TDT4745 Kunnskapssystemer, Fordypningsemne

Catastophic Forgetting in Neural Networks

Ole-Marius Moe-Helgesen
Håvard Stranden

Subject Teacher: Pinar Öztürk

December 2005

1 Summary

This report presents the work done by the authors during the course "TDT 4745 Kunnskapssystemer, fordypningsemne" at the Department of Computer and Information Science (IDI) at the Norwegian University of Science and Technology (NTNU) during the autumn of 2005. The problem that it studies is catastrophic forgetting in neural networks.

The report starts by explaining the problem of catastrophic forgetting, and gives a broad survey of various solutions to the problem. Two of the most well known solutions to the problem are reproduced according to their original experimental descriptions, and the results are found to be reproducible. Further studies are made using alternative measurements of catastrophic forgetting, which reveal that the solutions do not eliminate catastrophic forgetting, and hardly reduce catastrophic forgetting at all in some cases.

Contents

1 Summary **2**

2 Introduction **7**

2.1 Neural networks 7

2.1.1 Structure 8

2.1.2 Training 10

3 Overview **12**

3.1 The problem 12

3.2 Motivation 13

3.3 Mathematical Explanation for Catastrophic Interference . . . 15

3.3.1 Orthogonality 16

3.3.2 Frean and Robins' Analysis 16

4 Solutions **19**

4.1 Rehearsal of previously learned patterns 19

4.1.1 The Rehearsal Buffer Model 19

4.1.2 Sweep rehearsal 20

4.1.3 Problems with the rehearsal solutions 20

4.2 Pseudorehearsal 21

4.3 Activation sharpening 23

4.4 Dual-weight models 25

4.4.1 Hinton and Plaut's "deblurring" network model 25

4.4.2 Gardner-Medwin's model of recognition memory 26

4.4.3 Levy and Bairaktaris' high capacity dual-weight model 27

4.5 Dual-network models 30

4.5.1 French's "pseudo-recurrent" dual-network model 31
4.6 Important aspects of evaluation 32
4.6.1 Important input pattern characteristics 33
4.6.2 Important learning characteristics 33
4.7 Conclusion 34

5 Measuring catastrophic forgetting **37**
5.1 Measuring recognition of learned patterns 37
5.1.1 The average squared error measurement 37
5.1.2 Relative amount of wrong outputs 38
5.1.3 Amount of correctly recognized patterns 38
5.1.4 A change-in-error measurement 39
5.2 Serial position plots 39
5.3 The standard discriminability statistic 40
5.4 Conclusion 42

6 Evaluation of solutions **43**
6.1 Chosen solutions and justification 43
6.2 Pseudorehearsal 44
6.2.1 Experimental description 44
6.2.2 Measuring performance 47
6.2.3 Results 48
6.2.4 Discussion 51
6.2.5 Conclusion 52
6.3 Activation sharpening 52
6.3.1 Experimental description 53
6.3.2 Results 56

6.3.3 Discussion 57

6.3.4 Conclusion 59

6.4 Comparison of the two solutions 60

6.4.1 Pseudorehearsal with French's measurement method . 60

6.4.2 Activation sharpening with Robins' measurement method 60

6.4.3 Correctly recognized patterns with pseudorehearsal . . 62

6.4.4 Correctly recognized patterns with activation sharpening 64

6.4.5 Conclusion 65

7 Conclusion 66

8 Further work 67

A Experimental workbench 72

A.1 Architecture 72

A.2 Configuration file format 73

A.3 The core library 75

A.3.1 The parser class 75

A.3.2 The plugin loader 76

A.4 Interfaces 77

A.4.1 The network interface 77

A.4.2 The trainer interface 79

A.4.3 The input loader interface 80

A.4.4 The weight loader interface 80

A.4.5 The weight saver interface 81

A.4.6 The activation function interface 81

A.5 Using the eGAS library 82

B Plots from pseudorehearsal experiments **83**

2 Introduction

In this report we will look at a fundamental problem in neural networks: catastrophic forgetting. We will study the work of researchers in the field, provide an analysis of their results and reproduce their experiments. The questions we seek answers to are:

(1) What work has been done to overcome the problem of catastrophic forgetting in neural networks?

(2) How general are the results from this work?

(3) How well do the existing solutions perform on real-world problems.

When studying the relevant research to find the answers to these questions, some clarification about the word 'general' in this context is in order. The most important dimension of variation is that of the input data. For a solution to be able to work well for real-world problems, it must not put restrictions on how the inputs are grouped in the input space. For instance must not a *general* solution require that the inputs are all orthogonal (we will return to orthogonality in section 3.3) to each other. For practical reasons, it is also desirable that the solution can work on different network architectures.

The remainder of this chapter will be used to introduce important concepts regarding neural networks. Then in the next chapter a thorough introduction to catastrophic forgetting will be given. In chapter four we will introduce several results from recent research into catastrophic forgetting. Chapter five is used to introduce methods of measuring catastrophic forgetting, before we in chapter six present the results from our reproduction of some of the experiments presented earlier. Chapter seven will conclude the project with our conclusions and views on possible future studies.

2.1 Neural networks

Before introducing catastrophic forgetting it is prudent to spend some time refreshing the basics of neural networks. This section contains some important terminology that will be used throughout the report. It should however not be seen as an exhaustive guide to neural networks, as that is clearly out of scope for this report. An understanding of what neural networks are and

how they operate is expected from the reader, but this part of the report will serve to establish the terms we will use. For more details, there are many books, tutorials and papers available, among others *The Essence of Neural Networks* by Robert Callan [1].

2.1.1 Structure

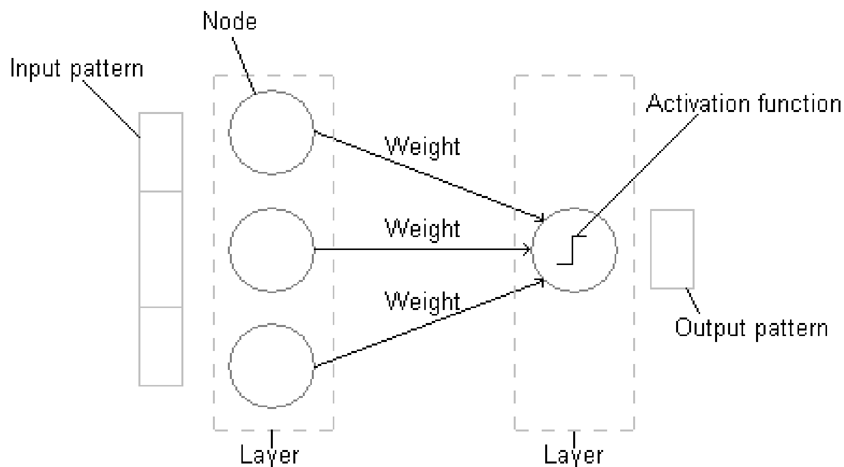

Figure 1: Conceptual illustration of a fully connected feed-forward ANN. Three nodes are grouped together in one layer, and one node is in the next layer. The black box illustrates the activation function that transforms the net input to output.

Artificial Neural Networks or *ANNs* are a distributed form of memory. They consist of a set of *nodes* connected to each other. Each of these connections has one or more *weights*, and it is in these weights that the storage of information takes place. We say that ANNs are *distributed* because each piece of information that we want to store is stored in the whole of the network and not just in one or a few weights. If each stored piece of information is only represented in a small part of the network, we say that the storage is *localized*.

Each node has a number of inputs and outputs. The output value of a node is determined by its total *weighted input* or *net input*, often denoted *net*. When determining the output of a given node, the net input is fed through an *activation function*. There are an infinite number of ways to do activation, but the sigmoidal function $1/(1 + e^{-x})$, step or sign function and linear function $ax + b$ are used most often. Figure 2 shows a plot with the

three functions.

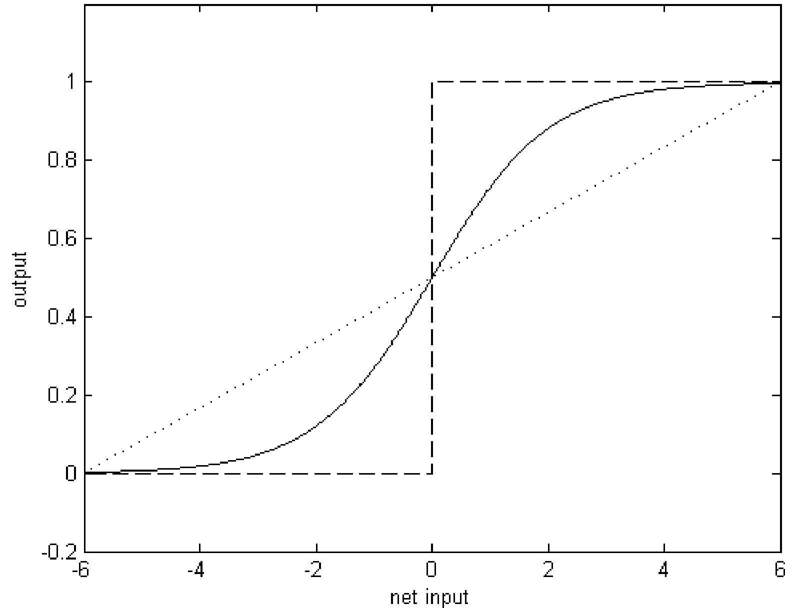

Figure 2: Three activation functions: Sigmoid function $\frac{1}{1+e^{-x}}$ (solid line), Linear function $\frac{x}{12} + 0.5$ (dotted line), and step function $sign(x)$ (dashed line).

In figure 1 a conceptual view of a neural network is shown. In addition to the terms established earlier, one can see that nodes are often grouped together in *layers* where every node in a given layer performs the same function. A common type of network which we will frequently discuss is the *feed-forward network*. In this network all the connections are from layer i to layer $i + 1$. In this network the layers are divided into three distinct types: One *input layer* which is provided with input data, any number of *hidden layers* where the first receives input from the input layer and the last provides input to the *output layer*. For the rest of the hidden layers the connections are from each hidden layer to the next. Note that it is possible with zero, one or more hidden layers. If there are no hidden layers the input layer is directly connected to the output layer, and we have a *linear* network. If there is one hidden layer, the same layer is connected to both the input and output layer. A network is said to be *fully connected* if, for all layers after the input layer, all nodes in layer $i - 1$ are connected to all nodes in layer i.

We will look at other network models as well later in this report, but these will be explained when encountered.

2.1.2 Training

For ANNs to be able to recognize some type of input they must be trained to perform that recognition. Training the network is also often referred to as *learning*. A goal of this learning is to be able to generalize the input so that an *input vector* (or *input pattern*) with noise can be recognized as the original input when presented to the network. The most common way of training a feed-forward network is through *back-propagation* [2]. This leads to the term *feed-forward back-propagation network* or *FFBP*. The back-propagation algorithm is a supervised learning algorithm where each *data item* during training consists of an *input pattern* and a *target output pattern*. The input layer is *fed* the input pattern, and then the network is asked to provide an output. Both input and output patterns are usually a vector of real numbers. Feeding an input pattern to a neural network means to set the value in each element of the vector at the corresponding node in the input layer. For instance, if a network is to be fed the input pattern (1,-1,0), the first node in the input layer would be given the value 1, the second the value -1, and the third the value 0.

After feeding the input, the output the network computes is compared to the target output pattern in the data item and an error is recorded. Next, the error is back-propagated through the network: each weight is adjusted with “its part of the blame” for the error. In concept, this means that connections that had a high contribution to the error will have a large adjustment done to them, while connections with a smaller contribution will experience a smaller change. The back-propagation process is controlled by two parameters; the *learning rate* that determines how much the calculated error should contribute to the new weight change, and the *momentum* that determines how much the previous weight change should contribute the new weight change. If there is no previous weight change, this contribution is zero. Both the learning rate and the momentum are typically set to a value between zero and one. The learning rate is usually fairly low, and the momentum fairly high.

Feeding *all* the data items in the training set and updating the weights in the network for all of them is called an *epoch*. The stop criteria of back-propagation is, depending on the training set used, when the error for all the data items in the training set is within some limit, or that the weights in the networks has converged. Training the network until it has reached the given stop criteria is called to train the network *to acceptance* or *to criterion*.

How many elements a neural network can store depends on the degree of

connectivity, the training, and the topology of the network. If a network has reached its theoretical limit on the number of data items to store, we say that it has reached its *saturation point*.

3 Overview

In this chapter we will provide an overview of the catastrophic forgetting problem, look at why it is such an important problem to solve, and provide some insight into why it occurs.

We will begin by explaining what precisely the problem is, and see how it relates to earlier work.

In the second section of the chapter we will try to provide proper motivation for solving the problem of catastrophic forgetting. We will argue that there are two main reasons for why one wants solutions to the problem: (1) To use artificial neural networks in problems where sequential learning is involved, and (2) To be able to properly explain how mammal brains, and human brains in particular, handle learning.

We will conclude this chapter by providing some insights into the mathematical explanations that has been put forth regarding the catastrophic forgetting problem.

3.1 The problem

Catastrophic forgetting, also referred to as catastrophic interference or the sequential learning problem, was discovered in the late 1980's by two research groups separately: McCloskey and Cohen in 1989[3] and Ratcliff in 1990[4]. In short catastrophic forgetting is when old information is erased instantly or almost instantly when trying to store new information together with the old.

This problem can be seen in conjunction with a general dilemma Grossberg coined *the stability-plasticity dilemma* in 1980[5]. In the preface to this article, published in his 1982 book[6], he writes:

> How can an organism's adaptive mechanisms be stable enough to resist environmental fluctuations which do not alter its behavioral success, but plastic enough to rapidly change in response to environmental demands that do alter its behavioral success?

In artificial neural networks the problem is that the networks are too plastic. The weights of the network will readily and rapidly adapt to new information, but often at the expense of previously learned information. When dealing

with ANNs it is relatively easy to create a network that can remember a set of input-pattern-to-output-pattern associations *if these patterns are all available at the same time*. What is a lot more difficult though is to provide the network with the patterns in *sequential order* while still being able to recognize both the new and the old patterns. An example used by Grossberg explains it in an easy-to-understand way:

> The seriousness of this problem can be dramatized by imagining that you have grown up in Boston before moving to Los Angeles, but periodically return to Boston to visit your parents. Although you may need to learn many new things to enjoy life in Los Angeles, these new learning experiences do not prevent you from knowing how to find you parent's house or otherwise remembering Boston.

For neural networks this is the exact situation. As an example can one teach a network to learn to associate portraits with the name of the person depicted. If however one provides the images in two batches, say women in the first and men in second, training the network to learn all the women's names first and all the men's names afterwards, one is likely to end up with a network that can recognize all of the men and none of the women.

Overcoming this sequential learning problem is what we will look at in this project, and to do so we will begin by properly establishing the motivation for solving the catastrophic forgetting problem.

3.2 Motivation

As mentioned in the previous section, the effect of catastrophic interference is that neural networks usually fail when learning information sequentially. This is a problem for two reasons.

The first reason is from a practical point of view, where it is desirable to have a learning system that does not require all patterns it should learn to be available when learning is to take place. There are many cases where concurrent learning like this is undesirable or even impossible. Consider a robot walking around in the world while learning to associate names with items. If learning is only possible concurrently, the robot will have to learn all previously learned associations every time a new object is encountered. Even though that can be possible in one particular robot, the storage requirements

increases as a function linear to the number of items to remember. If we have to remember all the old associations outside the neural network, what then is the point of having the neural network at all? In some systems this rehearsal is not even possible. However in chapter 4 of this report, we will see that many of the proposed solutions to the problem of catastrophic interference originates from this basic idea of rehearsing old information.

The second reason why catastrophic interference is a problem is in the larger perspective, where neural networks are studied as a means to understand the brain of humans or other animals. The expected behavior for an artificial brain, as for humans, is not to forget all old information when learning something new. As an example will a healthy human that after many years move to a new home not completely and almost instantly forget the layout of his old home. As long as catastrophic interference occurs in the artificial models for cognition, such models cannot be accepted as feasible models of a human brain.

There is however evidence for catastrophic interference in other animals. French and Ferran [7] performed an experiment where they tested time perception in rats. In their experiment they had a system where the amount of food the rats received was dependent on the duration that a lever was pushed down. The number of seconds that would give the maximum amount of food was the target number the rats were trained to learn. The rats were trained to learn 8 second and 40 second durations, using both sequential and concurrent training. The results from that experiment indicated that rats suffer from catastrophic forgetting when learning length in the time dimension. With "length in the time dimension" we mean a lapse of time, in this case 8 or 40 seconds. This in turn indicates that rats have a different way of storing temporal information like the one trained in the experiment than other kinds of information. *One of these ways of storage exhibit signs of catastrophic forgetting while the other does not.*

But even if some biological brains show signs of catastrophic interference, it is clear that they do not catastrophically forget in all cases. In the neural networks used today however, catastrophic forgetting is seen in most cases. We will see some explanations as to why this happens in section 3.3. There is also experimental evidence for catastrophic-like forgetting in human brains, but then due to physical damage and not interference by new information. J. McClelland et al. argues [8] that when the two hippocampi are damaged, catastrophic forgetting *of recent memories* can result. Already in 1957, when a patient had both his hippocampi removed in surgery, it was discovered that the hippocampal system was related to short-term memory [9]. The result

was that this patient could not remember anything of recent events nor could he remember new situations. He could however recall events that took place *before* his admittance into the hospital some years earlier. There has been much research done in this area, and it has - among other things - helped lead to the current artificial neural networks as a model of the hippocampus.

McClelland et al. theorizes that the older memories still retained during a leisure to the hippocampus are stored in what is called the neocortal system, often called the neocortex. It is not a clear line between what is in the neocortal and in the hippocampal systems, but the basal ganglia and the amygdala is frequently thought to be part of the neocortex. The neocortal system is part of the cerebral cortex that covers the four lobes of the brain. It is believed that the neocortex has developed in higher mammals and is where much of our "newly" developed higher intelligence takes place. McClelland, et al. put forth the theory that new memories are being learned in the hippocampus like it was discovered by Scoville and Milner, but that they are transferred to the neocortex at a later stage and kept there.

