12-2002

Developmental Flight Test of a Powered Approach Stability Augmentation System on the U.S. Navy's E- 2C Hawkeye 2000 Aircraft

Robert K. Williams
University of Tennessee - Knoxville

To the Graduate Council:

I am submitting herewith a thesis written by Robert K. Williams entitled "Developmental Flight Test of a Powered Approach Stability Augmentation System on the U.S. Navy's E- 2C Hawkeye 2000 Aircraft." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Aviation Systems.

Ralph Kimberlin, Major Professor

We have read this thesis and recommend its acceptance:

U. P. Solies, G. W. Garrison

Accepted for the Council:
Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

To the Graduate Council:

I am submitting herewith a thesis written by Simon Lucas Winberg entitled "Neural Supernets: Structuring Artificial Neural Networks According to Cluster Analysis." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Bruce A. Whitehead
Major Professor

We have read this thesis
and recommend its acceptance:

Kenneth R. Kimble

Bruce W. Bomar

Accepted for the Council:

Anne Mayhew
Vice Provost
and Dean of Graduate Studies

(Original signatures are on file with official student records.)

NEURAL SUPERNETS:
STRUCTURING ARTIFICIAL NEURAL NETWORKS ACCORDING TO CLUSTER ANALYSIS

A Thesis
Presented for the
Master of Science
Degree
The University of Tennessee, Knoxville

Simon Lucas Winberg
December 2002

ACKNOWLEDGMENTS

I thank my thesis committee for their support and guidance throughout the development of my thesis. Thanks to Dr. Bruce Whitehead for his guidance, for chairing my thesis committee, and improving my understanding of neural networks. Thanks to Dr. Kenneth Kimble for helping to guide my efforts and the many hours we spent discussing clustering and other heuristic methods. Thanks to Dr. Bruce Bomar for his suggestions and for serving in my committee.

ABSTRACT

A typical feed forward neural network relies solely on its training algorithm, such as backprop or quickprop, to determine suitable weight values for an architecture chosen by the human operator. The architecture itself is typically a fully connected structuring of neurons and synapses where each hidden neuron is connected to every neuron in the next layer. Such architecture does not reflect the structure of the data used to train it. Similarly, in the case where random initial weight values are used, these initial weights are also unlikely to relate to the training set. Thus the job of the training algorithm is to adjust these weights without any initial suggestion for the structure and general trends present in the training data.

This thesis investigates the effect of restructuring a typical fully connected architecture into a collection of *subnets* and processing *modules* that exhibit an application specific ordering. The conglomeration of these modules and subnets will be called a *supernet*.

The processing modules use techniques such as cluster analysis to find general patterns within the training set – somewhat like a low-resolution representation of trends within the data. The subnets are then used for "drilling deeper" into examples that exhibit these trends to produce a higher-resolution representation of aspects within the training set. Additional modules, referred to as *dicer* and *splicer* agents in the text, are used to respectively direct training examples to certain subnets, and join outputs from different subnets. The resultant structure of a supernet is similar to that of a neural network, but instead of being made up purely of neurons and synapses, a supernet is rather an assemblage of processing modules and neural subnets connected with dicers and splicers.

The goal of this thesis is to demonstrate practical advantages of supernets and how they can improve performance of standard training algorithms. A selection of supernet models

are reviewed and compared with a standard fully connected neural network structure.

The results of this practical evaluation show that there are definite benefits to the supernet technique, such as the Classification Based on Subnet Error (CBSE) model that works well with clusters that exhibit dissimilar local behavior. Since a supernet is designed specifically for certain types of application, it cannot be expected to improve performance for general applications. Thus it is necessary to develop a specially tailored supernet for the particular type of application for which it will be used. Although this can result in more work for the neural network developer, it is possible to radically reduce this workload by reusing modules and having a quick and simple means to connect them. If similar types of applications occur frequently, then the effort in developing supernets for these reoccurring applications should provide long-term benefits, since in the long run it will be easier to acquire trained neural networks for these applications. Such an effect is not necessarily achieved using standard neural networks since they are designed for universal use and as such do not have a structure that is optimized for specific types of application.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

η	Learning Rate (for an arbitrary layer in an ANN)
σ	Standard deviation
$\boldsymbol{\delta^{out}}$	Error signal
ANN	Artificial Neural Network
$\boldsymbol{\mathcal{B}}$	A base model
BM	Base Model
$\boldsymbol{C}$	An arbitrary cluster
CBSE	Classification Based on Subnet Error supernet
CC	Cluster Clue supernet
CDD	Concise Description Diagram
CT	Convergence Time
$\boldsymbol{ces}$	Combined Error Signal vector
$\boldsymbol{c_i}$	Center vector for cluster i
CM	Cluster Module
CPS	Cluster Prediction Subnet
$\mathbf{D}$	Distance matrix
$\boldsymbol{d}$	Distance vector
DES	Detail Extrapolation Subnet
DG	Distance Glance
d_j	Distance element of a distance vector $\boldsymbol{d}$
DMM	Determinate Membership Module
$\boldsymbol{e}$	Arbitrary error vector
EP	Embarrassingly Parallel
$\boldsymbol{F}$	A function
FMM	Fuzzy Membership Module
FP	Fuzzy-Prop supernet

HMM	Hybrid Membership Module
$\mathcal{M}$	Used to represent an arbitrary supernet model
$\boldsymbol{m}$	Vector of membership possibility values
MCC	Model Comparison Coordinate
MCC_{early}	Early performance measure comprising horizontal component of MCC.
MCC_{final}	Final performance measure comprising vertical component of MCC.
$M_{det}(x,i)$	Determinate (0 or 1) membership function of vector **x** being in cluster i
MFS	Membership Function Surface
$M_{fuz}(x,i)$	Fuzzy [0,1] membership function of vector $\boldsymbol{x}$ being in cluster i
$M_{hyb}(x,i)$	Hybrid [0,1] membership function of vector $\boldsymbol{x}$ being in cluster i
m_i	A member possibility value for the i^{th} cluster
MM	Membership Module
MP	Multi-Prop supernet model
MTE	Mean Training Error
$MTE_{early}(\mathcal{M},\boldsymbol{P})$	Early Mean Training Error for a model $\mathcal{M}$ and problem set $\boldsymbol{P}$
MVE	Mean Validation Error
$MVE_{final}(\mathcal{M},\boldsymbol{P})$	Final Mean Validation Error for a model $\mathcal{M}$ and problem set $\boldsymbol{P}$
n	An arbitrary number
Nc	Number of clusters
Nh	Number of hidden nodes in total
Nh_i	Number of hidden nodes for a certain subnet (i = subset number)
Ni	Number of inputs, i.e. dimension of input space
NMSE	Normalized Mean Square Error
No	Number of outputs, i.e. dimension of output space
Np	Number of Problem Sets in a problem collection $\boldsymbol{p}$
Nx	Number of training examples
OPS	Output Prediction Surface
$\boldsymbol{P}$	An arbitrary Problem set
$\boldsymbol{p}$	A collection of problem sets
RCT	Relative Convergence Time

r_i	The radius for the i^{th} cluster
$\boldsymbol{S}$	A training set or sample set
$\boldsymbol{V}$	Weight matrix for hidden layer of backprop subnet
v_{jk}	Single weights value for a node in the hidden layer / element of $\boldsymbol{V}$
$\boldsymbol{W}$	Weight matrix for output layer of backprop subnet
$\boldsymbol{w}$	Weight vector
w_{ij}	Single weights value for a node in the output layer / element of $\boldsymbol{W}$
$\boldsymbol{x}$	Arbitrary input vector for a module or supernet
$\boldsymbol{y}$	Intermediate output vector
y	Intermediate output scalar value
$\boldsymbol{y}^{\boldsymbol{\varepsilon}}$	Error vector propagated backward to hidden layer
$\boldsymbol{z}^{\boldsymbol{\varepsilon}}$	Error vector propagated backward to output layer
$\boldsymbol{z}^{\boldsymbol{des}}$	Represents a desired output used in training a network
$\boldsymbol{z}^{\boldsymbol{error}}$	Error vector for a subnet (vector difference $\boldsymbol{z}^{\boldsymbol{out}} - \boldsymbol{z}^{\boldsymbol{des}}$)
$\boldsymbol{z}^{\boldsymbol{out}}$	Represents the predicted output for a network

1 INTRODUCTION

Clustering is the process of grouping items together based on self-similarity. This is one of the most natural ways to split a large, disordered set of data into a more ordered collection of subsets where elements in a certain subset are more closely related to one another than to elements in other subsets.

An everyday illustration of effective clustering is to consider how a person might construct a difficult, 1000-piece jigsaw puzzle.

Initially all 1000 pieces are in a disorganized pile lying on a table. Some pieces are face-up while others show only a cardboard backing. The usual strategy that most sensible people use when building a puzzle involves organizing the pieces before actually attempting to connect the pieces together. It would indeed make things far more difficult if this step was skipped and the person building the puzzle decided to start by selecting one piece from the top of the pile, putting it face-up on the table and then attempting to find a matching piece in the rest of the pile. Such a technique might work with very simple puzzles, but for larger puzzles the only outcome would be frustration and time wastage.

