NTNU
Innovation and Creativity

Benchmarking Catastrophic Forgetting in Neural Networks

Ole-Marius Moe-Helgesen

Master of Science in Computer Science
Submission date: June 2006
Supervisor: Pinar Öztürk, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Problem Description

Catastrophic Forgetting, a common problem when neural networks are provided with information sequentially, is a subject for an increasing amount of research. There is currently no uniform way to measure and test artificial neural networks for catastrophic forgetting. Is it possible to compile a set of rules and guidelines for measuring and testing neural networks for catastrophic forgetting?

If so, how well does existing proposals to solving Catastrophic Forgetting work? Are the solutions applicable on real-world data as well as the randomized data structures used in experiments?

Assignment given: 20. January 2006
Supervisor: Pinar Öztürk, IDI

Abstract

Catastrophic Forgetting is a behavior seen in artificial neural networks (ANNs) when new information overwrites old in such a way that the old information is no longer usable. Since this happens very rapidly in ANNs, it leads to both major practical problems and problems using the artificial networks as models for the human brain.

In this thesis I will approach the problem from the practical viewpoint and attempt to provide rules, guidelines, datasets and analysis methods that can aid researchers better analyze new ANN models in terms of catastrophic forgetting and thus lead to better solutions.

I suggest two methods of analysis that measure the overlap between input patterns in the input space. I will show strong indications that these measurements can predict if a back-propagation network will retain information better or worse.

I will also provide source code implemented in Matlab for analyzing datasets, both with the new suggested measurements and other existing ones, and for running experiments measuring the catastrophic forgetting.

Preface

This Master's thesis has been written as part of my studies at the Norwegian University of Science and Technology in Trondheim, Norway. It has been done at the Faculty of Technology, Mathematics and Electrical Engineering, Department of Computer and Information Science, Self-Organizing Systems Group.

Acknowledgments

I would like to thank my supervisor and guidance teacher Pinar Öztürk who both spiked my interest in the field, led me to interesting research studies and kept me going with critical questions and interesting ideas. I would also like to thank the other students that has been part of the colloquiums that's been done at the SOS group the year I have been studying catastrophic forgetting: Axel Tidemann, Firas Risnes Barakat and Rikke Amalie Løvlid. I would also like to thank Håvard Stranden which whom I wrote the project report last fall together with. Much of the work in this thesis is based on the work we did together.

Trondheim, June 2006

Contents

List of Tables

List of Figures

1 Introduction

Last fall I worked on reviewing the problem of catastrophic forgetting and solutions provided to solve it. In the project written by myself and Håvard Stranden[1] we also reproduced two important experiments from the literature and reported the results. The work in this thesis is a continuation of the work done then.

In this chapter I will introduce the problem of catastrophic forgetting, provide motivation for solving the problem, present a brief review of some proposed solutions, and present the motivation for why a benchmark for catastrophic forgetting is needed.

I will also provide some information on artificial neural networks themselves. While this information might be well known to the reader, it also serves to establish the terminology regarding neural networks used in this thesis. For a more detailed introduction to both catastrophic forgetting and neural networks in general, I refer the reader to the above mentioned report by myself and Stranden.

Before diving into the details though, I want to provide some motivation for why this thesis has been written and why it has been written the way it has.

1.1 Motivation

The work I had done preceding this thesis together with Håvard Stranden [1] revealed that scientific reports and papers regarding catastrophic forgetting were lacking both in terms of the necessary details for reproducing the experiments and in terms of how general the results were. What we experienced when testing our implementation of architectures described in scientific papers was that we had to make several assumptions regarding parameters, network topology, training rules and so on. If all these details had been accurately reported, the experiments could have been reproduced and studied further much faster than what was the case. We also saw that the data used for testing these networks were questionable at best. They were artificial and randomly generated and with very few attributes (such as how they were distributed in the input space, how many classes there were, etc) deliberately chosen by the researchers.

With that in mind, I set out with several goals for this thesis

1. Can a set of rules and guidelines be set up for testing catastrophic forgetting such that other researchers can reproduce experiments without having to make assumptions on how the original experiment was done.
2. Does a set of real-world datasets to be used for catastrophic forgetting tests exist?
3. Can datasets be analyzed before training commences to see if they are liable to be easily forgotten? Will some datasets suffer more from catastrophic forgetting than others?
4. How well does the architectures we studied last fall perform on such real-world datasets?

I discovered quickly that the same issues with reporting we had seen in the papers on catastrophic forgetting papers, others had seen concerning the ANN and AI communities in general, and much work on providing rules and guidelines had already been done. I looked at several of these in order to compile some common guidelines for catastrophic forgetting in particular.

1.2 Reader's Guide

In chapter 2 I will present my studies into current approaches to benchmark neural networks. The goal of that chapter was to provide information usable to answer question one above. In chapter 3 I will work on answering the third question. I approach that issue by looking at a taxonomy of features devised for datasets in general, and proceed by formulating two more features specifically developed with catastrophic forgetting in mind.

In chapter 4, I discuss how best to measure forgetting in neural networks. This is essential both to aid researcher and thus answer the first problem posed and to facilitate the experiments performed later in this thesis.

In chapter 5 I provide a summary of the earlier chapters and discuss some of the issues and suggestions provided by the papers and reports studied. This chapter will provide my answer for how best to solve question one above.

In chapter 6 I describe the implementational work done for this thesis. All of this work is done in Matlab with the exception of some shell scrips for Unix used to generate and divide the datasets. Chapter 7 describes the experiments done in the Matlab setting. It also provides the results and discusses

these. These experiments will both attempt to study the measurements suggested earlier, test the existing architectures as described in question four above, and test other dataset attributes for connections with catastrophic forgetting.

In the final chapter, chapter 8 I provide my conclusions for the thesis and look at future work based on what I have done.

1.3 Artificial Neural Networks

Neural Networks are at the core of this thesis, as it is the loss of information in such networks that is being studied. These networks are inspired by the neural connections in the brain of humans and other animals. It is in the connections between the neurons that memories are stored, and similarly is it in the weights on arcs connecting nodes that ANNs store data.

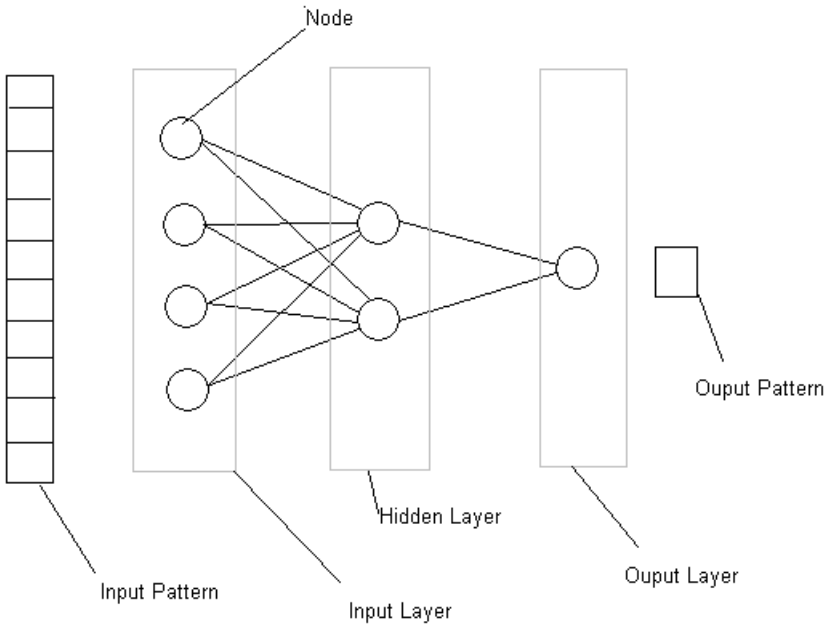

Figure 1: A conceptual view of an artificial neural network showing a three-layered network with input and output patterns, weights and nodes.

Any piece of information stored in an ANN is *distributed* across many or all the weights in the network. An output from the network is produced by combining the output of nodes with the weights on the arcs. The node outputs is again produced by taking the weighed sum of all outputs leading

to that node and passing that sum through an *activation function*. More specifically, the network is divided into several *layers*. When producing an output to a given stimuli, an *input pattern* is used and provided as inputs to the first layer (or *input layer*). These inputs are then multiplied by the weights attached to the arcs connecting the input layer to the next layer to produce, at each node, a *net input* or *weighted input*. This net input is the passed through an activation function, often the sigmoidal function $\frac{1}{1+e^{-1}}$, to produce the *output* at each node. This process is then repeated for each of the layers in the network until an output is produced at the final layer.

This output is then compared to the *target output* or *output pattern* associated with the current input pattern. The combined input and output pattern is called a training or test *instance*, or an *exemplar*. The difference between the target output and the actual output is calculated, and the weights are then updated in some fashion so they will reduce this error. Different algorithms can be used for this purpose, with *back-propagation* being the most common.

Figure 1 shows a conceptual view of an ANN with some of the most important terms labeled.

1.4 Catastrophic Forgetting

Catastrophic forgetting is the inability of ANNs to keep old information while at the same time learning new. It was discovered at the end of the 1980's by two separate research groups: McCloskey and Cohen[2] and Ratcliff[3]. It's been studied extensively in the 15 years passed since then, and several solutions has been suggested. Although it was discovered in the late 80's for neural networks, catastrophic forgetting is in fact a special case of the general dilemma termed *the stability-plasticity dilemma* in 1980[4] by Grossberg. The dilemma is that the adaptive mechanisms of an organism needs to be plastic enough to adapt to new changes, but at the same time stable enough to retain its old information. Make it too stable and it can't learn new information, but make it too plastic and it learns everything new while forgetting the old. Catastrophic forgetting is a manifestation of this "too plastic" behavior.

Catastrophic forgetting is an interesting problem seen from two important research perspectives: (1)Memory models that exhibit catastrophic forgetting are not plausible models of a human brain and (2) computer systems that forget old information when learning new is not suitable for a large range of problem domains. In this thesis the problem is approached from the second

viewpoint and thus only evaluated in terms of performance.

1.5 Approaches to solving catastrophic forgetting

There has been many papers published on research into catastrophic forgetting the last 15 years. In the project written by Håvard Stranden and myself the fall of 2005 [1] several of these were studied. We selected two for additional studies: The Pseudorehearsal solution developed by Anthony Robins and the Activation Sharpening algorithm by Robert M. French.

The Pseudorehearsal solution is based on the idea that old information should be learned alongside the new to make sure that the old is still retained. But to avoid having to explicitly store old patterns, one creates *pseudo patterns* for this purpose. These pseudo patterns are created by applying random inputs to the network and measuring the output that is produced. The analogy used is that if what one is training the network to do is to approximate a function, and the training patterns are the points on this function graph one wants to learn, the pseudo patterns are random samples along the curve.

The Activation Sharpening algorithm was developed after French studied the internal overlap of storage in neural networks. He hypothesized that if one could reduce the internal overlap, there wouldn't be as much data overwrite when learning new patterns. This was done by "sharpening" the nodes such that nodes with high activation got even higher activation and nodes with lower activation got even lower.

There have been other approaches to solving catastrophic forgetting as well, and several of these include dual-network architectures where one network serves as short-term memory and the other as long-term. These models are also more biologically plausible than for instance the pseudo rehearsal solution as evidence is found. It is however the two solutions we worked with last fall I have chosen to continue working with in this thesis.

1.6 Benchmarking

Neural Networks are highly complex data structures and so is the data that we use these data structures to analyze and work with. In fact they are so complex that we are at this point not near being able to perform complete formal studies on them. Due to this it is necessary to use empirical testing and experiments to perform research, improve on the existing models and to

discover new and better models. The challenge is to perform these experiments properly and in such a way that the conclusions that are drawn are in fact correct. From the middle of the 1990's, several researchers started acknowledging that the neural network community was lacking in this area. Several papers and some software packages were published discussing and providing solutions to this. In this thesis I will build on these works, from both the neural network and machine learning communities, and include *benchmarking of catastrophic forgetting* as well. As discussed above, catastrophic forgetting a serious problem when dealing with neural networks and proper testing is necessary if one is to be able to reduce or remove it as a general problem.

2 Existing Benchmark Packages

In this chapter I will look existing ways to benchmark neural network. This includes both methods for evaluating networks and datasets used during such evaluations. The methods form the basis for my suggestion for how to benchmark for catastrophic forgetting as formulated in question one in the motivation. I will describe the contents of the studied packages and discuss some of the arguments made by their authors.

I will however start with some information on the UCI repository as this is used by all of the packages studied in this chapter.

2.1 UCI Repository

The University of California, Irvine Machine Learning Repository is "a collection of problem databases, domain theories and data generators that can be used by machine learning researchers"[5]. With a few exceptions, all of the data in the repository is freely available to use as long as proper credit is given. The repository is available on the Internet from open HTTP and FTP servers[1]

The UCI repository is the de facto standard place to find datasets to use when testing a new machine learning algorithm or method in today's research community, and the datasets are as such widely used. All the packages discussed in this chapter uses the UCI repository.

The information provided on each dataset from the UCI website is limited. For each dataset the website provides information about the origin of the dataset, published works using the dataset, size of the dataset and input and output vectors, how the instances are distributed across the classes, and domain information regarding what the individual attributes in each data instance is to be interpreted as.

At the time of writing this thesis, 103 public documented datasets and four undocumented ones were available from the UCI website.

[1] Available from http://www.ics.uci.edu/ mlearn/MLRepository.html

2.2 Proben1

Proben1 is a package containing a technical report [6] with guidelines for benchmarking neural networks as well as a set of problems to use for these benchmarks. The technical report "PROBEN1 - A Set of Neural Network Benchmark Problems and Benchmarking Rules" that accompany the package is written by Lutz Prechelt from the university of Karlsruhe, Germany. The Proben1 package is available online from a public FTP server[2]

The Proben1 developers has done some analysis of the datasets provided with the package. They identified a number of different features in datasets that could be of interest, 16 in total, but have not made any attempt to completely classify the datasets according to those features. These features are mostly the same features as those introduced by Zijian Zheng[7], and they will be introduced and discussed thoroughly in section 3.1.

In addition to these features, Proben1 contains some useful guidelines for benchmarking. These rules cover both how one should conduct testing and how one should report them. Prechelt argues that without complete and explicit reporting of *all* parameters that can influence the results, "the experiments become irreproducible and the comparability of the results is hampered". These benchmarking rules are summed up in a 8-point list of information that should be included in every report about neural network experiments. The list, with some additional comments, is reproduced below.

1. **Problem**: Name, address, version/variant, etc. All the information necessary to uniquely identify the problem so other researchers can experiment with the exact same data.

2. **Datasets**: Training set, validation set, test set. How the dataset is split into smaller subsets for various phases must be specified. Are the instances randomly selected for the various sets or is some other algorithm used to achieve specific characteristics in and between the datasets?

3. **Network**: Nodes, connections, activation functions. The topology of the network is important, so it follows that one must report in detail how the network is constructed: how is the input and output fed and read? Are all layers fully connected? Is the same activation function used in all nodes? Which activation function(s) is(are) used?

[2]Proben1 is downloadble from ftp://ftp.ira.uka.de/pub/neuron/proben1.tar.gz

4. **Initialization**: Initialization values and the ranges of the random values if some values are randomly initialized is important to report. There can be great performance differences between a network with weights randomly initialized in the range [-1,1] and the same network with weights in the range [-10,10], so remember to report the range or distribution of random values as well.

5. **Algorithm**: Specific parameters and parameter adaption rules. Most update and training algorithms have some parameters that can be tuned, and as all ANN researchers have experienced changing these will have great impact on how well the algorithm in question performs. It is therefore vital that one reports *all* parameters used to initialize an algorithm. Providing pseudo code as well will help to expose any assumption that has been implicitly made.

6. **Termination**: Criteria for terminating, changing algorithm phases and restarting. It's important to be very specific about termination criteria for when to stop training, accept a test example as correct, and when to restart training due to the network not converging as wanted. Again providing pseudo code or regular code will help readers see *exactly* how this is done.

7. **Error Function**: Which error function or function, normalization rules. Closely related to termination criteria is the error function used to calculate how well the network performs. It is hard or impossible to reproduce results if a different error function from the original is used, so again be explicit and don't make assumptions about your readers having the same background as you.

8. **Experiment Parameters**: Number of runs, rules for excluding runs. Sometimes runs needs to be excluded from the averaged results due to the network not converging in the time alloted, it converging to a local maxima not providing good enough performance on the training or verification set, or for other reasons. Be sure that such possible factors are reported as well as the number of runs used to produce the reported results.

Finally Proben1 contains twelve benchmarking problems, and some analysis of the associated datasets. Some basic attribute features (see section 3.1) has been identified, some higher level analysis of instance and class features has been done, and they have all been encoded in a uniform way so they can easily be used for benchmarking.

All the datasets provided with Proben1 are from the UCI repository and are real-world problems. In the accompanying technical report, Prechelt argues that artificial datasets are not needed. His arguments for this is that for older problems that they are too easy to solve if the challenge of the dataset is known (such as the often used XOR-problem that is not linearly separable), and thus that solving these problems will not tell if a solution is applicable on real-world problems. He also argues against more complex artificial problems: No artificial problem can be certain to resemble *any* real-world situation, so even if a network can achieve good performance on a complex artificial dataset, it provides no guarantees on how it will perform in any real-world situation. Using real-world problems on the other hand will at least guarantee good performance in *some* situations.

An argument against using only real-world problems, and one that Prechelt briefly mentions, is that using artificial problems one can choose specific issues or features in a dataset to test a solution on. This can not guarantee that a solution will work well on real-world problems, but it *can* show that a solution will *not* work well, and at the same time give a strong indication as to *why* it will not work well. This will require the researchers to provide datasets that have been properly analyzed for features that can cause challenges for an ANN solution.

2.3 DELVE

DELVE - Data for Evaluating Learning in Valid Experiments - is a package consisting of three major parts to aid machine learning researchers. DELVE is not currently under development though, and the last updates were done in 2003. DELVE is still available from the project website[3] hosted by the University of Toronto.

The three parts of DELVE are:

1. *A software environment* for analyzing and manipulating datasets.
2. A package of *datasets* to use. The datasets are either *regression* or *classification* datasets. The datasets are divided into three categories: Development datasets to tune the learning algorithms, assessment datasets to use for testing, and historical datasets used previously in the literature.

[3] http://www.cs.toronto.edu/ delve/

3. A repository of *existing learning methods* that can be used to test datasets or to compare new learning methods against.

The goal of DELVE is, according to the user manual[8],

> [...]to help researchers and potential users to assess learning methods in a way that is relevant to real-world problems and that allows for statistically-valid comparisons of different methods.

The last part of this goal statement is important as the DELVE manual also provides a chapter on how to analyze and report results in such a way that they are statistically valid.

The datasets of DELVE are available from the project website and is divided into the above mentioned categories. Most of the datasets are from the UCI repository and has been reformatted to fit the DELVE data format. The only reported features of the datasets are simple features: number of attributes and the number of cases in addition to previously conducted experiments done using each dataset. In is unclear what further analysis the DELVE developers performed before selecting each dataset for inclusion into the DELVE package.

