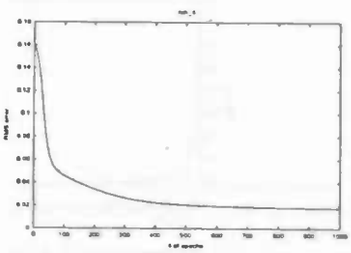
Experiment 2.4; Run 1



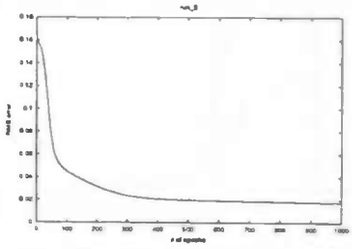
Experiment 2.4; Run 2



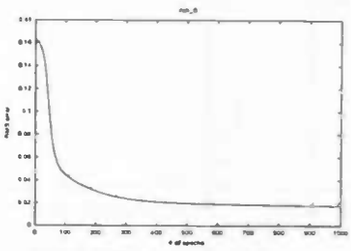
Experiment 2.4; Run 3



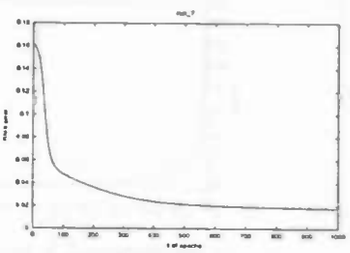
Experiment 2.4; Run 4



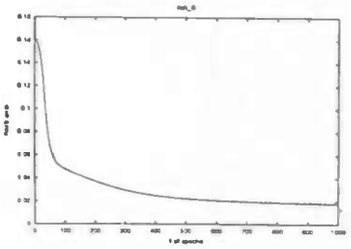
Experiment 2.4; Run 5



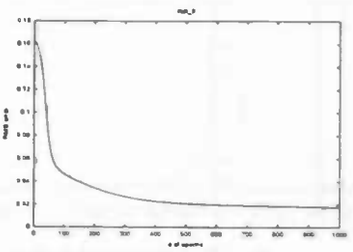
Experiment 2.4; Run 6



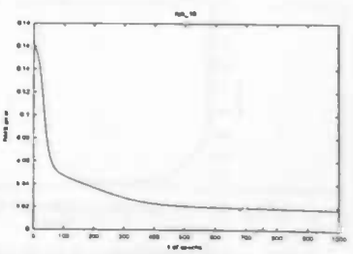
Experiment 2.4; Run 7



Experiment 2.4; Run 8



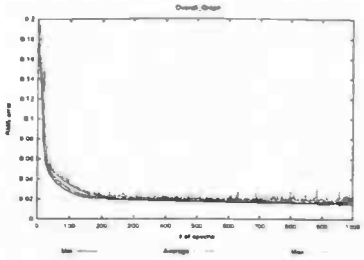
Experiment 2.4; Run 9



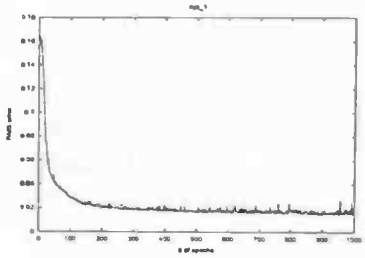
Experiment 2.4; Run 10

Experiment 2.5

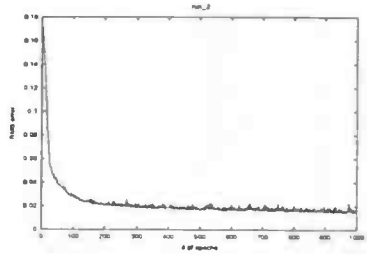
Natural exponent
AL (random) experiment
Learning rate = 0.7