In chapter 4 where we look at various proposed solutions, we will also examine ANN systems that work from this same idea: that memories are stored in two separate systems, one short-term plastic memory and one long term stable memory.

Catastrophic forgetting was first seen in feed-forward back-propagation networks [3] [4], but also in other network types such as Hopfield [10] and Elman [11] networks has the problem been identified and solutions applied to reduce it. It is however the hetero-associative feed-forward networks that most of the research and focus has been directed at.

3.3 Mathematical Explanation for Catastrophic Interference

So why does catastrophic forgetting happen in neural networks? In the following pages we are going to look at some of the more formal explanations. To begin with, we will look at the concept of orthogonality. This is a familiar term from linear algebra, and is also important when understanding catastrophic interference.

3.3.1 Orthogonality

In mathematics, orthogonal means perpendicular, or to be at right angles. Two vectors are orthogonal to each others if their dot product is zero. The definition of the dot product - or scalar product - of two vectors **a** and **b** is:

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i$$

where a_i is the ith element of **a** and b_i the ith element of **b**. In euclidean space it can be expressed as.

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}||\mathbf{b}| \cos\theta \quad (1)$$

In neural networks this is an important concept because orthogonality is a measure of the representational overlap of the n-dimensional input vectors in the n-dimensional euclidean input space. Two vectors that are orthogonal to each others are represented in different parts of the input space. As an example are the three input vectors (1,0,0), (0,1,0), and (0,0,1) all orthogonal to each others. What we can say is that the more orthogonal the inputs are, the more localized are each of the input patters. They are all represented in different regions of the input space.

From the definition in equation 1 it is also clear that if we keep the length of the two vectors constant and change the angle θ between them, the dot product will have the form of a sine wave with zero when the vectors are orthogonal to each other.

3.3.2 Frean and Robins' Analysis

Frean and Robins[12] performed a study related to pseudorehearsal in linear neural networks where they showed that the more orthogonal the input vectors that are to be learned are to each other, the less catastrophic the interference will be. The following general equations are from their article and will show that this is in fact true *when dealing with linear networks*. The network used has a single input layer, and one output node. It will be provided with two data items, A and B. Both of these have an input vector and a target output value. The following terms will be used in the equations:

a, **b**	Input vector for data items A and B.
a_i, b_i	The *i*th element of the vector.
w	Weights between the input layer and the output node.
t_A, t_B	Target output for data items A and B.
err_A, err_B	The error produced by the network when given a data item.

At the beginning, the network is trained to learn data item A so that $err_A = 0$. If we at this time provide the network with data item B, the error will be:

$$err_B = t_B - \mathbf{w} \cdot \mathbf{b} \quad (2)$$

A generalization of equation2 obviously holds for any data item provided to the network. For data item A, the error if we make a change of $\Delta\mathbf{w}$ to the weights will be:

$$err'_A = -\Delta\mathbf{w} \cdot \mathbf{a} \quad (3)$$

Now, to be able to make our new network correctly recognize B, $\Delta\mathbf{w}$ is given by:

$$\Delta\mathbf{w} = \delta\mathbf{b} \quad (4)$$

This is obviously very general, but will cover most of the commonly used learning rules for neural networks, including the delta rule [2]. If we want to remove any error for set B, i.e get $err'_B = 0$, the weight change is given by:

$$\begin{aligned} err'_B &= t_B - \mathbf{w} \cdot \mathbf{b} \\ 0 &= t_B - (\mathbf{w} + \Delta\mathbf{w}) \cdot \mathbf{b} \\ \Delta\mathbf{w} \cdot \mathbf{b} &= t_B - \mathbf{w} \cdot \mathbf{b} \\ \Delta\mathbf{w} &= \frac{t_B - \mathbf{w} \cdot \mathbf{b}}{\mathbf{b}} \\ \Delta\mathbf{w} &= \frac{err_B}{\mathbf{b}} \end{aligned}$$

Using this result and equation 4 we see that δ is given by:

$$\begin{aligned} \Delta\mathbf{w} &= \delta\mathbf{b} \\ \delta &= |\frac{\Delta\mathbf{w}}{\mathbf{b}}| \\ &= \frac{|\Delta\mathbf{w}|}{|\mathbf{b}|} \\ &= \frac{|\frac{err_B}{\mathbf{b}}|}{|\mathbf{b}|} \\ \delta &= \frac{err_B}{|\mathbf{b}|^2} \end{aligned}$$

Substituting back into equation 4, the required weight change is:

$$\Delta \mathbf{w} = \frac{err_B}{|\mathbf{b}|^2}\mathbf{b} \tag{5}$$

This can also be recognized as the delta rule weight change with learning rate $1/|\mathbf{b}|^2$. Using the result in equation 5 together with the expression for the new err'_A in equation 3, the error when being provided data set A after having changed the weights to classify data set B is given by:

$$\begin{aligned} err'_A &= -\frac{err_B}{|\mathbf{b}|^2}\mathbf{b}\cdot\mathbf{a} \\ err'_A &= -\frac{\mathbf{a}\cdot\mathbf{b}}{\mathbf{b}\cdot\mathbf{b}}err_B \end{aligned} \tag{6}$$

The final equation(6) shows the interesting result. We can see that the dot product between vectors **a** and **b** will determine the error for data set A. In the previous section on orthogonality we showed that the dot product will approach - and reach - zero when the two vectors reach orthogonality. In other words, the level of forgetting about the previous data (err'_A), will be lower the more orthogonal the input vectors are.

On the other hand, it should be clear that when a network contains only orthogonal input, the storage capacity of the network is drastically reduced. Clearly there is some middle ground where catastrophic forgetting is reduced while the network still maintains high storage capacity when compared to other models.

It is not clear how this transfers to non-linear networks, i.e. networks with hidden layers. French points out in [13] that if the information in neural networks is very distributed and with high overlap, catastrophic forgetting is more prevalent. High overlap of information, i.e. that many connections contribute to many data sets, is exactly the same as the low level of orthogonality in the linear case used here. It is however our belief that a certain degree of orthogonality in the input vectors does not guarantee that the information in the network is stored with a similar degree of distribution.

4 Solutions

Several people have tried to come up with solutions to either overcome or reduce the problem of catastrophic forgetting. French has written two articles [14] [15] that give a rough taxonomy of the various existing solutions. This chapter gives an overview of various solutions that have been presented by various researchers, highlights their strengths and weaknesses, and compares them to each other. The implementations of solutions that are discussed in this chapter have been selected as representative examples of their respective solutions from the attention that has been given to them in articles that discuss catastrophic forgetting. They are presented in a pragmatic manner according to what they do to overcome the problem.

4.1 Rehearsal of previously learned patterns

As we have mentioned earlier, the problem of catastrophic forgetting was first highlighted by McCloskey and Cohen in [3] and Ratcliff in [4], the latter of which also provided one of the early ways to reduce the problem: retrain the network on previously learned patterns as new patterns are learned, so it will be “reminded” of its previous knowledge. One implementation of this type of rehearsal is given by Ratcliff [4], and is called the *Rehearsal Buffer Model*.

4.1.1 The Rehearsal Buffer Model

The Rehearsal Buffer Model was also reviewed by Robins [16], who named it the *recency rehearsal mechanism*. With the Rehearsal Buffer Model, when a network is trained on a new pattern, that pattern is put in a set together with the n most recently learned previous patterns. The resulting set, of size $n+1$, is named a rehearsal buffer. The network is then trained on this pattern set instead of being trained on the single new pattern. Figure 3 illustrates the approach. The Rehearsal Buffer Model approach has been found to reduce the problem of catastrophic forgetting [4] [16].

A research done by Robins in [16] researched some variations of the model. These variations train a network in the same way as the Rehearsal Buffer Model, but instead select the previously learned patterns randomly, or choose patterns that were least recently learned. All these variations of ways of selecting previous patterns show no performance difference in practice.

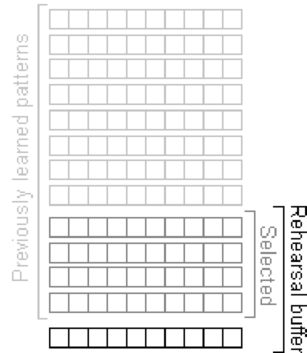

Figure 3: The rehearsal buffer for $n = 4$ when using the most recently learned previous patterns.

4.1.2 Sweep rehearsal

Sweep rehearsal is a particular variation of the Rehearsal Buffer Model introduced by Robins in [16]. In sweep rehearsal, the previously learned patterns to be used in rehearsal are selected randomly *for each epoch*, as oppose to the Rehearsal Buffer Model, where the selection of the most recent patterns is done only once. This process is repeated until the new pattern to be learned is learned to acceptance. Sweep rehearsal shows a significantly better performance than the previously presented rehearsal methods [16]. It is better because it will train the network on more of the previously learned patterns, and not discriminate any previously learned patterns [17], since it selects these patterns randomly for each epoch instead of only once.

4.1.3 Problems with the rehearsal solutions

The main problem with the rehearsal solutions is that the need to interleave previously trained patterns with new training patterns requires explicit storage of some or all of the previously trained patterns. This requirement not only increases the required storage space for a neural network, but also contradicts one of the purposes of a neural network: to be able to learn patterns without further explicit storage of the patterns it has learned. If one requires explicit storage of some or all trained patterns for a neural network, then other learning algorithms such as the k-nearest neighbor algorithm are likely

to be just as efficient.[1] When all known patterns are stored explicitly along with their desired output, it will perform as good as, and maybe even better than, a neural network if the similarity measure is appropriate.

Another problem with the rehearsal solution is the fact that it is not biologically plausible. When, for instance, humans learn a new word, they do not need to retrain every previously learned word to remember those. It does not seem plausible that we have these two types of storage in our brains; a neural network that recognizes the learned words, and a permanent storage mechanism with some explicit representation of each word (or the most recently learned, if we are to use the Rehearsal Buffer Model).

4.2 Pseudorehearsal

Another solution to the problem of catastrophic forgetting is named *pseudorehearsal*. The solution was first proposed by Robins in [17], and is stunningly simple.

The motivation for the pseudorehearsal solution is to get rid of the requirement to store previously learned patterns. The problem that arises is how to retain this information when it is not explicitly stored. Clearly there must be some way of fetching it from the network, but this is one of the main problems of neural networks; their exact operation and internal representation is not well known and understood. Therefore, a way to reproduce the previously learned data so they can be rehearsed must be found. This is what pseudorehearsal attempts to do.

When using pseudorehearsal to train a new item, a set of random input patterns are generated as random permutations with 50% zeros and 50% ones[2]. These random input patterns are then fed through the network in the normal way, generating an output vector. The input patterns and their corresponding outputs are then added to the training set, using the output each of them gave as that patterns associated output. Training is then performed in the normal fashion, using one of the previously described models for rehearsal, with the random input patterns and their associated outputs instead of the previously trained patterns (or a subset of them).

[1] The k-nearest neighbor algorithm classifies patterns by comparing them to the k nearest patterns according to some similarity measure. It then assigns the new pattern the class of the pattern to which it is most similar.

[2] This random pattern generation method was used in Robins' experiment [17], other generation methods may of course be used.

The random input patterns and their associated outputs are often referred to as *pseudopatterns* or *pseudo-items*. These pseudopatterns, quoting Robins in [17]:

> [...] serve as a kind of "map" of the function appropriate to reproducing the actual population.

The accuracy of this "map" depends on the number of pseudopatterns that are generated (if all possible pseudopatterns are generated, the "map" will be complete and represent all the outputs the network is capable of producing).

Analysis of the pseudorehearsal solution shows that it performs just as good as, and in some cases better than, the rehearsal methods that use previously trained data (see [17]). A later analysis by Frean and Robins [12] suggests that pseudorehearsal works only for fairly large dimensions of output, due to the effects of dimensionality on orthogonality of patterns in the Euclidean space (see 3.3.1 for a discussion of orthogonality). The analysis concludes that when orthogonality between patterns is high, pseudorehearsal tends to disrupt or even destroy the learning of new patterns. As the dimensionality increases, and orthogonality thus decreases, pseudorehearsal moves from disrupting learning of new patterns to preventing forgetting of old patterns.

Little has been said about the biological plausibility of the pseudorehearsal method when it has been studied. In one of his taxonomy articles [15], French also remarks that this question needs answering. We consider it safe to say that even though no thorough studies have been done, the pseudorehearsal method is more plausible than the rehearsal methods requiring explicit storage of previously learned patterns. One can imagine that there is a pseudopattern-like storage in recurrent connections in the human brain that leaves behind some kind of map of the network, and is used when feeding the new pattern that is to be learned. We know that there are recurrent connections in the human brain, so it does not seem too far fetched to imagine that a mechanism like this can exist. Still, establishing whether or not such a hypothesis holds requires much research and is far beyond the scope of this report.

As a conclusion, pseudorehearsal seems to be a far more feasible solution than rehearsal of previously trained patterns, since it removes the requirement of explicit storage of new items. Pseudorehearsal generally works best when the input dimensionality is high. When input dimensionality is low, pseudorehearsal may disrupt or destroy the learning of new patterns. In general, when input patterns are relatively orthogonal, pseudorehearsal worsens

the learning. Since catastrophic forgetting has an inverse relationship with the degree of orthogonality of patterns, pseudorehearsal should only be used when catastrophic forgetting is actually present. Thus, an analysis of forgetting in the network is required before deciding whether or not to use pseudorehearsal.

4.3 Activation sharpening

Robert M. French suggested in 1991 [13] that catastrophic forgetting was a direct consequence of the distributed nature of a neural network; almost all nodes contribute to storing of every pattern. In his article, French proposes a model that reduces the distributiveness of the representation of the learned patterns.

The approach uses an algorithm called activation sharpening that reduces the distributiveness of training on a new pattern to a subset of the nodes in the hidden layer. Activation sharpening is done on k nodes in the hidden layer. The hidden layer activations are calculated as follows:

$$A_{new} = A_{old} + \alpha(1 - A_{old}) \tag{7}$$

$$A_{new} = A_{old} - \alpha A_{old} \tag{8}$$

where A_{old} is the hidden layer activation value calculated from the activation function, α is a sharpening factor that determines how much the nodes should be sharpened, and A_{new} is the sharpened activation. Equation 7 is used on the k nodes that are to be sharpened, and equation 8 is used on the other nodes in the hidden layer. The algorithm performs a forward sweep through the network, and marks the k most active nodes for sharpening. These k nodes then receive sharpening according to the equations above. The difference in activation is then used as the error measure, and propagated back through the network using standard back-propagation. After that, an additional forward sweep is done, and finally a back-propagation sweep using the standard error measure. The effect is that the distribution of the learning is reduced through a reduction of the chance of weight changes in the unsharpened nodes. The reason for this lies in the back-propagation algorithm, which changes the weights very little when a node has activation close to zero.

The theory that gives rise to the activation sharpening algorithm says that interference is assumed to occur when the same nodes are active for different patterns. Since the activation sharpening effectively reduces the number of

nodes that experience a large weight change, it also reduces the amount of catastrophic forgetting.

Looking at the downsides of this approach, The semi-distribution of learning in a sharpened network implies that there is a limit to how many patterns that can be stored before interference occurs when activation sharpening is used, but the exact value of this limit is unknown, as is whether the limit is higher or lower than the limit for a standard feed-forward network of the same size. French suggests in his article that the optimal number of nodes to sharpen is the smallest k such that

$$m < nCk$$

where m is the number of data items to learn, n is the number of hidden nodes, C is some unknown constant. He does not elaborate further on the correctness of this equation.

Another downside to the activation sharpening algorithm is the fact that as the storage of learned patterns is less distributed through the network, the network loses more and more of its ability to generalize. An important question left open by French is therefore how many nodes that are sharpened (i.e. what k) when there is an "optimal" trade-off between the reduction of catastrophic forgetting and the ability to generalize.

The biological plausibility of activation sharpening is unknown, but as a thought experiment, one may imagine that there is some kind of function similar to this in the human brain. The brain is used to store and process vast amounts of information, and if every connection in the long-term memory in the brain was to contribute to storing every single memory, catastrophic-like forgetting would certainly seem possible, especially if the brain's functionality is otherwise similar to that of a neural network. To solve this problem, perhaps our brains utilize some kind of method to avoid fully distributed storage, while still being able to generalize and relate memories to each other. Although there may be some mechanism like this in the brain, it is unlikely that this works like activation sharpening does. The solution requires knowledge of the activation level of all the nodes to be able to determine which nodes that should be sharpened, and it is highly unlikely that this kind of centralized knowledge of all connections in the brain exists.

4.4 Dual-weight models

The idea to use multiple sets of weights comes from biology, where a study by Kupferman in 1979 [18] proved that synaptic activation changes in the brain occur at many different time scales. Some change rapidly, and some change more slowly. This evidence has led to a number of solutions that try to model this mechanism in neural networks, using different methodologies in different ways. Some were made with reduction of catastrophic forgetting in mind, others were made for other purposes. We will look at three models that each have received some attention since their publication.

4.4.1 Hinton and Plaut's "deblurring" network model

Hinton and Plaut suggested a model whose purpose is to recall previously stored patterns better when trained on new sets. The model was introduced in [19], and uses two sets of weights for a network. As a direct reference to the stability-plasticity problem, the first set of weights is stable and changes slowly, the other set is plastic and changes fast[3].

The model proposed by Hinton and Plaut uses a standard feed-forward back-propagation network with two sets of weights. The net inputs produced to each unit from these two different sets are summed when producing output, and both weight sets are trained with the same error when back-propagating. In addition, the fast weight sets is reduced in magnitude with a factor h after each weight change. This makes the fast weights decay rapidly to zero. An illustration of this dual-weight model can be seen in figure 4. The results of using this model to learn different sets of patterns show that the fast weights temporarily cancel out the changes in the slow weights caused by subsequent pattern set learning, so that the original set can be easily restored by rehearsing on just a subset of it.