As mentioned, most of the datasets in DELVE are real-world data from the UCI repository, but they will allow submissions of artificial datasets as well. These artificial datasets are mostly datasets resembling real-world examples, such as movement data from a robot arm. There is no artificial datasets that pose specific a priori challenges to the learning methods. In the DELVE manual they argue that such artificial datasets can be used to answer specific questions though, for example what the effect is of adding extra noise or irrelevant attributes is.

The software part of DELVE will has not been studied and will not be discussed in this thesis.

2.4 ELENA

ELENA, Enhanced Learning for Evolutive Neural Architecture[4], is a research project that was run as part of the ESPRIT[5] programme. One of the goals of

[4]Online at the website http://www.elena-project.org

[5]European Strategic Program on Research in Information Technology, a research program funded by the European Union

the project was to produce a benchmarking solution to use for experiments. The results from this part of the project is given in a technical report from 1995[9].

ELENA contains a database of benchmarking problems, and a research report which covers data analysis, performance evaluation as well as a comparison between both existing classifiers such as K-nearest-neighbor and Learning Vector Quantizer, as well as new classifiers introduced during the ELENA project, such as Piecewise Linear Separation. The ELENA database is divided into two parts: a part with artificial datasets and a part with real world problems.

Three artificial classification problems were generated for "rapid test purposes", and with the following requirements:

1. Varied dimension of the input vectors
2. Only two classes in each problem
3. High degree of intersection between the classes the vectors were to be classified as
4. Highly non-linear boundaries between these classes
5. Already published results on these problems

The second of these requirements for the problem sets is perhaps the most interesting: The ELENA researchers believe that two-class problems will "yield answers to the most essential questions" They don't provide an argument for this, but refer to a 1988 paper by Kohonen, Barna and Chrisley[10]. In this paper no proper argument is given for this statement either, as they also "[...]believed it[two-class problems] yields answers to the most essential questions".

When benchmarking for catastrophic forgetting though, it is more desirable with three or more classes in the benchmark problems. The reasoning for this is explained in section 5.1.

The real-world problems used in ELENA were all selected from the UCI research database. For the real-world problems, having more than two classes were a desired property in at least one dataset to identify possible problems the artificial two-class problems would not have.

3 Dataset Analysis

In this chapter I will look at various ways to analyze the datasets used for training neural networks. Such analysis help researchers identify features in the data, which are essential when one is to select datasets for evaulating the performance of neural networks.

In addition to presenting some some well know features, based on the work of Zijian Zheng[7] and partly on that of Lutz Prechelt in his work on the Proben1 package[6], I will introduce two new measurements that can be particularly useful when researchers are concerned with testing for catastrophic forgetting: *Input Overlap*(IO) and *Grouped Input Overlap*(GIO).

Information is a highly ambiguous term, which got different meaning for different groups of people, and that is often used undefined. In the context of data set analysis I use *data* to mean raw data, and *information* to mean data that is shaped to a form where it got meaning for humans. These definitions are in accordance with common usage in information science literature, such as [11] and [12]. For the datasets that this project is concerned with, the raw numbers and bits stored in the vectors and neural network weights are typically data, while the properly formatted inputs and outputs from the network, the analyzed dataset features, and the stored dataset are all considered information.

This chapter will provide foundations for my answer to both the second and third question from the introduction. The real-world datasets will be selected based on the information provided here, and IO and GIO are meant to be used as measurements to check how liable datasets are to be forgotten.

3.1 Zheng's Benchmark

Zijian Zheng introduced in the paper "A benchmark for classifier learning"[7] a taxonomy of 16 features that serve to describe datasets. These features are split into three categories:

- Features that look at the *attributes* of the dataset. In the terminology of this report, this means looking at the items of the input patterns. For instance will the first of item of each input pattern be the same attribute for different instances. Figure 2 shows the concept.
- Features that look at the *instances* of the dataset. This means exam-

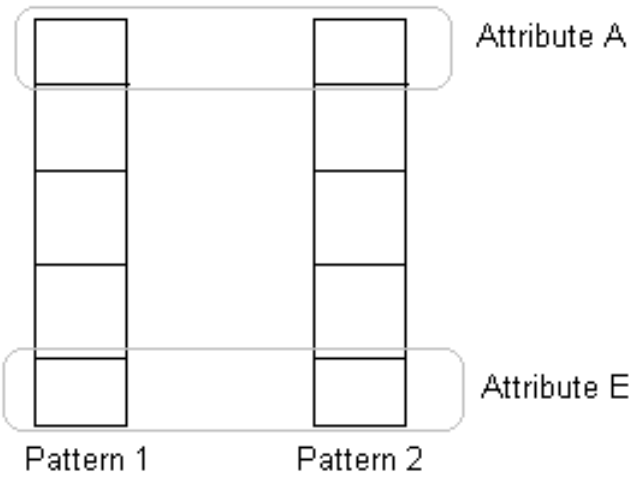

Figure 2: A two-instance dataset with 5 attributes.

ining the input and output vectors of the individual instances, but not viewing the classification of the whole dataset.

- Features that examine the *classes*. These include the most advanced and computationally expensive features.

All the 16 identified features are normalized into nominal values. Zheng does this by introducing terms such as "low", "high" and "medium" where these refer to the first, fourth and second, and third quadrants of all the values for that attribute respectively. The reason for doing this is to make it possible to later select an adequate set of test cases from the UCI repository such that all of these nominal values are covered. The problem with this approach will of course present itself when one introduces a new problem set into the benchmarking suite at a later stage. If this problem set causes features of existing problems to be in a different quadrant, the one does no longer have the wanted coverage and the selection of datasets might have to be repeated again from start.

As mentioned above, the benchmark problems Zheng selected were all from the UCI repository. He selected this using a coverage algorithm to have at least two problems with each possible feature value, and the problems were only selected from datasets recently used in published literature.

In the following sections, a quick introduction to the 16 features presented by Zheng is given.

3.1.1 Attribute Features

- **Type of attributes**
  This relates to how the attributes, or elements in the data vectors, are encoded. For instance boolean attributes, 1-of-C outputs[6], integral attributes or real-valued attributes. How one encodes the input to and output from a neural network is of vital importance for how well the network will perform. If for instance the output in a three-class problem is encoded as an integer in the range [0-,2] one is likely to see more mis-classifications than if one uses 1-of-3 encoding. Consider the case where an instance between class C_1 and class C_2 is presented to the network. One is likely to see the network trying to classify it as both, leading to class $C_3(1+2)$ which is obviously wrong. With a 1-of-3 encoding, one would instead have it classified as either C_1 or C_2, or perhaps both. In any case it is closer to the expected behavior than C_3. Even when it is not clear what the best way to encode a problem is, performance can change significantly if it is changed, so explicitly identifying the encoding it is of vital importance.

- **Number of attributes**
  Related to the encoding of the inputs and outputs is the size of the input and output layers in the network. The performance of a given network architecture or training algorithm will depend on the topology of the network. Changing the size of the layers interfacing with the outside of the network will often have great impact on this topology, and thus also be a deciding factor in the overall performance.

- **Number of different nominal attribute values**
  If the attributes are encoded nominally, this will typically be quite low, but if the attributes are of continuous or integral type, the number of nominal values can be high.

- **Number of irrelevant attributes**
  This is a boolean attribute: Are there one or more attributes that are irrelevant to the correct classification of *all* examples? Usually information about this is provided as part of the problem description as it is impossible to accurately extract irrelevant attributes from a random sample of instances and be certain one correctly generalizes over the entire population.

[6] 1-of-C is a commonly used way to encode the output in a classification problem: Each output is boolean valued and the target output is a "true" or 1 for the correct class and "false", -1 or 0 for all others

3.1.2 Instance Features

- **Presence of noise in attribute values**
  Boolean value: Does one or more attributes have noise in them or are they all accurate?

- **Presence of noise in class memberships**
  Boolean value: Are all instances classified accurately or does one or more have noise?

- **Frequency of missing attribute values**
  How often does instances have missing attributes?

- **Data-set size**
  The number of different data items in the dataset is important as it will help determine how different any two test runs on the dataset will be. When testing the network, the dataset must be split into two or more subsets. One must be used as *training set* and one as *test set*. It is also not uncommon to use a third subset as *validation set*. This will work as a test set during training, but not to test the performance once the stopping criterion is reached.

- **Dataset density**
  The dataset density is a measurement for the average "distance" between each input example in the dataset. It does not look at the actual values of the examples however except to calculate the size of each dimension in the input space, so this measurement is a only an indication about how "full" the input space in fact is.

  It is calculated as

$$\text{Density} = \frac{\text{Number-of-examples}}{\prod_{i=1}^{n} N_i} \tag{1}$$

  where n is the number of attributes and N_i is the number of values the attribute at position i can have. If an attribute is continuous N_i is the number of different values it got in the dataset.

3.1.3 Class Features

- **Number of classes**
  How many classes are there in total for the dataset?

- **Default accuracy**
  The default accuracy is obtained by calculating the relative frequency of the most common class in the dataset. For instance if a dataset contains 100 examples in classes C_1, C_2 and C_3 with 60 examples being in class C_1, 30 in class C_2 and 10 in class C_3, the default accuracy is $\frac{60}{100}100\% = 60\%$.

  This number tells us how well the most naïve classifier, one that will classify every given example as the class that had the highest frequency, will work. This default classifier is often used to benchmark new algorithms against.

- **Predictive accuracy**
  This is the highest previously achieved accuracy for the given dataset and is a measurement for how hard a given dataset is to correctly classify. Zheng retrieves this from the UCI repository which includes information about how well published algorithms have performed on every dataset.

  The problem with using this feature is that it will give a bias toward the more used datasets when looking for hard problems (and similarly toward lesser used datasets when looking for easy problems). Problems that have been studied extensively and tested with many algorithms are more likely to have a best reported result that is relatively higher to a theoretical maximum than problem sets that are not studied as extensively. Due to this, I recommend against using this feature to determine the difficulty of a problem as Zheng has done.

  A better approach would be to apply a common set of known classifiers on the datasets and select the best from these instead of the best reported. The obvious problem with this is the resources necessary to tune and test all the algorithms for each problem, something which is out of scope for this thesis.

- **Relative accuracy**
  The relative accuracy is a measurement of how well the default accuracy is when one relates it to the best known accuracy of the problem and is given as

$$\text{Relative accuracy} = \frac{\text{Predictive Accuracy} - \text{Default Accuracy}}{100\% - \text{Default Accuracy}} \cdot 100\%$$

  Since this depends on the predictive accuracy it is subject to the same problems as described in the previous section, and I advise against

using this as a measurement of the *difficulty of the domain* as Zheng suggests.

- **Entropy**
  Entropy, or more specifically information entropy or Shannon's Entropy, is a common measure that is used to describe the purity of a signal. It is used in telecommunications when dealing with noisy transmission lines, and also in machine learning[13] where it is very useful when analyzing data sets to see how much information is really stored.

  If we have a set S of training instances for the neural network, and each of these has an expected output of one of c possible values, the entropy of S is given as

$$Entropy(S) = \sum_{i=1}^{c} -p_i \log_2 p_i \tag{2}$$

  p_i here is the probability that a random sample from S is classified as class i. Formula 2 gives the minimum number of bits required encode the classification for each data item. For datasets used for benchmarking neural networks, this is very useful as it shows how much information that's provided by each data item. For instance if all the data items in the dataset are of the same class, $Entropy(S)$ will be 0: No bits, or no data at all, is required to be transmitted to tell a receiver the class of a given data item.

- **Average Information Score**
  The information score is a measurement introduced by Kononenko and Bratko[14](not read) to take into account the distribution of the different classes in a dataset.

  The definition uses I_{e_j} as the information score of a given classifier on test example e_j. This is defined as:

$$I_{e_j} = \begin{cases} -log_2 P(C_{e_j}) + log_2 P(C_{e_j}) & \text{if } P'(C_{e_j}) \geq P(C_{e_j}) \\ -(-log_2 P(1 - C_{e_j}) + log_2 P(1 - C_{e_j})) & \text{if } P'(C_{e_j}) < P(C_{e_j}) \end{cases}$$

  where $P(C_{e_j})$ is the prior probability, or relative frequency, of class C_{e_j} and $P'(C_{e_j})$ is the posterior probability of the same class as calculated by a chosen classifier, for instance an ANN or a decision tree algorith.

The average information score is then defined as

$$I_{average} = \frac{1}{n}\sum_{j=1}^{n} I_{e_j}$$

Since an existing classifier must be used, this attribute is again subject to the costs related to tuning and experimentation and inductive biases in the classifier used, and results must interpreted with that in mind.

- **Relative Information Score**
  The relative information score is calculated simply by seeing how much the average information score is compared to the entropy of the test set:

$$I_{relative} = \frac{I_{average}}{Entropy_{test}} 100\%$$

3.2 Input Overlap Measurement

In this section I will introduce the *Input Overlap Measurement*. The purpose of this measurement is to provide a tool for neural network researchers to use when analyzing datasets for training and testing. As Frean and Robins pointed out[15] and that was discussed in the 2005 project by myself and Stranden[1], orthogonality of the input data and the level of catastrophic forgetting to expect from a network has a clear relationship. If a level of internal orthogonality can be determined for a dataset, analyzing the results will be easier for researchers.

I will begin this section by reproducing some of the information on orthogonality that was written for the 2005 project mentioned above. Then I will use this information to create two measurements with basis in the geometric interpretation of the input vectors: The Input Overlap Measurement (this section) and the Grouped Input Overlap Measurement (next section).

3.2.1 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 $\mathbf{a}$ and $\mathbf{b}$ is:

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

where a_i is the ith element of $\mathbf{a}$ and b_i the ith element of $\mathbf{b}$. In euclidean space it can be expressed as.

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}||\mathbf{b}| \cos\theta \quad (3)$$

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 3 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.

When measuring the orthogonality of two vectors, it is important to keep all vectors normalized. Considering the three vectors (1,0), (2,0) and (3,0) in the 2-dimensional space, it is clear that they are all represented in one dimension only. Since we are only concerned with the orthogonality and not the distance between the vectors, the level of orthogonality between any two of these three vectors should be the same. The way to achieve this is to normalize the vectors (reduce the length of the vector to one while keeping the same direction) before calculation the orthogonality.

Since the length of both vectors are one, 3 is reduced to a simple cosine calculation. This is also the measurement that will be used to measure the orthogonality between two vectors $\mathbf{a}$ and $\mathbf{b}$:

$$Orth_{\mathbf{ab}} = \cos\theta \quad (4)$$

3.2.2 Input Overlap

Before defining the input overlap, the total orthogonality of an input set must be defined as well. In equation 5 below this is shown as the summed orthogonality between all elements in the set of vectors in.

$$TotOrth_{in} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} (Orth_{\mathbf{in[i]in[j]}}) \quad (5)$$

Using this, we can now define, in equation 6, the input overlap of a set of input vectors to be the averaged orthogonality measurement per input vector:

$$InOverlap_{in} = \frac{TotOrth_{in}}{n} \tag{6}$$

Some interesting results from this formula are:

- *InOverlap* will be in the range $[0, \frac{1}{2}(n-1)]$.
  Proof: The lowest possible orthogonality for any two vectors is 0, so it follows that the lowest possible *sum* of orthogonalities is 0 as well. The number of terms in the sum in equation 5 is given by

$$\begin{aligned}(n-1) + (n-2) + (n-3)...1 & \\ &= \sum_{i=1}^{n-1} i \\ &= \frac{1}{2}((n-1)^2 + (n-1)) \\ &= \frac{1}{2}(n^2 - n)\end{aligned}$$

  Each of these terms got a maximum value of 1, so the highest total value for $TotOrth$ is $\frac{1}{2}(n^2-n)$ as well. It then follows that the highest value that *InOverlap* can have is

$$\begin{aligned}Max(InOverlap) &= \frac{\frac{1}{2}(n^2-n)}{n} \\ &= \frac{1}{2}(n-1)\end{aligned}$$

- An overlap of 0 is only possible if the size of the input set is lower than or equal to the number of dimensions of the input vectors.
  Proof: When the overlap is 0, we got:

$$\begin{aligned}InOverlap_{in} &= 0 \\ \frac{TotOrth_{in}}{n} &= 0 \\ TotOrth_{in} &= 0\end{aligned}$$

  For $TortOrth_{in}$ to be 0 all of the vectors in in must be mutually orthogonal. In any space $\mathbf{R}^n$, at most n vectors can be mutually orthogonal.

From this follows that the input set can be at most the same size as the number of dimensions the vectors are represented in if the overlap is to be 0.

- An overlap of $\frac{1}{2}(n-1)$ is only possible if for any two vectors **v1** and **v2** in in, a scalar c exists such that $\mathbf{v1} = c\mathbf{v2}$. The geometrical interpretation of this is that a straight line can be drawn through all the points expressed by the vectors. For neural networks it means that all the data is stored in one dimension only (even if that dimension is a linear combination of other dimensions). This is the highest possible overlap of data.

- If $InOverlap_{in}$ is calculated to be I and a new input vector, **new** is added to in, $InOverlap_{\{in,\mathbf{new}\}}$ will have the same scalar relationship to I as $\sum_{i=1}^{n} Orth_{\mathbf{in[i]new}}$ has to I. In other words: if the summed orthogonality between a new vector to all existing vectors is higher than the existing overlap, the overlap will increase, if it is lower the overlap will decrease and if it is the same, the overlap will stay the same as well.

3.2.3 Relative Information Overlap

When comparing the overlap in different datasets, a possible useful measurement is the *relative* overlap. This is calculated by simply dividing the overlap by its theoretical max:

$$\begin{aligned} RelInOverlap_in &= \frac{InOverlap_in}{max(InOverlap_in)} \\ &= 2\frac{InOverlap_in}{n-1} \end{aligned}$$

3.2.4 Usefulness of the Input Overlap Method

An obvious limitation of the IO method is that it does not account for clustering of data in the training set. It is only concerned with how many data vectors one tries to store and how high the overlap is between these vectors. When the goal is generalization, as it often is, this overlap can be be desirable to classify new exemplars that are close in Euclidean space as the same class. The goal of the method though is that it can give an indication of how well the input data is distributed in the input space.

3.3 Grouped Input Overlap

A major flaw in the input overlap measurement introduced in the last section is that it does not account for which patterns that have already been learned and which are being learned afterward. To attempt to create a method that takes this into account, i suggest another measurement, termed Grouped Input Overlap(GIO). Instead of measuring the overlap between *all* input vectors, the GIO method only measures the overlap between vectors that are not in the same training group. For example if a network is supposed to train vectors belonging to class C_1 and C_2 first, and those belonging to class C_3 afterward, GIO would group all C_1 and C_2 vectors together in one set and all C_3 vectors in another, and measure the overlap between these two sets.

More formally, we divide the input set in into two disjoint sets A and B such that $A \bigcup B = in$. Then the total *grouped* orthogonality is given as:

$$TotGrOrth_{A,B} = \sum_{i=1}^{n}\sum_{j=1}^{m}(Orth_{\mathbf{A[i]B[j]}}) \tag{7}$$

where n is the number of vectors in set A and m is the number of vectors in set B.

The GIO is then defined as:

$$GrInOverlap_{A,B} = \frac{TotGrOrth_{A,B}}{n+m} \tag{8}$$

where again n is the number of vectors in set A and m is the number of vectors in set B.

This time the range of the measurement is give as $[0, \frac{n \cdot m}{n+m}]$.