The typical organizational strategy for building a puzzle (*Jigsaw Puzzle*, 1997/2002) is to first categorize the pieces according to corner and side pieces, then to arrange all pieces (maintaining the separation of the sidepieces) into piles of similar-colored pieces, and then possibly to divide these colored piles even further according to the number of connections each piece has. Only then, when all the pieces are organized should the puzzle be build, focusing on getting the sides of the puzzle completed first. The general strategy in connecting the pieces is to get the easy pieces connected first so that when it

comes to selecting more difficult pieces to join there are fewer options to try.

In all these steps it is quite easy to delegate tasks to other people who are willing to help with the puzzle. For instance during the first step of organizing the pieces into piles people can be designated to look for certain types of puzzle pieces. Then, when it comes to connecting pieces, each person concentrates on building one or more sub-puzzles with the ability to add to other sub-puzzles if suitable pieces are spotted.

This discussion shows that the method of solving this particular problem effectively relies on imposing an order on how the puzzle pieces are grouped together and on choosing the right set of procedures to follow when connecting the pieces.

Although the act of building a puzzle is a physical task performed by a human, there are some parallels that can be drawn between building a jigsaw puzzle and using a collection of simple functions to approximate a more intricate function. The human puzzle builder is attempting to join a set of pieces that fit together in a specific way so that the end result of this construction represents a conceptual whole. In a similar way a function can be approximated by "fitting" together an appropriate selection of simple functions. For example consider how the exponential function can be approximated by the first few terms in its Taylor series for values of x between 0 and 1 using the basic functions 1, x, x^2, x^3 as follows:

$$exp(x) \approx 1 + x + x^2/2 + x^3/6$$

Here these basic functions, or *function kernels*, have been joined together using a weighted sum of their outputs. This weighted sum produces the requested result (illustrated in **Figure 1**), which is an approximation to the exponential function.

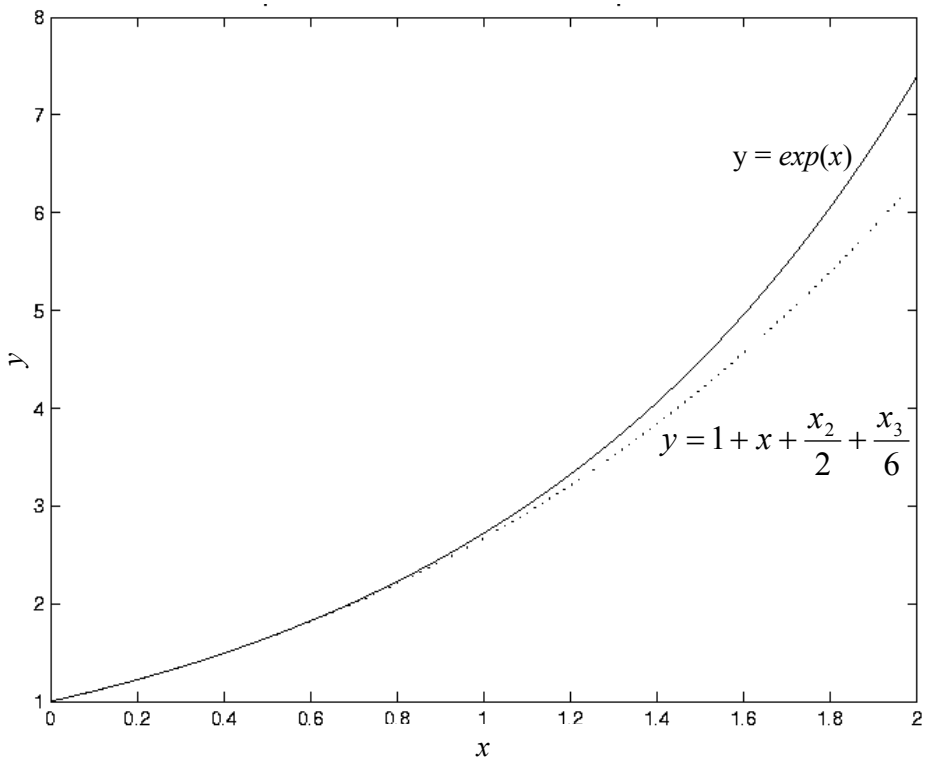


Figure 1: Exponent Function Approximation.

An Artificial Neural Networks (ANN), when used for function approximation, is attempting to perform such an operation automatically by combining a set of function kernels in such a way that together they produce an approximation for a certain function. An ANN is also commonly referred to as a *universal function approximator* because it is meant to be capable of generating an approximation for *any* given function. Thus the problem an ANN faces when doing this can be considerably more difficult than building a jigsaw puzzle.

Instead of being limited to two dimensions, like a jigsaw puzzle, a general ANN needs to be able to produce approximations for functions operating in *n* dimensions. And instead of a finite number of possible permutations in which puzzle pieces can be joined, an ANN has to deal with a practically infinite number of permutations in ways to connect its functional kernels. Thus an ANN has to deal with both the "curse of dimensionality" (Duda & Hart, 1973) and extreme permutations in it attempt to find a suitable solution.

However, an ANN also has a few strategies that can make its task easier, choices that are not available for jigsaw puzzles. An ANN can choose its function kernels (or functional "puzzle pieces") and how to connect them. In contrast, jigsaw puzzles have a specific set of pieces that can be connected together in only one way.

Sigmoid functions are one of the easiest and most effective choices for function kernels. They are essentially a kind of smoothed step function and therefore can be used to approximate transitions within a data set. The output of a sigmoid is symmetric and bounded. The transition between these bounds can be made either gradual or sharp so as to better mimic transitions within in the original data.

A sigmoid in 3D looks somewhat like an elastic sheet that can be stretched out to "cover" the training data. Although the sigmoid has what appears to be an uninteresting basic shape, this is probably the most powerful aspect of the sigmoid because it is usually easier to construct a complex structure using simple shapes that it is to use more complex

shapes that have many features (the features tend to get in the way and make the problem harder). A combination of these basic sigmoid shapes leads to far more interesting and detailed shapes, as shown in **Figure 2**. Each additional sigmoid has an effect of introducing additional folds to the "sheet" used to cover the data.

1.1 Problem Statement

Standard training methods for neural networks, such as backpropagation (discussed in **Section 2.2**), are excellent for "fitting" function kernels together. They generally make small changes to weight values in the network until the network suitably approximates the global behavior of its training data. But such a method does not have a means to make explicit use of any prior knowledge regarding the training data, such as how examples could be ordered, which examples have similar output behavior, which examples are most different from one another, etc. Using the jigsaw analogy, a standard training algorithm appears to be doing two things simultaneously: attempting to both organize and connect pieces at the same time. The normal way of using these algorithms may be the best way to train a neural network if no prior knowledge is available regarding the training data in use. However, if prior knowledge is available, it should be possible to use these training algorithms in a different way so that they produce better results. This is the hypothesis that motivated the development of this thesis.

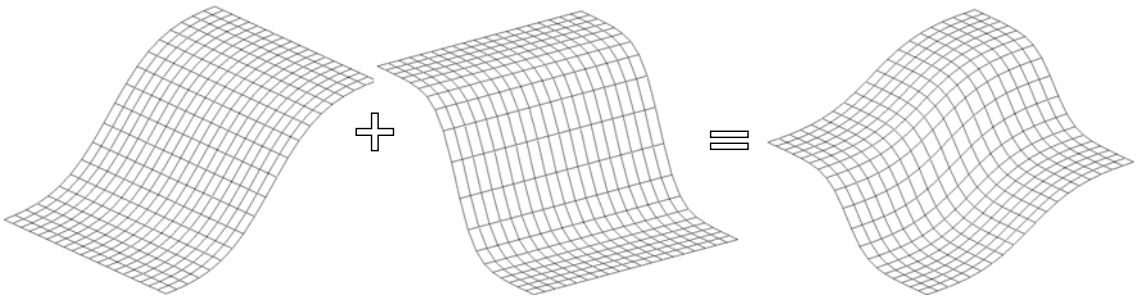


Figure 2: Combinations of Sigmoids.

1.2 The Supernet Concept

This thesis formulates a hybrid neural network, called a "supernet", that changes the form of a standard neural network so that it has separate pieces that are responsible for exploiting known properties of the training data. These pieces are used for the following high-level tasks:

- Organization and separation of data
- High- and low-level pattern discovery
- Transmission of data to appropriate sub-processes
- Combination of sub-process results

The architecture of a supernet has significant differences, when compared to that of a standard fully connected feed forward neural network model. Conceptually, the ordering procedures (referred to as "dicers") that are used to organize the supernet's training data have an effect of dissecting the fully connected structure of a standard model into pieces or subnets. These pieces are separately trained, but every training example does not necessarily pass through the entire network. In fact some training examples may be cast out entirely if they are deemed redundant. This behavior is quite unlike that of the standard model. The outputs of subnets are then joined using "splicing" agents to produce a final output of the supernet itself. This concept is illustrated in **Figure 3**.