Since the distributed nature of a network implies that many weights are involved in the storage of each pattern, retraining on some of these patterns will force the weights to move back to the state where they recalled these patterns, and thus intuitively reinforce the memory of all the patterns in the original set to some extent. Thus, the solution actually exploits the distributed nature of a neural network to recall previously stored patterns.

[3]Note that this terminology differs from Hinton and Plaut's article, where the term 'plastic' is used about the slow weights. We choose to continue using the terminology that fits the definition of the stability-plasticity dilemma outlined in section 3.1.

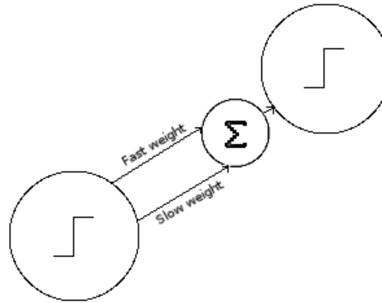

Figure 4: Two nodes using Hinton and Plaut's summing dual-weight model. The activation produced by the fast and slow weights are summed together to produce the input from one node to another.

For a mathematical analysis of this effect, termed the "deblurring" effect by Hinton and Plaut, see [19].

4.4.2 Gardner-Medwin's model of recognition memory

Another dual-weight model was proposed by Gardner-Medwin [20]. His aim was to build a high-quality model of short- and long-term memory. The central difference between this model and Hinton and Plaut's model is the use of multiplicative weights, as oppose to summed weights. The justification for the use of multiplication lies in the biological observation that different synapses in the brain act on different scales of time (i.e. slower and faster memory), and their changes *combine* to produce the resulting activation. An illustration of nodes in this model is shown in figure 5. The model uses Hebbian learning, and implements an optimization mechanism that reduces overlap when training the long-term memory, thus reducing catastrophic forgetting and increasing the networks capacity.

The short-term weights are binary and act as a form of activators that are changed if a Hebb rule is satisfied. This yields a model for recognition memory, where it is easier to recall a pattern that is already stored in long-term memory, since the combined activation will then be higher. A problem with Gardner-Medwin's model was the fact that the fast weights would initially be zero when a pure long-term memory recall was done, since the weights would then have decayed. To overcome this problem, Gardner-Medwin suggested "booting" the network by setting short-term weights to some other

value. This method is compared to the biological process of recall in humans, where long-term recall may be a time demanding process often influenced by recent memories to some extent.

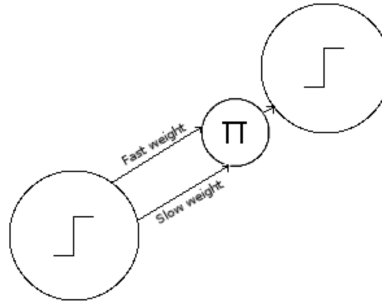

Figure 5: Two nodes using Gardner-Medwin's multiplicative dual-weight model. The activation produced by the fast and slow weights are multiplied with each other to produce the input from one node to another.

4.4.3 Levy and Bairaktaris' high capacity dual-weight model

Finally, Levy and Bairaktaris [21] extended Gardner-Medwin's model to use two independent sets of weights for long- and short-term memory. Although its capability to reduce or overcome catastrophic forgetting is unknown, it is one of the latest additions to the available dual-weight models. Arguably, French views this model as a kind of hybrid between the dual-weight solution and the dual-network solution [15], presented in a later section. He argues that one can just as well view the solution as two identical networks with a single set of weights each, where one network has slow weights, and another has fast weights. Still, the solution was viewed as a dual-weight model by its creators, so we will view it as such.

Levy and Bairaktaris' model is vaguely described in the article, which makes it hard to give a thorough description of it. It uses Hebbian learning for the short-term weights, and a deterministic Boltzmann machine learning algorithm called Mean Field Theory (MFT) for the long-term weights [4]. The interaction between the two weights is based on a mode switch from short-term to long-term mode, where the information from the previous mode is

[4] For more information on deterministic Boltzmann machines, see among others [22]

left behind as a node activation and thus interacts with the current mode. An illustration of two nodes in this model is shown in figure 7.

In short-term mode, the network acts as a two-layer bidirectional associative memory (BAM). A BAM is as extension of the Hopfield network that implements an additional layer, so that the network has an input layer and an output layer. This model is capable of hetero-association, in contrast to a Hopfield network, which can only do auto-association. The BAM implemented in Levy and Bairaktaris' model also includes some refinements to increase its capacity for storing of non-orthogonal patterns. These improvements were developed by Bairaktaris in [23]. The reason for this improvement lies in the way Hebb's rule trains the network. Networks trained with Hebb's rule have serious capacity issues with non-orthogonal patterns, due to the nature of the training rule. It can be shown that Hebbian learning can train a two-layer network to learn a set of input/output pattern association if the different input patterns in the input/output pattern associations are mutually orthogonal. However, if the input patterns are not mutually orthogonal, interference may occur, and the network may not be able to learn the various associations [23].

To overcome this, Bairaktaris created a *random internal representation* for his two-layered BAM. This random internal representation is a randomly generated pattern (vector of real numbers). The network is trained to learn how to associate a given input pattern to this random pattern, and then to associate the random pattern with the corresponding output pattern. The mechanism is illustrated in figure 6. If the dimensionality of these random vectors are larger than the input/output pattern dimensionalities, the dimensions of the combined input/output space that these represent is increased. According to Bairaktaris, this leads to less overlap between patterns in the internal representation.

The long-term mode uses the same network, but with a Boltzmann learning algorithm called *Mean Field Theory*, described by Peterson and Hartman [24]. Essentially, the Mean Field Theory output function is a continuous deterministic form of the Boltzmann machine output function. The outputs are continuously varying using a non-probabilistic sigmoidal function. MFT shapes its sigmoidal curve according to a global temperature parameter, analogous to the parameter to the energy functions used in algorithms like simulated annealing. The temperature parameter is gradually decreased during training, and as it decreases, the sigmoidal function approaches a step function.

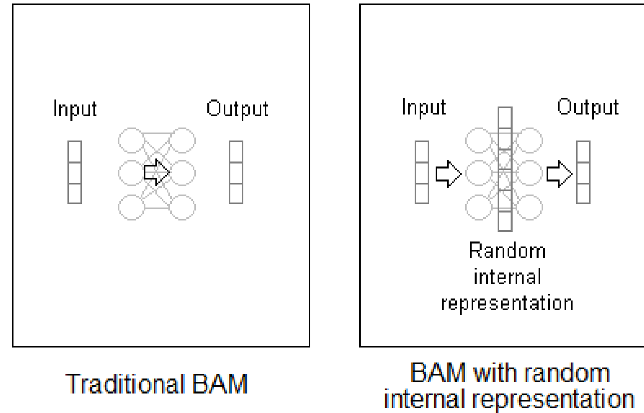

Figure 6: Illustration of Bairaktaris' random internal representation mechanism

Their article [21] demonstrated the effectiveness of the model by training the long-term memory with 7x5 images of letters from the Roman alphabet. The short-term memory was then used as a buffer for fast retrieval of patterns. The short-term memory has a fairly low capacity compared to the long-term memory. In the testing case with the alphabet, the capacity was 4-5 patterns without interference. This capacity can, according to Levy and Bairaktaris, be increased by increasing the number of hidden units. This increases the dimensionality of the internal representation, and should thus, according to Bairaktaris in [23], increase capacity for both the short-term and long-term memories.

What their demonstration does not do, is to actually measure the capacity of the long-term memory, and compare it to other known network models. This leaves it unknown whether or not the model actually has a higher capacity, and thus reduces catastrophic forgetting to some event. Also, it is worth noting that, although the capacity may be higher, it is not stated whether or not capacity is influenced by the orthogonality of patterns. It is fair to assume that at least the short-term memory is influenced by this, since it uses a modified Hebbian learning rule, although the rule has been modified to cope better with patterns that are not orthogonal.

As a conclusion, we can observe that the model of Levy and Bairaktaris was designed to have a higher capacity than other network, and that had properties that were complementary to human memory. The short- and

long-term memories have certain properties that can do this. Notably, the short-term memory is easily trained, but have a low capacity. The long-term memory is slow to train, but has a high capacity. It is unknown whether there actually is an increase in capacity, and it is also unknown whether this increase reduces catastrophic forgetting.

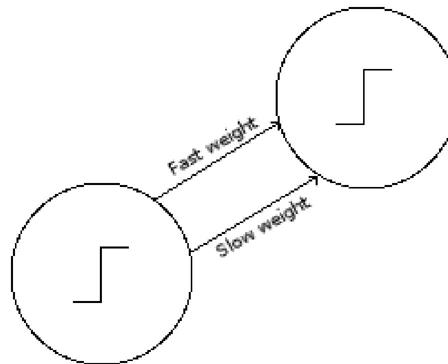

Figure 7: Two nodes using Levy and Bairaktaris' independent dual-weight model. The activation produced by the fast and slow weights are independent, but act in different time steps and leave activations behind to influence each other.

4.5 Dual-network models

Dual-network models try to reduce catastrophic forgetting by separating what is being learned from what already has been learned, i.e. separating the internal representations of a distributed network during learning. This class of networks have been implemented by several people, notably Ans & Roussett [25] and French [26]. Figure 8 illustrates a typical dual-network model.

The idea of using two networks comes from observations of human memory. These ideas are explained in an article by McClelland, McNaughton and O'Reilly [8], that outlines why there are complementary learning models in both the hippocampi and the neocortex of human brains. The article suggests that new information is learned in the hippocampi, and then slowly transferred to the neocortex. This reduces catastrophic forgetting. Still,

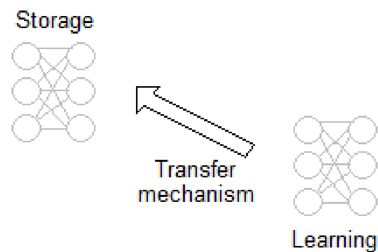

Figure 8: A typical dual-network model

the article leaves one big question unanswered; how is the transfer between these two regions done? Ans & Roussett and French came up with the same answer to this question, which is to use Robins' pseudopattern mechanism from [17] to transfer information from one network to the other. Although their solutions differ in some other aspects, this pseudopattern based transfer mechanism is the central aspect of both of them. Since Ans & Roussett's solution is somewhat more complex than French's, the rest of this section presents the network French developed to provide an understanding of the details of the operation of a dual-network. For details on Ans & Rousett's solution we refer to their article [25].

4.5.1 French's "pseudo-recurrent" dual-network model

In the dual-network model, one of the networks is used for short-term storage (i.e. learning). The other network is used for for long-term storage (i.e. memory). When training a dual-network of this type, the patterns to be trained are first fed through the network in a normal manner. The outputs from the long-term storage area are inhibited (not used), and only the outputs from the short-term storage area are used during back-propagation. However, before back-propagation takes place, a number of pseudopatterns (detailed in section 4.2) are generated by the network and fed through it. The outputs from the learning area are inhibited, and the outputs from the memory area are used instead. This set of pseudopatterns is then interleaved with the input pattern and its output from the learning area. The learning area is then trained to acceptance with this new set of patterns.

When French tested his network on sequential learning of 20 patterns [26], the network exhibited gradual forgetting in a very realistic manner. French, Ans and Roussett claim in a later analysis [27] that this shows that their dual-network models eliminate catastrophic forgetting. There are, however,

some memory mechanisms in the human brain that these networks will not model nicely. French, Ans and Roussett point out three important possible limitations. The first is related to snap-shot memories. Humans tend to remember in detail a certain event at a certain point in time. It is uncertain how well the dual-network model copes with this kind of memory.

Another important question is related to the way the transfer is done in the networks. Pseudopatterns have been shown to be a powerful mechanism for preventing catastrophic forgetting, but they do nothing to put the learning of a new pattern in context with similar patterns. It would be good, perhaps even optimal, if the mechanism for learning should retrain the patterns which have the largest risk of interfering with the new pattern as this new pattern is learned (i.e. the learned patterns that are least orthogonal to the new pattern). Pseudopatterns make no guarantees in this respect, and so it is still unknown whether this is a limitation or not. Note that this question also applies to the pseudorehearsal mechanism presented earlier, and to any other mechanism involving the use of pseudopatterns.

4.6 Important aspects of evaluation

With the different solutions presented, we see that there are differences in the approaches that are taken when the various solutions are tested. A good evaluation should perform as thorough tests as possible, to identify as many strengths and weaknesses as possible.

There are two aspects of the tests that are essential in understanding the capabilities of each solution:

- The characteristics of the input patterns that are used during testing
- The characteristics of the way the patterns are learned

Both these characteristics must be analyzed in the context of a given solution. The mechanism contained in the solution may or may not have an impact on the interpretation of them, and they may or may not be relevant at all for the solution.

4.6.1 Important input pattern characteristics

When analyzing input patterns, it is important to note whether the patterns are tailored in some way to match the solution that is being analyzed. Such tailoring may be a source of artificially good results, but may also be necessary to make the solution work at all. There is little sense in criticizing a tailoring when the lack of that tailoring will render the solution useless. If the tailoring, on the other hand, seems to lead to especially good performance, then one should certainly try to identify if and why the tailoring does so, and, if possible, try to find a way to remove this unfair biasing of the performance of the solution. Also, if the tailoring renders the solution less practical for certain applications, this clearly has an impact on the generality and usability of the solution.

It is also important to identify the orthogonality of the input patterns that each solution is tested with. If the input patterns are very orthogonal, catastrophic forgetting is less likely to occur at all. If they are less orthogonal, catastrophic forgetting is more likely to occur, but the occurrence will depend more on other characteristics of the solution.

A final aspect of the input pattern characteristics is the number of patterns that the solution is tested with. Like all memories, neural networks have saturation points. Different solutions to the problem of catastrophic forgetting may or may not have an impact on the saturation point of a network. Some solutions may not even have an obvious saturation point (such as networks that extend their topology to cope with the number of input patterns in one way or another), while others have a relatively small capacity before saturation occurs. It is therefore useful to test or analyze the saturation point of a solution. This saturation analysis must also be viewed in correlation with the performance of the network before saturation; certain solutions may be very well suited for solving problems of a certain size if catastrophic forgetting does not occur before saturation, regardless of the other characteristics of the input patterns.

4.6.2 Important learning characteristics

It is important to remember that some solutions do not differ between sequential learning and learning all patterns at once. This is the case for i.e. Hopfield networks, where there is no difference between learning all patterns at once or sequentially, since “all patterns at once” just trains each pattern

sequentially. The learning algorithm makes no incremental adjustments to the weights; the weights are calculated once and then stored. In general, all solutions with such a one-shot learning approach will exhibit the same behavior whether patterns are trained all at once or sequentially.

For solutions that do differ between sequential learning and all-at-once learning, i.e. the back-propagation method, it is of course essential to test the solutions capability to reduce or overcome catastrophic forgetting when learning patterns sequentially, since that is when catastrophic forgetting may occur. If all patterns are learned at once, the network will not exhibit any catastrophic forgetting until it has reached its saturation point. When patterns are learned sequentially, on the other hand, catastrophic forgetting may occur.

A final concern is the interpretation of the term sequential learning. The term does not mean that patterns must be learned one by one. Learning sets of patterns sequentially is also sequential learning. This is illustrated in figure 9. Since the latter type of sequentiality differs from the first, there is also a need to look at the type of sequentiality in correlation with characteristics of the input patterns, to identify possible features that may have an impact on the performance of the solution being tested. It may also be that certain input pattern sets are not meant to be trained one by one, but rather as smaller sets, because they are in some way correlated and have characteristics that make it natural to present them to a network in that way. In this case, training them one by one would not be realistic, and thus lead to a performance measurement that is not as relevant.

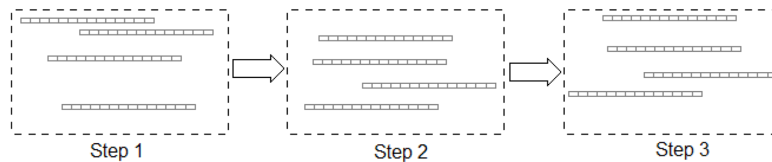

Figure 9: Three steps in learning of a sequence with four patterns in each set in the sequence. In each step the four items are learned concurrently.

4.7 Conclusion

This chapter has presented various solutions to the problem of catastrophic forgetting. We have grouped these solutions in a pragmatic manner after what they do to overcome the problem, and presented concrete implementa-

tions of each of them. Their strengths and weaknesses have been presented briefly, and according to what has been researched about them so far.

The solutions can mainly be grouped into four types:

- Rehearsal solutions
  - The Rehearsal Buffer Model
  - Sweep rehearsal
  - Pseudorehearsal
- Dual-weight architectures
  - Hinton and Plaut's "deblurring" network
  - Gardner-Medwin's model of recognition memory
  - Levy and Bairaktaris' high capacity dual-weight model
- Dual-network architectures
  - French's "pseudo-recurrent" dual-network model
- Others
  - Activation sharpening

Unfortunately, there exists no evaluation that compares the different solutions. It is likely that different solutions are more suited for a certain application than others, and that there is no "best" solution.

Solutions vary both in complexity and performance, but what can be said about them all is that in order to be sure of how good each of them are, formal proofs or extensive empirical testing is needed. We have looked at important aspects of the process of evaluating solutions. It is desirable to identify as many characteristics of the input set and learning method as possible, and consider all of them during evaluation. When evaluating an experiment that has been conducted to evaluate a solution, one should at least consider:

- Any tailoring of input sets that make them particularly suitable to the solution
- Orthogonality among the individual patterns in the input set
- The saturation point of a network when a given solution is used

- Whether or not sequential learning makes a difference in the network and solution being tested
- How many patterns there are in each set that the network is learning sequentially

There may also be other important characteristics that need consideration. Still, the guidelines provided here give a good foundation for evaluation of the different solutions.