Proof: For minimum value, the proof is the same as in the previous section. The maximum value of each of the terms in equation 7 is again 1. The number of terms is obviously $n \cdot m$, so it follows that $max(TotGrOrth) = n \cdot m$ and that:

$$Max(GrInOverlap_{A,B}) = \frac{n \cdot m}{n+m}$$

4 Measuring Catastrophic Forgetting

The empirical approach to measuring catastrophic forgetting is to create a test set and measure how good the network performs on that set at various stages. There are other approaches as well, like measuring the *internal* overlap in the network, but the empirical studies remains the most common. In this chapter I will look primarly at empirical methods, but also at one analytical.

Placing this chapter in context of the original four questions from the introduction, it is meant as an aid to answering the first. If one is to test for catastrophic forgetting, it is necessary to have one or more methods to use when seeing how much information is retained or forgotten.

4.1 Empirical Measurements

When studying how good the network performs, the typical experiment goes something like this:

1. Train the network using a training set until it performs satisfactory on a *test set*.
2. Train the network to learn a *second* training set
3. Test the network on the same test set as during the first training.

There can be variations of this, like the experiments performed by Robins in his pseudorehearsal experiments[16] where he did not provide all new instances at once in a second set, but instead fed them one at a time. Or French that instead of measuring how quickly one forgot the first test set measured how quickly one would relearn the first set after having trained on the second to acceptance.

The test described above is the simplest and the most intuitive when one wants to answer the question "How much of the old information does the network forget?"

The white box of the solution one can change is then how to do the performance test itself. This test is often called a *loss function*.

4.1.1 Binary loss functions

The most obvious approach is to simply count the number of test examples that produce the correct output. This is called a *0-1 loss*. 0-1 loss is a good empirical measurement as in a practical application the only interesting question is: *"is the network doing what it is supposed to do"*. Even though it is good to test the real-world applicability of an algorithm, it is not necessarily the best lost function to use when developing new algorithms as it does not discriminate between an algorithm that is almost correct and one that is far off from being correct. A variant of the 0-1 loss that can be used in certain cases is to count the number of correct outputs for each test example and give a score based on this. For instance if a network got five outputs and for a given test three of these were correct, it would get the a score of 0.6. This will better discriminate between an algorithm that is almost correct (say 4/5 or 0.8 and one that is far away from being correct (like 1/5 or 0.2) than a simple 0-1 loss function.

4.1.2 Averaged squared error

Another commonly used loss function, is the *averaged squared error*. This is calculated by taking the squared difference between the expected output and the actual output for each element in the output vector, then summing them for the entire output vector and taking the average as shown in formula 9.

$$e = \frac{1}{n} \sum_{i=1}^{n} (o_i - d_i)^2 \tag{9}$$

e is then the average squared error for a single pattern.

4.1.3 Averaged squared error delta

In the study done this fall by myself and Stranden, we suggested using a slightly altered version of the averaged squared error measurement for measuring how the error changes. I will now term this the *Averaged squared error delta*. When using this loss function both the output after the initial training, $o_{i_{old}}$ and the output after a second training step has been done, o_i is used. The averaged squared error delta is shown in formula 10.

$$e_{\Delta} = \frac{1}{n}\sum_{i=1}^{n}\left[(o_i - d_i)^2 - (o_{i_{old}} - d_i)^2\right] \quad (10)$$

4.2 Analytical Measurements

Empirical testing can be time consuming and prone to experiments not covering all wanted variables. As mentioned earlier the complexity is so high when dealing with neural networks though, that it is very hard to analytically analyze neural network solutions in regards to performance. The same holds true for the catastrophic forgetting aspect of neural networks, so advances in this direction is very few so far. One attempt was made by Robert French when studying Activation Sharpening. He introduced the concept of *Activation Overlap*. This measurement looks at the *lowest* activation for each hidden node over a set of patterns, and calculates the average of these. I.e for a network with k nodes in the hidden layer, the activation overlap when testing with a pattern set of size n is:

$$\text{Activation Overlap} = \frac{1}{n}\sum_{i=1}^{k}\left[min\left(o_i(I_1), o_i(I_2), ...o_i(I_n)\right)\right]$$

This measurement is not strictly analytical, but it attempts to evaluate catastrophic forgetting without measuring it. However when testing this at the project I did with Stranden last fall, we failed to find any correlation between activation overlap and the rate of forgetting.

When studying the measurement, it is revealed that it is unlikely to give a good indication of the CF performance of a network. For instance will a network where the information is very distributed and nodes only activate on one class each and then with a high output (like the wanted behavior from activation sharpening), the Activation Overlap will still be high even if there is obviously low overlap of activation.

5 Benchmarking Catastrophic Forgetting

In this chapter I will present a set of rules, guidelines and measurement methods for benchmarking catastrophic forgetting based on the analysis and existing work presented in chapters 2, 3, and 4. Together, these chapters form my answer to question one posed in the introduction.

5.1 Dataset selection

When selecting datasets for benchmarking catastrophic forgetting, one is faced with an additional challenge compared to benchmarking of the performance of networks with monolithic training: The data must be dividable into at least three distinct groups that contain different pieces of information. Two groups must be used for the first training step, and a third for the second training step. The simplest approach: to create two subsets of the training set and train one after another is likely to fail if there is redundancy in the training instances. Then one cannot be certain that the second subset contains the desirable amount of new information, and the test will then obviously be flawed.

Creating the desired three groups of input data can be done in two ways:

1. Dividing the data in such a way that new classes are introduced at each step in the training sequence. This is illustrated in Figure 3a. At the first step the network is trained to classify input data in two classes and at the second step a third class is introduced. 2+1 classes is the simplest way to do this, but of course more classes can be introduced at each step, and/or more training steps can be performed.

2. Dividing the data in such a way that the same classes cover different areas of the input space at each step in the training sequence. This is illustrated in Figure 3b and c. In 3b the data for the first and second training steps aren't properly divided so the network will classify the data introduced at the second step even before any training has been performed with those input examples and thus no changes will happen in the network weights during the training of the second set of input data. In 3c the data *is* properly segmented and the network is likely change during the second training step.

If the first approach is used, the datasets must contain at least three classes:

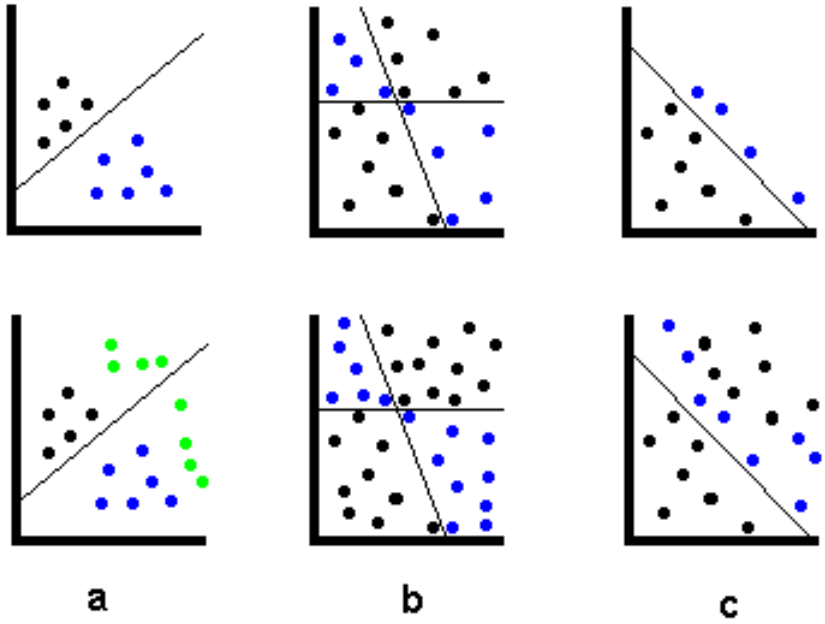

Figure 3: Each dot illustrates a training instance, with x and y coordinates being the two input attributes and the color its class. The lines illustrate how the network discriminates between classes. The top row shows the input space after training of the first input set is complete, the second row shows the input space after having new training examples introduced. The three columns illustrate having different classes introduced at each step(a), dividing the first and second training sets in such a way that new examples are prematurely classified correctly(b) and dividing the first and second training sets in such a way that the network must adapt its hyperplanes to the new information(c).

Two for an initial training and a third to introduce after the initial training is complete. If the dataset contains only two classes, the initial training is useless as it will train a network to classify the input the same way no matter what input it is given.

If the second approach is used, it is desirable that the examples of a class that is split are in areas of the input space that are far apart. This is to reduce the chance that the network classifies the examples in the second set correctly even before they are presented as training examples.

The problem with the second approach is that it is increasingly difficult to compile distinct groups as the number of dimensions in the input space increase and the intersection between the classes go up. Due to this and the simplicity of the first approach, I suggest using training data with three or

more classes *if possible* and segmenting the data as mentioned above when creating training sets for each training step.

5.1.1 Artificial and Real-World datasets

There has been several approaches on how to select to use either artificial data or real-world data. Proben1 contains only real-world problems (see 2.2, DELVE contains mostly real-world problems, but they argue that artificial data can be used to shed light on specific problems (see 2.3), ELENA contains both artificial and real-world problems (see 2.4), and Salzberg argues that "the use of artificial data should always be considered a way to test more precisely the strengths and weaknesses of a new algorithm"[17].

Based on these arguments and approaches, I suggest using an approach close to that chosen by Salzberg. First features must be identified as it was done by Zheng (see 3.1) in real-world data. There is no substitute for actual data, but one should make an informed choice when choosing datasets. Second additional problems one wants to address should be identified, and artificial problems created to examine these in particular. For instance with regards to catastrophic forgetting, examining different degrees of overlaps in the input space, various noise levels, degree of class intersection, etc are all features that one can expect to affect performance and would thus want particularly tailored datasets to test for.

I suggest using all features from chapter 3 except predictive and relative accuracy. These both depend on how extensively the datasets have been used in the past and will therefore not give information that can be used in an objective selection. It is also important to carefully review the inductive bias of the classifier used if the the Information Score measurements are to be used.

5.2 Measurement Methods

When measuring catastrophic forgetting one selects a test set and records performance after training a training set covering the same input and output space and again after training with new information. Performance can also be recording during training of the new information if one is interested in seeing how it develops.

When making a qualitative comparison between methods or parameters, at

least two general methods can be chosen:

- One can select a measurement level, set at a percentage of the best performance and see how many training epochs the network manages to stay above that level.
- One can attempt to fit the performance-epoch plot to a curve and look for a relationship where $\mathbf{f} = k\mathbf{h}$ with k being a constant and $\mathbf{f}$ and $\mathbf{h}$ are the fitted graphs of two different plots.

The "performance" can be measured in several ways as described in section 4.1. There is no best-way to do this measurement, as different setups, datasets and networks can work with different loss functions. What is important to remember though, is that it is possible to work with different error functions for training and testing. It can be suitable to have more detailed error information during training (and such use MSE), but to want a binary correct/wrong answer during testing and thus use a binary loss function then.

5.3 Statistical Guidelines

In addition to what is already mentioned, a few words of caution regarding statistics is in order. It has been pointed out that the reporting of results has been lacking when it comes to statistical accuracy. It is out of scope for this thesis to more than scratch the surface of this issue, and many others can and have done that very well already. For the interested readers, I want to refer to the excellent 1997 article "On Comparing Classifiers: Pitfalls to Avoid and a Recommended Approach" by Steven Salzberb[17]. Here he points out some very important things to remember. Very briefly some of these are:

- **The multiplicity effect**
  Many of the statistical tools that are available were not designed for computational experiments. The number of runs and experimental treatments are often much larger than one would perform using conventional means. As an example, consider a comparative study of fourteen classification algorithms using eleven different datasets. This leads to 154 variations, which were all compared to a default classifier using a two-tailed, paired t-test using a 0.05 level of significance. The multiplicity effect emerges here as this leads to an expected $154 \cdot 0.05 = 7.7$ "significant" results. This is probably not what the researchers wanted.

It is important to be aware of this, and choose significance levels and test methods accordingly before drawing conclusions.

- **Parameter Tuning**
  As everyone that has worked with neural networks are aware of, changing the parameters of the training algorithm (such as the learning rate and momentum of back-propagation) or network topology (such as the degree of connectivity if one is not using a fully connected network) is likely to greatly affect the performance of the neural network. Due to this, researchers frequently tune such parameters before performing tests. What is not always done though, is to make sure that a separate subset of the training data is used for this parameter tuning, and that it is not reused for training or testing later.

- **Valid Generalization**
  A wanted step for researchers when performing comparative studies, is to draw general conclusions based on a limited set of test cases. When doing this, it is important to have a few things in mind. First, one must realize that even if the selection of test problems is made a random from the increasingly popular UCI repository, one cannot draw conclusions that extend *beyond* the UCI repository itself as the UCI repository is *not* a random selection of all possible problems solvable by neural networks. For instance are many of the problems in the repository almost linearly solvable.

Again I refer to Salzberg's article for more details on these topics.

5.4 Reporting

As has been said earlier, accurate reporting is central when one is to do useful benchmarking. It is important that the results are reproducible, and a detailed description of the experiment is thus necessary. I recommend using the setup described for Proben1 in section 2.2, but where one remembers to add details on how the CF testing has been done to the algorithm and/or experiment parameter sections.

6 Implementation

In this chapter I will describe the implementational work done. This isn't directly related to the questions I seek to answer with the thesis, but for me to be able to perform experiments to validate or invalidate my claims, I needed to implement some code in Matlab.

I also provide a section with some examples on how to use the source code at the end of the chapter.

6.1 Design Choices

When creating the source code needed to perform my experiments it was important that the code would be re-usable by others. At the same time I had to adjust to two important restrictions:

- The datasets would be from the UCI repository
- The back-end ANN code was the one made at IDI by Diego Federici and Axel Tidemann.

Due to this, the datasets are in one part of the code adjusted to the UCI repository, and stays that way until they are ready to be handled by the neural networks. Then they are transformed to the format used in the existing ANN implementation.

The location of each source file is based on its function. The root of the source code tree contains the directories `ann`, `datasets`, `dataset_analysis`, experiments `and results`. More information in each of these is provided in the next few sections.

6.2 Datasets

The datasets used were all from the UCI repository and is available in the `datasets` subdirectory. In these directories are the generated datasets used as well as the programs used for generation. The source code for these programs are in section B.3.

6.3 Dataset Analysis

In chapter 3 I provided an overview of 16 attributes in datasets, and an additional two proposed measures (Input Overlap and Grouped Input Overlap). For all of the 16 attributes that could be analyzed automatically, provided the data was presented in the uniform UCI format, I have implemented code for this. In section B.1 the Matlab source code developed for extracting those attributes is presented. All of this source code is in the `dataset_analysis` subdirectory.

6.4 Artificial Neural Networks

For the ANN implementation, I have been fortunate enough to use existing code developed at IDI by Diego Federici and Axel Tidemann. This code was developed to work with recurrent artificial neural networks[7], so I made some minor additions to adapt the algorithms to non-recurrent networks.

The two earlier proposed solutions to catastrophic forgetting, pseudorehearsal and activation sharpening has also been implemented. This was for two reasons: (1) to use together with regular back-propagation as a way to evaluate the Input Overlap and Grouped Input Overlap measurements, and (2) to further analyze their performance, this time with real-world data from the UCI repository.

In section B.2 of the appendix is the source code for the neural network implementation and the algorithms used reproduced.

6.5 Running and testing

For this master thesis more than hundred different neural network tests has been run, often taking several hours per run. With the limited time alloted to the work, it was essential to provide proper tools for running these tests and analyzing the results. In sections B.4 and B.5 of the appendix is the source code used for running the experiments and for analyzing the results produced.

[7] A RANN is a network where temporal information can be stored by providing connections from the output layer to the input layer, thereby creating a network capable of storing temporal information and relational information between different data instances.

6.5.1 Activation Sharpening

I have implemented the code for running activation sharpening experiments like those described by French[18]. The source code is attached in appendix B.4.2. The idea of the activation sharpening algorithm is to differentiate more between different activations. In effect French wishes to reduce the distribution of knowledge such that the data needed to recognize each pattern is stored in fewer nodes that in a regular ANN. This is done by adding an extra training step before the usual back-propagation. During this extra training step, a new activation given by the formulas in equations 11 and 12 is calculated for each of the hidden layer nodes. The selection between equation 11 (to increase the activation) and equation 12 (to reduce the activation) is done by adjusting the number of nodes to sharpen. If there are n nodes in the hidden layer, k are getting increased activations and $n - k$ are getting reduced activations.

The error between this new activation and the actual activation is the back-propagated through the weights connecting the hidden layer to the input layer.

$$A_{new} = A_{old} + \alpha(1 - A_{old}) \tag{11}$$

$$A_{new} = A_{old} - \alpha A_{old} \tag{12}$$

In pseudo code the activation sharpening algorithm is reproduced in listing 1.