This thesis develops a selection of supernet models that exhibit various forms of ordering and joining operations. These models were developed for a certain set of problems in an attempt to improve on the performance characteristics shown by standard neural networks trained on the same problems.

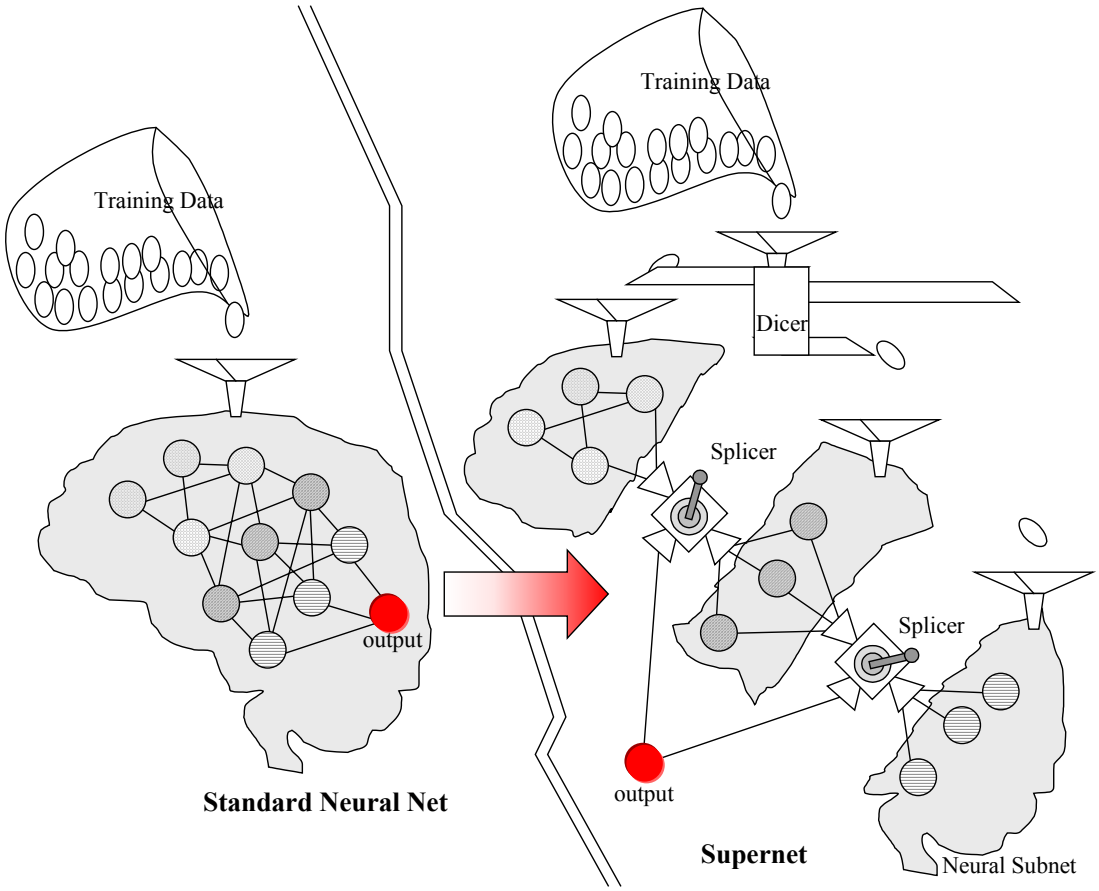


Figure 3: Conceptual Supernet Structure.

1.3 Critical Terminology

The following points introduce terminology that is used throughout this text:

- In the context of this thesis, the term "network" is used interchangeably to refer to the high-order operation of either a supernet or ANN.

- The *overseer* is a human who wants to acquire a trained network by making it "learn" from a particular training set.

- A *trained* network is one that has "learned" the patterns defining its training data and can use these patterns to predict outputs for inputs that were not given in the original training data, but are still within the *input domain*.

- The *input domain* of a network comprises the space of all meaningful inputs that can produce meaningful outputs and is dependent on how well the training data covers this domain.

- If a network is generally able to produce predictions within a certain error tolerance, the network is proclaimed to provide good *generalizations*, and no further training is required.

- A *supernet* is defined to be an interconnection of subnets and modules. The high-level behavior of a subnet, when treated as a single input-output process, is congruent to that of a conventional neural network since its purpose is also function approximation or classification. But the actual design and structure of supernets are not congruent to those of a conventional neural network.

- S*ubnets* are standard feed forward neural networks that are trained exclusively by a standard training algorithm (such as backprop). The subnets used in this thesis have a fully connected architecture comprising an input layer, single hidden layer and output layer. However, subnets need not be limited to this particular structure.

- Both supernets and subnets have two distinct operations: *train* and *yield*. The *train* operation is the process by which the network learns training data. The *yield* operation is the process by which a network predicts an output for a particular input.

- Any network that classifies data yields a response that is a *classification*, whereas a network that performs function approximation yields an *approximated output* for the function.

- *Training sets* comprise training data that is derived from a certain application (or process) and is used to train a network. The term "set" is used to indicate that the training data is organized into *sets* of *training examples*, where each example represents an input and consequential output derived from the application.

- *Validation sets* contain *validation examples* that relate to a specific training set. The validation examples are used to determine the degree to which a network can generalize to arbitrary examples within the network's input domain. These sets have the same structure as training sets, but should not contain the same examples as given in the training set. One or more validation sets can be used in testing the network, although in this thesis the validation examples for a certain training set are all collected into a single validation set.

- An *epoch* refers to one complete pass through a training set where each example is processed by the network.

- The *prediction error* is a quantity proportional to the difference between an example output and the predicted output for the corresponding example input.

- The *mean training error* (MTE) is the average prediction error that the network exhibits when predicting outputs for its own training set.

- The *mean validation error* (MVE) is the average prediction error exhibited by the network when predicting outputs for all validation sets related to its training set.

The most important performance characteristic is the MVE. Generally the overseer would select a certain MVE value that is adequate for the application concerned and then initiate the network's training processes and continue training the network until the MVE is achieved. The overseer is responsible for selecting suitable parameters to accomplish this.

1.4 Objectives

The focus of this thesis is to perform a practical investigation into the feasibility of the supernet concept. The desired outcomes of this investigation are:

- Determining the benefits supernets offer over standard neural network models
- Finding what the limitations of the supernet concept are
- Devising an effective means to develop and train supernets

1.5 Thesis Structure

The structure of this thesis is as follows:

- **Chapter 1** introduces the problem this thesis investigates and the outcomes it

aims to provide.

- **Chapter 2** provides a literature review that is drawn upon in the development of methods used in this thesis.
- **Chapter 3** develops methods for evaluating and creating supernets.
- **Chapter 4** provides detail on the reusable modules that are instrumental to the functioning of supernets.
- **Chapter 5** presents the design of each supernet model developed, together with test results and a discussion relating to possible optimizations for the models.
- **Chapter 6** evaluates the overall performance each supernet in relation to the other models and discusses for which types of problems these supernets are best designed to handle. A conclusion is provided that discussion what this thesis achieved together with suggestions for future research on the topic.

2 Background

This chapter reviews some of the core theories and methods drawn on in this thesis. **Chapter 4** provides a selection of algorithms for the methods reviewed in this chapter.

2.1 Artificial Neural Networks (ANNs)

An Artificial Neural Network (or ANN) is a data processing paradigm that was inspired by the way in which an organic (i.e. human or animal) brain functions. An ANN is an attempt to design a computer program that mimics the ability of the organic brain to estimate responses for certain stimuli based on only a small number of training examples (Abdi et al, 1999).

An organic brain features a huge number of neurons that are traditionally assumed to be connected together by synapses. Each synapse is a unidirectional connection that transmits either excite or inhibit messages from a source neuron to a destination neuron. An input stimulus gives rise to a sequence of these neuron activations, often called "firings", that propagates through parts of the brain at various levels of excitation or inhibition, finally terminating in some form of response or decision (even if this decision is to do nothing). This is all happening on an immense scale: the number of neurons in the brain is in the order of billions, while the number of synapses in a brain is many orders of magnitude greater than this. In fact there are as many as 10,000 synapses for each neuron (Haykin, 1994: 2). So even simple decisions are likely to be derived from a massive number of neural firings.

Even though a microprocessor, which performs events in the nanosecond range, might appear blindingly fast compared to the operation of neurons in a brain, which typically

perform events only in the millisecond range, the vastly parallel and interconnected nature of the brain greatly outperforms a microprocessor for highly complex tasks such as image recognition and converting sounds into meaning (Kosko, 1992: 2).