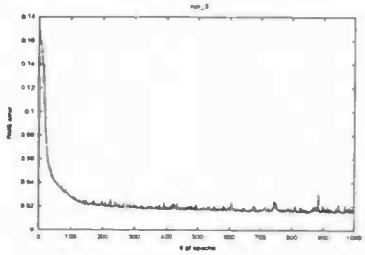
Experiment 2.5; Overall



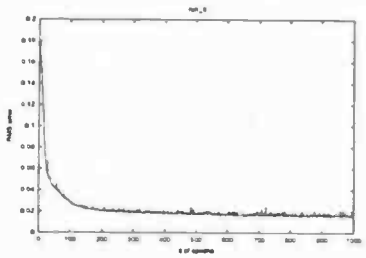
Experiment 2.5; Run 1



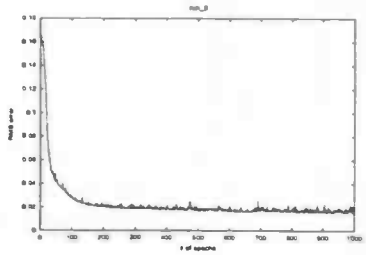
Experiment 2.5; Run 2



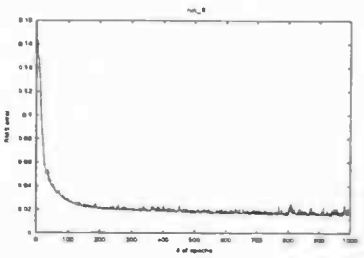
Experiment 2.5; Run 3



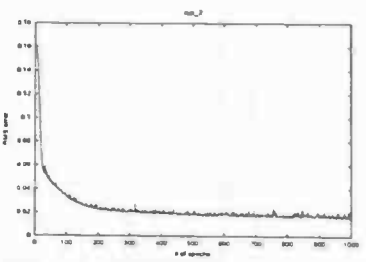
Experiment 2.5; Run 4



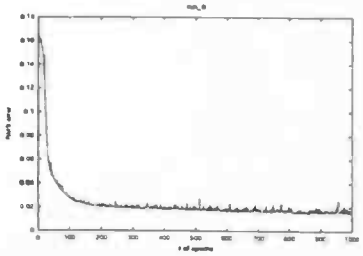
Experiment 2.5; Run 5



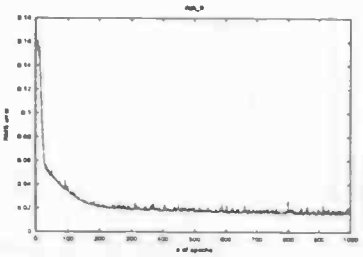
Experiment 2.5; Run 6



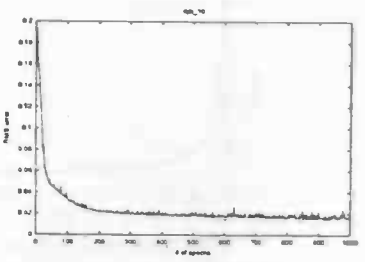
Experiment 2.5; Run 7



Experiment 2.5; Run 8



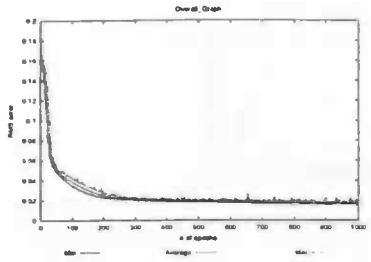
Experiment 2.5; Run 9



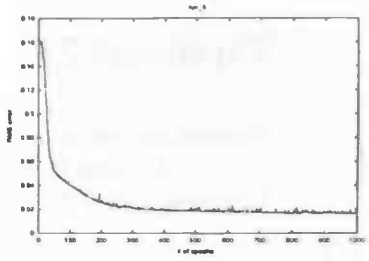
Experiment 2.5; Run 10

Experiment 2.6

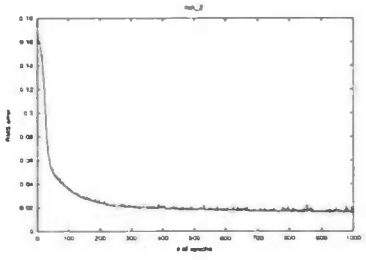
Natural exponent
AL (random) experiment
Learning rate = 0.5