5 Measuring catastrophic forgetting

Measuring the degree of catastrophic forgetting for various approaches and solutions in a uniform way is necessary for a fair, objective, and meaningful evaluation. However, given the diversity of these solutions both in terms of the way they work, the assumptions they make, and the architectures they act upon, measuring them all with a single method that takes all their performance aspects into account is hard, if not impossible. This section will examine various approaches to measure catastrophic forgetting, discuss their strengths and weaknesses, and suggest measurements that will serve part of the criteria used when evaluating the performance of the various solutions.

5.1 Measuring recognition of learned patterns

An obvious way to test a trained network for catastrophic forgetting is to test its ability to recognize patterns it has learned. To be able to do this, some form of error measurement must be used. This section describes various ways to measure errors in the recognition of patterns.

5.1.1 The average squared error measurement

For measuring recognition of a pattern, the average squared error measurement is often suitable. The average squared error is simply calculated from the formula

$$e = \frac{1}{n} \sum_{i=1}^{n} (o_i - d_i)^2$$

where e is the error, o_i is the output of the network at position i in a pattern, d_i is the desired output at position i in that pattern, and n is the length or number of elements in the pattern.

This measurement provides a good notion of how big the error for a particular pattern is. It should be remembered that in most cases, the network does not have an error-free performance on the old patterns in the first case. Therefore, when evaluating these forgetting measures it is important to take into account the errors that were measured at the moment where learning of an old pattern was stopped.

Averaging this error measurement over all patterns provides a fairly good measurement of the performance of the network for that set of patterns. To

evaluate catastrophic forgetting on the basis of this measurement, one can compare performance on an old set of patterns after a new set of patterns has been learned

1. Learn set A
2. Learn set B
3. Test average error on set A

This does not show any particular characteristics of the way the network has forgotten these patterns, such as how much worse the performance is on the *first* or oldest pattern in the set of old patterns compared to the *newest* pattern in that set. Serial position plots, which will be discussed later, can be used to fill in some of these blanks.

5.1.2 Relative amount of wrong outputs

Another way to measure performance is to calculate the number of wrong outputs across all nodes for each pattern, and then averaging that number across all patterns. To calculate the relative amount of wrong outputs, one simply calculates the number of wrong outputs for each pattern, divides that by the size of the pattern, according to a suitable criterion, and averages that number across all patterns.

This method gives an estimated measurement of how many of the patterns that are incorrect, and an accurate measurement of how many outputs that were wrong. If we get a value of 0.3, this means that 30% of the outputs were wrong.

5.1.3 Amount of correctly recognized patterns

A solution is never better than the number of patterns it recognized correctly. A network that does not exhibit catastrophic forgetting should recognize all patterns it has trained on correctly. Measuring how many patterns it recognizes is done by simply feeding all the patterns through the network, and counting the number of patterns that are not recognized. Each pattern is said to be recognized if all outputs are within a chosen and suitable criterion.

5.1.4 A change-in-error measurement

We suggest a measurement that tries to take into account the original errors on the old patterns. This measurement calculates the difference between the original error for a pattern, and the squared error for that pattern after the new pattern has been learned.

The measurement is calculated as follows:

$$e = \frac{1}{n} \sum_{i=1}^{n} \left[(o_i - d_i)^2 - (o_{i_{old}} - d_i)^2 \right]$$

where e is the error, o_i is the output of the network at position i in the pattern, $o_{i_{old}}$ is the old output of the network at position i, d_i is the desired output at position i in the pattern, and n is the length or number of items in the pattern.

If this measurement has a result greater than zero, then forgetting of the pattern has occurred. If the result is zero, then no forgetting has occurred, and if it is less than zero, the performance of the network has actually improved. The scalar interpretation of the value calculated by this measurement depends on the range and interpretation of the pattern for which the measurement was used. Generally, the greater the number is, the more forgetting has occurred.

5.2 Serial position plots

A serial position plot is a measurement that is done after all training is completed. It shows the average error of each pattern on the y-axis, and the *serial position* of that item on the x-axis. This measurement was first presented by Ratcliff in [4].

The plot is done by putting all the learned patterns in a sequence, starting with the one that was first learned (or the first pattern in the first set that was learned). The error of this pattern is plotted at $x = 1$. Then the error of the next pattern is plotted at $x = 2$, the error of the next after that at $x = 3$, and so on.

The resulting graph shows the performance of the network on all patterns known by it. Since the patterns are plotted in series, the plot gives a good view of any temporal effects on performance, which is very interesting when dealing with sequential learning.

An example of a serial position plot can be seen in figure 10, which shows the performance of the network in Robins' article on each individual pattern after learning an initial set of 20 patterns, and then 10 new items one by one (i.e. sequentially) without any mechanism to prevent catastrophic forgetting.

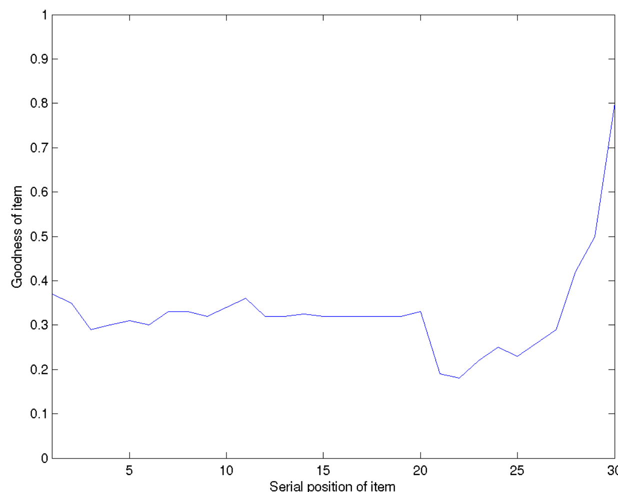

Figure 10: Example serial position plot of the network in Robins' experiment [17] without prevention of catastrophic forgetting

5.3 The standard discriminability statistic

The standard discriminability statistic was suggested by Ratcliff in [4]. It was designed to measure how well a network discriminates old patterns from new patterns.

First of all, this measurement requires that all patterns are normalized to the range $[-1, 1]$. The dot product between a given input pattern and its corresponding output pattern is calculated, and divided by the length of the pattern to provide a match value. The higher the match value, the better the recognition of the pattern (intuitively, a match value of zero means the input and output patterns are orthogonal and thus as unrelated as they can be). The mean match value for all items is calculated across several Monte Carlo trials[5].

[5] Monte Carlo trials are trials that test a physical, deterministic system using a statistical approach. For more information on Monte Carlo methods, see [28].

Now, when a new set of patterns is introduced to the network, the mean match for that set after some degree of learning is calculated in the same manner. These means are then used to calculate what Ratcliff named the standard discriminability statistic, old-new discriminability, or d' measurement:

$$d' = \frac{\mu_{old} - \mu_{new}}{\sigma_{new}}$$

Here μ is the mean and σ is the standard deviation of either old and new items, as indicated by their subscripts.

To provide an understanding of what d' is, we must take a look at its components and see what they describe when related to each other in this manner. The d' measurement can be interpreted as a variation on the well-known z-value for samples on a Gaussian distribution. The z-value is defined as

$$z = \frac{x - \mu}{\sigma}$$

Here μ is the mean and σ is the standard deviation, and x is the sample to be measured. The z-value describes how far the sample x is from the mean in terms of standard deviations. Consider an example where $\mu = 1000$, $\sigma = 10$, and $x = 900$. The z-value will then be $z = \frac{900-1000}{10} = -10$, which means that x lies 10 standard deviations below the mean.

When using the mean of the old set of patterns, μ_{old}, as the sample for which the z-value is calculated, the z-value becomes d', and describes how far the mean of the match for old patterns lies from the mean of the match for new patterns in terms of standard deviations for matches for new patterns. In other words, it shows what happens to the ability the network has to match old patterns when it learns new patterns. The higher d' is, the larger the difference between the old match value and the new match value is.

When the d' measurement was used by Ratcliff in [4] to analyze the Buffer Rehearsal Model, as presented in section 4.1, the old patterns were those in the rehearsal buffer, and the new items were those to be learned. Ratcliff found that d' was always positive when using his model, meaning that the old patterns were kept in memory, and that the network on average performed better on those than on the new patterns.

Finally, we will try to make some predictions on what we will see when d' is used as a measure for catastrophic forgetting. Intuitively, we would expect d' to be negative when new items are trained to criterion without further retraining the old patterns, as long as the orthogonality between old and new patterns is low. When the orthogonality is high, a network should be

able to distinguish old patterns from new patterns nicely, and d' should be close to zero. This reasoning shows that it is important to relate d' to the orthogonality among the two sets.

Also, when retraining old patterns or taking some other precaution to preserve them, we would expect the d' value to be above zero if, on average, the old patterns are preserved more than the old patterns are learned. This means that a network is somewhat unable to learn new patterns, as it is trying to retain its already learned knowledge as much as possible. This was named *catastrophic remembering* by among others Robins in [16]. If the d' value is below zero, this should mean that average performance on old patterns are worse than performance on new patterns, meaning that the network has forgotten its old patterns to some extent. Again, the ideal case is when d' is close to zero, which means that the balance between remembering old patterns and learning new patterns is optimal.

5.4 Conclusion

The above sections have presented various ways of measuring catastrophic forgetting. Each of them provide a measurement of the error a network displays when fed with a particular pattern or a set of patterns.

There is no best method or most complete method; each described method gives its own notion of error and how errors relate to each other. We therefore suggest that several measurements should be used when the different solutions are evaluated, to provide an as full picture as possible of the performance of the network. This performance should then be tested empirically across a wide range of different applications, and reviewed according to and in correlation with the criteria given in chapter 4, providing an as good as possible understanding of the quality of the solution being tested.

6 Evaluation of solutions

In this chapter we will reproduce two selected solutions to the problem of catastrophic forgetting. The chapter is organized into four main sections. The first gives a justification for which solutions we chose to reproduce. This justification is followed by a section for each of the evaluations we have conducted. Each of these sections starts by giving a description of the experiment that is done, followed by a presentation of the results from that experiment. After that, a discussion of the results is given. This discussion is mainly aimed at comparing our results to those obtained in the original experiments. Finally, we provide a more generalized discussion about the solutions and a conclusion for further work.

6.1 Chosen solutions and justification

The solutions we have chosen are:

- Robins' pseudorehearsal solution (See section 4.2)
- French's activation sharpening (See section 4.3)

The first solution we chose is the pseudorehearsal solution. We chose this solution for several reasons. First of all, this solution has been in use since it was developed by Robins. Both French and Ans & Roussett used it as the mechanism for transferring information between the two networks in their dual-network models, discussed in section 4.5. It has also been extensively analyzed by Frean and Robins [12]. In addition , it can be compared to a wide range of other experiments, namely Ratcliff's Rehearsal Buffer Model (discussed in section 4.1.1), Robins' sweep rehearsal solution (discussed in section 4.1.2, and the original pseudorehearsal experiments done by Robins (discussed in section 4.2). Because it remains relevant for the latest developments and is possible to compare to a great amount of research results, it is a natural solution to reproduce.

Activation sharpening has been chosen for two main reasons. First the paper with the experiment we are reproducing [13] is the first reported experiment by Robert M. French, one of the main figures in the field studying catastrophic interference. Even though activation sharpening has not fueled as many related studies as pseudorehearsal, many of French's later papers as well as other studies use results from this experiment. Second, the paper is

one of the first to point out the importance of orthogonality and sparseness in the artificial neural representations, and also provides a measuring method to help analyze this.

6.2 Pseudorehearsal

This section presents our reproduction of Robins' pseudorehearsal solution [17] by first providing a description of the original experiment, our interpretation of it, and differences between that experiment and our reproduction. Finally, we present the results from our experiment and provide a comparison of our results to those found in the original experiment.

6.2.1 Experimental description

The original experiment performed by Robins used a standard feed-forward back-propagation network with a single hidden layer. The input layer had 32 nodes, the hidden layer had 16 nodes, and the output layer had 32 nodes. The learning rate of the network was set to 0.3, and the momentum was set to 0.5. The training was stopped when an item was learned so that its output was within 6% of the desired output. This was the same network configuration as Ratcliff used when testing his Rehearsal Buffer Model [4].

The network was first trained to learn a set of 20 binary items, in which the patterns consisted of binary values that were set to either 0 or 1 at equal probability. The network was supposed to learn these items both in an auto-associative manner, e.g. learn to reproduce the input pattern at the output nodes, and in a hetero-associative manner, e.g. learn to reproduce a (possibly) different output pattern for a given input pattern.

After the network has learned the initial set of 20 binary input patterns, a set of pseudopatterns was generated. The size of this set was varied to be either 8, 32, or 128 during the experiment. The pseudopatterns were generated in the same way as the original items, and fed through the network to produce their corresponding output. These pseudopatterns and their outputs were then stored for use when introducing new items sequentially.

To test the ability pseudorehearsal has to alleviate catastrophic forgetting, 10 new items were presented to the network *one by one* (i.e. sequentially). These training sequences are referred to as *intervening trials* by Robins and Ratcliff. Each of these new items were learned along with 3 pseudopatterns, randomly

selected from the generated set. The resulting training set, of size $n = 4$, was then trained to criterion on the network. This way of using pseudorehearsal is termed *random pseudorehearsal* by Robins, since it randomly selects 3 pseudopatterns from the total set of pseudopatterns for each new item to be learned. After each of these intervening trials, the network's performance on the 20 original items was tested to see if its ability to recognize them had decreased, and how much it had decreased.

Robins also tested *sweep pseudorehearsal* to improve random pseudorehearsal, in which he selected the pseudopatterns to be used in training *for each epoch*, just like sweep rehearsal (described in section 4.1.2) does to improve the Rehearsal Buffer Model. We decided to test only the random pseudorehearsal solution, since that is the one that is most similar to the rehearsal experiments done by Ratcliff, and thus has more material available for comparison. Although Ratcliff only used auto-association in his experiments, we decided to do the hetero-associative experiment as well to test the random pseudorehearsal solution as thoroughly as possible.

The pseudo code in listing 1 provides a short, concise description of the random pseudorehearsal experiment we will conduct.

Listing 1: Pseudorehearsal pseudo code

```
function pseudopattern(poolSize):
  # Generate auto- or hetero-associative items
  initialSet := generateDataItems(20)

  # Learn initial patterns
  train(initialSet)

  # Test the performance on those 20 items
  performance[0] := testOn(initialSet)

  # Generate a pool of 8, 32, or 128 pseudopatterns
  pseudoPool := generatePseudo(poolSize)

  for i in 1 to 10:
    # Generate new auto- or hetero-associative item
    newItem := generateDataItems(1)

    # Learn the item along with three randomly
    # selected pseudopatterns
    newSet[0] := newItem
    for j in 1 to 3:
      newSet[j] := randomFrom(pseudoPool)
    train(newSet)

    # Test the networks performance on
    # the original 20 items
    performance[i] := testOn(initialSet)
```

When attempting to reproduce Robins' experiment, we made some adjustments to the original experiment. These adjustments were the following:

1. **Random and/or sweep pseudorehearsal**
   We decided to test only the random pseudorehearsal solution, since that is the one that is most similar to the rehearsal experiments done by Ratcliff, and thus has more material available for comparison. Although Ratcliff only used auto-association in his experiments, we decided to do the hetero-associative experiment as well to test the random pseudorehearsal solution as thoroughly as possible.

2. **Learning rate**
   When training with Robins' learning rate, we could not get the network to converge properly. We found that this was a consequence of the parameters used by Robins. In cases where a single item disagreed on a certain output compared to the other patterns, the network would often get stuck in an optima that made its output correct for the majority of the items in the set. For instance, when 19 of 20 items wanted the output of node 3 to be 1, the output of node 3 would move closer and closer to 1, even though the last item wanted the output to be 0. This is partially a consequence of the high learning rate used, and lowering it from 0.3 to 0.1 made the network converge. The other reason is the strict training criterion. Both of these values were lowered gradually until the network converged in all our tests.

3. **Acceptance criterion**
   We interpreted Robins' criterion as meaning that an output had to be equal to or larger than 0.94 to be interpreted as a 1, and equal to or less than 0.06 to be interpreted as a 0, since 0.06 is 6% of the range of possible outputs.

   We discovered that the trouble we had making the network converge was partially due to this strict training criterion. Training to such strict values tends to make the weights too adapted to some of the patterns, which makes the network use a very long time to adjust its weights in cases when a single item disagrees with all the other patterns on a certain output. To avoid this, we lowered the criterion from 6% to 20%, which, along with the lowered learning rate, made the network converge successfully. This means that an output had to be equal to or larger than 0.8 to be interpreted as a 1, and equal to or less than 0.2 to be interpreted as a 0.

6.2.2 Measuring performance

For reviewing the network on recognition of the base population after each new pattern is introduced, a plot of the networks performance on the base population was used. The plot shows the performance on the base population on the y-axis after learning n items sequentially, indicated on the x-axis. In other words, the value for $x = 0$ indicates the performance on recognition of the 20 items after having learned only those, $x = 1$ indicates the performance after learning one new item, and so on. All tests were averaged over 50 runs to eliminate any anomalies.

The changes we made to the criterion for accepting items as learned meant that we had to change the way we measured errors as well. We chose to use an error measurement that is more suitable to our criterion; a measurement that counts the number of incorrect outputs for a given item. This measurement method was also described and discussed in section 5.1.2. As previously mentioned, we used a criterion of 20% in our experiment. This translates to accepting output values equal to or less than 0.2 as 0, and accepting output values equal to or greater than 0.8 as 1. Our measurement measures the average number of nodes with incorrect output for each item, and then averages that across all patterns. This means that our measurement actually shows how many percent of the total number of outputs that are wrong across all items.

A consequence of our choice is that the numbers from our measurement are not directly comparable to those in Robins' experiment. What is comparable, though, is the shape of the graph, i.e. way the performance develops as the network is trained on intervening items.