As in the case of an organic brain, an ANN has certain input stimuli, synapses and artificial neurons called nodes. But in the case of an ANN the input stimuli are numbers in a vector, the synapses simply pass on weighted values from one node to another, and the nodes themselves are mathematical functions.

A typical ANN, such as the one shown in **Figure 4** has at least three layers: an input layer, one or more hidden layers and an output layer. The input layer contains one node for each element of the input vector (***x***) and each input node is fed an element of the input vector (i.e. the *i'th* neuron is fed with x_i) and passes this value on to the first hidden layer.

The layers between the input and output layers are called hidden layers because the nodes in these layers neither receive input directly from sources, nor send values directly to destinations that are external to the network.

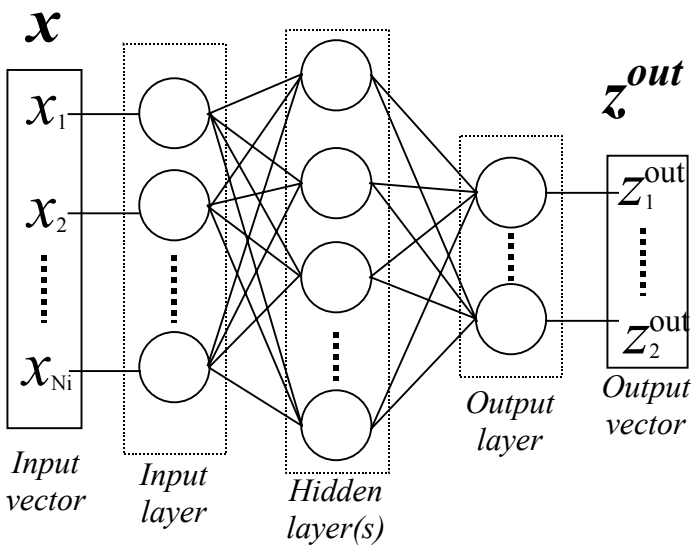


Figure 4: Fully Connected Artificial Neural Network.

In a fully connected feed forward artificial neural network each input node passes its value on to every node in the first hidden layer, then every node in the first hidden layer passes its output to every node in the second hidden layer, and so on until the last hidden layer passes all its outputs to every node in the output layer. The outputs that are produced by the nodes in the output layer are assembled into the final output vector ($\boldsymbol{z}^{out}$) that forms the predicted output of the network.

From this discussion, a feed forward network can be summarized as one that passes results only forward from one layer to the next and does not have any loop back connections. Although feedback network models do exist, such as Fuzzy Cognitive Maps (Kosko, 1992: 152), only feed forward networks are considered in this thesis.

The actual operation of nodes within an ANN determines the overall operation of the ANN. The operation of input nodes is trivial since they merely pass on input values unchanged. However, the processing power of the neural network lies in its hidden nodes and output nodes.

Figure 5 illustrates a typical node found in either a hidden layer or output layer. Generally such a node has multiple inputs represented by elements of a vector $\boldsymbol{x}$ and an equal number of weight values, represented by a vector $\boldsymbol{w}$. The elements of the input vector $\boldsymbol{x}$ are combined using the weight values in $\boldsymbol{w}$ to form a scalar value that is then passed through the *transfer function* $\boldsymbol{f}$. There are various ways to combine the input elements of $\boldsymbol{x}$ using the weights $\boldsymbol{w}$, but the easiest and by far most common method is to use the dot product of $\boldsymbol{w}$ and $\boldsymbol{x}$. The result returned by $\boldsymbol{f(w^T.x)}$ is the output of the node and is shown on the diagram as the scalar value y.

If the node is in the hidden layer then output y is passed on as an input element to the nodes in the next layer. If the node is in the output layer, then y is actually an element of the final output vector $\boldsymbol{z}^{out}$ of the neural network.

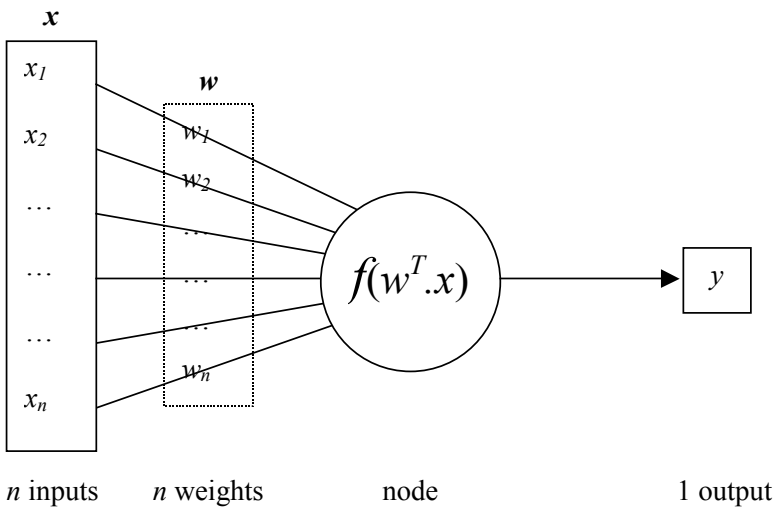


Figure 5: A hidden or output node.

The choice of ***f*** depends more on the training data used and the application in which the network is used, but generally choosing ***f*** from the family of sigmoid functions produces adequate results (Theodoridis & Koutroumbas, 1999: 92).

From the design of a single node, it becomes apparent that the weights are in effect mimicking the operation of synapses in a brain, since in a neural network the weight values are a form of excitation or inhibition message. The weight values are also the only values that are not fixed by the operation of the network. So the way in which a network is trained is by adjusting its weight values so that the predicted output, z^{out}, of the network is suitably close to the example outputs given in the training set.

There are two classes of neural network training methods: either *supervised* or *unsupervised* methods. Supervised methods train a network using an external "teacher" (Haykin, 1994: 57) that tells the network what output should be given for a certain input. Training examples for such algorithms comprise an input part and a corresponding output part. The backpropagation algorithm described in **Section 2.2** is an example of a supervised training method.

Unsupervised learning (or self-organization) occurs where there is no external "teacher" available to correct the learning performed by the network. The training examples therefore have only an input part and no output part. The free parameters (not necessarily weights) of such networks are generally tuned according to statistical regularities in the data (Haykin, 1994: 65). Clustering is a form of unsupervised learning and is used by sub-processes in some of the supernet models (see **Section 4.1**).

An important factor regarding the operation of a neural net is the choice of how output from the network is used for a certain application. In the case of *conditional* outputs, such as when an ANN is controlling a system, **binary** or **bipolar** outputs may be desired (Kosko, 1992: 43). Binary signals are either **0** (for false) or **1** (for true). However, since neural nets produce *fuzzy* outputs, forcing these outputs to be *binary* would hide the uncertainty of these results. This could have a detrimental effect on the training of the network because it would be difficult to determine by how much a predicted output is in error when compared to a desired output (i.e. it would either be completely wrong or completely right). Bipolar signals are often used instead as they are real values (usually in the range **[-1, 1]**) and can provide the training algorithm with a means to determine a degree of "falseness" or "trueness" based on the magnitude of negative or positive results.

As mentioned in **Section 1.3**, the aim of a neural network is not to produce perfect predictions for its training set, but rather to *generalize*, being able to produce suitably correct predictions for arbitrary examples not found in the training set. Thus the choice of a neural net's architecture and the mechanism by which it is trained should be optimized for this concept of generalization.

2.2 Backpropagation

Backpropagation is a supervised learning algorithm that computes the synaptic weights for a feed forward neural network. It is in essence a minimization process which finds optimal weight values that causes the output error of the network to reach a global minimum.

There are two basic modes in which supervised training algorithms do this: either using *batch mode* or *incremental mode*. In the batch mode weights are changed by an averaged amount only at the end of each epoch, while the incremental mode changes the weight for each training example. The operation mode used for this thesis is incremental, as this is well supported by the EbTide neural network testing software discussed in **Section 2.5**

Backpropagation is a form of gradient descent that operates in the weight space. Therefore each transfer function used by each node in the network must be differentiable (Haykin, 1994: 185). Sigmoid and gaussian functions are used in the code developed for this thesis; they are both differentiable functions.