Listing 1: Activation sharpening algorithm

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

  # Sort nodes by output value
  sort(nodes)

  # Increase the activation for the k nodes with
  # highest activation
  for i in 1 to k:
      nodes[i].newAct := nodes[i].oldAct +
       alpha*(1-nodes[i].oldAct)
  end
```

```
# Decrease the value of the other n-k nodes
for i in k+1 to n:
    nodes[i].newAct := nodes[i].oldAct -
     alpha*nodes[i].oldAct

# Propagate the difference between new and
# old activation to the weights between the
# input and hidden layer. WeightUpdate is
# the common update rule with learning rate
# used in backprop
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()
```

6.5.2 Pseudorehearsal

The idea for pseudorehearsal followed from the thought that one should rehearse on old information to make sure it was remembered. But instead of storing old patterns to use together with the new ones when training, random patterns were generated. This would remove the requirement to store the old patterns while - in theory - making sure the old information was kept.

In the original pseudorehearsal experiment a pool of 8, 32 or 128 patterns were generated and selected three patterns from this pool at random each time a new instance was being learned. However my experiments used the commonly used batch training method where a larger set of training instances are being learned at the same time. To adapt the pseudorehearsal algorithm to this, I used the *entire* pool for training. When training a second set of instances, a pool of patterns were generated and added to the training and testing sets. Simple pseudo code for the implemented code, leaving details for logging and testing out, is in listing 2. The complete source code is in appendix B.4.1.

Listing 2: Pseudorehearsal pseudo code

```
function pseudoRehearsal(poolSize, maxEpoch):

  # Learn initial training set
  epoch = 0
  while(test(initialTestSet)==failed or epoch>maxEpoch)
    train(initialTrainingSet)
    epoch := epoch+1
  end

  # Generate a pool of 8, 32, or 128 (and in some cases
  # more) pseudo patterns
  pseudoSet := generatePseudo(poolSize)

  # Combine the pseudo pool together with the second
  # training and test sets
  trainingSet := secondTrainingSet + pseduoSet
  testSet :=secondTestSet + pseudoSet

  # Learn the combined second set
  epoch = 0
  while(test(testSet))==failed or epoch>maxEpoch)
    train(trainingSet)
    epoch := epoch+1
  end
```

6.6 Usage

The first thing to do when using the code is to change working directory to the root of the source code tree and add paths recursively:
`» addpath(genpath(pwd))`

Then all functions should be added and ready to be used.

The tests are run by the functions called `run_X_Y_test` where `X` is the dataset and `Y` is the test type. For instance will `run_led_pseudo_test` run the pseudorehearsal experiment with the LED dataset. The test files only take a file name as input parameter, while the rest of the information is edited when the experimental settings changes. In many ways they are more configuration files just holding variable data than programs themselves.

To run a test more than once and get the averaged results, the function `run_averaged_experiment` is used. This takes a function reference, a base file name and a number of runs as parameters in addition to a parameter telling the program if all runs are to use the same filename or not. For instance will the following run 20 pseudorehearsal experiments with the LED7 data without irrelevant attributes or noise and with 300 instances:

```
>> runs = run_averaged_experiment('led7_300_0noise',
          0, @run_led_test, 20)
```

Here `runs` will be a structure containing both complete data for each of the 20 runs as well as statistical data like mean and standard deviation from the mean. The structures called `first`, `second` and `third` are data for the first training batch (initial training), second training batch and the performance on the first test set while training the second training set.

This will use the datasets `led7_300_0noise1.data` , `led7_300_0noise2.data` , ... `led7_300_0noise20.data` . The datasets must be generated before running the experiments from Matlab. This can be done with the Unix bash script `makecustom.sh` . To generate the datasets in the above example, the command

```
./makecustom.sh 20 300 0 0
```

would have been executed.

For analyzing the data, some functions are available. The function used during experiment 1 in the next chapter analyzes one set of data from the

pseudo pattern experiments and one back-propagation data (all averaged). The data in the pseudopattern experiment must be saved with the names `log_N` where `N` is the number of pseudo patterns used. Likewise the data in back-propagation tests must be saved with the name `runs`. The analysis function takes a base filename to use when searching for the back-propagation and pseudorehearsal log files. For instance:

```
>> runs = run_averaged_experiment('led7_300_10noise', 0, ...
          @run_led_pseudo_test, 20);
>> log_8 = runs.avg.third.mean;
>> save pseudo_led7 log_8
>> runs =  run_averaged_experiment('led7_300_10noise', 0, ...
          @run_led_test, 20);
>> save cf_led7
>> [normal_best, pseudo_best] = analyze_cf_results('led7')
```

will run a test where the configuration is 8 pseudo patterns (the configuration file `run_led_pseudo_test` must have been edited first), save this, run a back-propagation test, save that as well, and then analyze to see when they fall below 50% performance (for details on this see next chapter).

The dataset analysis part of the source code works on datasets loaded from their UCI repository data files through one of the `load_X` where `X` is dataset type like waveform or LED. These return the input and output in two different matrices, such that the input attributes for instance one is in row one of the input matrix and the output class for that is in row one of the output matrix. Some analysis method requires one of these and some both. For instance can the entropy and input overlap of a LED dataset be checked with:

```
>> [input, output] = read_led('led7_300_10noise1.data');
>> entropy(output)

ans =

    3.2856
>> input_overlap(input)

ans =

  105.4585
```

For IO and GIO, there is also functions that average results using a number of datasets much like the neural network tests. For instance to calculate the average input overlap for all the 20 datasets used above:

```
>> io = average_input_overlap('led7_300_10noise', @read_led, 20)