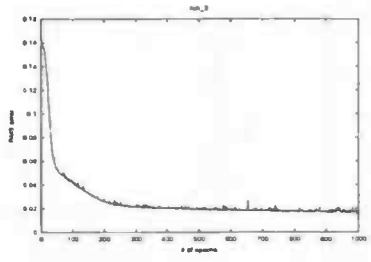
Experiment 2.6; Overall



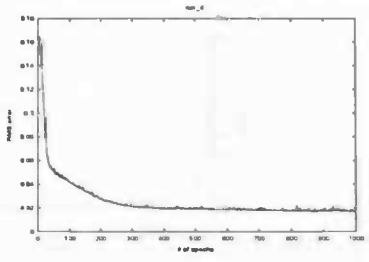
Experiment 2.6; Run 1



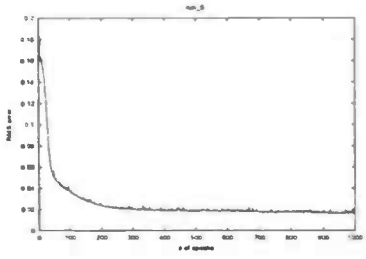
Experiment 2.6; Run 2



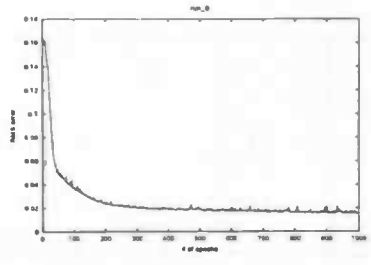
Experiment 2.6; Run 3



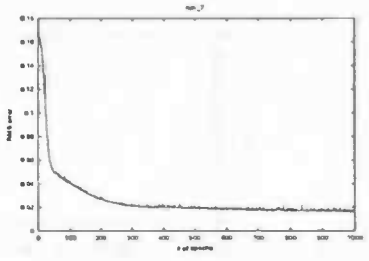
Experiment 2.6; Run 4



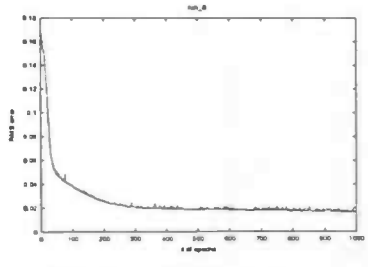
Experiment 2.6; Run 5



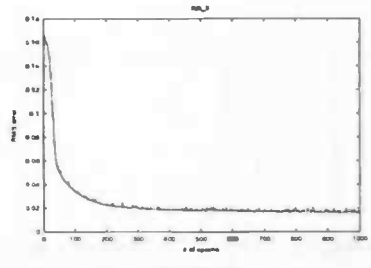
Experiment 2.6; Run 6



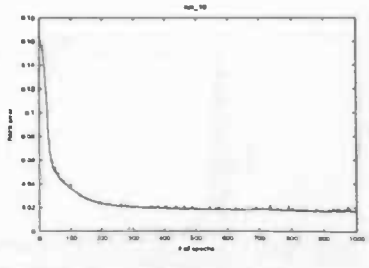
Experiment 2.6; Run 7



Experiment 2.6; Run 8



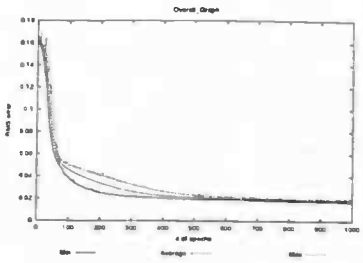
Experiment 2.6; Run 9



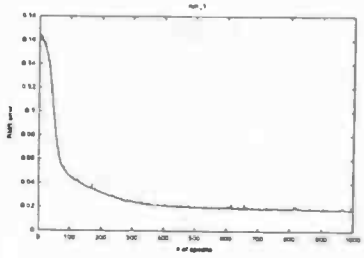
Experiment 2.6; Run 10

Experiment 2.7

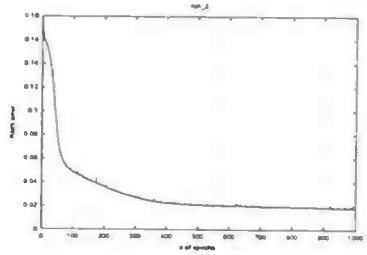
Natural exponent
AL (random) experiment
Learning rate = 0.3