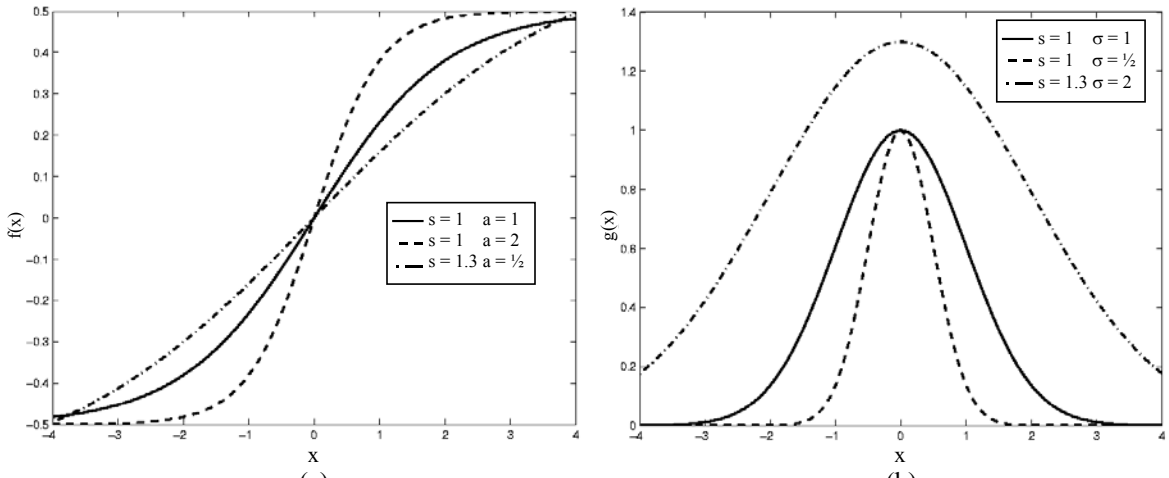
Generally, the choice of using sigmoids or gaussians depends on what type of training data is being used. Remember from **Chapter 1** that sigmoids are like flexible sheets that are good for "stretching" over training data. So if the training data has mostly smooth areas and few "dips" and "peaks" then a sigmoid should be able to approximate such data well. The gaussian, on the other hand, is better suited to data that is more "bumpy". Seeing as the shape of a gaussian is itself a "bump" these functions are expected to work well for approximating "bumpy" data. Gaussians can be used as a kernel function for a backpropagation network or as Radial Basis Functions (RBF) in RBF Networks (discussed in **Section 2.3**).

The formulae for the scaled sigmoid and gaussian functions are shown below together with their first derivative:

Scaled Sigmoid: $f(x) = s \cdot \left(\frac{1}{1+e^{-ax}} - 0.5 \right)$ $\qquad f'(x) = \frac{as \cdot e^{-ax}}{(1+e^{-ax})^2}$

Scaled Gaussian: $g(x) = s \cdot \left(e^{-x^2/2\sigma^2} \right)$ $\qquad g'(x) = -s \cdot \left(\frac{x \cdot e^{-x^2/2\sigma^2}}{\sigma^2} \right)$

The *s* parameter represents a scaling factor for both functions, *a* changes the slope of the sigmoid and σ is the standard deviation for the gaussian that determines how steep its graph is. **Figure 6** illustrates the effect of these parameters. Notice that smaller values of *a* cause the sigmoid to be become drawn out, making the transition more gradual. Similarly, small values of σ cause the gaussians to be steeper. The parameter *s* is used to ensure that the range of a function is sufficient to reach all the desired output levels, while the *a* and σ parameters are used to tweak the functions allowing for more convenient weight ranges (floating point values have a limited precision after all).



(a) (b)

Figure 6: Effect of altering sigmoid and gaussian parameters.

(a) Asymmetric sigmoids and (b) gaussians.

The Backpropagation algorithm is a two-stage process. In the first stage a training example ($\boldsymbol{x}$) is fed forward through the network producing a predicted output vector ($\boldsymbol{z}^{out}$). The next stage computes an error vector ($\boldsymbol{e}$) that is the vector difference between $\boldsymbol{z}^{out}$ and the desired output ($\boldsymbol{z}^{des}$). This error vector is transformed into an error signal (δ^{out}) using the derivatives of the output nodes' transfer functions and their weights. This is explained in Abdi et al (1999: 74). This error signal is then propagated further backward one layer at a time in the same way.

Once the error signal has been backpropagated through the network using the connection weights, the weights are adjusted so as to minimize the mean squared error between the desired and actual outputs. The weights are changed iteratively by a small amount. The amount by which the weights are changed depends on the *learning rate* (η) used for the node in question. Generally all nodes in a single layer all have the same learning rate.

The backpropagation algorithm is applied for a certain number of epochs until the error is reduced to a desired level (to the desired mean training error). If the error value does not converge towards a small enough value or converges too slowly, then the training has to be restarted using different values for the training parameters (i.e. the learning rate or number of hidden nodes needs to be modified) or different initial weight values.

The standard backpropagation algorithm is given in **Section 4.3**. Further detail regarding backpropagation and a detailed explanation of is implemented is given in Saarinel el al (1992: 31-42).

2.3 Radial Basis Functions (RBFs)

A Radial Basis Function (RBF) network is a form of neural network whose kernels functions are radially (i.e. spherically) symmetric around a center point. The nodes in an

RBF networks also have weight vectors that converts the input vector to the node into a scalar value. However, the weight vector is commonly called a centroid as it represents the point around which input vectors produce symmetric output.

At a high level, the basic form of an RBF network is equivalent to that of a general neural network described in **Section 2.1** since it is a feed forward network that has three layers (where the input and output layers are joined by a single hidden layer). However, its inner workings are slightly different. An RBF network generally uses a form of Euclidean distance function instead of a simple dot product as a means to map its input vector to a scalar value. This scalar value is then passed through a gaussian function. Thus a standard RBF network has the following form of function kernel:

$f(\|x - c_i\|)$, $f(x) = e^{-x^2/2\sigma^2}$ where σ is the standard deviation of the gaussian, and $\boldsymbol{c_i}$ is a centroid

A view of the output produced by an RBF kernel function operating on a grid of 2D input points (the x-y plane) is given in **Figure 7**. This illustrates the symmetric behavior of RBFs.

The term "Radial Basis Function" thus refers to using a radial distance from a certain centroid to calculate the output of the function. Consequently, for a specific RBF function, all inputs that are an equal distance from the centroid all have the same output.

The RBF network learns a dataset by moving its centers around so that the combined outputs of the RBF functions approximate a given training set. There are various implementations of such training algorithms. For instance, a simple method is to use the k-means clustering algorithm to determine the positions of the centroids, but there are more optimal methods, such as the genetic algorithms devised by Whitehead and Choate (1996).

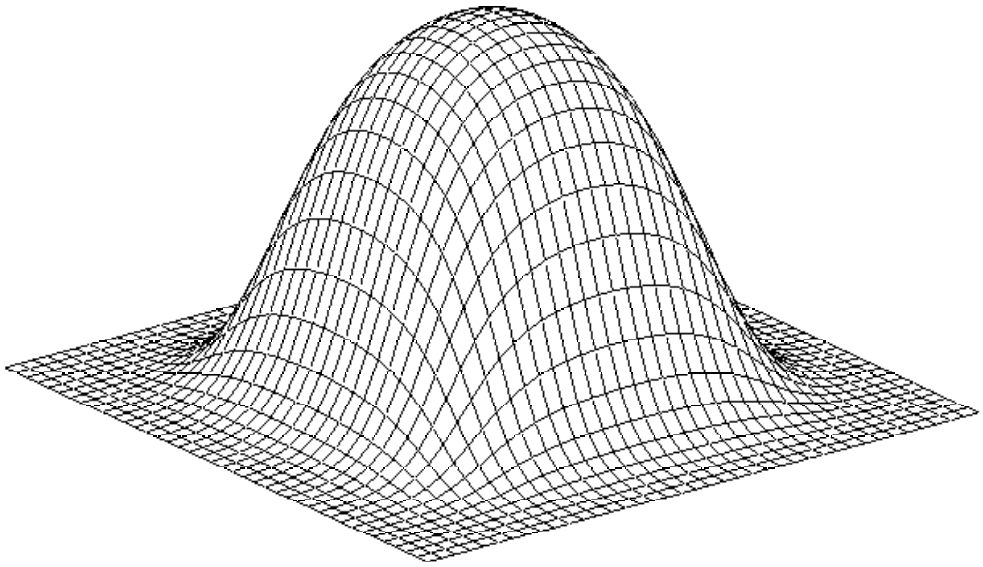


Figure 7: 3D gaussian Surface.

Another significant difference between networks that use sigmoid functions and RBF networks is that changing the weights used with sigmoid functions has a global effect on the predicted values of the network (Moody and Darken, 1989). But, changes to centroids that are used with gaussian functions have only a local effect on the predicted results of the network. This is caused by the way in which sigmoid functions are used to approximate training data as a combination of steps. In such a situation, each step adds or subtracts to the value of the other steps, thus when one of the weight values is changed in the network, it is necessary to adjust all the other weight values so as to compensate for the global change in the predicted output.

When the centroids are adjusted in a RBF networks, the predicted results of the network are only changed locally. Thus, when a change is made to one of the centroids used in a RBF network, the only compensation required involves adjustment to the centroids that are close to the centroid that was changed. Thus RBF networks only experience a local change in their predicted output when a single centroid is changed.