io =

               mean: 100.6291
                std: 1.9641
    theoretical_max: 149.5000
            percent: 67.3105
```

7 Experiments

In this chapter I will describe the experiments performed for this thesis, report the results and discuss these.

To aid in answering the questions asked at the beginning of this thesis, there were three issues I wanted to study by experimentation:

1. Can the Input Overlap and Grouped Input Overlap be used to predict Catastrophic Forgetting? Does a correlation between these measurements and the rate of forgetting exist?
2. How does different attributes in the dataset affect the rate of forgetting?
3. How well does the studied solutions work with datasets expressing real knowledge?

The first issue was important as it could give researchers and engineers information *before* training their networks on how much information the networks would expect to forget.

Answering the second question could help them improve performance as they would have more information on how to change the datasets if they experienced high rate of forgetting.

The third question is interesting as the two proposed solutions had, to my knowledge, not previously been tested using real-world datasets, but only with randomly generated artificial data.

To select datasets for these experiments, I used the selection Zheng made in his report, but with the added requirement that the data had to contain *at least* three classes. This requirement was added so the datasets would follow the requirement set out in section 5.1. This left five datasets. One (The Lymphography set) was discarded as it contained four classes, but in two of these were only two and four instances which left little room for training and testing with three different classes. Another (the NetTalk dataset) was no longer available from the repository and could not be used. That left only three suitable data sets: The LED problems LED-7 and LED-24 and the Waveform problem (all described in detail later in the chapter). After expanding these datasets into a total of six problems they were tested using the following setup from chapter 5:

1. Divide the dataset into two disjunct subsets: The first set **A** containing all instances falling in all but one output class and the second set **B** containing the rest of the instances.

2. Train the network using **A** (dividing it into training and test sets first)

3. Use the network that performed best on **A**, and train that to learn **B**. During this second training, measure the network's performance on the test subset of **A**.

4. Repeat step 1 to 3 20 times and record average results.

7.1 Problem Descriptions

This section describes in details the experiments and problem domains in accordance to the setup in section 2.2. In table 1 are the results from Zheng's analysis on the three datasets used, expanded to the six experiments

7.1.1 LED 7

1. **Problem:** This problem is the "LED display domain" from the UCI repository, found in the "led-display-creator" directory. It contains seven boolean attributes signaling if each of seven diodes is on or off, and ten output classes, each being one of the ten decimal digits. The dataset was used without noise and with 10% noise.

2. **Datasets:** 20 datasets of 300 instances each were generated from the program available from the UCI repository. For each experiment, one dataset was used, and one output class in this dataset selected at random for the second training batch. The instances falling into one of the remaining nine classes were used for the first training batch. For both the first and second training batch, 70% of the instances were used for training and 30% for testing. Tuning the parameters was done with a separately generated dataset.

3. **Network:** The network used was a 7-40-10 fully-connected feed-forward network. Each of the ten output nodes were interpreted as binary signals with a one signaling that the network classified the input being in that output class and a zero as the network not being in that output class. The sigmoidal activation function was used in all nodes.

4. **Initialization:** All initial weight values were selected from a normal distribution with mean at 0 and with a standard deviation of 0.2.

5. **Algorithm:** Different training algorithms were used for the different tests. The details about these can be found in section 6. For the back-propagation tests, the learning rate was set to 0.9, and no momentum term was used.

6. **Termination:** Training was done until a 100% classifaction rate was achieved, or training had been done for the maximum number of epochs. Once perfect classification rate or max epoch was reached, training moved on to the next set were the same training and testing was done, this time on the second training set. Once this was terminated in the same way, the experiment was concluded.

7. **Error Function:** For each of the ten output nodes the node was considered correct if it was within 0.49 from the target output (0 or 1). MSE was calculated for training, and binary loss was used when testing for acceptance. A training instance was considered correctly classified if all output nodes were correct, and considered as incorrect if one or more nodes were incorrect.

8. **Experimentation Parameters:** 3000 epochs in each training run, 20 runs with reported results taken as the average of these. All runs were used for reporting, even if the network did not converge during the first training run.

7.1.2 LED-24

1. **Problem:** This problem is the "LED display domain" from the UCI repository, found in the "led-display-creator" directory. It contains 24 boolean attributes encoding the status of 24 diodes (on or off), and ten output classes, each being one of the ten decimal digits. The problem is the same as the LED-7 problem, but with an additional 17 irrelevant attributes added. The dataset was used without noise and with 10% noise.

2. **Datasets:** 20 datasets of 300 instances each were generated from the program available from the UCI repository. For each experiment, one dataset was used, and one output class in this dataset selected at random for the second training batch. The instances falling into one of the remaining nine classes were used for the first training batch. For both

the first and second training batch, 70% of the instances were used for training and 30% for testing. Tuning the parameters was done with a separately generated dataset.

3. **Network:** The network used was a 24-40-10 fully-connected feed-forward network. Each of the ten output nodes were interpreted as binary signals with a one signaling that the network classified the input being in that output class and a zero as the network not being in that output class. The sigmoidal activation function was used in all nodes.

4. **Initialization:** All initial weight values were selected from a normal distribution with mean at 0 and with a standard deviation of 0.2.

5. **Algorithm:** Different training algorithms were used for the different tests. The details about these can be found in chapter 6. For the back-propagation tests, the learning rate was set to 0.9, and no momentum term was used.

6. **Termination:** Training was done until a 100% classifaction rate was achieved, or training had been done for the maximum number of epochs. Once perfect classification rate or max epoch was reached, training moved on to the next set were the same training and testing was done, this time on the second training set. Once this was terminated in the same way, the experiment was concluded.

7. **Error Function:** For each of the ten output nodes the node was considered correct if it was within 0.49 from the target output (0 or 1). MSE was calculated for training, and binary loss was used when testing for acceptance. A training instance was considered correctly classified if all output nodes were correct, and considered as incorrect if one or more nodes were incorrect.

8. **Experimentation Parameters:** 3000 epochs in each training run, 20 runs with reported results taken as the average of these. All runs were used for reporting, even if the network did not converge during the first training run.

7.1.3 Waveform

1. **Problem:** This problem is the "waveform" problem from the UCI repository. The problem consists of a C program that generates waves.

Each wave is constructed from one of three base forms (that acts as output classes), and 21 input attributes. All of these attributes got noise added from a normal distribution with mean 0 and variance 1.

2. **Datasets:** The waveform tests were done using the dataset provided on the UCI repository that contains 5000 instances. Due to the generator program's dependency on an old statistical tool, it was not possible to generate more instances.

   One test was run with all 5000 instances using the first two thirds for training and the last third for testing. One of the three output classes were selected for the second training batch at random, and the last two used for the first training batch.

   The second test was run using 500 instances. These were selected at random from the population of 5000. Again two thirds were used for training and one third for testing. Selecting instances for the second training batch was also done in the same way.

   Tuning the parameteres for the tests was done by a similar subset of 500 different from those used in the 500 tests.

3. **Network:** The network used was a 21-40-3 fully-connected feed-forward network. Each of the three output nodes were interpreted as binary signals with a one signaling that the network classified the input being in that output class and a zero as the network not being in that output class. The sigmoidal activation function was used in all nodes.

4. **Initialization:** All initial weight values were selected from a normal distribution with mean at 0 and with a standard deviation of 0.2.

5. **Algorithm:** Different training algorithms were used for the different tests. The details about these can be found in chapter 6. For the back-propagation tests, the learning rate was set to 0.2, and no momentum term was used.

6. **Termination:** Training was done until a 100% classifaction rate was achieved, or training had been done for the maximum number of epochs. Once perfect classification rate or max epoch was reached, training moved on to the next set were the same training and testing was done, this time on the second training set. Once this was terminated in the same way, the experiment was concluded.

7. **Error Function:** For each of the ten output nodes the node was considered correct if it was within 0.49 from the target output (0 or 1).

MSE was calculated for training, and binary loss was used when testing for acceptance. A training instance was considered correctly classified if all output nodes were correct, and considered as incorrect if one or more nodes were incorrect.

8. **Experimentation Parameters:** 1000 epochs in each training run, 20 runs with reported results taken as the average of these. All runs were used for reporting, even if the network did not converge during the first training run.

Feature	LED-7 no noise	LED-24 no noise	LED-7 10% noise	LED-24 10% noise	Waveform 500	Waveform 5000
Type of Attributes	Binary				Continuous	
Size	300				500	5000
# of Attributes	7	24	7	24	21	21
Missing Attributes	0					
# Nominal Values	2 per attribute				N/A	
# Irrelevant Attributes	0	17	0	17	0	0
Attribute Noise	No		Yes			
Class Noise	No					
Missing Attributes	No					
Dataset size	300				500	5000
Dataset density	2.34	$1.79 \cdot 10^{-5}$	2.34	$1.79 \cdot 10^{-5}$	$7.25 \cdot 10^{-51}$	$7.68 \cdot 10^{-57}$
# Classes	10				3	
Default Accuracy	10.0				33.9	
Predictive Accuracy	100		70.0	71.0	86.0	
Relative Accuracy	100		66.7	77.8	78.8	
Entropy	3.29				1.58	
Information Score	Needs runs from other classifying methods, and is not used					

Table 1: The table shows the attributes for the different datasets used in experiment 1. For predictive (and relative) accuracy the values reported by Zheng is used. For the other attributes, calculations has been done with the datasets used in the experiments.

7.2 Experimental setup

I this section I describe the experiments run using the artificial neural nNetworks setup in Matlab. I first describe the originally planned tests, how they were performed, why one of them failed and why more experiments were needed. Then I describe the expanded LED tests that were run to get more data for drawing conclusions regarding the input overlap measurements.

7.2.1 Experiment 1 - Testing original datasets

For testing the input overlap measurement, I used the datasets selected by Zheng from the UCI repository, with the additional requirement that each dataset also contained three or more output classes. This was to ensure that the test for catastrophic forgetting could be performed according to the guidelines in section 5.1.

The datasets fulfilling these requirements were the LED-7, LED-24 and Waveform datasets described in the previous section. For all experiments, 20 equal runs with different random seeds were performed, and the averaged results reported in the next sections.

The following three tests for catastrophic forgetting were planned:

1. How quickly is 50% of the original knowledge lost when tested with a regular back-propagation network. I.e for how many training epochs does the network contain half or more of the original knowledge?

2. How quickly is 50% of the original knowledge lost with the best-performing pseudo-rehearsal experiment. For each dataset, 3-5 pool sizes were used for pseudo-rehearsal. This was 8, 32 and 128 as used in the original experiment by Robins as well as pool sizes relative to the size of the second training set: equal the size and half the size. If the size of the second training set was less than 256, only the equal-sized pool was tested, and if it was less than 128 only the original three pool sizes were tested.

3. How quickly is 50% of the original knowledge lost when using the activation sharpening algorithm? However, this test was discarded during experimentation as networks trained with activation sharpening failed to classify even the first training set properly. Without the network remembering first training, measuring forgetting would be impossible.

50% was chosen rather arbitrarily, but with a few restrictions: The selected level should show a significant drop in performance, but not too low to be subject to acceptance due to patterns being recognized as a result of noise.

As described in the problem description, 40 hidden nodes were used in all the tests. This was chosen as hidden layers sizes at that magnitude has been successful for many problems. The fact that the number of hidden nodes are kept constant between the experiments and the number of inputs and outputs are varied means that there will be a difference in how much storage space is available to the different tests. It is not possible to keep both these variables constant at the same though and one can only speculate as to what differences varying either of these will have. Considering these conditions the simplest solution (keeping the number of hidden nodes constant) was chosen. For reference the number of total weights is in table 7.2.1

Experiment	Inputs	Hidden size	Outputs	Total weights
LED-7	7	40	10	730
LED-24	24	40	10	1410
Waveform	21	40	3	1003

Table 2: The total number of weights for the networks in experiment 1. The calculation is $(input + 1) * hidden + (hidden + 1) * output$. The extra added nodes are bias nodes.

7.2.2 Experiment 2 - Expanded LED tests

Since only six experiments were run as a result of the datasets selected from the Zheng sets, further testing was needed. These new experiments were all done using the LED dataset. Since the LED dataset was generated from a C program available from the UCI repository, three variables could be tuned freely: The number of instances, the percentage of noise and the number of irrelevant attributes. By changing these attributes, I hoped to be able to see further connections between the input overlap and the rate of forgetting.

The Waveform dataset was also provided with a generator, but as that source code was dependent on external programs no longer available and thus new instances could not be generated, only the LED problem was chosen for further study.

The new tests performed were the following:

- Keep the number of inputs constant at 7 and the number of instances constant at 300, while varying the noise from 0 to 25%.
- Keep the number of inputs constant at 7 and the noise constant at 0%, while varying the number of instances from 100 to 2000.
- Keep the level of noise constant at 0% and the number of inputs constant at 300, while varying the number of irrelevant attributes between 0 and 17 (and thus the number of inputs from 7 to 24).

In all tests, the questions I wanted answer to were

> How does the input overlap measurements change with the changing variables? How does the rate of forgetting change? Is it a relation between the two?

Except for the changes described above, the experiments were equal to those described in section 7.1.1.

This time, a level of 75% recognition was chosen as a check-point. The reason for this was that with the very noisy datasets, performance was so low (going as low as 45% after training the first set) that it would never go below 50% recognition.

7.3 Results

In this section I describe and analyze the results from the two experiments run.

7.3.1 Experiment 1

In table 3 are the summarized results from those tests from the first experiment that were not dropped, as well as the calculated Input Overlap and Grouped Input Overlap for the datasets. In table 4 are the same results and the calculated Grouped Input Overlap. In appendix A are more detailed results from these experiments.

To be able to visually look for a relationship between how quickly the network forgets and these measurements, plots of the observations can be seen in figures 4(Input Overlap) and 5(Grouped Input Overlap). From these plots and the results in the tables I observed the following:

- Measured in percentage of maximum, all datasets are roughly between 55 and 80% for both IO and GIO.
- Measured in percentage of maximum, both IO and GIO are virtually equal for the two different-sized waveform tests.
- Measured in percentage of maximum, GIO is significantly lower for the waveform datasets than for the led datasets.
- Isolating the LED tests (the four observations with highest epoch), the LED-24 test with noise deviates from a linear relationship in both GIO and IO tests.
- Regarding pseudorehearsal, performance is a lot better for the LED tests than for the waveform tests.

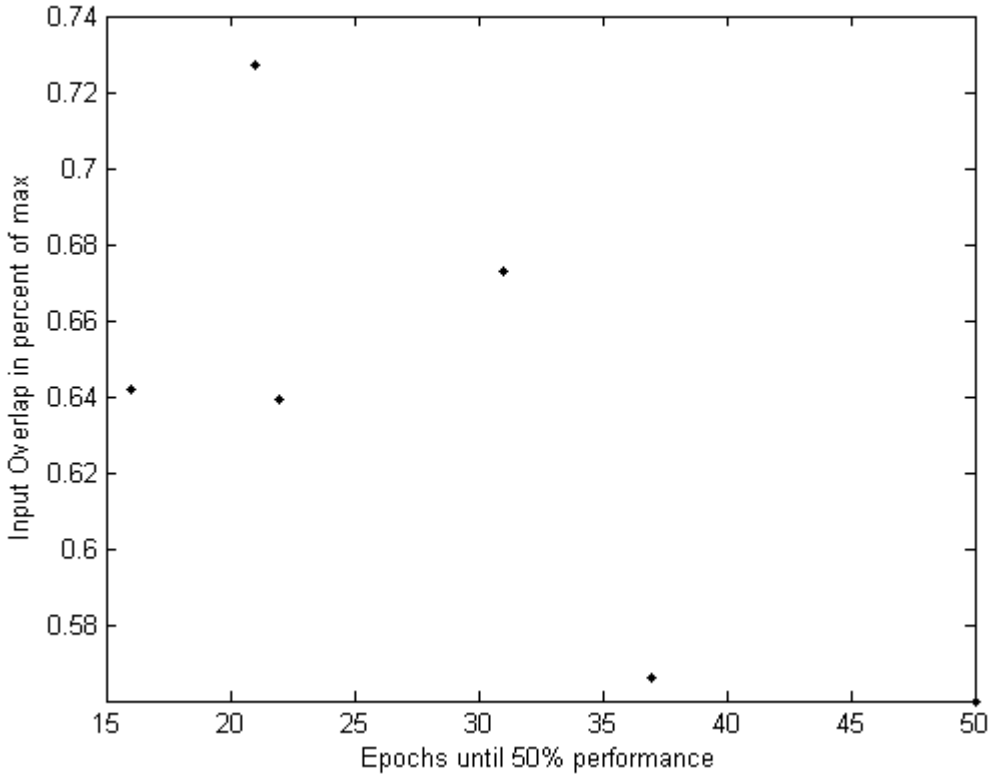

Figure 4: The figure shows the relationship between the Input overlap and the number of epochs before a network trained with back-propagation had forgotten 50% of the originally learned data

Name	Noise	Instances(n)	Input Overlap	Max IO ($\frac{1}{2}(n-1)$)	Input Overlap % of max	Basic Network	Pseudo Rehearsal
LED-7	No	300	108.65	149.5	72.7%	21	Never
LED-7	10%	300	100.63	149.5	67.3%	31	469
LED-24	No	300	83.69	149.5	56.0%	50	401
LED-24	10%	300	84.65	149.5	56.6%	37	362
Waveform	Yes	5000	1598.3	2499.5	63.9%	22	7
Waveform	Yes	500	159.8	249.5	64.0%	16	34

Table 3: T

his table shows the results from running the first experiment and the calculated Input Overlap, the maximum Input Overlap, and the Input Overlap relative to the max. The numbers in the two right-most columns are the number of epochs the network was trained on the second training set before it had a performance less than 50% of the best on the first test set. For pseudorehearsal the reported number is that from the *best* pseudorehearsal run.Input Overlap results summary

Name	Noise	Instances(n)	Grouped Input Overlap	Max GIO ($\frac{n \cdot m}{n+m}$)	GIO % of max	Basic Network	Pseudo Rehearsal
LED-7	No	300	20.8	28.6	72.6%	21	Never
LED-7	10%	300	17.4	26.7	65.1%	31	1946
LED-24	No	300	15.5	26.6	58.2%	50	401
LED-24	10%	300	14.2	26.1	54.2%	37	350
Waveform	Yes	5000	652.8	1111	58.7%	22	7
Waveform	Yes	500	64.75	112.8	57.4%	16	34

Table 4: This table shows the results from running the first experiment and the calculated Grouped Input Overlap, the maximum Grouped Input Overlap, and the Grouped Input Overlap relative to the max. The numbers in the two right-most columns are the number of epochs the network was trained on the second training set before it had a performance less than 50% of the best on the first test set. For pseudorehearsal the reported number is that from the *best* pseudorehearsal run.

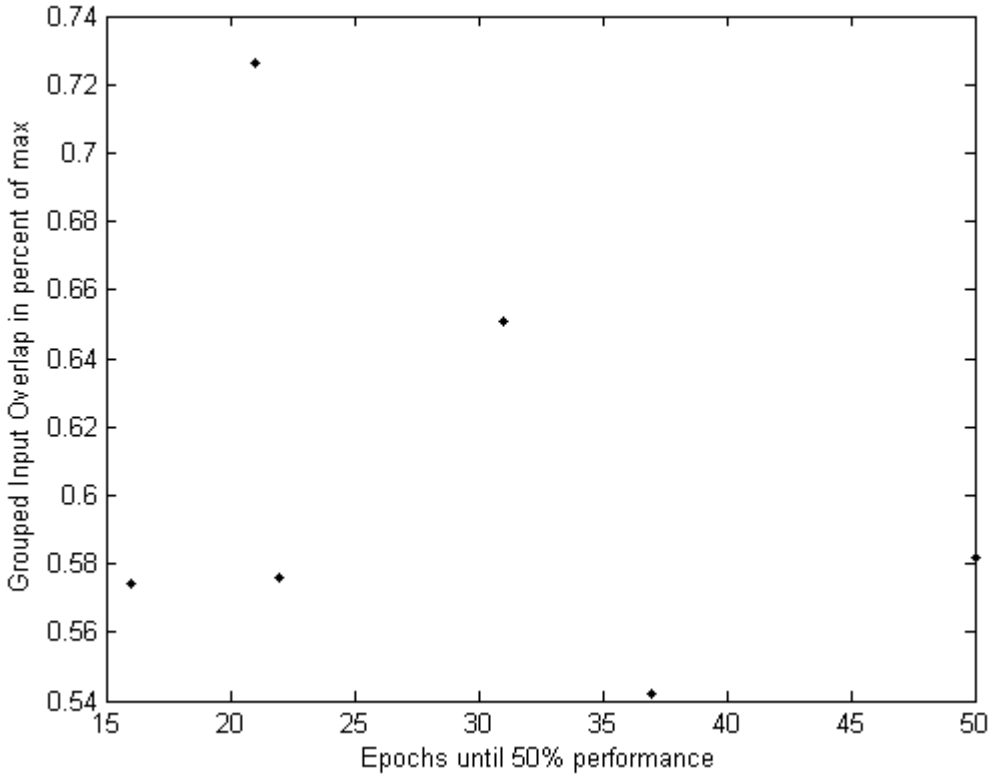