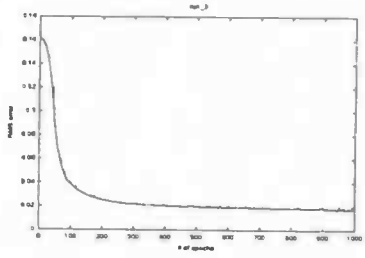
Experiment 2.7; Overall



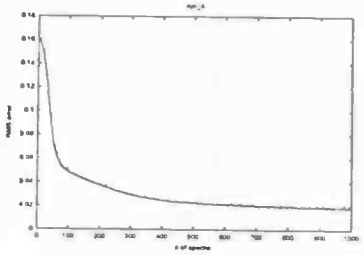
Experiment 2.7; Run 1



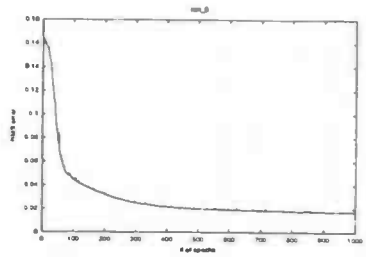
Experiment 2.7; Run 2



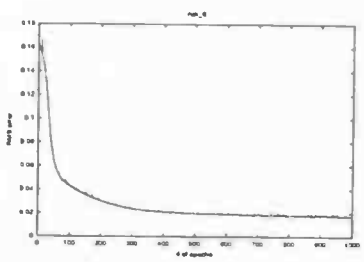
Experiment 2.7; Run 3



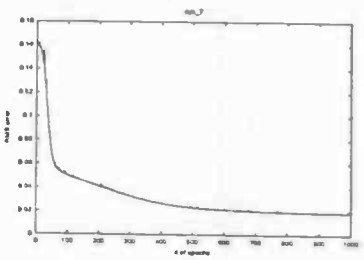
Experiment 2.7; Run 4



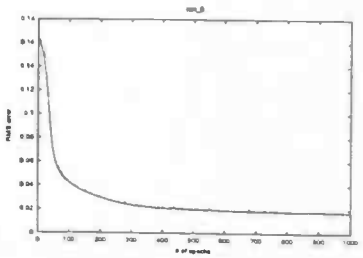
Experiment 2.7; Run 5



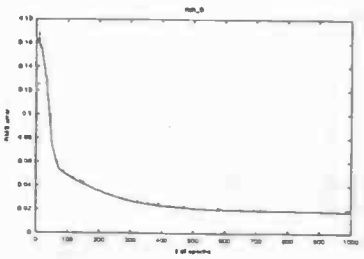
Experiment 2.7; Run 6



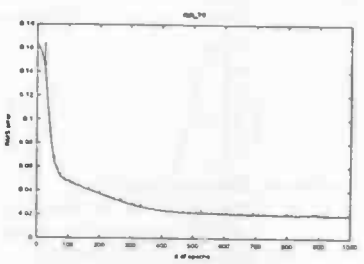
Experiment 2.7; Run 7



Experiment 2.7; Run 8



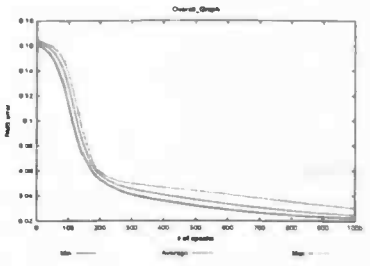
Experiment 2.7; Run 9



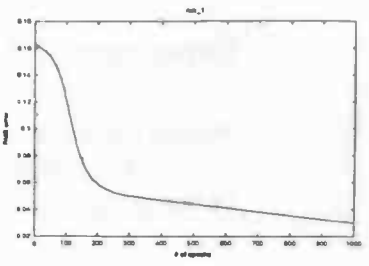
Experiment 2.7; Run 10

Experiment 2.8

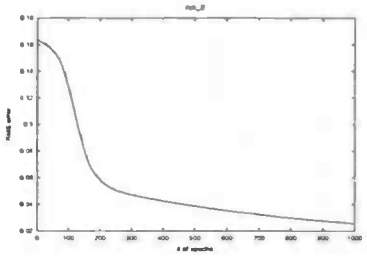
Natural exponent
AL (random) experiment
Learning rate = 0.1