A network model that exhibits local changes during each training example is likely to learn its data more quickly than a network that exhibits global changes for each training example (Dwinnell, 1998). This generally occurs because in the local case there are fewer updates required to compensate for changes in the weight values that in the global case. Thus a RBF networks generally requires fewer training epochs to learn its training data than is require by a standard network using sigmoids.

2.4 The Fuzzy Membership Function

A cluster is generally thought of as a grouping of elements that are gathered together according to self-similarity or by proximity. However, in some situations it is not always possible to determine if a particular element is a member of a specific cluster. When attempting to partition training examples according to clusters, there may be a selection of examples that cannot be classified with absolute certainty. In such cases it is useful to employ the techniques of fuzzy logic to account for ambiguities in the data. Considering that neural networks offer an inherent fuzzy nature as well, it seems appropriate to use fuzzy clustering techniques on training data.

The term "fuzziness" in the field of Mathematics involves "multi-valence", and stems from the Heisenberg position-momentum uncertainty principle of quantum mechanics (Kosko, 1992: 3). Fuzzy truths correspond to truth, falsehood or a degree of indeterminacy between truth and falsehood. The fuzzy membership function (also commonly called the *multi-valued indicator function*), $M_{fuz}(\boldsymbol{x}, C)$, determines the degree of membership to which an element $\boldsymbol{x}$ belongs to a cluster C (Kosko, 1992: 6). The result is a value in the range [0,1], where the function is defined as:

$$M_{fuz}(\boldsymbol{x}, C) = \begin{cases} 1 \; if \;\; \boldsymbol{x} \in C \\ 0 \; if \;\; \boldsymbol{x} \notin C \\ \in (0,1) \; otherwise \end{cases}$$

If $\boldsymbol{x}$ is definitely in the cluster C, then the result of $M_{fuz}(\boldsymbol{x}, C)$ is 1. If $\boldsymbol{x}$ it is definitely *not* in C the result is 0. But if $\boldsymbol{x}$ is neither definitely in nor definitely not in C, then the result is some value between 0 and 1. Values closer to 1 indicate a greater possibility that $\boldsymbol{x}$ belongs in C, and values closer to 0 indicates the lesser possibility that $\boldsymbol{x}$ belongs in C. A value of ½ indicates total ambiguity, meaning it is impossible to tell whether $\boldsymbol{x}$ should or should not be in C.

The determinate (or binary) membership function $M_{det}(\boldsymbol{x}, C)$ is essentially a special case of the fuzzy membership function that is limited to an output of either 0 or 1. This form of fuzzy membership function is used for supernets that are designed to process data that contain non-overlapping clusters.

2.5 EbTide (Example-Based Training Interface Definition)

EbTide is a shared library developed at UTSI that provides a means to rapidly produce C++ prototypes for neural network models. EbTide provides a well-defined class interface to provide linkage between the shared library and the model prototype that is to be trained, fitted or adapted according to a set of examples. This interface is bi-directional, enabling the EbTide library to call procedures (such as train) in the model, and the model can invoke certain EbTide procedures. Definition files form part of the code written for a given neural network model, these files provides a means for EbTide to access state variables used by the model.

EbTide provides the following services:

- 3D Visualization for the training process
- Statistical reports used when evaluating the performance of a certain model
- A GUI (Graphical User Interface) for facilitating the operation of EbTide
- Streaming of model states to/from file

The 3D visualizations provided by EbTide show movement of hyper planes projected in 3D that indicates how the weight values change. Translucent "clouds" are used to show cluster volumes that the model has found. The position of data points indicates the input part of the examples (mapped into 3D) while the labels or colors of the data points provide an additional dimension that show desired outputs of the examples. **Figure 8*(a)*** illustrates a sample view of a training set containing 5 non-overlapping clusters.

A surface projection can be generated by the program that allows the user to view relationships between a grid of input vectors mapped into 2D and their corresponding predicted outputs mapped into 1D. **Figure 8*(b)*** shows an Output Prediction Surface (OPS) generated from the 5-cluster training data used in **Figure 8*(a)*** (notice that the central cluster had the highest average desired output).

These visualizations are useful for debugging a particular neural network model and to get a "feel" for what the results produced by the model look like. The statistics are particularly useful when it comes to evaluating the performance of a certain neural network model.

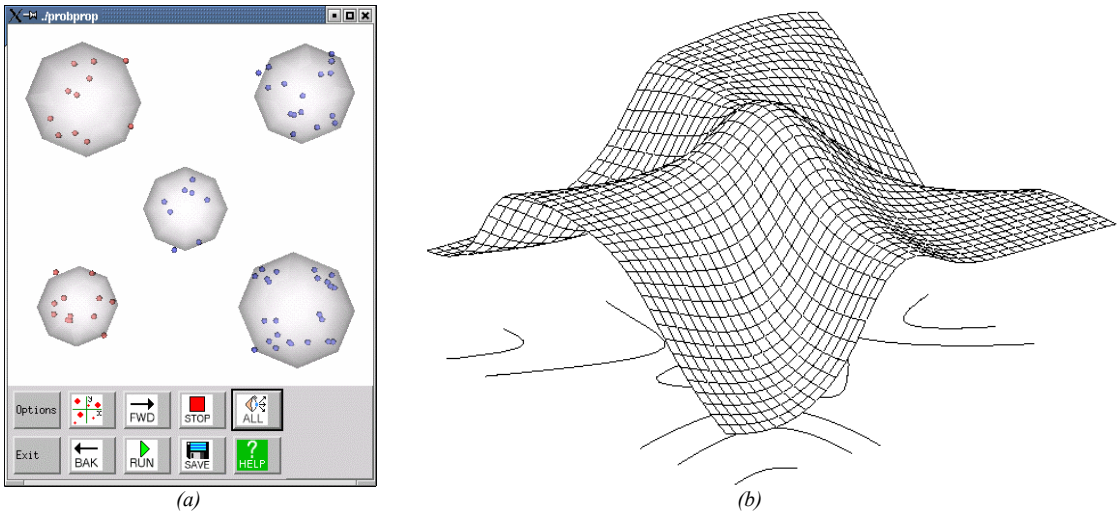


Figure 8: EbTide Vizualization Features.

(a) EbTide Cluster Vizualization, (b) EbTide OPS plotted in Matlab.

3 EVALUATION AND DEVELOPMENT

This chapter develops the design and evaluation strategies used to develop, test and compare supernet models. These strategies are categorized into three major parts: 1) evaluation techniques, 2) problem sets, and 3) techniques for accomplishing the tests quickly. The evaluation techniques are of particular importance since they are fundamental in determining the actual outcomes for this thesis.

3.1 Evaluation methods

There are three desired outcomes in the evaluation of a particular supernet model. Firstly, it needs to be determined whether a particular supernet can be trained more quickly and produce better predictions than a standard model. Secondly, it is necessary to decide if one supernet model is better or worse in terms of performance than an alternate supernet model. And thirdly, it would be useful to know which aspect of a particular supernet causes it to perform better or worse than an alternate model.

The technique used in this thesis was to develop a *base model* for comparison, which exhibits the performance of a standard neural network. This base model will provide a standard with which to measure the performance of other models; this standard will be a minimal performance measure that supernet models must attain. If a particular supernet model is found to offer the same or worse performance than the base model for a certain collection of problems, then that supernet model can be discarded for that application domain, as it provides no benefit over the base model.

The evaluation method comprises two sub-methods: a testing method and a comparison

method. The testing is performed on an individual supernet basis, with no reference to the base model or other supernets; it purely establishes on a statistical basis how well the supernet has managed to learn a specific set of problems. However, the comparison method is concerned with relating the performance of a certain supernet model with that of other models. The comparison method essentially produces a "normalized" performance rating in relation to the base model, and this measure is used to determine the success or failure of a particular supernet model.

3.1.1 The Base Model

The requirement of the base model is that it must be as close as possible to the most standard and commonly used model of fully connected feed forward neural networks. Since there's only really one choice in connection architecture, using fully connected nodes, the biggest decisions in choosing such a model are: 1) whether to use one or more hidden layers, and 2) which training algorithm should be used.

On the topic of choosing the number of hidden layers there is some debate as to whether or not a neural network with multiple hidden layers is more powerful than a network with a single hidden layer (Hornik et al, 1989). For ease of implementation and since most simple backpropagation neural networks have just one hidden layer, the base model chosen has a single hidden layer.

The choice of training algorithm is less significant since the goal of the thesis involves determining how best to use a standard training algorithm when prior information regarding the structure of the training set is known. However the most commonly used training method at present is backpropagation therefore this method was chosen for training the base model.

Thus the design of the base model is a traditional feed forward neural network that exhibits a fully connected architecture containing a single hidden layer. The number of

input nodes and output nodes are fixed according to the input and output dimensions of the training set. The number of hidden nodes is a free parameter, and so are the learning rates used to training nodes in the hidden layer and output layer.