Figure 5: The figure shows the relationship between the Grouped Input Overlap and the number of epochs before a network trained with back-propagation had forgotten 50% of the originally learned data

7.3.2 Experiment 2

The results from experiment 2, are shown in tables 5 (varied noise), 6 (varied number of irrelevant attributes), and 7 (varied number of instances used for training and testing). In all of these tables the relative Input Overlap and Grouped Input Overlap has been measured and is recorded. In figures 6 and 7 are plots with all the tests. In these the number of epochs until 25% knowledge was forgotten is plotted against either the IO or the GIO.

For the experiment with varied noise (table 5), it is unclear if a relationship exists. The slowest forgetting occurs with between 3% and 15% noise, while both the high and low ends of the test exhibit high rate of forgetting. It is worth mentioning that in the tests with noise above 15%, performance dropped drasticly when training the first set of instances compared to noise-free sets. This means that less patterns needs to be forgotten before the 25% limit is reached.

Noise Percent	IO % of max	GIO % of max	Epochs until 75% performance
0	72.7	69.7	8
3	71.3	67.2	24
5	69.7	64.9	19
10	67.3	64.0	22
15	64.5	64.1	24
20	62.1	62.8	10
25	59.8	60.3	10

Table 5: The table shows the results from the LED tests with noise varied between 0 and 25 per cent. Input Overlap and Grouped Input Overlap was calculated and the measured overlap in percentage of the theoretical max is reported in the above table. The last column shows the epoch in which the basic back-propagation network fell first below 75% recognition rate on the first test set.

For the test with varied number of irrelevant attributes (table 6), a relation between the rate of forgetting and the overlap measurements seems apparent. With 0 irrelevant attributes, both IO and GIO is at its highest while forgetting is occurs fastest and with the maximum number of 17 irrelevant attributes, IO and GIO is at its lowest while the network forgets the slowest.

Table 7 shows the results from varying the number of instances. The interesting part of this is that both the relative IO and relative GIO stays the

Irrelevant Attributes	IO % of max	GIO % of max	Epochs until 75% performance
0	72.7	70.5	8
1	70.5	66.1	9
2	68.6	67.0	9
4	65.9	65.8	16
6	63.7	61.5	16
8	62.4	60.1	15
10	61.1	59.8	18
12	60.2	59.8	17
14	59.2	57.5	18
16	58.4	56.8	22
17	58.4	56.4	22

Table 6: The table shows the results from the LED tests with the number of irrelevant attributes varied between 0 and 14. Input Overlap and Grouped Input Overlap was measured and are reported in percentage of their theoretical maximum in the table. The last column shows the epoch in which the basic back-propagation network first had less than 75% recognition rate.

same while the number of epochs until 75% performance is reached varies between 5 and 11. Both of these numbers are within the low 50% percentile of all the measurements while the IO and GIO is in the high 50% percentile.

Total Instances	IO % of max	GIO % of max	Epochs until 75% performance
100	72.6	68.7	11
300	72.7	69.7	8
500	73.0	69.7	7
750	72.9	69.5	5
1000	72.6	71.4	6
1500	72.7	72.1	11
2000	72.9	67.6	8

Table 7: The table shows the results from the LED tests with the number of instances varied between 100 and 2000. Input Overlap and Grouped Input Overlap was measured and are reported in percentage of their theoretical maximum in the table. The last column shows the epoch in which the basic back-propagation network first had less than 75% recognition rate.

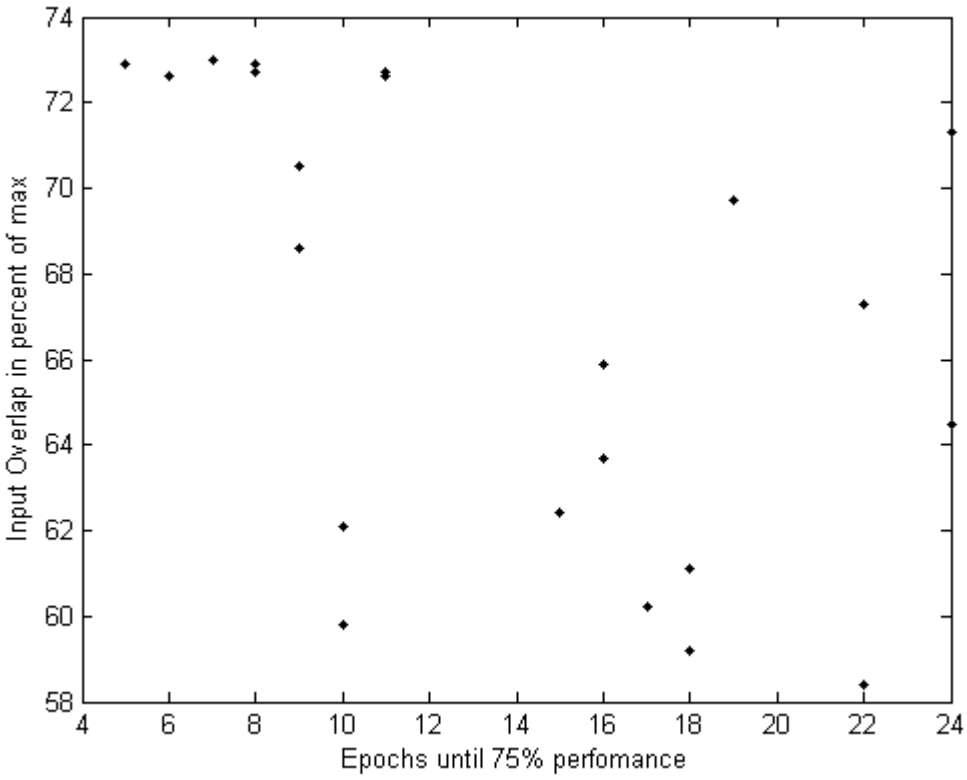

Figure 6: The figure shows the relationship between the Input Overlap and the number of epochs before a network trained with back-propagation had forgotten 25% of the originally learned data. The plot shows *all* the tests done in experiment 2.

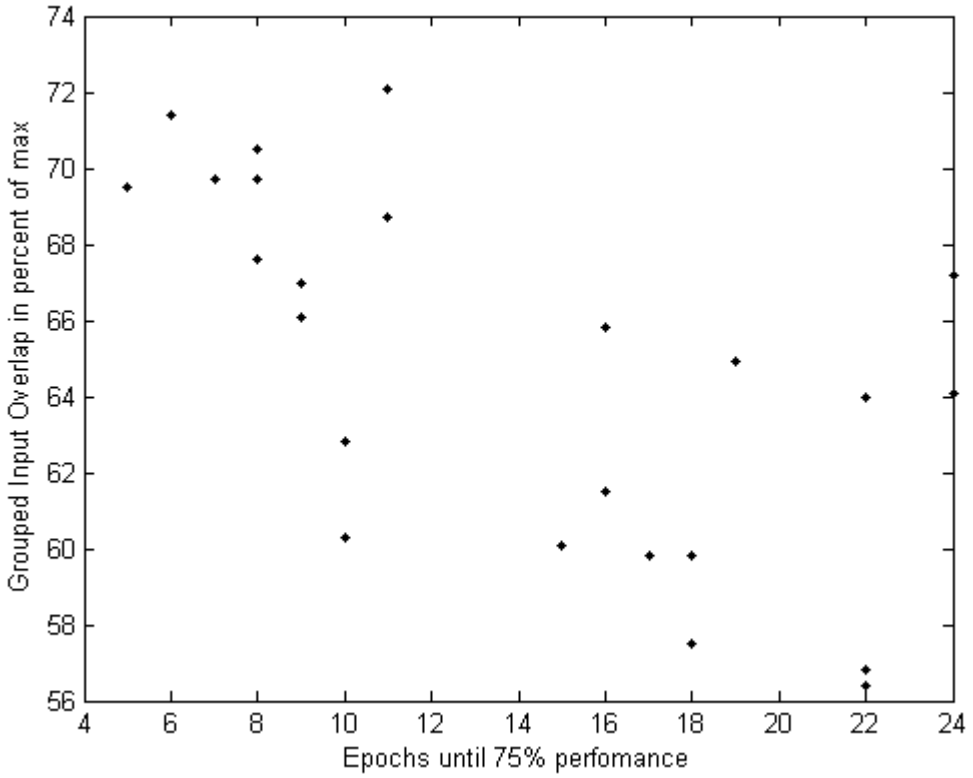

Figure 7: The figure shows the relationship between the Grouped Input Overlap and the number of epochs before a network trained with back-propagation had forgotten 25% of the originally learned data. The plot shows *all* the test done experiment 2.

7.4 Discussion

In this section I will attempt to answer the questions posed at the start of this chapter.

7.4.1 Relevancy of Input Overlap Measurements

Looking at the limited data from experiment one in figures 4 and 5, there seems to be a lack of connection between the Input Overlap measurements and the rate of forgetting when looking at all datasets at once. When one looks at the data from the second experiment, relations are more clear though. Looking at the plots, there is a connection, but with high variance, that shows that higher overlap leads to fewer epochs until the set level of recognition is reached. The connection is stronger for the Grouped Input Overlap measurement than for the Input Overlap measurement. The difference between GIO and IO is small though, and before running the experiments, I expected that GIO would exhibit better results than IO. A probable reason for this is that the classes in the experiments performed are quite uniformly distributed in the input space (due to how they are being generated) and thus there will be little change if overlap is measured against all other instances or only a subset of them.

Due to the limited data used, it is not possible to find a connection between GIO/IO and forgetting *across different datasets*, so by simply measuring the relative overlap one *cannot* with this data conclude about the datasets proneness to forgetting. At least can no such recommendations be given without further study.

7.4.2 Other attributes' relation to forgetting

The 16 attributes identified by Zheng and their values for the datasets used in experiment 1 is listed in table 1. From the experimental results, the following observations can be seen when looking for relations between these attributes and the rate of forgetting:

- Higher input dimensionality consistently leads to less forgetting. This is seen both in the difference between the waveform and LED tests in experiment 1, and when changing the number of irrelevant attributes (and thus the number of dimensions in the input space). This is a

natural consequence of increasing the storage space available without increasing the amount of information to store.

- A related observation can be made when the number of instances are varied. As the *information* these instances store (how to divide the input space for classification) is not changed much when increasing the number of instances, we cannot see a significant change in the rate of forgetting either. Between the Waveform 500 and Waveform 5000 tests there is a small difference where the 500 instances tests exhibit a bit higher rate of forgetting (16 vs 22 epochs). For the expanded LED tests with varied number of instances in experiment 2, it seems that the number of instances to learn has no effect on the rate of forgetting.
- It is unclear how noise affects forgetting as it reduced the performance of the regular training greatly in the LED tests performed.
- From the available data, it is not possible to see a connection between entropy and forgetting. Since the entropy is unchanged in most of the tests done, it is not enough data to conclude in any direciton though.

7.4.3 Performance of implemented solutions

Analyzing pseudorehearsal first, it is clear from the tests that more pseudopatterns leads to better recognition rate. This is not surprising. What is interesting though, is that one needs to provide quite a few pseudopatterns before performance is considerably better than a regular back-propagation trained network. When viewing the results table, remember that for the LED tests only 300 instances were used, and as such only $(300/10) * 0.7 = 21$ instances were used for training. Using half the amount of pseudopatterns as regular patterns for training, the performance is very close to that of back-propagation. Only when having an order of magnitude more patterns can major differences be seen, and even then will the network quickly forget a lot of the original information. Clearly the pseudorehearsal solution can perform better than backprop, but still not well enough to be applicable for most engineering issues where one wants to retain the old information.

There is also a noticeable difference between how it performs on the waveform problem with a much larger domain of possible inputs (inputs being real-valued between approximately -10 and 10) then the LED problems with inputs being binary valued. The faster forgetting on the waveform problem can be due to several issues:

- The datasets are larger. Larger datasets could lead to more training per epoch and more overfitting, thus also faster forgetting per epoch. This is refuted by the results from the LED test with varied number of instances. Here there is no connection between the number of instances and the rate of forgetting.
- There are only three classes in the waveform problem as opposed to ten in the LED problem. This leads to the first training only discriminating between two classes. Fewer classes means that it is likely to be a larger area the hyperplanes created by the neurons' activation function can operate in and still classify the problem correctly. Then when learning a second set, the hyperplanes aren't fitted very well and is quick to change away from correct classification.
- A higher learning rate had to be used during training in order to get the network to converge properly. This means that each epoch will change the network more away from the it's starting point at the beginning of the epoch than if it was trained with a lower learning rate.

Regarding activation sharpening, it is hard to say why it did not work well with the datasets used. It can be that the intersection between the classes were too high, and thus dividing the hyperspace became harder when fewer nodes were allowed too high activation. Since further studies into *why* the activation sharpening method failed has not been done, no conclusions can be made.

8 Conclusions

At the beginning of this thesis I forumlated four questions that I wanted answers to. Throughout the rest of the thesis I've provided answers to those, and I will now sum up my conclusions.

Question one was to see if a set of rules and guidelines could be set up to test for catastrophic forgetting. This question has been answered primarily by researching previous work for neural network testing and presenting that work here. It has also been answered by providing concrete ways to test for forgetting. One of those methods has also been used in other experiments in this thesis. The conclusion is that such guidelines *can* be provided, and I have given some of them. With further study there will without doubt be more ways to perform testing and probably better ways as well, but I have provided a uniform way to test such that one can compare results between different network architectures, algorithms and datasets.

The second question was to look for real-world datasets to aid testing and to make sure the tests done were more reliable. Here I found the selection done by Zheng, but due to additional restrictions for testing for catastrophic forgetting, the selection had to be more limited and thus the number and variety of the datasets became very low. I have not found a general set of datasets that without doubt covers all attributes in such a way that a test can be said to be "general" if it performs well on all those datasets.

The third question, and the one which most of the experimental work for this thesis has been done for, was regarding the analysis of datasets. My conclusion here is that *there is a connection between forgetting and the two measurements I designed to measure overlap in the input space: Input Overlap and Grouped Input Overlap.* When there is a significant change in either of these due to the alteration of one or more dataset parameters, the networks trained with backpropagation will also show a change in the speed which information is lost. It is also clear that the interesting way to compare IO and GIO is to compare their values *relative to the maximum* and not their absolutes. However if either of these are to be used as an attribute when selecting datasets or as an indication prior to training as to how it is expected that the network will perform, one must perform more tests with datasets that have relative IO and GIO that are both higher and lower than what's been used for the experiments in this thesis.

The fourth and final question was related to the Activation Sharpening and Pseudorehearsal algorithms studied and how well they performed on datasets

other than the artificial ones that were produced for the original papers. Here I saw that performance for Activation Sharpening was very poor, in fact so poor that the algorithm could not be used for further studies. The pseudorehearsal solution proved to work better than back-propagation, but significantly better *only* if a large number of pseudo patterns were generated. This leads to practical problems with the solution since the generation and storage of these patterns can be expensive in terms of memory of computation. Since the pseudoerehearsal solution never *removed* the forgetting, but only delayed it, many potential applications will also find it unsuitable.

8.1 Future Work

The main direction to take the work done here further, is by expanding the experiments done in Matlab. Due to the problems encountered when selecting datasets, the variation in the dataset attributes was not as good as wanted. By selecting more datasets with variations in the attributes from the ones used, better conclusions can be drawn. To be able to make such a selection though, more datasets must first be analyzed so one can choose which to select to use as a benchmark set.

Another direction to continue this work, is to expand the network architectures and training algorithms. All the tests run in this thesis has been with setups very close to the standard fully-connected one-layered feed-forward back-propagation network. There has been many advances into architectures better suited for particular problems, and also some with CF in mind. Especially dual-network architectures are interesting in that regard.

I would also have liked to test the network developed by Axel Tidemann in his master's thesis [19], but unfortunately I did not have the time available to do so.

Further work can also be done with specific dataset attributes in mind, particularily entropy.

References

[1] Ole-Marius Moe-Helgesen and Håvard Stranden. Catastrophic forgetting in neural networks, 2005.

[2] 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.

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

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

[5] C.L. Blake D.J. Newman, S. Hettich and C.J. Merz. UCI repository of machine learning databases, 1998.

[6] Lutz Prechelt. Proben1: A set of neural network benchmark problems and benchmarking rules. Technical Report 21/94, 1994.

[7] Zijian Zheng. A benchmark for classifier learning. Technical Report TR474, N.S.W Australia 2006, 1993.

[8] C. E. Rasmussen, R. M. Neal, G. E. Hinton, D. van Camp, M. Revow, Z. Ghahramani, R. Kustra, and R. Tibshrani. The delve manual, 1996.

[9] A. Guérin-Dugué R. Chentouf C. Aviles-Cruz J. Madrenas M. Moreno J.L. Voz F. Blayo, Y. Cheneval. Deliverable r3–b4–p task b4 : Benchmarks. Technical Report R3-B4-P, 1995.

[10] Barna G. Kohonen, T. and R. Chrisley. Statistical pattern recognition with neural networks: Benchmarking studies. volume 1, pages 61–68, 1988.

[11] Kenneth C. Laudon and Jane P. Laudon. *Managment Information Systems*. Prentice Hall, 2004.

[12] Keri E. Pearlson. *Managing and Using Information Systems - A Strategic Approach*. Joh Wiley & Sons, Inc., 2001.

[13] Tom M. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[14] I. Kononenko and I. Bratko. Information-based evaluation criterion for classifier's performance. *Machine Learning*, 6:67–80, 1991.

[15] 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.

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

[17] Steven Salzberg. On comparing classifiers: Pitfalls to avoid and a recommended approach. *Data Mining and Knowledge Discovery*, 1(3):317–328, 1997.

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

[19] Axel Tidemann. Dancing robots. *Master's Thesis*, 2006.

A Results

This appendix contains complete results from the experiments run, as well as complete descriptions of the problems.

A.1 Led-7 tests

The results from the Led-7 tests without noise is in table 8, and performance plots are in figure 8. In table 9 are the results from the Led-7 tests with 10% noise, while the plots from these experiments are in figure 9.

Test	First epoch with $\leq 50\%$ recognition rate
Backpropagation	21
Pseudorehearsal, pool size 8	25
Pseudorehearsal, pool size 32	74
Pseudorehearsal, pool size 128	Never

Table 8: R
esults from the Led test without noise and without irrelevant attributes. Table shows regular backpropagation, and pseudorehearsal with 8, 32 and 128 pseudo patterns. Results are from tests on 20 different datasets and are averaged.

Test	First epoch with $\leq 50\%$ recognition rate
Backpropagation	31
Pseudorehearsal, pool size 8	39
Pseudorehearsal, pool size 32	104
Pseudorehearsal, pool size 128	469

Table 9: Led-7 (10% noise) results

Dataset	Measurement	Theoretical Max	Value	% of max max	Standard Deviation
Led-7 No noise	Input Overlap	149.5	108.7	72.7%	1.272
Led-7 10% noise	Input Overlap	149.5	100.6	67.3%	1.964
Led-7 No noise	Grouped Input Overlap	28.6	20.8	72.6%	3.28
Led-7 10% noise	Grouped Input Overlap	26.7	17.4	65.1%	2.89

Table 10: Led-7 input overlap analysis results

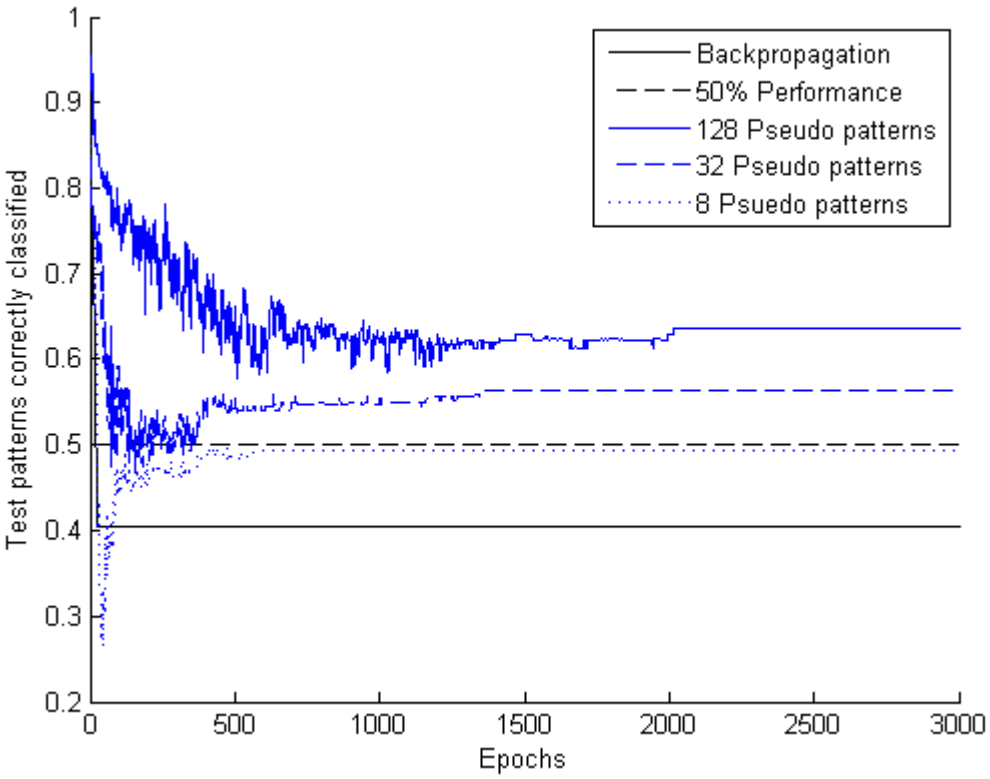

Figure 8: Led7 no-noise performance plot

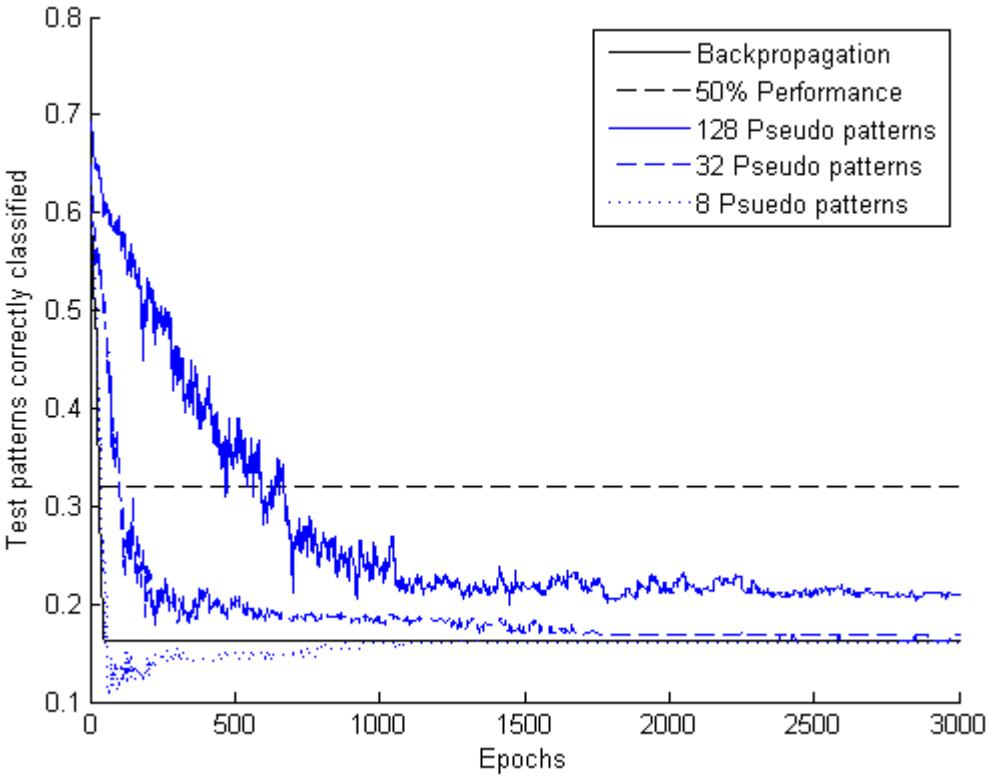

Figure 9: Led7 10% noise performance plot

A.2 Led-24 tests

The results from the Led-24 tests without noise is in table 11, with performance plots in figure 10. The results from the Led-24 tests with 10% noise are in table 12 and plots from those tests in figure 11.

Test	First epoch with $\leq 50\%$ recognition rate
Backpropagation	50
Pseudorehearsal, pool size 8	44
Pseudorehearsal, pool size 32	144
Pseudorehearsal, pool size 128	401

Table 11: Led-24 (no noise) results

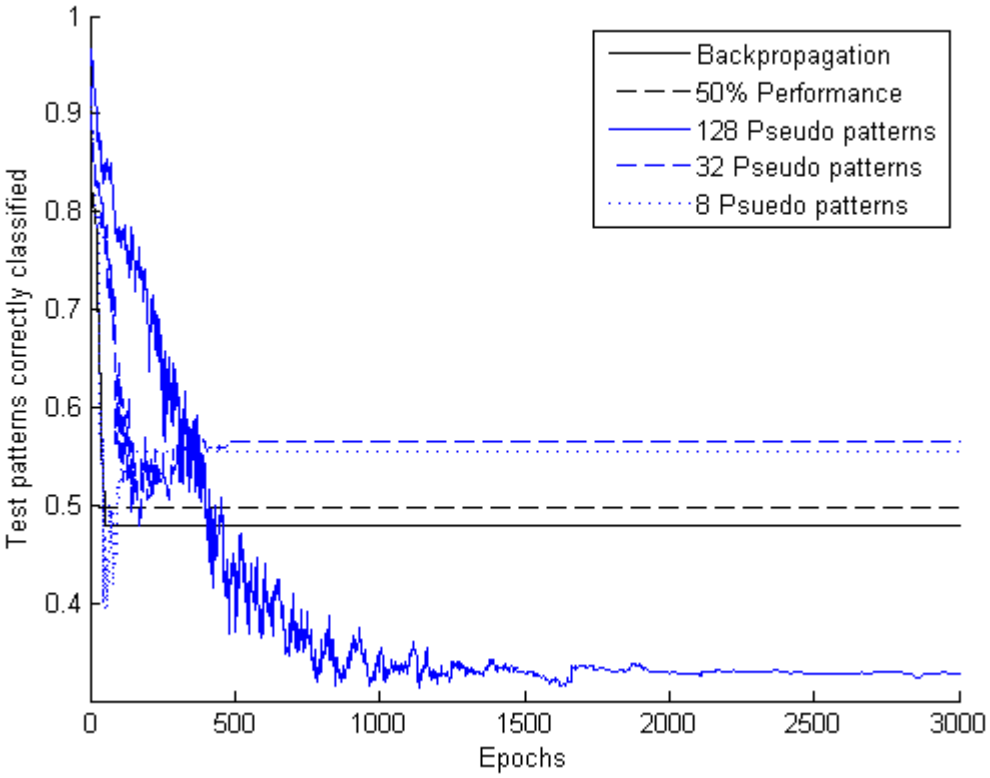

Figure 10: Led24 no-noise performance plot

Test	First epoch with $\leq 50\%$ recognition rate
Backpropagation	37
Pseudorehearsal, pool size 8	42
Pseudorehearsal, pool size 32	99
Pseudorehearsal, pool size 128	362

Table 12: Led-24 (10% noise) catastrophic forgetting results

Dataset	Measurement	Theoretical Max	Value	% of max	Std Dev
Led-24 No noise	Input Overlap	149.5	87.32	58.4%	0.667
Led-24 10% noise	Input Overlap	149.5	83.69	56.0%	0.934
Led-24 No noise	Grouped Input Overlap	26.6	15.5	58.2%	1.93
Led-24 10% noise	Grouped Input Overlap	26.1	14.2	54.2%	1.59

Table 13: Led-24 input overlap analysis results

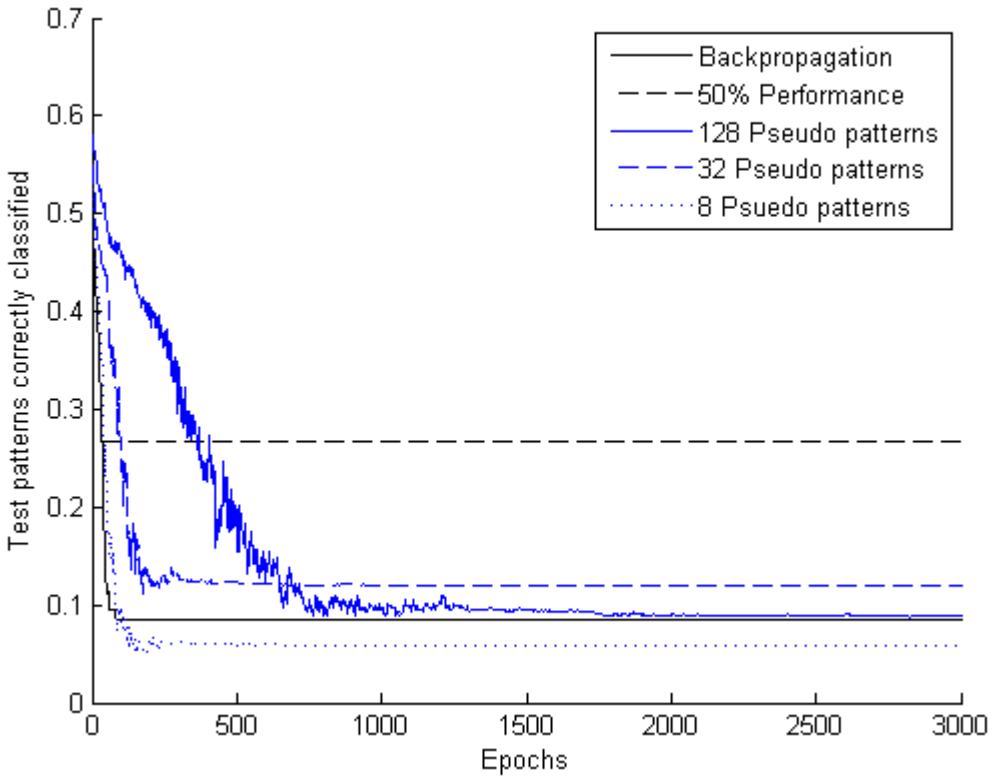

Figure 11: Led24 10% noise performance plot

A.3 Waveform tests

The results from the Waveform tests with 5000 instances is in table 14 and from the tests with 500 instances in table 15. Figures 12 and 13 show the performance plots for backpropagation and pseudorehearsal for the waveform data with 500 and 5000 instances respectfully.

Test	First epoch with $\leq 50\%$ recognition rate
Backpropagation	22
Pseudorehearsal, pool size 8	4
Pseudorehearsal, pool size 32	5
Pseudorehearsal, pool size 128	5
Pseudorehearsal, pool size 500	7

Table 14: Waveform with 5000 instances forgetting results

Test	First epoch with $\leq 50\%$ recognition rate
Backpropagation	16
Pseudorehearsal, pool size 8	11
Pseudorehearsal, pool size 32	17
Pseudorehearsal, pool size 128	34

Table 15: Waveform with 500 instances

Dataset	Measurement	Theoretical Max	Value	% of max	Std Dev
Waveform 500	Input Overlap	159.8	249.5	64.0%	1.02
Waveform 5000	Input Overlap	1598.3	2499.5	63.9%	0.00
Waveform 500	Grouped Input Overlap	64.75	112.83	57.4%	1.83
Waveform 5000	Grouped Input Overlap	652.8	1111	58.8%	15.6

Table 16: Waveform input overlap analysis results

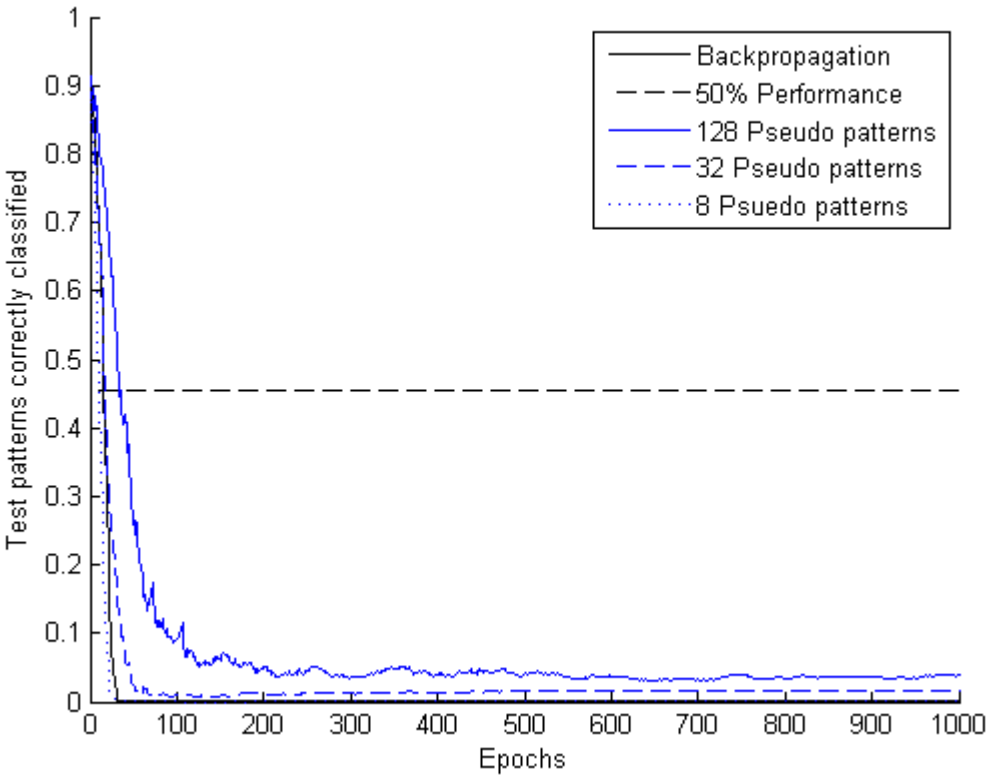

Figure 12: Waveform with 500 instances performance plot

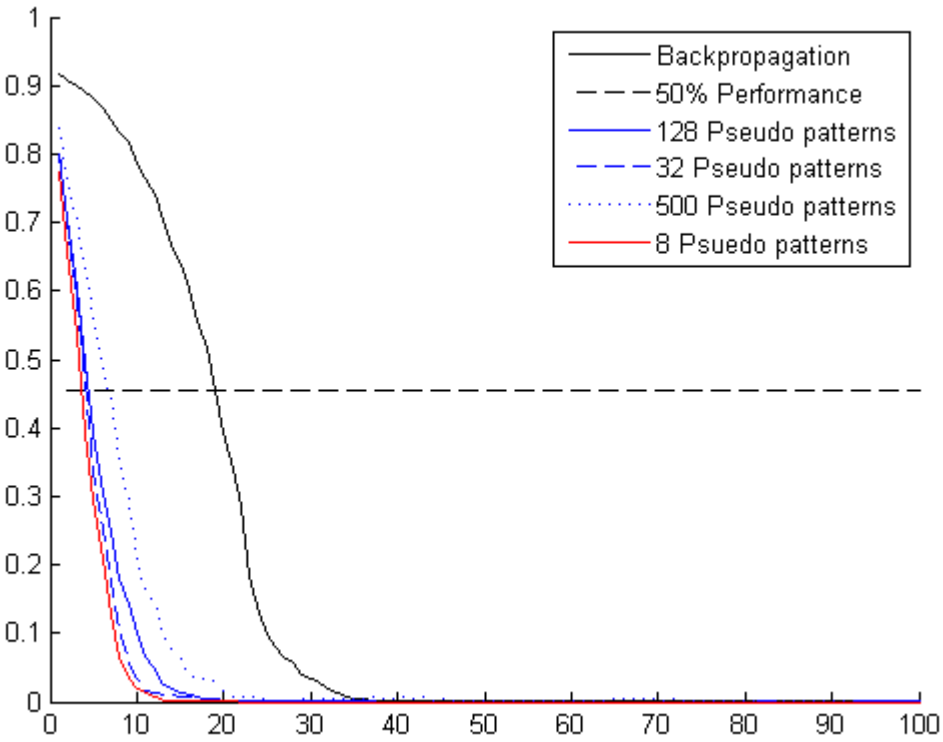

Figure 13: Waveform with 500 instances performance plot

B Implemented Code

In this appendix important parts of the source code for this thesis will be reproduced. More information about the implementation can be found in chapter 6.

B.1 Dataset Analysis

This section contains source code used to analyse datasets.

Listing 3: entropy.m: Source code for calculating entropy of a dataset