6.2.3 Results

We started with hetero-associative items. To have something to compare the pseudorehearsal results to, we first ran the experiment without pseudorehearsal, simply introducing the 10 new items one by one. We expected the performance of the network to decrease rapidly as more of these items were introduced. As pseudorehearsal was introduced, we expected the network to perform better, and the performance degradation to happen more gradually. Figure 11 shows the performance of the network without pseudorehearsal, and with pseudorehearsal and different numbers of pseudopatterns.

As seen in the figure, without pseudorehearsal, performance decreases rapidly as new items are introduced sequentially. We see a rapid fall in performance on the original items after introducing only one new item to the network. The decrease in performance continues, and the network ends up with about 73% correct outputs across all the original items.

In Robins' experiment, the mean goodness of the base population was 0.35 after training on the 10 new items sequentially. In our experiment, 73% of the outputs are correct after training. In other words, although the numbers are not directly comparable, seems that our network performs better than Robins does. The explanation for this lies in our requirements for accepting an item as learned, which are more relaxed than in Robins' experiment.

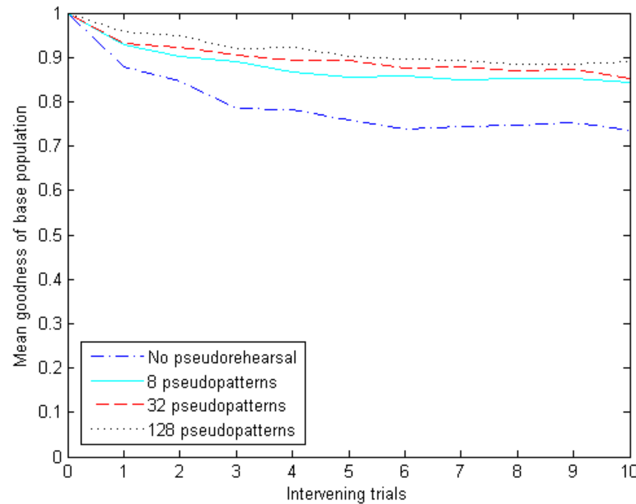

Figure 11: Comparison of random pseudorehearsal with 8, 32, and 128 binary pseudopatterns when training on hetero-associative items

In spite of exhibiting a performance that seems somewhat better then what Robins' network shows, the network still shows a rapid decrease in performance as the first items are trained. We thus conclude that this behavior is satisfactory.

When testing with pseudorehearsal, we first used 8 pseudopatterns from which 3 were selected for use each time a new intervening item was learned. The graph in figure 11 clearly shows that performance has improved. Compared to the performance without pseudorehearsal, the decrease develops more gradually. The network ends up with about 84% correct outputs on the original items, which is a clear improvement. In Robins' experiment, the network ended up with a goodness of about 0.6 on the original items. This is an increase in goodness of about 25% compared to the performance without pseudorehearsal. Our performance increase is about 15%, and is thus less than what Robins achieved. Again, we think that this difference emerges from the adjustments we made to the learning rate and criterion.

With a larger pseudopattern set, this time consisting of 32 items, there are no major differences from when there were only 8 pseudopatterns. This time, we end with about 85% correct outputs, which gives an improvement of 14% compared to the performance without pseudorehearsal. Robins' experiment showed the same behavior for 32 pseudopatterns as it did for 8, and it seems

that our network behaves in the same way.

Finally, when we increase the number of pseudopatterns to 128 the network ends up with about 89% correct outputs. This is a bit better than what was the case with 8 and 32 pseudopatterns. It is also a bit different from the results Robins achieved, which showed no difference in performance for different numbers of pseudopatterns. Robins explains his consistent performance with the fact that random pseudorehearsal uses only 3 of the generated pseudopatterns no matter how many pseudopatterns there are to choose from. We argue that this is incorrect; the 3 pseudopatterns are chosen from the set each time a new introduction is used. Thus, it is likely that the network will be trained on a better approximation of the original data when using a larger number of pseudopatterns. Our conclusion is thus that the behavior we see is correct, and aligns with our own expectations.

Having seen that the behavior of the network with and without pseudorehearsal for hetero-associative items, we now turn to the auto-associative items. Again, we first tested the performance without pseudorehearsal, and then with pseudorehearsal, using the same numbers of pseudopatterns. Figure 12 shows the development in these cases.

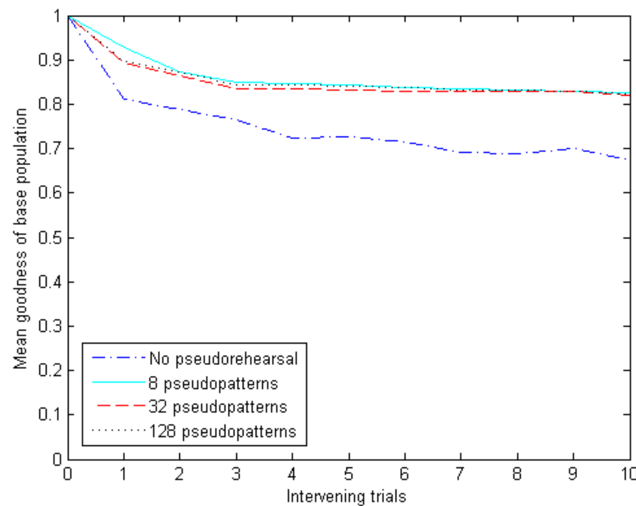

Figure 12: Comparison of random pseudorehearsal with 8, 32, and 128 binary pseudopatterns when training on auto-associative items

Without pseudorehearsal the network ended up with getting about 67% of the outputs correct when tested on the original items. This is a low per-

formance compared to what we saw for hetero-associative patterns, but low performance also indicates that the network behaves as expected.

Moving on to testing with pseudorehearsal, we tested the network with 8 pseudopatterns in the training set. The network again showed a substantial increase in performance, and now gives 82% correct outputs. This is a performance increase of 18% compared to the performance without pseudorehearsal. The network ends up with the same performance with 32 pseudopatterns.

Finally, when testing with 128 pseudopatterns, the network improved a bit more, and ended up with giving 83% correct outputs. As we argued in the hetero-associative case, this is what we expected, and a plausible behavior for the random pseudorehearsal solution. The results are as expected, and satisfactory, although the differences between different numbers of pseudopatterns are not as large as for the hetero-associative case.

6.2.4 Discussion

As the results show, we achieved improved performance when using random pseudorehearsal, when compared to standard back-propagation learning without any rehearsal mechanism. Our relative performance increase, however, is less than what Robins achieved. We conclude that the reason for this lies in the adjustment of the criterion we did. Since we had a more relaxed criterion than what Robins had, our network accepted worse performance than what Robins' network did. It is likely that this led to the smaller increase in performance. Still, we observe essentially the same behavior as Robins did: The network performed far better with pseudorehearsal than without it, and the number of pseudopatterns used has little effect on the performance of the solution.

Robins argued in his article that a neural network can be viewed as a function approximator, and discussed his solution when viewing a network in this way. In this discussion, he pointed out that when the network makes a function approximation that is partially based on noisy data, the pseudorehearsal solution is likely to achieve less increase in performance, since it will sample the noisy data as well as the actual data, and thus train the network partially on the noise as well [17]. The relaxed criterion used in our reproduction of his experiment will make the function approximation of the network less accurate. Although this less accurate approximation is not the same as noise, it is still less accurate and thus gives the same behavior as noisy data may do.

The pseudorehearsal solution may thus, when applied to the network with the parameters in our experiment, sample inaccurate points of the function the network has approximated. These “bad” pseudopatterns will contribute to making the network perform worse on the original items.

It is worth noting that sweep pseudorehearsal, which Robins also tested, selects new pseudopatterns from the set once for each epoch, just as the sweep rehearsal mechanism described in section 4.1.2 does. Although we did not test sweep pseudorehearsal, it is likely that it is less sensitive to the “bad” pseudopatterns when the set of pseudopatterns is “large enough”, since the noisy bits will then have less emphasis on the resulting function approximation. This would probably give a better performance in cases like ours, and as Robins shows in his article, the sweep pseudorehearsal solution performs better than the random pseudorehearsal solution in general.

6.2.5 Conclusion

We conclude that Robins’ pseudorehearsal solution alleviates catastrophic forgetting, and performs as expected and showed by us and by Robins. Although the learning rate and criterion we used are different from what Robins used, and we use a different method for calculating performance or goodness, we still see the same developments in performance as Robins did in his tests. The network rapidly forgets items without pseudorehearsal, and forgetting occurs more gradually and less catastrophic when using pseudorehearsal.

6.3 Activation sharpening

The second experiment we reproduced was the activation sharpening experiment done by Robert M. French. The results from that experiment were originally presented in his 1991 article *Using Semi-Distributed Representations to Overcome Catastrophic Forgetting in Connectionist Networks*[13]. We will begin by introducing our reproduction of his experiment, along with any differences between the original experiment and our reproduction. we then continue by presenting our results, before concluding with an analysis.

6.3.1 Experimental description

As described in section 4.3, the activation sharpening algorithm works by calculating a new "sharpened" output value for each node based solely on the output value itself in addition to a sharpening factor. The nodes with the highest activations in the output layer will have their output further increased, or sharpened, while the rest of the nodes will have their output decreased. The formulas for this was:

$$A_{new} = A_{old} + \alpha(1 - A_{old}) \quad (9)$$

$$A_{new} = A_{old} - \alpha A_{old} \quad (10)$$

The difference between the sharpened activation and the old activation was used as an error and back-propagated through the network. After this was done, a regular back-propagation sweep was performed.

The network used for the experiment was a 3-layered fully-connected network with 8 nodes in each layer. The momentum was set to 0.9, the learning rate 0.2, and the *sharpening factor* α to 0.2.

The network was trained to learn 11 different associations, then presented with a 12th association to learn. When the network had learned this new association, one of the original 11 associations was chosen and the network was tested on this. The number of nodes to sharpen was varied from 1 to 7. In the graphs later presented 0 sharpened nodes means that standard back-propagation and not activation sharpening was used as a training algorithm.

Several parameters and details were missing from French's report regarding the experiments. We have made the following assumptions in our tests:

1. **Encoding of input and output patterns**
   French gives no indication of how this is done. We have chosen to use *binary encodings* of both patterns in the data items.

2. **Auto- or hetero-associativity**
   French does not state how the associations between the input and output patterns should be made, i.e. if the input pattern and the output pattern should be the same or not. The fact that the network used has the same number of nodes in the input and output layers suggests auto-associativity. We have tested both, but after seeing no significant difference in the results we have concentrated on hetero-associative data items. The reason for this is that the set of hetero-associative data

items is a superset of the set of auto-associative data items. By using the larger possible set of data items we reduce the chance of the network only being suitable for specific situations, in particular we are making sure we are not training a network only to learn a function similar to multiplication with the identity matrix[6].

3. **Distribution of patterns**
French gives no description about how the data items are distributed in the 8-dimensional input space formed by both the eight input nodes As we showed in section 3.3, the orthogonality of the inputs are critical to the amount of overlap and thus also interference in the network. We have chosen to pick patterns randomly from a uniformly distributed population.

4. **Learning rate and momentum in the propagation step of the activation sharpening**
No mention of the use of learning rate or momentum in this step was made. French states that a learning rate and momentum was used, but not if this was only for the complete back-propagation following the activation sharpening or for that as well as the activation sharpening propagation itself. After testing with neither, with only learning rate and with learning rate and momentum, we concluded that only a learning rate was used. If no learning rate (or a learning rate equal to one) was used, results deteriorate dramatically and activation sharpening performs worse than back-propagation on French's tests. If a momentum term was used, the network would fail to reach acceptance during the initial training. We have used a learning rate equal to that used in the back-propagation settings and no momentum in all later tests. This weight update rule is shown in listing 2

Listing 2: Activation Sharpening weight update rule

```
function weightUpdate(Weight weight):
  newVal := learningRate *
              (weight.to.Anew-weight.to.Aold) *
              weight.from.output
  return newVal
```

[6]An identity matrix of size n is the nxn matrix that has 1 in the diagonal from top-left to bottom-right and 0 elsewhere. When a vector of size n is multiplied by its identity matrix, the result will be the vector itself. When training a neural network on auto-associative data items, it is important to make sure one is not simply creating the a network representing the identity matrix as that does not possess any generalization possibilities.

5. **Acceptance criterion**
   French does not specify how acceptance of an association is established, and thus also not when to stop training. Since this is a fundamental aspect of the experiment, this is also likely to have great impact on the results of the tests themselves. We have chosen to used the same acceptance criterion as we used in the pseudorehearsal tests: To accept any output higher than 0.8 as a 1 and any output lower than 0.2 as a 0. A data item is correctly classified if all outputs are correct, and a set of data items is trained if they are all successfully classified.

To test the network performance French introduced a measurement method where the number of epochs needed to relearn an origin pattern was used. We have used the same measurement in these experiments.

French also introduced what he termed *activation overlap*. The idea of this measurement is to tell how much the information about a set if data items overlaps in the internal representation in a network. It is calculated by taking the average of the minimal activation level of each hidden node. As an example, consider a network with three hidden layer nodes where we want to calculate the overlap for two associations. For the first input, the activation is (1.0, 0.1, 0.3) and for the second (0.2, 0.0, 0.5). The activation overlap is then (0.2+0.0+0.3)/3 = 0.17

To give a more detailed description on the experiment, we refer to listing 3 for the activation sharpening algorithm used.

Listing 3: Activation sharpening algorithm

```
function activationSharpening():
  nodes := hidden layer nodes

  # Sort nodes by output value
  sort(nodes)

  # Sharpen the k nodes with highest activation
  for i in 1 to k:
     nodes[i].Anew := sharpen(nodes[i].Aold)

  # Blunt the other nodes
  for i in k+1 to n:
     nodes[i].Anew := blunt(nodes[i].Aold)

  # Propagate the difference between new and
  # old activation to the weights between the
  # input and hidden layer
  for i in 1 to n:
    for each weight where weight.to==nodes[i]:
      dw := weightUpdate(weight)
      weight := weight + dw

  # Perform usual back-propagation
  backprop()
```

6.3.2 Results

Before introducing the experimental results we saw in our experiments, we want to provide a summary of French's own results as a basis for comparison.

With regards to the number of epochs required to relearn the original associations, French reported that this would go from 330 without activation sharpening, drop to 81 and 53 for one and two sharpened nodes respectively and then again raise to again to approximately 175 for 3-node sharpening before reaching levels above that of standard back-propagation when sharpening four or more nodes.

As mentioned in section 4.3, French suggests that the optimal number of

nodes to sharpen is the smallest k such that

$$m < nCk \tag{11}$$

where m is the number of data items to learn, n is the number of hidden nodes, and C is some unknown constant. In other words, the optimal number of nodes to sharpen is dependent only on the network structure and the number of data items to learn.

Our test on the number of epochs needed to relearn a pattern is shown in figure 13. Here we see that the results are better if trained with a learning rate of 0.2, as opposed to no learning rate (or a learning rate of 1). If a momentum term is added to the weight function as well, like that in listing 4, the network would fail to converge during training.

Listing 4: Activation Sharpening weight update rule with momentum term

```
function weightUpdate(Weight weight):
  newVal := learningRate*
            (weight.to.Anew-weight.to.Aold)*
            weight.from.output
            + weight.lastChange * momentum
  weight.lastChange := newVal
  return newVal
```

However the variation in the results are large, with a standard deviation ranging from 100 epochs with one sharpened node and using a learning rate to 430 epochs for six sharpened nodes without learning rate. Even so, the main features of the curves are consistent throughout all trials: 1-node sharpening always show the best results, 2-node sharpening gives about the same results as regular back-propagation, and more sharpened nodes give worse results.

We also calculated the activation overlap in the trained networks. Our results here show no significant difference in the overlap as defined by French. For all tests with 0-7 sharpened nodes, the overlap differs between 0.25 and 0.30 on averaged runs having standard deviation of 0.1

6.3.3 Discussion

Our experiments clearly show the same improvement in epochs needed to re-learn a pattern when using activation sharpening as French reported. Figure

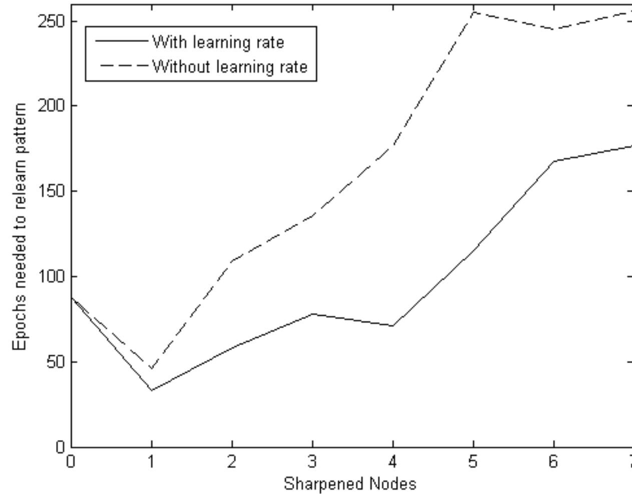

Figure 13: Number of epochs needed to relearn a pattern. Tested with different parameters for the activation sharpening propagation step. The graphs are the results of 50 averaged runs.

14 shows how both our and French's experiments with the results scaled to show back-propagation at a reference level of 100. As it can be seen from that figure, in our experiment the improvement is highest when only one node is sharpened, as opposed to two nodes as French reports. Since the network structure and the number of data items to learn is the same in our experiments as in those conducted by French, French's theory given in equation 11 seems to be lacking. As reported in section 6.3.1 pattern encoding, propagation parameters, pattern distribution, and acceptance criterion are all possibly different in our experiment. Since those are the possible differences between the experiments, one or more of those must also account for the change in the optimal number of nodes to sharpen from two in French's experiments to one in ours.

With regards to the activation overlap our results differ greatly from those reported by French. Where he reported curves that somewhat followed the curves of epochs needed to relearn the pattern, our results show *equal activation overlap for any number of sharpened nodes.* It is not certain what the reason for this difference in results stem from, but again the reason it must come from one of the earlier mentioned points where our experiments and French's might differ.