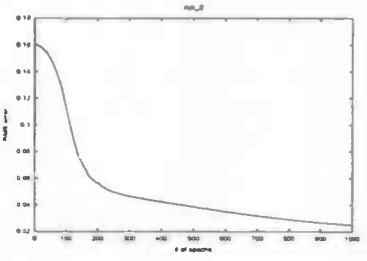
Experiment 2.8; Overall



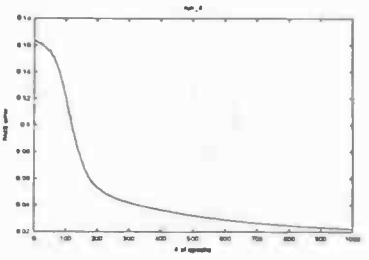
Experiment 2.8; Run 1



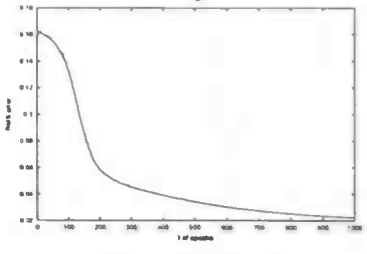
Experiment 2.8; Run 2



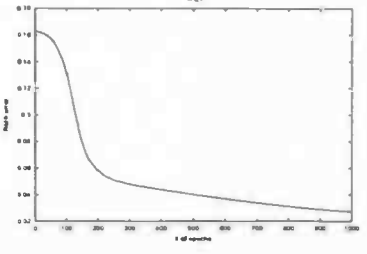
Experiment 2.8; Run 3



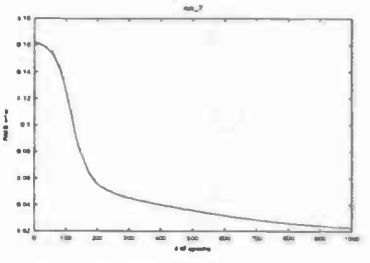
Experiment 2.8; Run 4



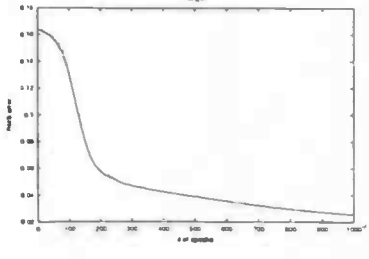
Experiment 2.8; Run 5



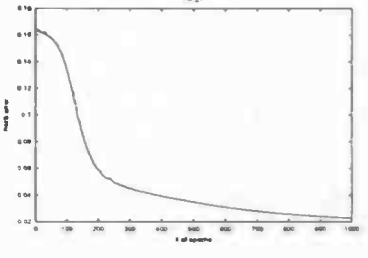
Experiment 2.8; Run 6



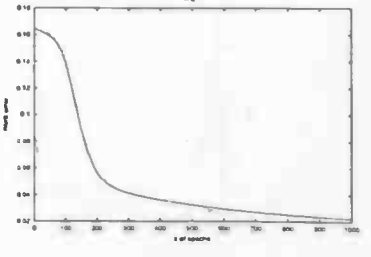
Experiment 2.8; Run 7



Experiment 2.8; Run 8



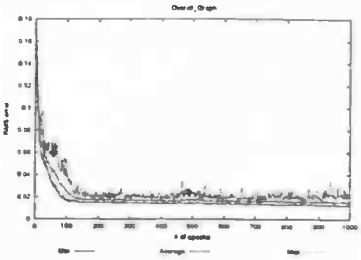
Experiment 2.8; Run 9



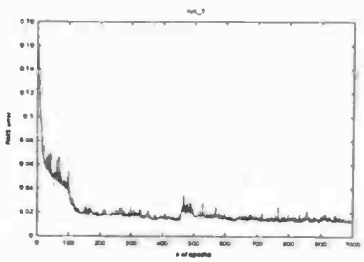
Experiment 2.8; Run 10

Experiment 2.9

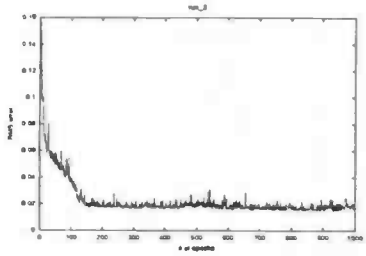
Natural exponent
AL (maximum) experiment
Learning rate = 0.7