```
% Calculate the entropy of a dataset.
% function E = entropy(outputs)
%
% Inputs: A vector containing all the output class values for the dataset.
% Each element in the vector should be the output of one instance in the
% dataset.
%
% Output: The calculated entropy
function E = entropy(output)
    E = 0;
    for i = 1:size(output,2)
        num = size(output(output(:,i)==1),1);
        p = num/size(output,1);
        E = E-(p*log2(p));
    end
```

Listing 4: density.m: Source code for calculating the density of a dataset

```
%Calculates the dataset density as defined by Zheng
% function dens = density(dataset)
%
% Inputs: A dataset matrix with each row being the input vector of one
% instance
%
% Output: The dataset density
function [dens] = density(dataset)
    dset_size = size(dataset);
    n = dset_size(2);

    prod = 1;
    for i = 1 : n
        prod = prod*size(unique(dataset(:,i)),1);
    end

    dens = dset_size(1) / prod;
```

Listing 5: default_accuracy.m: Source code for calculating the default accuracy of a dataset

```
% Calculates the default accuracy of a dataset as defined my Zheng
% This is the relative frequency of the most common output class
%
% function  acc = default_accuracy(output)
%
% Input: A matrix containing one row for each instance and one binary
% column for each output class
%
% Output: The default accuracy
function  acc = default_accuracy(output)
    maxval = 0;
    for i = size(output,2)
        val = size(output(output(:,i)==1),1);
        if(val>maxval)
```

```
            maxval = val;
        end
    end
    acc=maxval/size(output,1);
```

Listing 6: relative_accuracy.m: Source code for calculating the relative accuracy of a dataset

```
% Calculates the relative accuracy as the how good the default accuracy is
% in relation to the best recorded accuracy on the domain.
%
% function rel = relative_accuracy(predictive, default)
%
% Inputs:
% predicive: The best recorded accuracy
% default: The default accuracy
%
% Output: The relative accuracy
function rel = relative_accuracy(predictive, default)
    rel = 100*((predictive-default)/(1-default));
```

B.1.1 Input Overlap Measurement

For calculating the input overlap measurement, the functions in listings 7, 8 and 9 were used. For calculating the grouped input overlap, the source code in listings 10 and11 were used.

Listing 7: input_overlap.m: Source code for calculating the input overlap of a dataset

```
% Calculate the input overlap for a given matrix of input vectors.
%
% function overlap = input_overlap(input)
%
% Inputs:
% input: A matrix containing the input vectors to calculate the overlap for
%        Each row of the vector should contain one input vector.
%
% Outputs:
% overlap: The total input overlap for the dataset

function overlap = input_overlap(input)
    n = size(input,1);
    overlap = total_orthogonality(input)/n;
```

Listing 8: total_orthogonality.m: Source code for calculating the total orthogonality of a set of vectors

```
% Calculate the total orthogonality of a set of vectors as the summed
% cosine of the angle between any two vectors.
%
% Inputs:
% vects: A MxN matrix where each of the M rows contains an vector in
% N-dimensional space.
```

```
%
% Output: The summed orthogonality
function ans = total_orthogonality(vects)
    n = size(vects,1);
    ans = 0;
    %TODO: Two nested for-loops is very far from being optimal in Matlab.
    %Some internal functions might help on getting this done faster.
    for i = 1 : n-1
        irow = vects(i,:);
        for j = i+1 : n
            jrow = vects(j, :);
            ans = ans+orthogonality(irow, jrow);
        end
    end
```

Listing 9: orthogonality.m: Source code for calculating the orthogonality between two vectors

```
% Calculate the level of orthogonality between vectors a and b. This is
% defined as the cosine of the angle between them/
%
% function ans = orthogonality(a, b)
%
% Inputs:
% a, b: Two vectors. length(a) must be equal to length(b)
%
% Output: cos(angle(norm(a),norm(b)))
function ans = orthogonality(a, b)
    norm_a = norm(a);
    if(norm_a==0)
        norm_a = 1;
    end

    norm_b = norm(b);
    if(norm_b==0)
        norm_b = 1;
    end

    ans = dot(a/norm_a, b/norm_b);
```

Listing 10: grouped_input_overlap.m: Source code for calculating the grouped input overlap of a dataset

```
% Calculate the grouped input overlap for two sets of input vectors.
%
% function overlap = grouped_input_overlap(input)
%
% Inputs:
% first_set: A matrix containing the input vectors in one training set.
%            Each row of the vector should contain one input vector.
% second_set: A matrix containing the input vectors for the second training
% set.  This set should have the same format as first_set
%
% Outputs:
% overlap: The total input overlap for the dataset

function overlap = grouped_input_overlap(first_set, second_set)
    n = size(first_set,1)+size(second_set,1);
    overlap = total_grouped_orthogonality(first_set, second_set)/n;
```

Listing 11: total_orthogonality.m: Source code for calculating the total grouped orthogonality for two sets of vectors

```
% Calculate the total orthogonality of a set of vectors as the summed
% cosine of the angle between any two vectors.
%
% Inputs:
% vects: A MxN matrix where each of the M rows contains an vector in
% N-dimensional space.
%
% Output: The summed orthogonality
function ans = total_orthogonality(vects)
    n = size(vects,1);
    ans = 0;
    %TODO: Two nested for-loops is very far from being optimal in Matlab.
    %Some internal functions might help on getting this done faster.
    for i = 1 : n-1
        irow = vects(i,:);
        for j = i+1 : n
            jrow = vects(j, :);
            ans = ans+orthogonality(irow, jrow);
        end
    end
```

B.2 Neural Network Testbench

The source code presented in this section is used to extend the neural network code provided by other researchers (former and present) at IDI (Axel Tidemann and Diego Federici) to work with the networks wanted for this thesis.

Listing 12: test_ann.m: Source code for how network performance was tested during the experiments

```
% Function for testing the performance of a neural network
%
% function correct = test_ann(net, in, out, criterion)
%
% Inputs:
% net: The network to test
% in: An array of input patterns to use as test instances.
% out: An array of output patterns to use as test instances. in and out
% should be arranged in such as way that in(:,I) is the input pattern for
% out(:,I) for all I.
% criterion: How much the actual output can diverge from the desired output
% while still being counted as a correct classification.
%
% Outputs:
% The number of patterns where all outputs where within the criterion of
% the desired outputs.
function correct = test_ann(net, in, out, criterion)
    correct = 0;
    num_test_patts = size(in,2);
    for j = 1 : num_test_patts
        tnet = ann_activate(net, in(:,j));
        err = out(:,j) - tnet.L{3};
        if(abs(err)<criterion)
            correct = correct+1;
        end
    end
```

Listing 13: ann_activate.m: Source code for activating a regular ANN without recurrent connections

```
function net = ann_activate(net, X)
    net.L{1}(net.size(1)+1:end) = 0;
    net.L{2} = 0;
    net.context(1:end) = 0;
    net = rnn_activate(net,X);
    net.L{1}(net.size(1)+1:end) = 0;
```

Listing 14: train_epoch.m: Source code for training a neural network for a complete epoch

```
function net = train_epoch(net, in, out)
    %Get a random order to train the exemplars in
    order = randperm(size(in,2));

    for j = 1 : size(order)
        n = order(j);
```

```
        net = ann_activate(net, in(:,n));
        err = out(:,n) - net.L{3};
        net = rnn_train(net, err, 1);
    end
```

B.3 Datasets

This section contains the source code created to manipulate or generate the datasets used for the experiments.

Listing 15: make20.sh: Source code for generating the data patterns for the LED tests

```
for i in `seq 1 20`;
do
  ./led24 300 $RANDOM led24noise$i.data 10
  ./led24 300 $RANDOM led24nonoise$i.data 0
  ./led 300 $RANDOM led7noise$i.data 10
  ./led 300 $RANDOM led7nonoise$i.data 0
done
```

Listing 16: make500s.sh: Source code for making the datasets for the Waveform tests with 500 instances

```
# Script for generating 20 datasets with 500 instances from an original dataset
# with 5000.  The script creates the datafiles waveform500_1.data to
# waveform500_20.data.  Each file contains 500 instances starting at a random
# point in the original file and moving forward.
for i in `seq 1 20`;
do
   #Generate a random number between 500 and 5000
   RAN_NUM=`echo "500+($RANDOM/32767)*4500" | bc -l | awk -F"." '{print $1}'`

   #Select the instances from RAN_NUM-500 to RAN_NUM
   cat waveform.data | tail -n $RAN_NUM | head -n 500 > waveform500_$i.data
done
```

B.4 Experiments

This section contains source code produced explicitly for the experiements performed for this thesis and the datasets used in these experiments.

Listing 17: average_input_overlap.m: Source code for calculating the average IO of a group of datasets

```
% Calculate the average input overlap from a set of data files
%
% function avg_io = input_overlap_averaged(file_prefix, load_fun, num)
%
% Inputs:
% file_prefix: The prefix of the data files. The file names must be
%   <file_prefix><id>.data where id is in the range [1,num]
% load_fun: The dataset loading function to use for loading each data set.
%   This function must accept one parameter (the file name) and return two
%   values: The input and output data as matrixes with one instance per
%   row.
% num: The number of data files to use for the calculation
%
```

```
% Output: A data structure containing the mean, standard, theoretical
% maximum, and percentage of theoretical maximum of the calculated input
% overlaps
function avg_io = input_overlap_averaged(file_prefix, load_fun, num)
    ios = zeros(1, num);
    max_ios = zeros(1, num);
    nominal_ios = zeros(1,num);
    for i = 1:num
        [input, output] = load_fun(strcat(file_prefix, int2str(i), ...
            '.data'));
        ios(i) = input_overlap(input);
        max_ios(i) = max_io(input);
        nominal_ios(i) = ios(i)/max_ios(i);
        %Calculate average grouped input overlap for each
        for j=1:size(output,2)

    end
    avg_io.mean = mean(ios);
    avg_io.std = std(ios);
    avg_io.theoretical_max = mean(max_ios);
    avg_io.percent = 100*(avg_io.mean/avg_io.theoretical_max);
end
```

Listing 18: grouped_input_overlap_averaged.m: Source code for calculating the average GIO of a group of datasets

```
% Calculate the average input overlap from a set of data files
%
% function avg_io = input_overlap_averaged(file_prefix, load_fun, num)
%
% Inputs:
% file_prefix: The prefix of the data files. The file names must be
%   <file_prefix><id>.data where id is in the range [1,num]
% load_fun: The dataset loading function to use for loading each data set.
%   This function must accept one parameter (the file name) and return two
%   values: The input and output data as matrixes with one instance per
%   row.
% num: The number of data files to use for the calculation
%
% Output: A data structure containing the mean, standard, theoretical
% maximum, and percentage of theoretical maximum of the calculated input
% overlaps
function avg_gio = grouped_input_overlap_averaged(file_prefix, load_fun, ...
    num)
    gios = zeros(1, num);
    max_gios = zeros(1, num);
    nominal_gios = zeros(1,num);
    for i = 1:num
        [input, output] = load_fun(strcat(file_prefix, int2str(i), ...
            '.data'));

        %Select a random class for second set
        second_class = floor(rand*size(output,2))+1;

        %Create First and Second set
        indexes = output(:,second_class)==0;
        first_set = submatrix(input, indexes);
        indexes = output(:,second_class)==1;
        second_set = submatrix(input, indexes);

        %Calculate overlap and max
```

```
        gios(i) = grouped_input_overlap(first_set, second_set);
        max_gios(i) = max_gio(first_set, second_set);
        nominal_gios(i) = gios(i)/max_gios(i);
    end
    avg_gio.mean = mean(gios);
    avg_gio.std = std(gios);
    avg_gio.theoretical_max = mean(max_gios);
    avg_gio.percent = 100*(avg_gio.mean/avg_gio.theoretical_max);
```

Listing 19: average_grouped_input_overlap.m: Source code for calculating the average GIO of a dataset

```
% Calculates the average grouped input overlap with each of the different
% output classes being in a separate subset in each sample.
%
% function avg_gio = average_grouped_input_overlap(input, output)
%
% Inputs:
% input: A matrix containing an input pattern at each row
% output: A matrix containing an output pattern at each row. The output
% must be with 1-of-C encoding where each column is a different class with
% a 1 if the instance is in that class and a 0 otherwise
%
% Output: An average GIO structure containing the mean, max, mean relative
% to max and standard deviation
function avg_gio = average_grouped_input_overlap(input, output)
    classes = size(output,2);
    overlaps = zeros(1, classes);
    max_gios = zeros(1, classes);
    for i = 1:classes
        indexes = output(:,i)==0;
        first_set = submatrix(input, indexes);
        indexes = output(:,i)==1;
        second_set = submatrix(input, indexes);
        overlaps(i) = grouped_input_overlap(first_set, second_set);
        max_gios(i) = max_gio(first_set, second_set);
    end
    avg_gio.mean = mean(overlaps);
    avg_gio.std = std(overlaps);
    avg_gio.theoretical_max = mean(max_gios);
    avg_gio.percent = 100*(avg_gio.mean/avg_gio.theoretical_max);
```

Listing 20: run_cf_experiment.m: Source code for running a catatsrophic forgetting experiement using regular backpropagation

```
% Function used to run an experiment to test for catastrophic forgetting
% using a regular fully-connected feed-forward network trained with
% backpropagation.
%
%function [first,second,third] = run_cf_experiment(first, second,
%         train_percentage, test_percentage, epochs, learning_rate,
%         hidden_size, criterion)
%
% Inputs:
% first: A data structure set with dataset to use for the first batch of
%   training and testing. This should be set in first.all.in and
%   first.all.out
% second A data structure containing the dataset to use for the second
%   batch of training. Similiar structure as for first
% train_percentage: How much of the dataset to use for training, in the
```

```
%   range [0-1]. This will be taken from the beginning of the dataset
%   vectors
% test_percentage: How much of the dataset to use for training, in the
%   range [0-1]. This will be taken from the end of the datase vector. So
%   if train_percentage+test_percentage>1, there will be overlap in the
%   training and testing sets
% epochs: Integer determining the number of epochs the experiment will run
%   for
% learning_rate: The learning rate to use for the backprop training
% hidden_size: The number of nodes in the hidden layer
% criterion: A value k such that if t<=o+k and t>=o-k, the output o
%   classifies as the target t. This value is used when testing the network
%   to see if it classifies test patterns correctly.
%
% Outputs:
% All the outputs have had an additional field testlog added.  This
% contains a number denoting how big proportion of the test instances that
% were correctly classified in each run. These are padded so they are
% contains elements equal to the number of epochs even if the network
% converges earlier. first and second contains in addition fields X.train
% and X.test containing the training and testing instances used.
% first: Data for the first training batch performed
% second: Data for the second training batch performed
% third: Data for testing the first test batch during the second training.
function [first,second,third] = run_cf_experiment(first, second, ...
    train_percentage, test_percentage, epochs, learning_rate, ...
    hidden_size, criterion)

    %Calculate number of instances to use for training and testing
    train_num = round(size(first.all.in,1)*train_percentage);
    test_num = round(size(first.all.in,1)*test_percentage);

    %Create input and output sets properly structured for the neural
    %network
    first.train.in = permute(first.all.in(1:train_num,:), [2,1]);
    first.train.out = permute(first.all.out(1:train_num,:), [2,1]);
    first.test.in = permute(first.all.in(end-test_num:end,:), [2 1]);
    first.test.out = permute(first.all.out(end-test_num:end,:), [2 1]);
    first.testlog = [];

    %Repeat for second training period
    train_num = round(size(second.all.in,1)*train_percentage);
    test_num = round(size(second.all.in,1)*test_percentage);
    second.train.in = permute(second.all.in(1:train_num,:), [2 1]);
    second.train.out = permute(second.all.out(1:train_num,:), [2 1]);
    second.test.in = permute(second.all.in(end-test_num:end,:), [2 1]);
    second.test.out = permute(second.all.out(end-test_num:end,:), [2 1]);
    second.testlog = [];

    %Keep a log of how well the performance on the first dataset develops
    %while learning the second
    third.testlog = [];