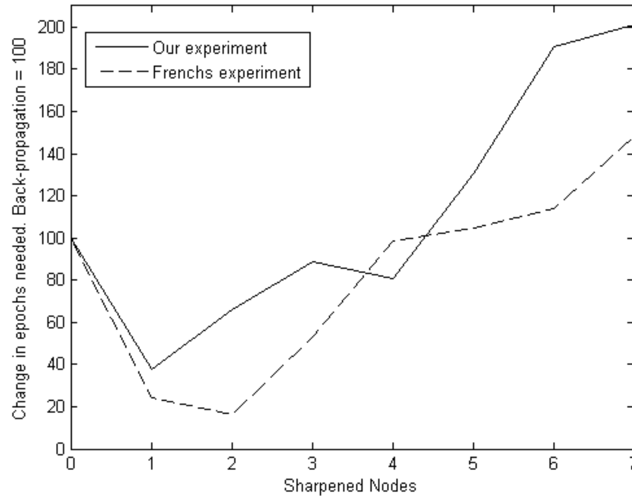

Figure 14: Improvement in epochs needed to relearn pattern. Back-propagation is set at reference level 100 and other results are scaled accordingly.

Since the activation overlap is a result of the input patterns being used, it is our theory that how we create the input patterns cause the lack of change in activation overlap. We select our input patterns from a uniform distribution with binary patterns. Patterns that are not randomly and uniformly selected will exhibit different activation overlaps. Also, selecting from a larger set of patters, by for instance using real-valued inputs instead of binary inputs, will make it possible for the network to discriminate between the input patterns better as there will be more differences.

6.3.4 Conclusion

We have showed that French's reported results in general are reproducible. We have seen that the network can have a reduction in relearning time when activation sharpening is used. We have also argued that French's theory that only network structure and the number of data items to learn determines the optimal number of nodes to sharpen is not correct. Our experimental results support that claim.

Our experiments also show that activation sharpening not necessarily results in a reduced activation overlap. In fact, our experiments with uniformly distributed binary input patterns show no significant change at all in the

activation overlap after applying the activation sharpening algorithm.

6.4 Comparison of the two solutions

Now that we have tested both solutions, we turn to comparing them against each other. To our knowledge, this has never been done before. To compare the two solutions against each other, we used the measurement from Robins' experiment on French's activation sharpening, and vice versa. We also counted the number of correctly recognized patterns after training on 10 intervening items for pseudorehearsal, and after 1 intervening item with activation sharpening. This measurement shows how many patterns that are actually recognized during each of the experiments. A comparison will also be made between the number of correctly recognized patterns after 1 intervening item has been presented with pseudorehearsal and activation sharpening.

6.4.1 Pseudorehearsal with French's measurement method

When testing the pseudorehearsal solution with French's measurement method, we used French's network with hetero-associative patterns, and generated and used pseudopatterns as described in the pseudorehearsal experiment. To measure performance, we used French's measurement method, which measures how many epochs it takes to learn a randomly chosen pattern from the original set after an intervening item has been learned by the network. As in the pseudorehearsal experiment, all results were averaged over 50 runs.

When testing without pseudorehearsal, the network learned the original pattern after about 79 epochs on average. When using 8 pseudopatterns, the required number of epochs was down to about 24. In other words, the improvement is drastic. For 32 pseudopatterns the result was also about 24 epochs, and finally, for 128 pseudopatterns, the network used about 18 epochs on average to relearn the original item. The results are shown in figure 15.

6.4.2 Activation sharpening with Robins' measurement method

When testing Robins' measurement method on a network trained with activation sharpening, we followed the procedure given in section 6.2.1, but as we had no new patterns (i.e. pseudopatterns) to provide the network with, each intervening set of patterns consisted only of one new association to learn.

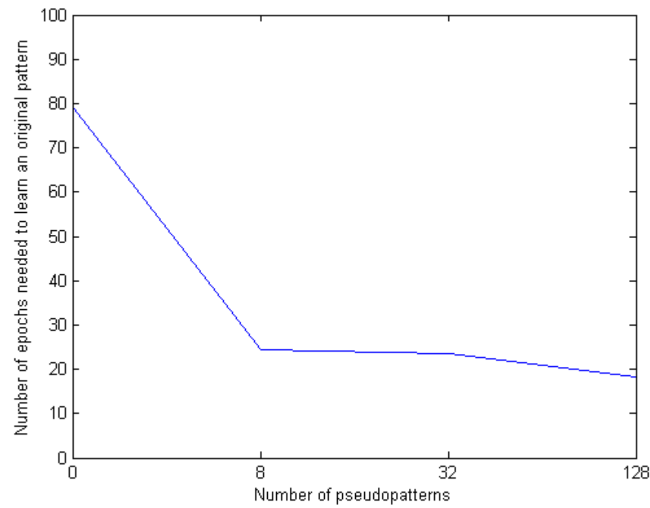

Figure 15: Performance of random pseudorehearsal in terms of the number of epochs required to relearn one of the original items after learning an intervening item.

The experimental results can be seen in figure 16.

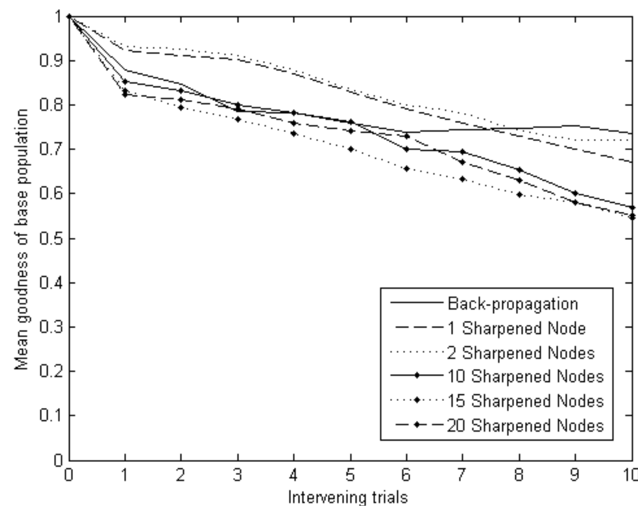

Figure 16: Performance of a network trained with the activation sharpening algorithm, measured in mean goodness as defined in section 6.2.

Here we see a performance that is quickly degrading to below that of regular

back-propagation. It only outperform back-propagation when sharpening less than four nodes, and when sharpening 10, 15, and 20 nodes is performance significantly lower than back-propagation.

Another noticeable feature of the graphs is that while performance is comparably high for the first few interleaving trials, it degrades rapidly after the first 2-3. Even though these experiments are too limited for us to draw generalized conclusions, they indicate that the *older* data, i.e. the data which it is longer since the network learned, is less stable when using the activation sharpening algorithm (see 3.1 for a discussion on stability-plasticity) than when using either regular back-propagation or pseudorehearsal. This is likely to be due to the increased alterations done in the existing weights when learning a new pattern. Our results measuring the activation overlap earlier indicates that the algorithm has failed to distribute the information, and that there is still significant information overlap in the network. Since the activation sharpening algorithm now performs two passes of back-propagation and thus weight alterations, instead of only one as back-propagation does, more information will be lost in the weights as well.

6.4.3 Correctly recognized patterns with pseudorehearsal

As we have shown, measuring the average goodness as the relative amount of wrong outputs gives the same results as in Robins' original experiment. However, this measurement does not show how many patterns that are correctly recognized after the intervening items have been introduced. Measuring this gives a pragmatic and accurate view of how good the solution is with the test patterns used. A way to measure the amount of correctly recognized patterns was described in section 5.1.3.

We measured how many of the original patterns that were recognized after each intervening item was introduced. We expected this to show a rough correlation with the results from the original reproduction described in section 6.2, but the new results actually show something quite different. Although the network was shown to produce fewer wrong outputs in total when pseudorehearsal is used, the network does not recognize any more patterns. In fact, it ends up recognizing one or none of the original patterns it trained on. This is the case both for auto-associative and hetero-associative training items, as shown in figures 17 and 18.

These graphs reveal that pseudorehearsal performs worse than what the earlier results show. Although these new results also show that the method does

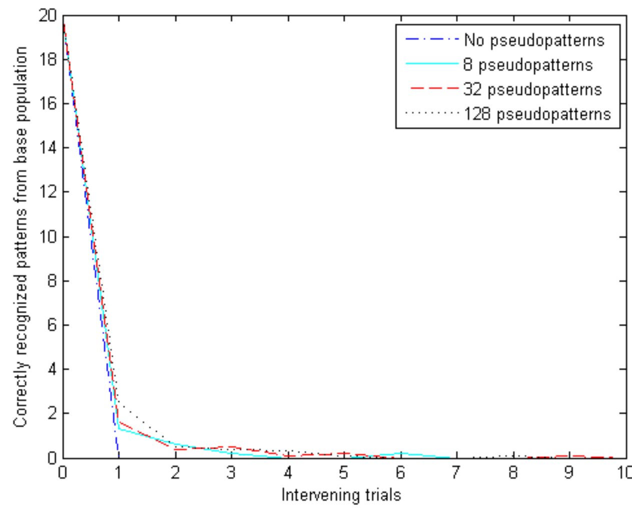

Figure 17: Performance of random pseudorehearsal measured as the number of correctly recognized original hetero-associative items.

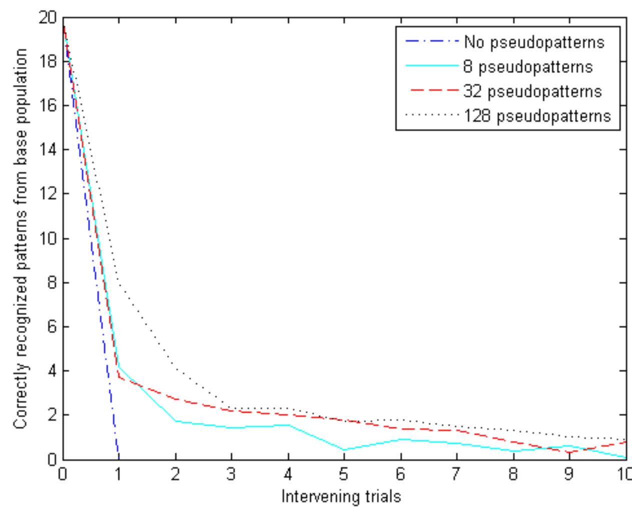

Figure 18: Performance of random pseudorehearsal measured as the number of correctly recognized original auto-associative items.

make the network perform better with pseudorehearsal than without it, the forgetting in the network is still catastrophic. Previous measurements have not shown this, since they have only taken into account the average error on

each output node or the average number of nodes with wrong output across all patterns. The new results show that although the number of errors is decreased in total, the number of correctly recognized patterns is still far too low, and the network still exhibits catastrophic forgetting of what it has learned.

The reason for this rather surprising result is not too difficult to see. Although the network has far less error on average with pseudorehearsal, a single output error is enough to say render the recognition of a pattern as failed. This is the case for many problems, i.e. types of input sets, such as the ones used in our experiments, and the original experiments we have reproduced.

6.4.4 Correctly recognized patterns with activation sharpening

The measurement used in the previous section was also used on networks trained with the activation sharpening algorithm, and the results from these two are presented in figure 19.

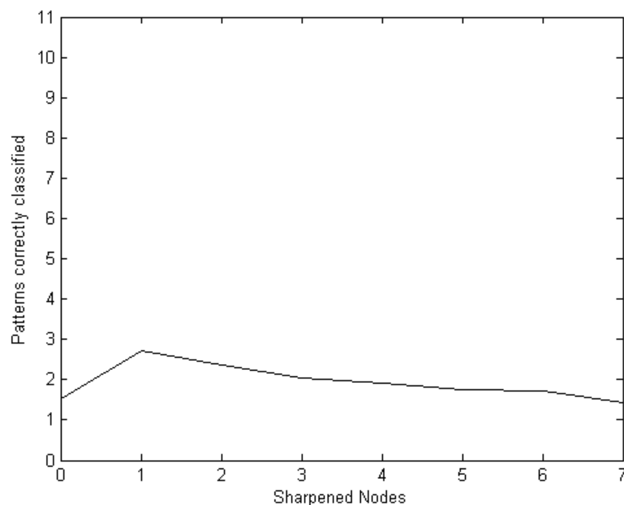

Figure 19: Number of correctly recognized patterns in the 8-8-8 network trained with activation sharpening, averaged over 50 runs.

The results here are consistent with those from section 6.3: regular back-propagation exhibits the worst results, one node sharpening exhibits the best, and as more nodes are sharpened the network performs progressively worse. The interesting aspect is again that *No matter how many nodes are*

sharpened, no more than three of eleven patterns are correctly classified. This clearly shows that forgetting of the original patterns is a *big* problem after activation sharpening as well, even if it is smaller than if without activation sharpening.

6.4.5 Conclusion

We have used two ways of measuring and comparing the two experiments in this section, only one of which was directly discussed in chapter 5. Even though this is a rather limited experimental basis, there are clear conclusions that can be drawn. The first of these is that neither solutions perform particularly well on the pragmatic test used. In a real-world setting neither would have performed to what could have been called an acceptable level.

We conclude that the measurements made by the original authors and those after them do not provide a realistic view of how good or bad their respective solutions are. Although the total error (or number of erroneous outputs) goes down with pseudorehearsal, almost every single one of the original patterns the network is supposed to remember is not recognized correctly. The network gives at least one erroneous output for all (or in some cases all but one) of the patterns. This shows that even with pseudorehearsal the network does still exhibit catastrophic forgetting, which means that the pseudorehearsal solution is not as good as the original results indicated. Also for activation sharpening our tests show that even if an improvement over back-propagation can be seen, this is not by far as good as originally reported.

7 Conclusion

At the beginning of this project, as reported in 2, we posed three questions that we were seeking answers to:

(1) What work has been done to overcome the problem of catastrophic forgetting in neural networks?

(2) How general are the results from this work?

(3) How well do the existing solutions perform on real-world problems.

When viewing this project in light of these questions, we are confident that we have given a good answer to the first. We have given a broad survey of the available solutions to the problem of catastrophic forgetting and identified their characteristics, and also done a survey of various methods for measuring catastrophic forgetting.

When seeking an answer to the second question, we have reproduced the experiments from two important early papers to affirm the results of these experiments. These were

- Robins' pseudorehearsal solution, and
- French's activation sharpening algorithm

The first of these solutions has been widely tested and accepted as a good solution to the problem [17] [12]. It has also been used for other applications, notably for transferring information between networks in dual-network models [15] [25] [26] [27]. The latter has not been used widely, but the work done during development of it highlighted several important aspects of catastrophic forgetting, and fueled further research [13] [15].

Our reproductions of the experiments mainly show that the solutions work as described, but also identify important weaknesses in both of the original experiments. In the case of pseudorehearsal, the parameters used in the experiment were found to be too strict to produce feasible results. We have showed that when the parameters are more relaxed, and the network actually converges, pseudorehearsal works as intended. In the case of activation sharpening, the results were reproducible, but we have argued that French's theory of the optimal number of nodes to sharpen is incorrect.

We also tried using pseudorehearsal on the network from French's experiment, and using activation sharpening on the network from Robins' experiment, while keeping the original way to measure performance in both cases. The results from these experiments were consistent with previous results for the pseudorehearsal solution. For activation sharpening we saw that the algorithm had worse performance on data that was learned longer ago than the other algorithms. This indicates that the activation sharpening algorithm will "disturb" the old information more than the other algorithms if it has previously failed to distribute the information and avoid information overlap. Since we did not see the low activation overlap that French reported, we cannot say how well activation sharpening would perform if lower overlap was the case.

It is our view that the most important result from these experiments is that **The tested solutions reduce the effect of catastrophic forgetting to some degree, but they are not good enough to serve as general solutions.**

The third question we did not find a good answer to, and doing that is one of the things we propose as future work in the next section.

8 Further work

The experiments that were duplicated are far from complete, and several important aspects have not been tested thoroughly. Chapter 4.6 provides a discussion of important aspects to consider when evaluating solutions to catastrophic forgetting, motivated by the need for a solution to be as general as possible. These give a rise to several areas that require further testing to determine a more general goodness of the solutions:

- Tests with input sets with different and known mutual orthogonalities among patterns
- Tests with various degrees of sequential learning (various numbers of items in each part of the sequence)
- Tests with large input sets to provoke saturation of the solution
- Tests with other types of learning than standard back-propagation, to see whether the solutions are usable in those cases as well

We find it desirable to establish a way to measure the mutual orthogonality of all input patterns that are used during testing. This will give us a much stronger foundation for discussing the quality of the evaluation in terms of the input patterns.

Finally, we present our idea for refinement of the pseudorehearsal solution. It would be a definite improvement to find a way to determine the quality of a generated pseudopattern, and use that quality measurement to select only "good" pseudopatterns for use during sequential learning. This quality measurement should be able to determine whether or not the input pattern is representative for one or more of the already learned patterns, or, when thinking in terms of function approximation, determine whether the sampling done by feeding it through the network is a sample that is on as close as possible to the approximated function.

With regards to the activation sharpening algorithm, one possibility might be the explore other ways in which the sharpening is done. French's idea was to distribute the information in the network, and other ways of doing this might provide better results.

We do not believe it is worth-while to work on improving French's formula for the optimal number of nodes to sharpen even though our results indicate that it is not correct. As our results also show that the activation sharpening algorithm itself must be improved for it to provide satisfactory results, such a formula is likely to be of little use.

A final natural extension of these experiments is to test their validity on real-world problems. This can both be to perform tests using actual robots, it can be using robotic simulators, or it can at least be on more actual problems than the abstract pattern associations we have used. Such simulations are more prone to emergent behavior, and both new solutions and problems can emerge as a result of testing in such settings.

References

[1] Robert Callan. *The essence of neural networks.* Prentice Hall Europe, Essece, England, 1999.

[2] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. *Parallel distributed processing: explorations in the microstructure of cognition, vol. 1: foundations*, pages 318–362, 1986.