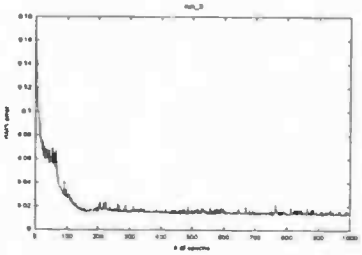
Experiment 2.9; Overall



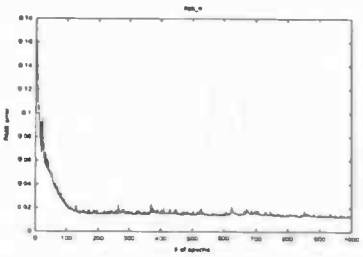
Experiment 2.9; Run 1



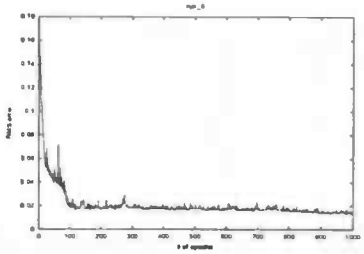
Experiment 2.9; Run 2



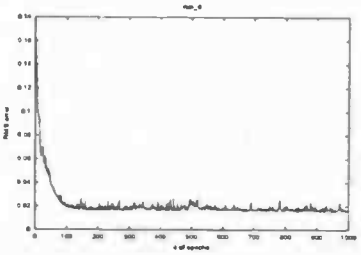
Experiment 2.9; Run 3



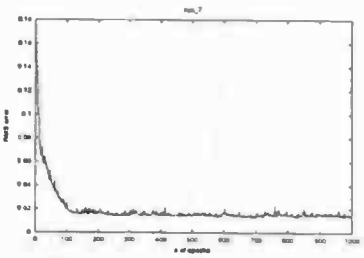
Experiment 2.9; Run 4



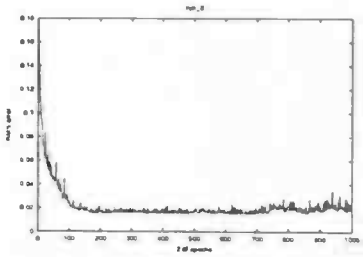
Experiment 2.9; Run 5



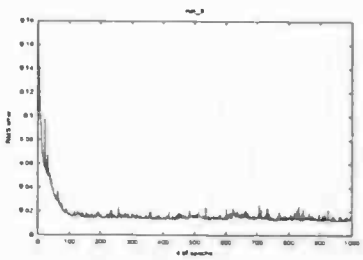
Experiment 2.9; Run 6



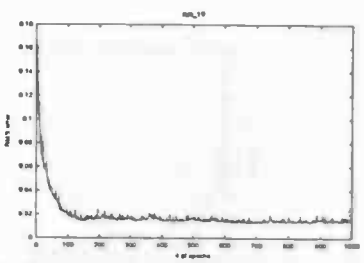
Experiment 2.9; Run 7



Experiment 2.9; Run 8



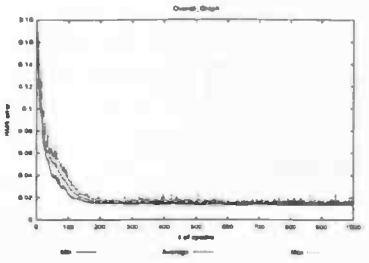
Experiment 2.9; Run 9



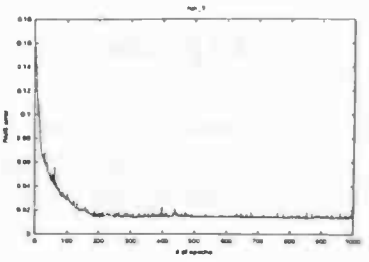
Experiment 2.9; Run 10

Experiment 2.10

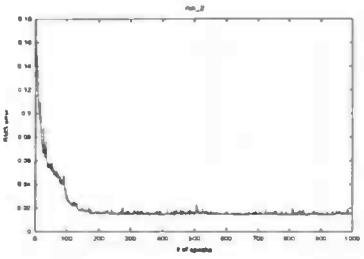
Natural exponent
AL (maximum) experiment
Learning rate = 0.5