    %Create a new network with one hidden layer
    network_size = [size(first.train.in,1), hidden_size, ...
                    size(first.train.out,1)];
    net = rnn(network_size, learning_rate);

    %Start training first batch, storing the best net for later use
    bestnet = net;
    for i = 1 : epochs
        net = train_epoch(net, first.train.in, first.train.out);
```

```
    %Test the network and calculate percentage
    correct = test_ann(net, first.test.in, first.test.out, criterion);
    percentage = correct/size(first.test.in,2);

    %Store the best net so far
    if(percentage>max(first.testlog));
        bestnet = net;
    end
    first.testlog(i) = percentage;

    %If 100% accuracy, stop training and set the testlog for the
    %remaining epochs to be 100% as well
    if(correct==size(first.test.out,2))
        first.testlog(i:epochs)=1;
        break
    end
end
 disp(sprintf('%s Best network classified %d%% correctly', ...
'Training of first batch complete.', max(first.testlog)*100));

% Now the network is trained as good as possible (or as good as we want
% it). The next step is to train it to learn new information, this is
% stored in second.all.in and second.all.out
net = bestnet;

for i = 1 : epochs
    net = train_epoch(net, second.train.in, second.train.out);

    correct = test_ann(net,second.test.in,second.test.out,criterion);
    percentage = correct/size(second.test.in,2);

    if(percentage > max(second.testlog))
        bestnet = net;
    end
    second.testlog(i) = percentage;

    %test again with the first testset to see how it performs
    third.testlog(i) = test_ann(net, first.test.in, ...
      first.test.out, criterion)/size(first.test.in,2);

    if(correct==size(second.test.out,2))
        second.testlog(i+1:epochs)=1;
        third.testlog(i+1:epochs)=third.testlog(i);
        break
    end
end

disp(sprintf('%s Best network classified %d%% correctly', ...
 'Training of second batch complete.', max(second.testlog)*100));
disp(sprintf('%s of initial dataset is %d%% correctly', ...
 'Experiment complete. Final performance', third.testlog(end)*100));
```

B.4.1 Pseudo rehearsal

Listing 21: run_pseudo_patt_experiment.m: Source code for running a catatsrophic forgetting experiement using pseudopattern training

```
% Function used to run an experiment to test for catastrophic forgetting
% using a regular fully-connected feed-forward network trained with pseudo
% rehearsal and backpropagation
%
% function [first,second,third] = run_pseudo_patt_experiment(first, second,...
%    train_percentage, test_percentage, epochs, learning_rate,...
%    hidden_size, criterion, pseudo_pool)
%
% Inputs:
% first: A data structure set with dataset to use for the first batch of
%   training and testing. This should be set in first.all.in and
%   first.all.out
% second A data structure containing the dataset to use for the second
%   batch of training. Similiar structure as for first
% train_percentage: How much of the dataset to use for training, in the
%   range [0-1]. This will be taken from the beginning of the dataset
%   vectors
% test_percentage: How much of the dataset to use for training, in the
%   range [0-1]. This will be taken from the end of the datase vector. So
%   if train_percentage+test_percentage>1, there will be overlap in the
%   training and testing sets
% epochs: Integer determining the number of epochs the experiment will run
%   for
% learning_rate: The learning rate to use for the backprop training
% hidden_size: The number of nodes in the hidden layer
% criterion: A value k such that if t<=o+k and t>=o-k, the output o
% classifies as the target t. This value is used when testing the network
%   to see if it classifies test patterns correctly.
% pseudo_pool: How many pseudo patterns to learn along side the second
%   training set
%
% Outputs:
% All the outputs have had an additional field testlog added.  This
% contains a number denoting how big proportion of the test instances that
% were correctly classified in each run. These are padded so they are
% contains elements equal to the number of epochs even if the network
% converges earlier. first and second contains in addition fields X.train
% and X.test containing the training and testing instances used.
% first: Data for the first training batch performed
% second: Data for the second training batch performed
% third: Data for testing the first test batch during the second training.

function [first,second,third] = run_pseudo_patt_experiment(first, second,...
    train_percentage, test_percentage, epochs, learning_rate,...
    hidden_size, criterion, pseudo_pool)

    %Calculate number of instances to use for training and testing
    train_num = round(size(first.all.in,1)*train_percentage);
    test_num = round(size(first.all.in,1)*test_percentage);

    %Create input and output sets properly structured for the neural
    %network
    first.train.in = permute(first.all.in(1:train_num,:), [2,1]);
    first.train.out = permute(first.all.out(1:train_num,:), [2,1]);
    first.test.in = permute(first.all.in(end-test_num:end,:), [2 1]);
    first.test.out = permute(first.all.out(end-test_num:end,:), [2 1]);
    first.testlog = [];

    %Repeat for second training period
    train_num = round(size(second.all.in,1)*train_percentage);
    test_num = round(size(second.all.in,1)*test_percentage);
    second.train.in = permute(second.all.in(1:train_num,:), [2 1]);
```

```
second.train.out = permute(second.all.out(1:train_num,:), [2 1]);
second.test.in = permute(second.all.in(end-test_num:end,:), [2 1]);
second.test.out = permute(second.all.out(end-test_num:end,:), [2 1]);
second.testlog = [];

%Keep a log of how well the performance on the first dataset develops
%while learning the second
third.testlog = [];

%Create a new network with one hidden layer
network_size = [size(first.train.in,1), hidden_size, ...
                size(first.train.out,1)];
net = rnn(network_size, learning_rate);

%Start training first batch, storing the best net for later use
bestnet = net;
for i = 1 : epochs
    net = train_epoch(net, first.train.in, first.train.out);

    %Test the network and calculate percentage
    correct = test_ann(net, first.test.in, first.test.out, criterion);
    percentage = correct/size(first.test.in,2);

    %Store the best net so far
    if(percentage>max(first.testlog));
        bestnet = net;
    end
    first.testlog(i) = percentage;

    %If 100% accuracy, stop training and set the testlog for the
    %remaining epochs to be 100% as well
    if(correct==size(first.test.out,2))
        first.testlog(i:epochs)=1;
        break
    end
end
disp(sprintf('%s Best network classified %d%% correctly', ...
'Training of first batch complete.', max(first.testlog)*100));
net = bestnet;

% Now the network is trained as good as possible (or as good as we want
% it). The next step is to train it to learn new information, this is
% stored in second.all.in and second.all.out.  Since we are doing
% pseudo rehearsal we'll also generate a set of pseudo patterns to use
% for training alongside the new patterns

% The pseudo pattern solution here differs somewhat from the one
% presented by Robins. While he was learning one new pattern at a time
% while testing his networks, I am learning all the new patterns at
% once (as it is usually done during ANN training).  To adapt the
% pseudo rehearsal solution to this training, I am adding the entire
% pool of pseudo patterns to the list of patterns to be learned,
% instead of randomly selecting three.

% Generate a pool of psuedopatterns
for i = 1 : pseudo_pool
    item.in = [];
    for j = 1 : size(first.train.in,1)
        item.in = [item.in rand*20-10];
    end
    item.in = permute(item.in, [2 1]);
    net = ann_activate(net, item.in);
```

```
        item.out = net.L{3};
        second.train.in = [second.train.in item.in];
        second.train.out = [second.train.out item.out];

        second.test.in = [second.test.in item.in];
        second.test.out = [second.test.out item.out];
    end


    % Train the second batch of patterns together with the pseudo pool (now
    % added to second.train)
    for i = 1 : epochs
        net = train_epoch(net, second.train.in, second.train.out);

        correct = test_ann(net, second.test.in, second.test.out, criterion);
        percentage = correct/size(second.test.in,2);

        if(percentage > max(second.testlog))
            bestnet = net;
        end
        second.testlog(i) = percentage;

        %test again with the first testset to see how it performs
        third.testlog(i) = test_ann(net, first.test.in, ...
          first.test.out, criterion)/size(first.test.in,2);

        if(correct==size(second.test.out,2))
            second.testlog(i+1:epochs)=1;
            third.testlog(i+1:epochs)=third.testlog(i);
            break
        end
    end

    disp(sprintf('%s Best network classified %d%% correctly', ...
     'Training of second batch complete.', max(second.testlog)*100));
    disp(sprintf('%s of initial dataset is %d%% correctly', ...
     'Experiment complete. Final performance', third.testlog(end)*100));
```

B.4.2 Activation Sharpening

Listing 22: train_actsharp_epoch.m: Source code for training an epoch using the activation sharpening algorithm. Replaces the regular "train_epoch.m". training

```
function net = train_actsharp_epoch(net, in, out, k, sharp_factor)
    %Get a random order to train the exemplars in
    order = randperm(size(in,2));

    for j = 1 : size(order)
        n = order(j);
        net = ann_activate(net, in(:,n));

        net = activation_sharpening(net, k, sharp_factor);

        net = ann_activate(net, in(:,n));
        err = out(:,n) - net.L{3};
        net = rnn_train(net, err, 1);
    end
```

Listing 23: activation_sharpening.m: Source code for performing the activation sharpening algorithm. training

```
% Peform the activation sharpening algorithm on an array of nodes.
% These nodes are assumed to be the output values of the hidden layer in a
% 3-layered neural network.
%
% function hidden_layer = activation_sharpening(hidden_layer, k,
% sharpening_factor)
%
% Inputs:
% net: A network already activated (using ann_activate/rnn_activate)
% k: The number of nodes to increase the output value of. n-k nodes will be
%    reduced
% sharpening_factor: How much to 'sharpen' the outputs. A high factor will
%    lead to greater differences between the hidden layer provided as input
%    and the hidden layer produced as output.
%
% Outputs:
% A network with sharpened nodes.
function net = activation_sharpening(net, k, sharpening_factor)

    % Sharpen a value.
    % Inputs:
    % val: The value to sharpen. This is assumed to be between 0 and 1
    % apply_sharpening: Boolean value determining saying if the value is
    % among the k highest (and should be increased) or among the k lowest
    % and should be lowered
    %
    % Output:
    % ans: The sharpened value
    function ans = sharpen(val, apply_sharpening)
        if apply_sharpening
            ans = val + sharpening_factor*(1-val);
        else
            ans = val - sharpening_factor*val;
        end
    end

    hidden_layer = net.L{2};
    sorted = sort(hidden_layer, 'descend');
    border_val = sorted(k+1);
    sharpen_nodes = hidden_layer>border_val;
    hidden_layer = arrayfun(@sharpen, hidden_layer, sharpen_nodes);
    hidden_errors = net.L{2}-hidden_layer;

    %Updating w1. Adding 1 for the input bias. Backpropagating error from
    %the hidden layer.
    for i=1:length(hidden_errors)
        net.w{1}(i,:) = net.w{1}(i,:) + net.lr .* hidden_errors(i) .* [ net.L{1}' 1 ];
    end
end
```

Listing 24: run_actsharp_experiment.m: Source code for running a catatsrophic forgetting experiement using activation sharpening training

```
% Function used to run an experiment to test for CF
% Datastructures with training and test data set are passed
% in, and the same datastructures with an experimental log is returned back
function [first,second,third] = run_actsharp_experiment(first, second, ...
    train_percentage, test_percentage, epochs, learning_rate, ...
```

```
hidden_size, criterion, actsharp_k, actsharp_factor)

%Calculate number of instances to use for training and testing
train_num = round(size(first.all.in,1)*train_percentage);
test_num = round(size(first.all.in,1)*test_percentage);

%Create input and output sets properly structured for the neural
%network
first.train.in = permute(first.all.in(1:train_num,:), [2,1]);
first.train.out = permute(first.all.out(1:train_num,:), [2,1]);
first.test.in = permute(first.all.in(end-test_num:end,:), [2 1]);
first.test.out = permute(first.all.out(end-test_num:end,:), [2 1]);
first.testlog = [];

%Repeat for second training period
train_num = round(size(second.all.in,1)*train_percentage);
test_num = round(size(second.all.in,1)*test_percentage);
second.train.in = permute(second.all.in(1:train_num,:), [2 1]);
second.train.out = permute(second.all.out(1:train_num,:), [2 1]);
second.test.in = permute(second.all.in(end-test_num:end,:), [2 1]);
second.test.out = permute(second.all.out(end-test_num:end,:), [2 1]);
second.testlog = [];

%Keep a log of how well the performance on the first dataset develops
%while learning the second
third.testlog = [];

%Create a new network with one hidden layer
network_size = [size(first.train.in,1), hidden_size, ...
    size(first.train.out,1)];
net = rnn(network_size, learning_rate);

% Train first batch, and store the best net for the next training
% batch.
bestnet = net;
for i = 1 : epochs
    net = train_actsharp_epoch(net, first.train.in, ...
        first.train.out, actsharp_k, actsharp_factor);

    %Test the network on a separate test set
    correct = test_ann(net, first.test.in, first.test.out, criterion);
    percentage = correct/size(first.test.in,2);

    %Store the best net so far
    if(percentage>max(first.testlog));
        bestnet = net;
    end
    first.testlog(i) = percentage;

    %If 100% accuracy, stop training and set the testlog for the
    %remaining epochs to be 100% as well
    if(correct==size(first.test.out,2))
        first.testlog(i:epochs)=1;
        break
    end
end
 disp(sprintf('%s Best network classified %d%% correctly', ...
'Training of first batch complete.', max(first.testlog)*100));

% Now the network is trained as good as possible (or as good as we want
% it). The next step is to train it to learn new information, this is
% stored in second.all.in and second.all.out
```

```
net = bestnet;

for i = 1 : epochs
    net = train_actsharp_epoch(net, second.train.in, ...
        second.train.out, actsharp_k, actsharp_factor);

    correct = test_ann(net, second.test.in, second.test.out,criterion);
    percentage = correct/size(second.test.in,2);

    if(percentage > max(second.testlog))
        bestnet = net;
    end
    second.testlog(i) = percentage;

    %test again with the first testset to see how it performs
    third.testlog(i) = test_ann(net, first.test.in, ...
        first.test.out, criterion)/size(first.test.in,2);

    if(correct==size(second.test.out,2))
        second.testlog(i+1:epochs)=1;
        third.testlog(i+1:epochs)=third.testlog(i);
        break
    end
end

disp(sprintf('%s Best network classified %d%% correctly', ...
 'Training of second batch complete.', max(second.testlog)*100));
disp(sprintf('%s of initial dataset is %d%% correctly', ...
 'Experiment complete. Final performance', third.testlog(end)*100));
```

B.5 Analysis

This section contains the source code used to analyze the results from the experiments.

Listing 25: analyze_cf_results.m: Source code for analyzing the results produced from backpropagation and pseudo rehearsal training in experiment 1

```
function [normal_best, pseudo_res] = analyze_cf_results(testname)

    % A helper function for converting the data structure names
    % into names suitable for display in a plot
    function ans = replace_name(inp)
        if strcmp(inp, 'log_8')
            ans = '8 Psuedo patterns';
        elseif strcmp(inp,'log_32')
            ans = '32 Pseudo patterns';
        elseif strcmp(inp,'log_128')
            ans = '128 Pseudo patterns';
        elseif strcmp(inp,'log_500')
            ans = '500 Pseudo patterns';
        else
            ans = inp;
        end
    end

    normal = load(strcat('cf_', testname, '.mat'));
    pseudo = load(strcat('pseudo_', testname, '.mat'));

    hold off;
    pseudo = orderfields(pseudo);
    names = fieldnames(pseudo);
    minlen = size(normal.runs.avg.third.mean, 2);
    halfval = max(normal.runs.avg.first.mean)/2;

    normal_best = find(normal.runs.avg.third.mean<halfval, 1, 'first');
    pseudo_res.best = -1;

    %Keep an array of names for the plot.
    plotnames = strvcat('Backpropagation', '50% Performance');

    %Loop for finding the best pseudo solution and for finding
    for i=1:size(names)
        p = pseudo.(char(names(i)));
        plotnames = strvcat(plotnames, replace_name(char(names(i))));
        l = size(p,2);
        if l<minlen;
            minlen = l;
        end
        half_this = find(p<halfval, 1, 'first');

        %Store the half-point for this pseudo test
        pseudo_res.(char(names(i))) = half_this;

        %If the half-point was not found, the best possible performance
        %was achieved, so update that
        if size(half_this,2)==0
            pseudo_res.best = zeros(1,0);
        end
```

```
        %Otherwise if the half-point is after the current best, update
        %current best as well
        if size(pseudo_res.best,2)~=0
            if half_this>pseudo_res.best
                pseudo_res.best = half_this;
            end
        end
    end

    %Create the line that will show the 50% performance point
    half = zeros(1, minlen);
    half(:,:) = halfval;

    %Start plotting
    hold on;
    plot(normal.runs.avg.third.mean(1:minlen), 'k');
    plot(half, 'k--');

    graphtypes = strvcat('-', '--', ':', 'r-', 'r-.', 'r--', 'r:');
    for i=1:size(names)
        p = pseudo.(char(names(i)));
        plot(p(1:minlen), graphtypes(i,:));
    end

    legend(plotnames);
    xlabel('Epochs');
    ylabel('Test patterns correctly classified');
    hold off;
end
```

Listing 26: analyze_varied_led_results.m: Source code for analysing the results from experiment 2.

```
function [out] = analyze_cf_results(in_data, io_data, gio_data, ...
    check_point, plotlength, info_str)

    % A helper function for converting the data structure names
    % into names suitable for display in a plot
    function ans = replace_name(inp)
        if strcmp(inp, 'log_8')
            ans = '8 Psuedo patterns';
        elseif strcmp(inp,'log_32')
            ans = '32 Pseudo patterns';
        elseif strcmp(inp,'log_128')
            ans = '128 Pseudo patterns';
        elseif strcmp(inp,'log_500')
            ans = '500 Pseudo patterns';
        else
            ans = inp;
        end
    end

    hold off;
    data = orderfields(in_data);
    names = fieldnames(data);
    check_val= check_point;

%   normal_best = find(normal.runs.avg.third.mean<halfval, 1, 'first');
%   pseudo_res.best = -1;

    %Keep an array of names for the plot.
```

```
%   plotnames = strvcat('Backpropagation', '50% Performance');

    %Loop for finding the best pseudo solution and for finding
    x_io = zeros(size(names));
    x_gio = zeros(size(names));
    epocs = zeros(size(names));
    x_irr = zeros(size(names));
    disp(sprintf('%s&IO&GIO&Epochs until 75% performance\\', info_str));
    for i=1:size(names)
        this_item = data.(char(names(i))).avg.third.mean;
        test_id = strrep(char(names(i)), 'log_', '');
        test_name = cat(2,'test_', test_id);
        out.(test_name).io = io_data.(cat(2,'io_',test_id)).percent;
        out.(test_name).gio = gio_data.(cat(2,'gio_',test_id)).percent;

        %plotnames = strvcat(plotnames, replace_name(char(names(i))));
        check_val = data.(char(names(i))).avg.first.mean(end)*check_point;
        at_checkpoint = find(this_item<check_val, 1, 'first');

        %Store where this test reached the check point
        out.(test_name).epoch = at_checkpoint;

        disp(sprintf('%s&%2.1f&%2.1f&%d\\\\', test_id, out.(test_name).io, ...
            out.(test_name).gio, out.(test_name).epoch));
        x_irr(i) = str2num(test_id);
        x_io(i) = out.(test_name).io;
        x_gio(i) = out.(test_name).gio;
        epochs(i) = out.(test_name).epoch;
    end
end
```