[3] Michael McCloskey and Neal J. Cohen. Catastrophic Interference in connectionist networks: The sequential learning problem. *The psychology of learning and motivation*, 24:109–165, 1989.

[4] R. Ratcliff. Connectionist models of recognition memory: Constraints imposed by learning and forgetting functions. *Psychological review*, 97:285–308, 1990.

[5] Stephen Grossberg. How does a brain build a cognitive code? *Psychological Review*, 81:1–51, 1980.

[6] Stephen Grossberg. *Studies of Mind and Brain.* D. Reidel Publishing Company, Boston, MA, 1982.

[7] Robert French and André Ferrara. Modeling time perception in rats: Evidence for catastrophic interference in animal learning. In *Proceedings of the 21st Annual Conference of the Cognitive Science Conference*, pages 173–178, NJ, 1999. LEA.

[8] B. McNaughton J. McClelland and R. O'Reilly. Why there are complementary learning systems in the hippocampus and neocortex: Insights from the successes and failures of connectionist models of learning and memory. *Psychological Review*, 102:419–457, 1995.

[9] W. B. Scoville and B. Milner. Loss of recent memory after bilateral hippocampal lesions. *J. Neurol. Neurosurg. Psychiatr.*, 20:11–21, 1957.

[10] A. Robins and S. Mccallum. Catastrophic Forgetting and the Pseudorehearsal Solution in Hopfield-type Networks. *Connection Science*, 10:121–135, 1998.

[11] Bernard Ans, Stéphane Rousset, Robert M. French, and Serban Musca. Preventing Catastrophic Interference in Multiple- Sequence Learning Using Coupled Reverberating Elman Networks.

[12] Marcus Frean and Anthony Robins. Catastrophic forgetting in simple neworks: an analysis of the pseudorehearsal solution. *Network: Computation in Neural Systems*, 10:227–236, 1999.

[13] Robert M. French. Using Semi-Distributed Representations to Overcome Catastrophic Forgetting in Connectionist Networks. Technical Report CRCC-TR-51-1991, 510 North Fess, Bloomington, Indiana, 1991.

[14] R. M. French. Catastrophic Forgetting in Connectionist Networks. In *Encyclopedia of Cognitive Science*, volume 1, pages 431–435, London, 2003. Nature Publishing Group.

[15] R. M. French. Catastrophic Forgetting in Connectionist Networks: Causes, Consequences and Solutions. *Trends in Cognitive Science*, 3:128–135, 1999.

[16] A. Robins. Catastrophic forgetting in neural networks: the role of rehearsal mechanisms. In *Proceedings of the First New Zealand International Two-stream Conference on Artificial Neural Networks and Expert Systems*, Los Alamitos, 1993. IEEE Computer Society Press.

[17] A. Robins. Catastrophic Forgetting, Rehearsal and Pseudorehearsal. *Connection Science*, 7:123–146, 1995.

[18] Kupferman. Modulatory Actions of Neurotransmitters. *Annual Review of Neuroscience*, 2:447–465, 1979.

[19] Geoffrey E. Hinton and David C. Plaut. Using Fast Weights to Deblur Old Memories. In *Proceedings of the Ninth Annual Cognitive Science Society Conference*, pages 177–186, Hillsdale, NJ, 1987. Lawrence Erlbaum Associates.

[20] A. R. Gardner-Medwin. Doubly modifiable synapses: A model of short and long term autoassociative memory. In *Proceedings of the Royal Society, B238*, pages 137–154, 1989.

[21] Joseph P. Levy and Dimitrios Bairaktaris. Connectionist Dual-weight Architectures. *Language and cognitive processes*, 10:265–283, 1995.

[22] G. E. Hinton. Deterministic Boltzmann learning performs steepest descent in weight-space. In *Technical Report CRG-TR-89-1*. Department of Computer Science, University of Toronto, 1989.

[23] D. Bairaktaris. A model of auto-associative memory that stores and retrieves, successfully, data regardless of their orthogonality, randomness or size. In *Proceedings of the Hawaii International Conference on System Sciences '90*, Kona, Hawaii, 1990.

[24] C. Peterson and E. Hartman. Explorations of the mean field theory learning algorithm. *Neural networks*, 2, 1989.

[25] B. Ans and S. Rousset. Avoiding catastrophic forgetting by coupling two reverberating neural networks. *Academie des Sciences, Sciences de la vie*, 320:989–997, 1997.

[26] R. M. French. Using pseudo-recurrent connectionist networks to solve the problem of sequential learning. In *Proceedings of the 19th Annual Cognitive Science Society Conference*, NJ, 1997. LEA.

[27] B. Ans R. M. French and S. Rousset. Pseudopatterns and dual-network memory models: Advantages and shortcomings. In *Connectionist Models of Learning, Development and Evolution*, pages 13–22, London, 2001. Springer.

[28] John Hertz, Anders Krogh, and Richard G. Palmer. *Introduction to the theory of neural computation.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1991.

A Experimental workbench

This appendix presents eGAS, a workbench we made for use in our experiments. The appendix assumes that the reader is familiar with C++ and object-oriented programming. Still, readers which only know object-oriented programming ought to be able to read it.

Our workbench, eGAS, is implemented as a set of pluggable modules or shared libraries, and written in C++. The workbench was created from the following goals:

- eGAS should be able to implement any kind of neural network, i.e. all networks that can be modelled as some set of nodes with directional connections between them.
- It should be easy to implement new ways of feeding patterns to the network, new ways of setting weights in the network, and new algorithms for training the network. Each of these new implementations should be a module that only interacts with the eGAS core through the interfaces it exposes. Implementation of a new module should not require changes to the eGAS core or any already existing modules.
- eGAS should be configurable through a simple configuration file format. A configuration file should represent a total setup of eGAS for a simulation. Both the eGAS core and any modules used in a simulation should use this file for configuration.

These goals were fueled by the fact that most existing libraries or applications that simulate neural networks make certain assumptions about the networks they simulate, or make it relatively difficult to extend the existing code with new functionality like a new training algorithm. Since we had a need to implement new algorithms and new types of networks quite often, it was very important to make it easy to do so, and to share and reuse existing code whenever it was possible.

A.1 Architecture

The architecture of eGAS is very modular. The workbench consists of 6 module (or plugin) types, which are coupled with each other as shown in figure 20.

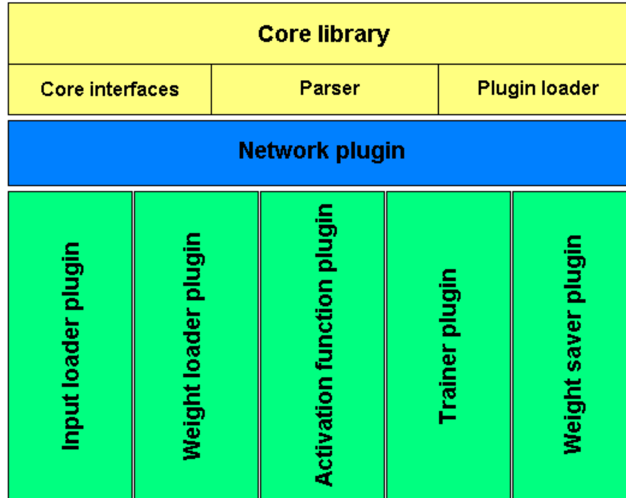

Figure 20: The eGAS architecture

The eGAS core library is the heart of the workbench, and contains the following:

- Interface definitions for all the various plugin types
- A basic network implementation that makes network plugin implementation easier
- A plugin loader that can load all of the plugin types and return an instance of them
- A parser for the configuration files

This lean and light-weight, but powerful core library serves as the basis of the eGAS workbench.

A.2 Configuration file format

The configuration file format in eGAS is simple, but powerful. A configuration file consists of two main parts:

- A declaration part that declares all the context types and their contained variables and types
- A configuration part that defines contexts and sets their variables to values

This results in a type-safe and easily readable configuration file format. The grammar of the file is best explained through a simple example:

```
declare root
{
    type network
}
declare network
{
    string name
    intlist layer_sizes
    type trainer
}
declare trainer
{
    string plugin
    real criterion
}


network my_network
{
    name = my test network
    layer_sizes = [2,2,1]
    trainer backprop
    {
        plugin = backprop
        criterion = 0.2
    }
}
```

This simplified example first declares the `root` context, which may only contain contexts of the type `network`. The `root` context specifies what contexts, strings, integers, etc. that may be specified in the root of the configuration file. Next, the file declares the `network` context, which may contain a string specifying its name, a list of integers that specifies the size of the layers (and implicitly the number of layers), and the trainer context, which specifies the trainer to use. The trainer context may contain a string specifying what plugin to use, and a real-valued number specifying the acceptance criterion for the trainer.

A.3 The core library

This section describes the classes that are implemented in the core library. The core library provides a set of classes that enables people to write simulations in an easy manner using eGAS.

A.3.1 The parser class

The parser consists of two classes; the `Parser` is the main class, and the `Context` class represents a context in the configuration file, e.g. the `network` context from the example configuration file above.

The `Parser` class contains the following important functions:

```
class Parser
{
  public:
    explicit Parser(const std::string& in);
    explicit Parser(std::ifstream& in_file);

    Context* getConfig() const;
};
```

The two different constructors take either an `std::string` or an `std::ifstream` reference to the configuration. Upon construction, the parser instance parses its configuration. The parsed configuration can then be fetched through the `getConfig()` function. This function returns a pointer to an instance of the `Context` class. This instance is always the root context of the configuration file. The `Context` class contains the following important functions:

```
class Context
{
  public:
    real_t getReal(const string_t& key) const;
    integer_t getInt(const string_t& key) const;
    string_t getString(const string_t& key) const;

    reallist_t getRealList(const string_t& key) const;
    intlist_t getIntList(const string_t& key) const;
    stringlist_t getStringList(const string_t& key) const;

    Context* getChild(const string_t& key) const;
    contextlist_t getChildren(const string_t& type = "") const;
};
```

The functions are used to fetch the various parameters that are set in various contexts. Note again that contexts may contain other contexts, which can be retrieved by specifying the name of the desired context, or by retrieving a list of all contexts of a given type.

As an example, if we look at the configuration file described earlier, the context returned by `getContext()` is the root of that configuration file. The only thing it contains is a context named my_network, so to get that we would call `getChild("my_network")` on the root context. To get the list of layer sizes from the network, we would then call `getIntList("layer_sizes")` on the network context.

A.3.2 The plugin loader

Since most of the parts of eGAS are plugins with interfaces defined in the core library, the core library provides a plugin loader that allows for convenient creation of instances of the various plugins. The plugin loader is a template class, and has the following interface:

```
template<typename T> class PluginLoader
{
  public:
    inline static PluginLoader<T>& instance();

    T* newObject(Context& context);
    void deleteObject(T* object);
};
```

The plugin loader is a singleton class with one instance per plugin type. To load a network, the following code is used:

```
#include "parser/parser.h" // The parser interface
#include "ann/core/network.h" // The network interface
#include "plugins/plugin_loader.h" // The plugin loader interface
#include <fstream>

using namespace egas;

int main(int argc, char* argv[])
{
  // Read configuration from a file
  std::ifstream f("config.conf");
  Parser parser(f);

  // Fetches the network named network from the configuration
```

```
  Context* rootContext = parser.getConfig();
  Context* networkContext = rootContext->getChild("network");

  // Get an instance of the plugin loader...
  PluginLoader<Network>& loader = PluginLoader<Network>::instance();
  // ...and load a network object with configuration
  Network* theNetwork = loader.newObject(*networkContext);

  // ... do what you want with the network object ...

  // ... and then delete it.
  loader.deleteObject(theNetwork);

  return EXIT_SUCCESS;
}
```

A.4 Interfaces

This section describes relevant parts of the various interfaces in the eGAS workbench. All these interfaces are implemented as plugins or shared libraries, for use with the plugin loader and the core library of eGAS.

A.4.1 The network interface

Having seen the configuration and plugin loader API of eGAS, we now turn to perhaps the most important class in the core library. The `Network` class implements basic functions for creating a network, but leaves the feeding and training functions abstract. The `Network` class contains the following important functions:

```
class Network
{
  public:
    explicit Network(Context& configuration);
    Layer& addLayer(Layer& l);
    void connectNodes(Node& from, Node& to);
    void connectLayersFully(Layer& from, Layer& to);
    void dropAllConnections();
    void build();

    virtual void feed() = 0;
    virtual bool learn() = 0;

    const LayerList getLayers();
```

```
    virtual reallist_t getInput() const;
    virtual reallist_t getOutput() const;

    void setWeightLoader(WeightLoader& wl);
    void setWeightSaver(WeightSaver& ws);
    void setInputLoader(InputLoader& il);
    void setTrainer(Trainer& t);


  protected:
    virtual void buildNetwork() = 0;
};
```

The interface has purely virtual functions, which means that every to be used in eGAS must be plugin that is a subclass of `Network` and implement the `feed()`, `learn()` and `buildNetwork()` functions. The interface provides implemented functions for adding layers, connecting nodes and layers, and dropping connections. It also provides functions for giving instances of the various plugins that are to be used in the network.

The `feed()` function should perform a feed of an input pattern given by the input loader. The `learn()` function should train the network, preferably by using its trainer instance.

An instance is constructed by passing in a `Context` instance with the configuration of the network. The basic `Network` class does not care about any parameters specified in this configuration. This means that a network plugin must define configuration parameters that it can use to create a network. A network plugin should create its network in the `buildNetwork()` function. To do this, the network plugin may use the `Layer`, `Node` and `Edge` classes, which are defined as follows:

```
class Layer
{
  public:
    Layer();
    Node& addNode(Node& node);
    const NodeList getNodes() const;
    int getPosition(const Node& node) const;
};

class Node
{
  public:
    Node(ActivationFunction* activation_function);

    void connectTo(Node& node, real_t weight);
```

```
    void connectFrom(Node& node, real_t weight);
    void dropConnection(Edge* connection);
    void dropAllConnections();

    const EdgeList getConnectionsTo();
    const EdgeList getConnectionsFrom();

    void setInput();
    real_t getOutput();
};

class Edge
{
  public:
    Edge(Node* from, Node* to, real_t weight);
    Node* const from;
    Node* const to;
    real_t value;
};
```

The `Layer` class implements functions for adding nodes and retrieving them, as well as finding their position in the layer.

The `Node` class implements functions for connecting nodes, setting the net input, and getting the output of the node. A node is created by passing in a pointer to the `ActivationFunction` plugin instance the node should use.

The `Edge` class implements a simple constructor that takes a source and target nodes, and the weight between them, as arguments. The members of the class are public, but cannot be changed.

The `Network` class and its components, defined by the `Layer`, `Node`, and `Edge` classes, provide a convenient way of creating a network plugin for the eGAS workbench.

A.4.2 The trainer interface

The trainer interface is the second most important interface in eGAS the `Network` interface. It is defined in the `Trainer` class, and is supposed to train a `Network` instance when given to it. The `Trainer` class is quite simple:

```
class Trainer
{
  public:
    Trainer(Context& configuration);
    virtual bool operator()(Network& network) = 0;
```

```
    virtual ~Trainer();
};
```

This means that to construct a `Trainer` instance, one must provide a configuration. The basic `Trainer` class does not require any particular parameters in this configuration. Their interpretation is left to the plugin that implements the interface. The training of a network occurs when `operator()` is called. This operator is required to return a boolean, specifying whether the training converged or not. This is analogous with the `learn()` function in the `Network` class.

A.4.3 The input loader interface

The input loader interface defines the interface for plugins that load input into the network from some arbitrary source:

```
class InputLoader
{
  public:
    InputLoader(Context& configuration);
    virtual bool operator()(Layer& input_layer) = 0;
    virtual ~InputLoader();
};
```

The interface is almost identical to the `Trainer` class, but the `operator()` function takes the input layer of the network as an argument. The purpose of the `operator()` function is to set the net input on each of the nodes in the input layer. Like the `Trainer` class, the `InputLoader` class accepts a `Context` instance specifying its configuration upon construction, but does not require any parameters to be set in this configuration.

A.4.4 The weight loader interface

The weight loader interface defines the interface for plugins that load weights into the network. This is meant to be used for weight initialization before any feeding or training occurs. The interface is very similar to the two previously described interfaces:

```
class WeightLoader
{
  public:
    WeightLoader(Context& configuration);
```

```
    virtual real_t operator()(const IndexedNode& from,
        const IndexedNode& to) = 0;
    virtual ~WeightLoader();
};
```

The `WeightLoader` has an `operator()` that accepts two `IndexedNode` instances as its arguments. The `IndexedNode` class is simply a `std::pair` of a `Node` and a `std::pair` with two integers, representing which layer a node is in, and its position in that layer. The weight loader is supposed to return the real-valued number that is the weight between these two nodes. The constructor behaves just like in the previously described classes.

A.4.5 The weight saver interface

The weight saver interface defines the interface for plugins that save the weights to some storage area. Typically, this storage area is a file, but it can be anything. Again, the interface is very general and puts very few restrictions on the implementing plugin:

```
class WeightSaver
{
  public:
    WeightSaver(Context& configuration);
    virtual void operator()(const Network& network) = 0;
    virtual ~WeightSaver();
};
```

The weight saver is expected to save the weights of the entire network when its `operator()` is called. The constructor behaves like in the previous classes.

A.4.6 The activation function interface

Finally, we present the activation function interface, defined by the `ActivationFunction` class:

```
class ActivationFunction
{
  public:
    ActivationFunction(Context& configuration);
    virtual real_t operator()(const real_t net_input) = 0;
    virtual ~ActivationFunction();
};
```

The `operator()` function of this class accepts a real-valued number representing the net input of a node, and returns a real-valued number representing the output of the node. Otherwise, the class behaves just like the other plugins.

A.5 Using the eGAS library

As the above sections show, the eGAS library can be used to implement various network plugins and plugins necesseary for that network, which can be loaded into an application using the core library. eGAS itself is only a library, and provides no end-user application. When conducting our experiments, we have written small applications that utilize the library to perform the required tasks.