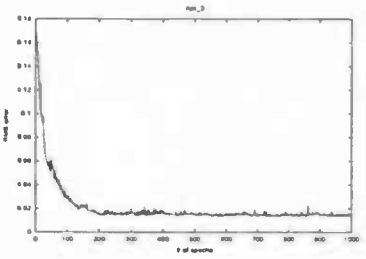
Experiment 2.10; Overall



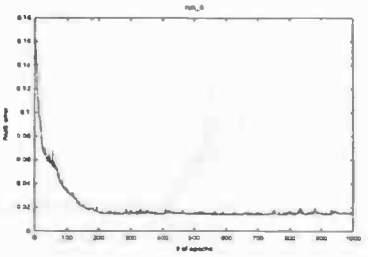
Experiment 2.10; Run 1



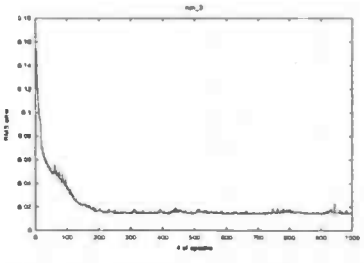
Experiment 2.10; Run 2



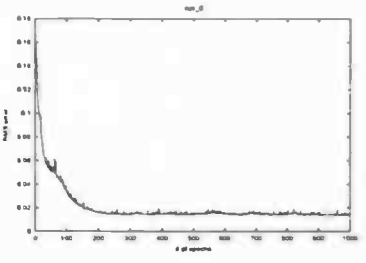
Experiment 2.10; Run 3



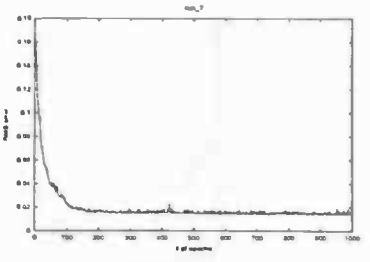
Experiment 2.10; Run 4



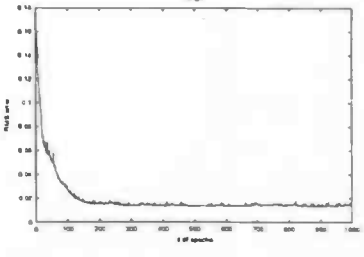
Experiment 2.10; Run 5



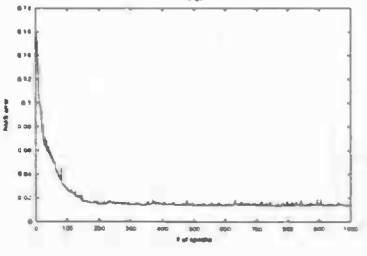
Experiment 2.10; Run 6



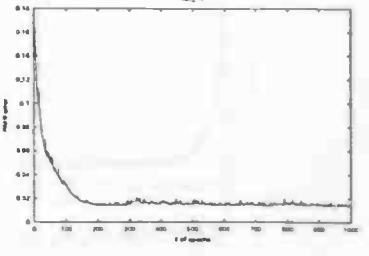
Experiment 2.10; Run 7



Experiment 2.10; Run 8



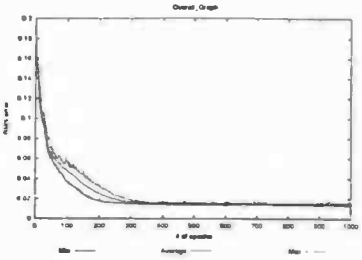
Experiment 2.10; Run 9



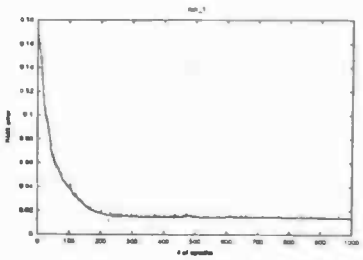
Experiment 2.10; Run 10

Experiment 2.11

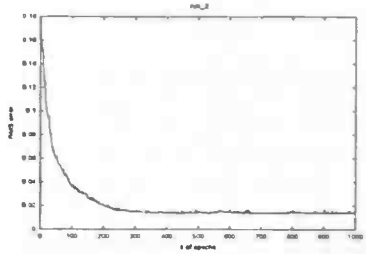
Natural exponent
AL (maximum) experiment
Learning rate = 0.3



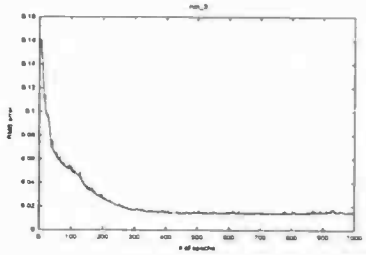
Experiment 2.11; Overall



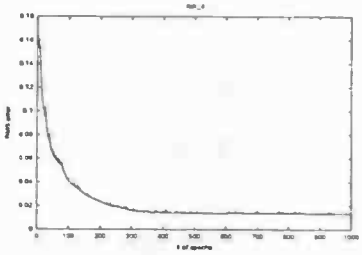
Experiment 2.11; Run 1



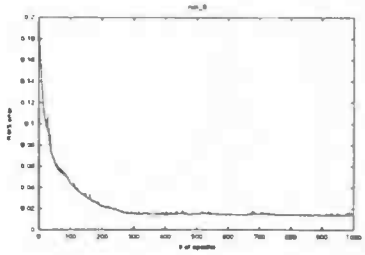
Experiment 2.11; Run 2



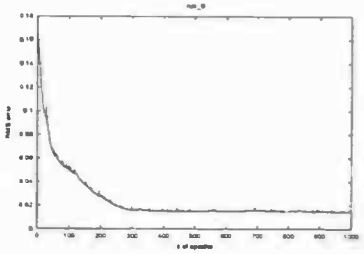
Experiment 2.11; Run 3



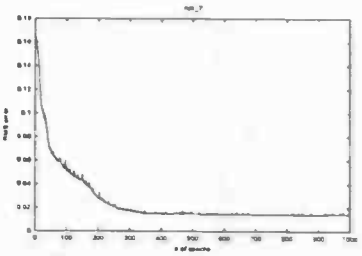
Experiment 2.11; Run 4



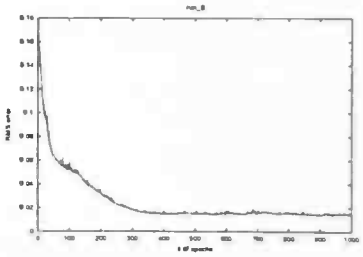
Experiment 2.11; Run 5



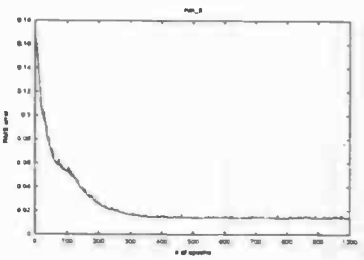
Experiment 2.11; Run 6



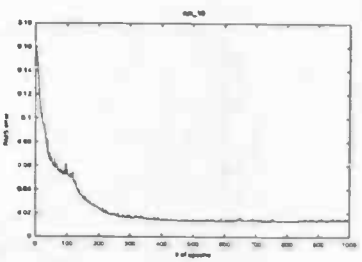
Experiment 2.11; Run 7



Experiment 2.11; Run 8



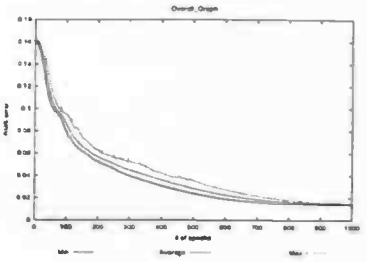
Experiment 2.11; Run 9



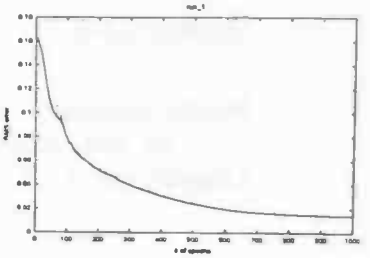
Experiment 2.11; Run 10

Experiment 2.12

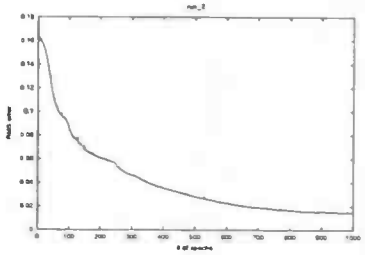
Natural exponent
AL (maximum) experiment
Learning rate = 0.1