3.1.2 Testing a Model

This thesis attempts to follow the validation guidelines as presented in Burke's (1993: 20-24) article on assessing neural networks. This article stresses the importance of using an error measure that is both normalized and which represents the standard deviation in addition to the mean of differences between a network's results and the desired results. It also indicates the necessity for testing a network with a validation set that is different from the training set. Such a technique provides a fairer indication of how well the network has learned the patterns governing the process that produced the training set.

Before continuing with a description of how a model is tested, a few definitions are required:

- A *sample set* (***S***) is used to refer to either a training set or a validation set (they both contain a set of examples).
- The number of examples in the sample set is *Nx*.
- For an arbitrary input ***x*** to the network, the corresponding desired output for this input is $z^{des}(x)$.
- When a network is fed the input ***x*** it yields a predicted result $z^{out}(x)$.

The *normalized mean square error* (NMSE) is a method for comparing the mean desired output of a sample set against the predicted outputs yielded by the network for inputs in the set. The NMSE is calculated in such a way that if its value is greater than 1, then the predictions are less accurate than the constant prediction of the sample mean. The NMSE is determined by dividing the *root mean square* (RMS) of prediction errors by the

standard deviation (σ) of the desired outputs.

$$NMSE = RMS/\sigma$$

$$RMS = \sqrt{\frac{1}{Nx}\sum_{x \in S}\left(z^{out}(x) - z^{des}(x)\right)}$$

$$\sigma = \sqrt{\frac{1}{Nx}\sum_{x \in S}\left(z^{out}(x) - \bar{z}^{des}\right)^2}$$

$$\bar{z}^{des} = \frac{1}{Nx}\sum_{x \in S}\mathbf{z}^{\mathbf{des}}(\mathbf{x})$$

During training an NMSE value will be calculated at the end of each epoch to indicate the *Mean Training Error* (MTE) present at the end of that particular epoch. These MTE values will show the *instantaneous performance* of the network for each training epoch. These values can then be graphed as a means to visualize the training progress. The *final MTE* value (the value calculated for the last training epoch) will indicate how well the model predicts its own training data at the end of training. Another NMSE measure will be calculated with respect to the *validation set* and this will be used as the *Mean Validation Error* (MVE) for the network. This MVE value (as mentioned in **Section 1.3**) indicates the level of generalization that the network has reached.

Each model tested in this thesis will be provided with a table of MTE and MVE results that documents how well the model performed on a common collection of problem sets.

3.1.3 Overfitting & Hidden Node Restrictions

Generally, the purpose of a neural network is to provide, on average, the best possible predictions for arbitrary input data that is not explicitly contained in the training data. The network is supposed to approximate outputs based on the patterns learned from the

training data. A problem known as "overfitting" occurs when a neural network has too many free parameters. This may cause the network to predict the training data very accurately, but is likely to seriously degrade the network's ability to generalize. This is caused by arbitrary inputs being predicted inaccurately due to there being too little restriction on the behavior of the predicted outputs in regions between the training examples. This problem can also occur when fitting a polynomial to a set of samples. Looking at the polynomial case helps to explain the problem of overfitting data.

Figure 9 shows two attempts at fitting two different polynomials to the values [3, 1, ½, 1, 3]. The underlying function from which the samples are taken is represented by the dashed line *f(x)*. **Figure 9(a)** shows a polynomial *p(x)* of degree 2 (i.e. only two degrees of freedom) that provides a good fit for the samples. In this case the curve varies consistently between the sample values. **Figure 9(b)** shows an undesirable fit using a polynomial of degree 12. In this case the polynomial curve intersects the samples, but the curve fluctuates excessively between the samples. This fluctuation is due to the curve having too many turning points (i.e. too many degrees of freedom), which cannot be suitably controlled using the available number of samples.

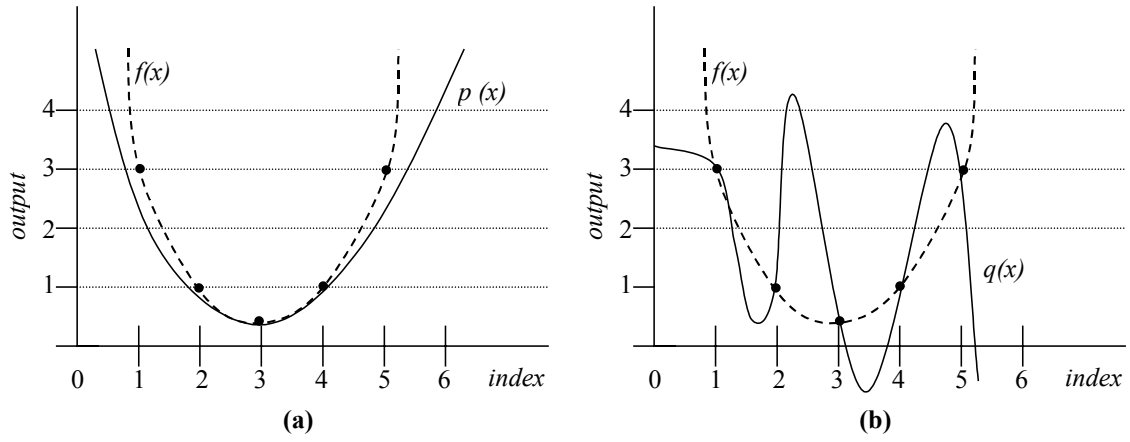


Figure 9: Demonstration of polynomial overfitting.

(a) Desirable polynomial fit, (b) Undesirable / Overfitted polynomial fit.

The weight values for the hidden nodes and output nodes are the only variables that change during training of a standard neural network structure (i.e. the base model). Thus the number of hidden nodes and output nodes used for a certain base model determines exactly how many degrees of freedom are apparent in that model. As in the polynomial case, if a neural network has too many degrees of freedom, then the network will contain some redundant nodes, which can potentially interfere with the way that the network interpolates results between training examples. Conversely, if a network has too few degrees of freedom, then the network become too restricted, being unable to accommodate anything other than the most general patterns present in a training set.

As a means to reduce the problem of overfitting training data, a restriction needs to be imposed on the number of hidden nodes used by a particular neural network model. This implies that some trial and error is required to find a suitable number of hidden nodes for a particular model, when that model is trained with a certain training set. Therefore, the base model and supernet models that are tested in this thesis are limited to a certain number of hidden nodes for each training problem. This in turn limits the degrees of freedom each model has, thus lessen the likelihood of overfitting training data.

3.1.4 Comparing Degrees of Freedom

As mentioned in **Section 3.1.3**, the degrees of freedom (*df*) for the base model is easily calculated from the number of hidden nodes (*Nh*) the model has for a certain problem set, given that the problem set contains *Ni* inputs and *No* outputs. The following formula can be used to calculate this:

$$df(\mathcal{B}) = Nh \cdot (Ni + No)$$

The degrees of freedom for a supernet are *not* necessarily determined just by the total number of nodes that the supernet has in its subnets. The degrees of freedom changes for

different regions of the input space: each cluster may have a different number of nodes used in predicting its local behavior.

The processing modules used to join the subnets may provide additional degrees of freedom for the supernet. Therefore the calculation of the number of degrees of freedom required for a supernet to train a certain problem set is dependent on the number of clusters in the problem set, the number of nodes required in each subnet to learn the behavior of a certain cluster, and the sub-processing modules used in the supernet. The number of inputs and outputs would also affect this.

The exact calculation of the degrees of freedom for a supernet may be rather complex; however the following procedure is a suggested means to approximate this calculation, using the following assumptions:

- Assume that the processing operations are entirely responsible for performing membership selection and training set segregation. Then the subnets are used entirely for learning individual clusters.
- Assume that each subnet has the same number of nodes (or use a mean value, calculated by dividing the total number of hidden nodes in the supernet by the number of subnets).
- Assume that each subnet has the same number of input and output dimensions as is present in the training set.
- Assume that clustering (i.e. the modules) adds $log_2(Nc)$ degrees of freedom to a supernet since a certain number of variables are required to represent the cluster spaces.

In such a case the calculation for the number of degrees of freedom in a supernet is:

$$df(\mathcal{M}) = \log_2(Nc) + Nh \cdot (Ni + No) ,$$

where:

- *Nh* = the total number of hidden nodes in the network,
- *Ni* = the average number of input dimensions, and
- *No* = the average number of output dimensions for each subnet.

Since the number of hidden nodes and the degrees of freedom required by a certain model to learn a particular training set may not be the same for other models learning the same training set, the number of hidden nodes and the degrees of freedom are provided as additional measures when testing the models.

3.1.5 Comparing Models

Supernets are compared by determining how well each model performs on a collection of problem sets. The technique chosen to do this effectively was to develop a so-called *Model Comparison Coordinate* (MCC) for each problem set that expresses a numerical performance rating for a network in relation to the base model. The MCC is calculated based on the individual test's performance on the model using the NMSE error computed from the tests (as described in **Section 3.1.2**).