B Plots from pseudorehearsal experiments

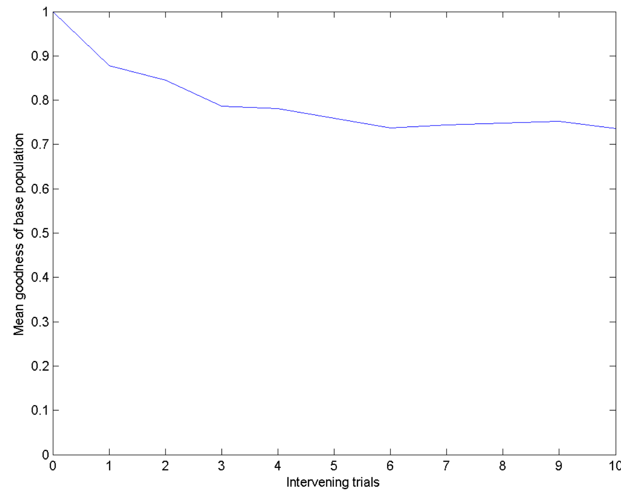

Figure 21: Performance without pseudorehearsal, training on hetero-associative items

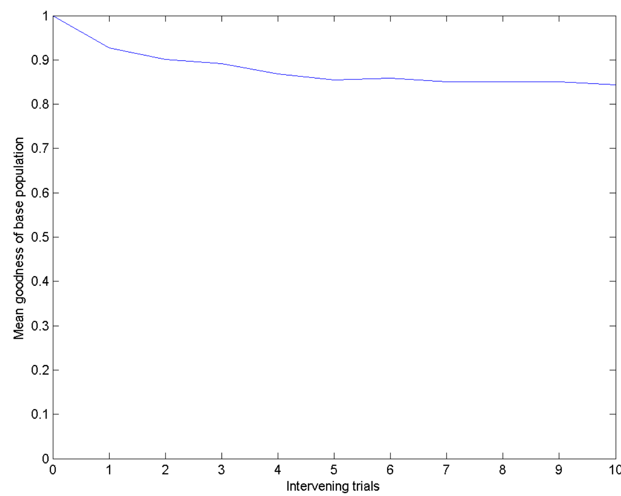

Figure 22: Performance using random pseudorehearsal with 8 pseudopatterns, training on hetero-associative items

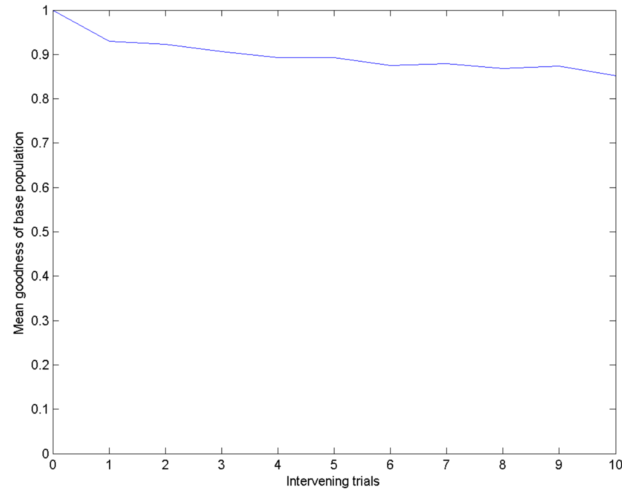

Figure 23: Performance using random pseudorehearsal with 32 pseudopatterns, training on hetero-associative items

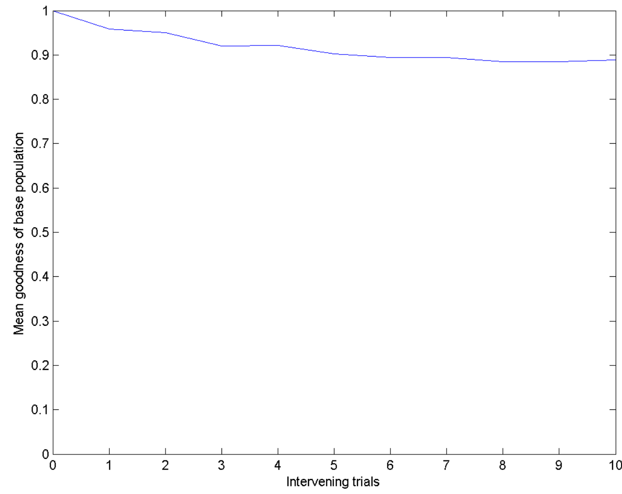

Figure 24: Performance using random pseudorehearsal with 128 pseudopatterns, training on hetero-associative items

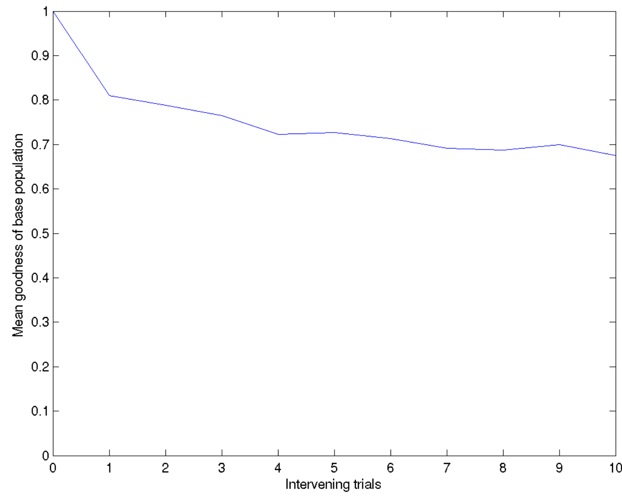

Figure 25: Performance without pseudorehearsal, training on auto-associative items

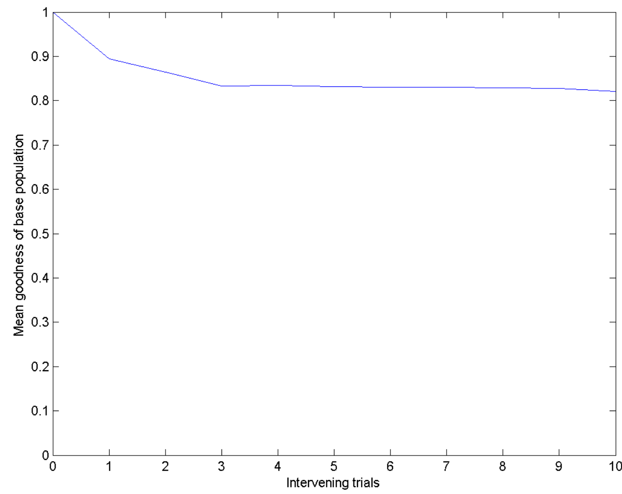

Figure 26: Performance using random pseudorehearsal with 8 pseudopatterns, training on auto-associative items

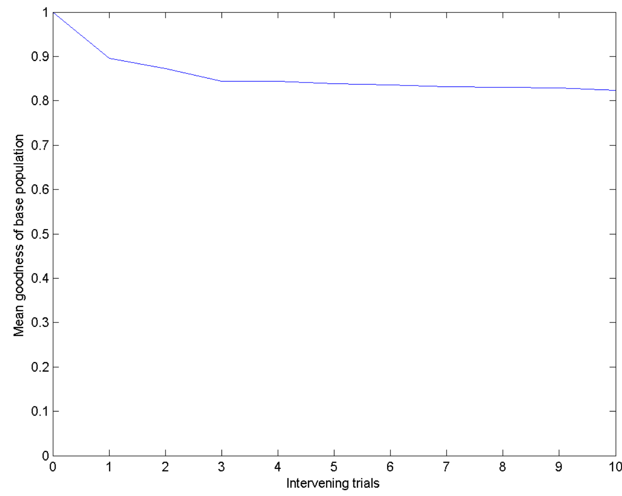

Figure 27: Performance using random pseudorehearsal with 32 pseudopatterns, training on auto-associative items

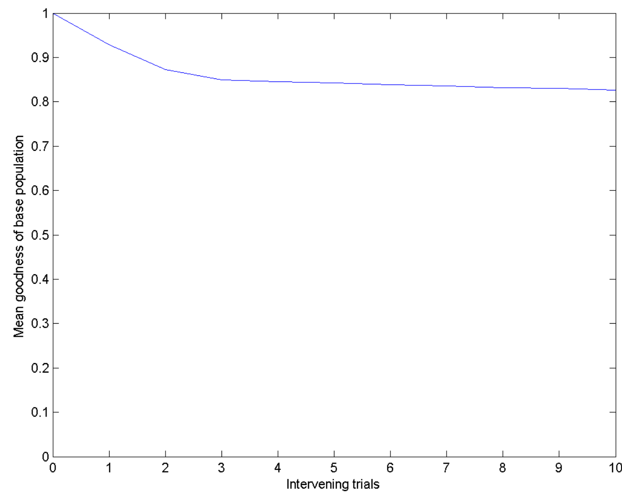

Figure 28: Performance using random pseudorehearsal with 128 binary pseudopatterns, training on auto-associative items

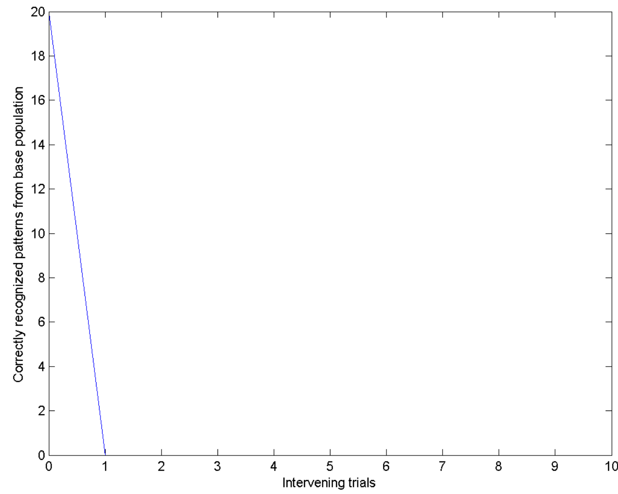

Figure 29: Number of correctly classified items without pseudorehearsal, training on hetero-associative items

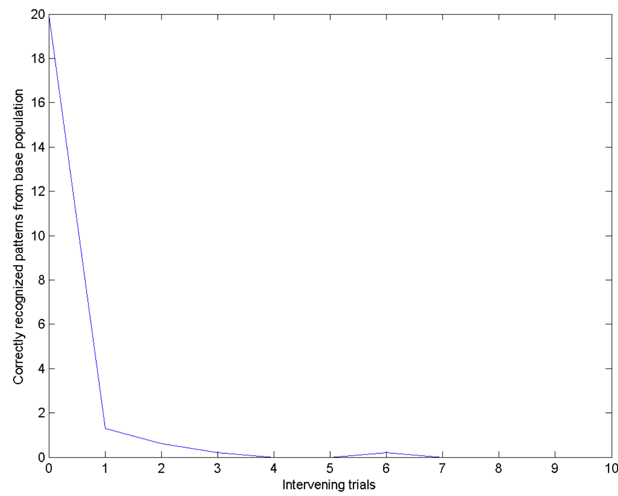

Figure 30: Number of correctly classified items using random pseudorehearsal with 8 binary pseudopatterns, training on hetero-associative items

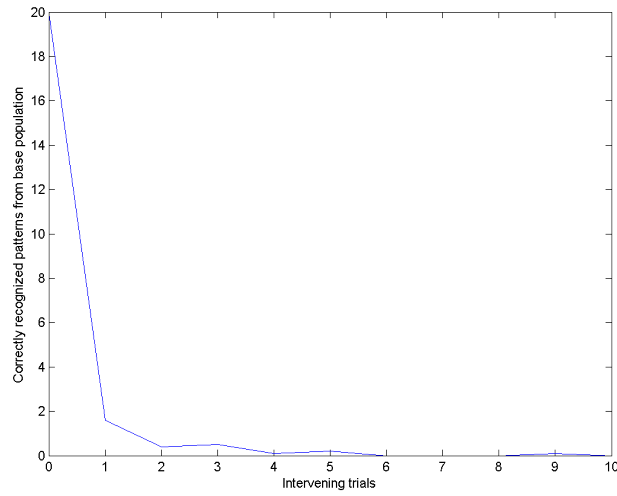

Figure 31: Number of correctly classified items using random pseudorehearsal with 32 binary pseudopatterns, training on hetero-associative items

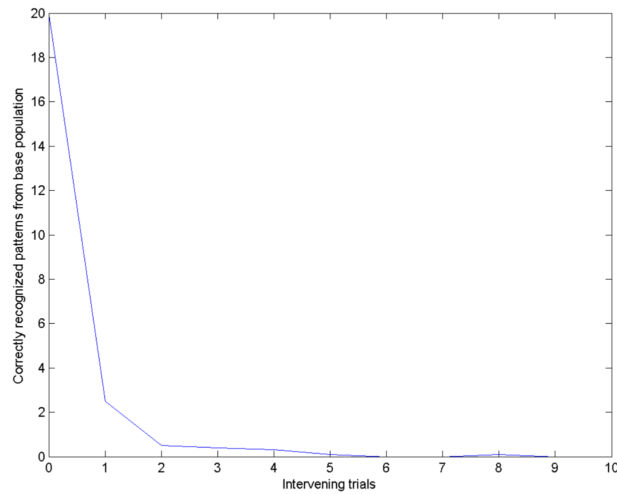

Figure 32: Number of correctly classified items using random pseudorehearsal with 128 binary pseudopatterns, training on hetero-associative items

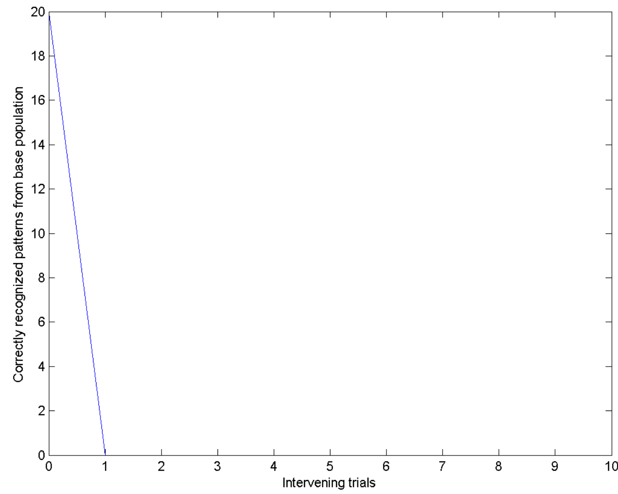

Figure 33: Number of correctly classified items without pseudorehearsal, training on auto-associative items

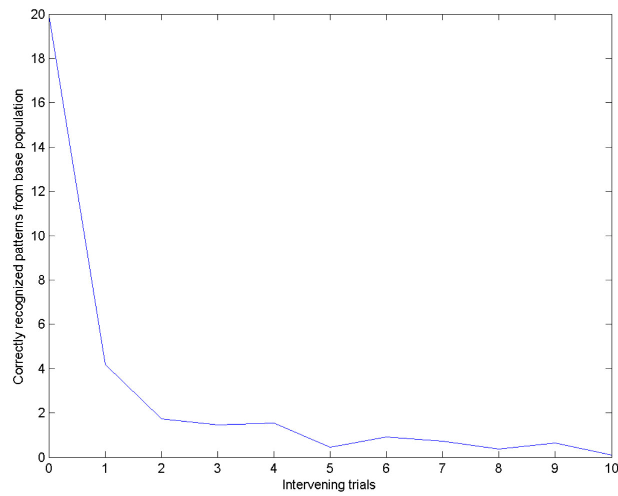

Figure 34: Number of correctly classified items using random pseudorehearsal with 8 binary pseudopatterns, training on auto-associative items

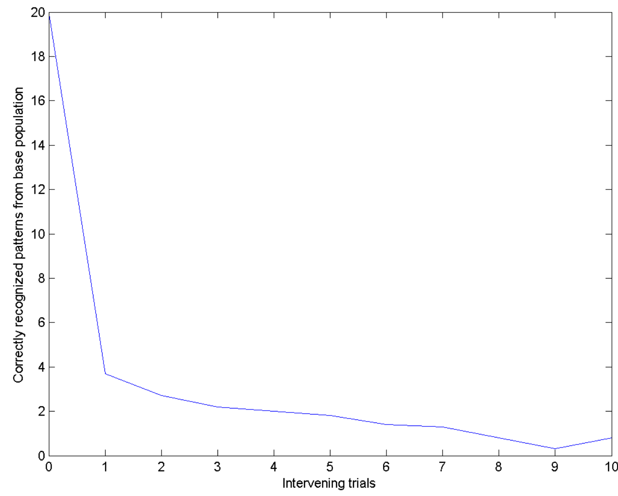

Figure 35: Number of correctly classified items using random pseudorehearsal with 32 binary pseudopatterns, training on auto-associative items

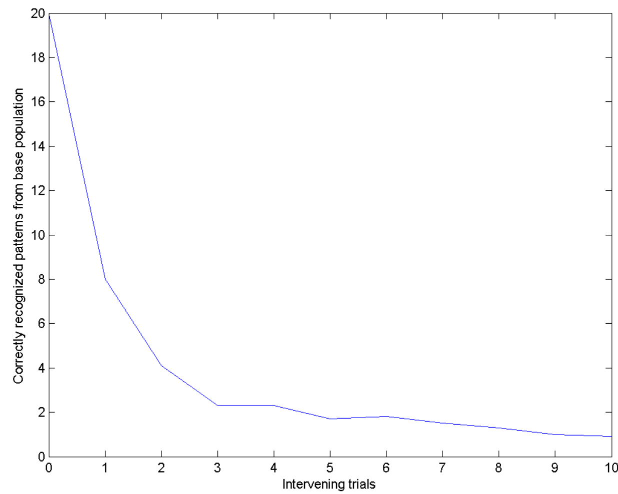

Figure 36: Number of correctly classified items using random pseudorehearsal with 128 binary pseudopatterns, training on auto-associative items