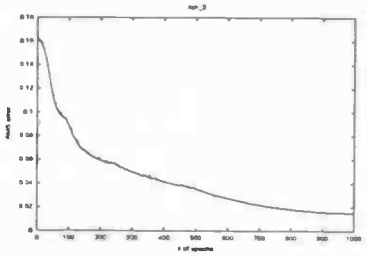
Experiment 2.12; Overall



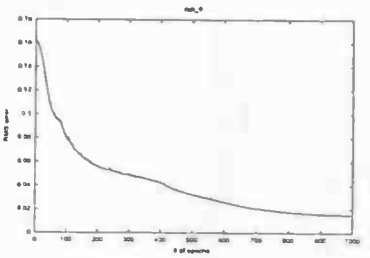
Experiment 2.12; Run 1



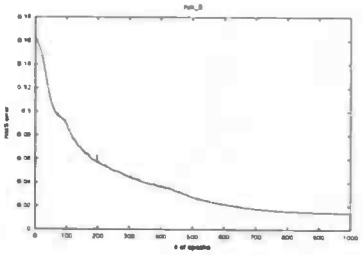
Experiment 2.12; Run 2



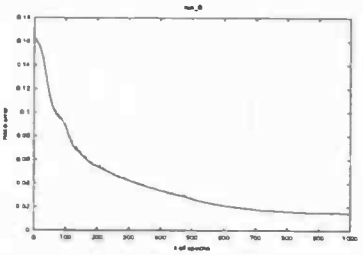
Experiment 2.12; Run 3



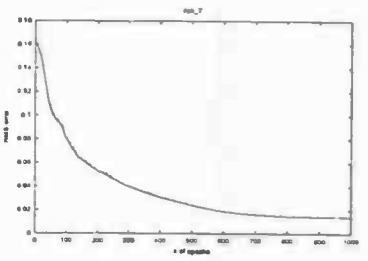
Experiment 2.12; Run 4



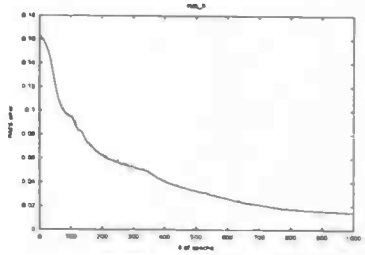
Experiment 2.12; Run 5



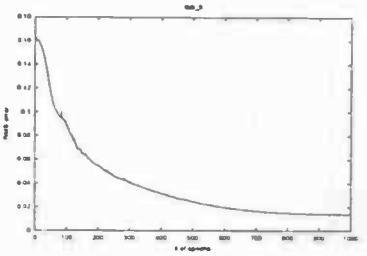
Experiment 2.12; Run 6



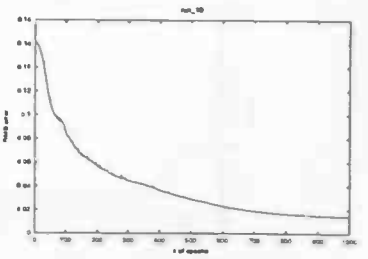
Experiment 2.12; Run 7



Experiment 2.12; Run 8



Experiment 2.12; Run 9



Experiment 2.12; Run 10

References

[1] L. Breiman. "Bagging predictors". University of California, Department of Statistics, Technical Report No. 416, 1994.

[2] S. Hansen and P. Salamon. "Neural network ensembles". IEEE Transactions on Pattern Analysis and Machine Intelligence, 12(10): 993-1000, 1990.

[3] M. Perrone and L.N. Cooper. "When networks disagree: Ensemble methods for hybrid neural networks". In *R.J. Mammone (Ed) Neural Networks for Speech and Image Processing*, Chapman Hall, 1993.

[4] S. Hashem and B. Schmeiser. "Approximating a Function and its Derivatives using MSE-Optimal Linear Combinations of Trained Feedforward Neural Networks". In *Proceedings of the World Congress on Neural Networks* vol 1, pp 617-620, Lawrence Erlbaum Associates, New Jersey, 1993.

[5] D.H. Wolpert. "Stacked generalization". *Neural Networks*, 5, pp 241-259, 1992

[6] M.I. Jordan and R.A. Jacobs. "Hierarchical mixtures of experts and the EM algorithm". *Neural Computation*, 6, pp 181-214, 1994

[7] S. Haykin. "Neural Networks A Comprehensive Foundation", IEEE Press, pp 1-20.