The evaluation is performed in relation to a common group of problem sets $\boldsymbol{p}$ that represents a particular broad *application domain* for which an efficient supernet model is desired. A particular problem $\boldsymbol{P} \in \boldsymbol{p}$ represents a *specific application* within the broader application domain $\boldsymbol{p}$. An efficient supernet is one that generally works well for all the problems in application domain $\boldsymbol{p}$. A simple heuristic for choosing an efficient supernet for such an application domain could be to select the supernet that has the lowest *average MCC* for problems within the domain. However, there is a potential risk in this approach that outlier MCC values might bias the average. This method is made more reliable by using a large number of problems that represent the same application domain.

The MCC comprises a pair of absolute valued components, MCC_{early} and MCC_{final}, which express train and yield performance for a supernet $\mathcal{M}$ in relation to a base model $\mathcal{B}$. The MCC_{final} component is calculated according to the MVE value which the supernet achieved at the end of training divided by the MVE for the base model. The MCC_{early} component is calculated by dividing the MTE value obtained for the supernet after only a small number of training epochs have been completed by the MTE value for the base model after it has completed the same small number of epochs.

Each network is limited to the *same* number of training epochs for a particular problem $\boldsymbol{P}$. The number of training epochs for problem $\boldsymbol{P}$ is equal to the number of epochs that the base model required to reach a certain MVE value. The number of training epochs used in the MCC_{early} component is taken at 10% of this number.

The MCC for a specific problem set $\boldsymbol{P}$, a Base Model $\mathcal{B}$, and a supernet model $\mathcal{M}$ is given by the coordinate $\boldsymbol{(}\, MCC_{early}(\boldsymbol{P},\mathcal{M})\, \boldsymbol{,}\, MCC_{final}(\boldsymbol{P},\mathcal{M})\, \boldsymbol{)}$ calculated as follows:

$$\left.\begin{aligned} MCC_{final}(\mathcal{M},\boldsymbol{P}) &= \frac{MVE_{final}(\mathcal{M},\boldsymbol{P})}{MVE_{final}(\mathcal{B},\boldsymbol{P})} \\ MCC_{early}(\mathcal{M},\boldsymbol{P}) &= \frac{MTE_{early}(\mathcal{M},\boldsymbol{P})}{MTE_{early}(\mathcal{B},\boldsymbol{P})} \end{aligned}\right\} \; for\ \boldsymbol{P} \in \boldsymbol{p}$$

The average MCC is the mean of all the coordinates for these sample problems representing the application domain $\boldsymbol{p}$:

$$\overline{MCC}(\mathcal{M},\boldsymbol{p}) = \left(\frac{1}{Np}\sum_{i=1}^{Np} MCC_{early}(\mathcal{M},\boldsymbol{P_i}), \frac{1}{Np}\sum_{i=1}^{Np} MCC_{final}(\mathcal{M},\boldsymbol{P_i}) \right),$$
$$where\ \boldsymbol{p} = \{\boldsymbol{P_1}, \boldsymbol{P_2}, ..., \boldsymbol{P}_{Np}\}$$

The intention of the MCC_{early} component is to indicate which model is the quickest to adapt to its training data when given a very limited number of training epochs. The MCC_{final} component shows which model performs best in the general case in which the number of epochs are less restricted.

The plot of the MCC coordinates gives a concise representation of each supernet's performance in comparison to the base model. **Figure 10** illustrates a general MCC plot in which the origin, that represents the base model $\mathcal{B}$, is at position (1,1). The horizontal axis represents the MCC_{early} component while the vertical axis represents the MCC_{final} component. Thus supermodels that show better performance than the base model in terms of both early and final results will be in the bottom left quadrant, while models that show worse performance are in the upper right quadrant. The limits on the axes prevent MCC points from gathering too closely in the center. An MCC coordinate of (0,0) is unlikely to occur for a supernet unless it produces a NMSE of zero when the base model produces a nonzero NMSE for the same data.

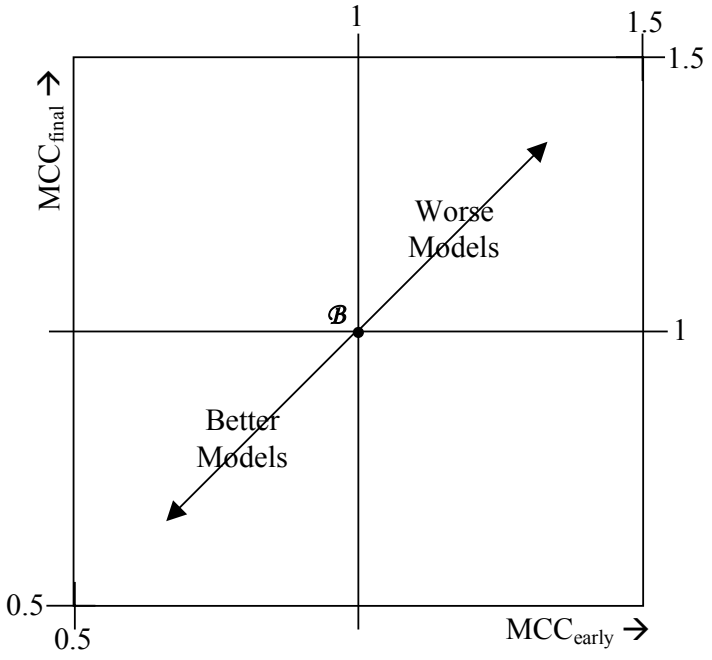


Figure 10: A general MCC Plot.

3.1.6 The 30% Relative Convergence Time

In most situations, a training performance graph exhibits an initial "knee" that asymptotes to a certain minimum training error (see **Figure 11**). The learning rate in the region before the "knee" takes place quickly, where the network is producing an initial rough approximation of the training data. The asymptotic part of the graph that occurs after the "knee" indicates that the network is gradually refining this initial approximation, and shows the training error is converging towards a certain minimum value at a forever-decreasing rate.

Generally, choosing an NMSE slightly below the asymptote shown in the training performance graph provides a fairly close estimate for the final training error in the case where the number of training epochs is not limited (provided there are no further sudden improvements in the training performance – shown by the dotted line in **Figure 11**).

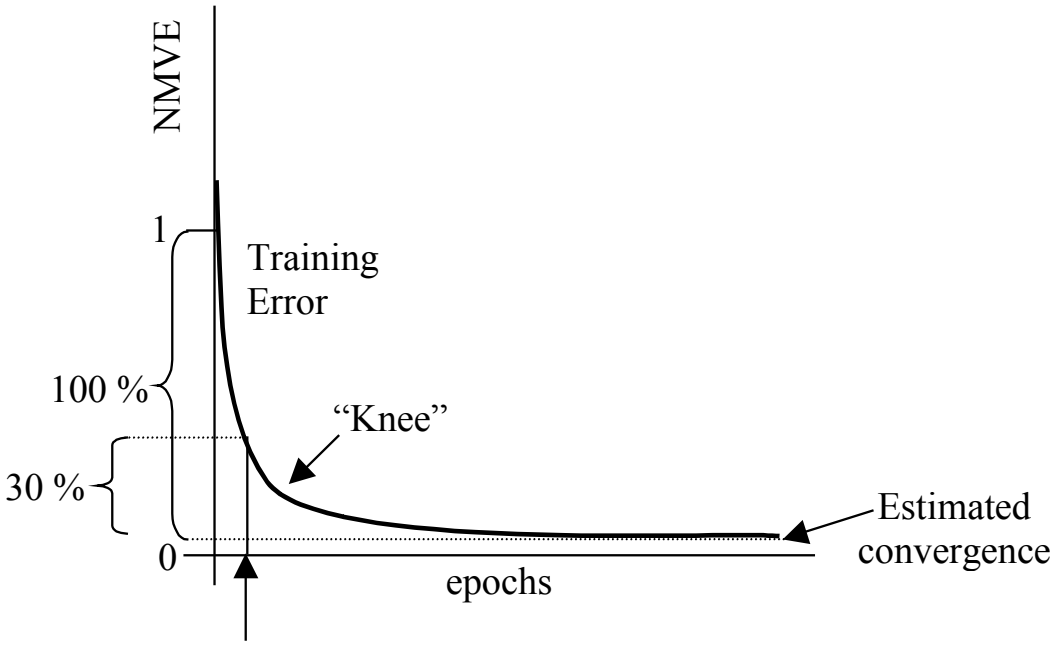


Figure 11: Convergence of training error.

From the estimate for the final training it is possible to determine a convergence time that indicates how quickly a model converges to this final training error. The convergence time can be calculated by determining how many epochs are required to reach a certain percentage of the final mean training error (for instance a value 30% above the final training error, where 100% above the final training error is defined to be a NMSE value of 1). This indicates how quickly the state of a model being trained on a certain problem converges to a final state. For instance, one model may take 20 epochs to reach its 30% point; while another model may need 30 epochs to reach it's 30% point. Both models may converge to different final training error values. The convergence time merely indicates which model takes the least time to get to 30% of its final convergence.

It is useful to compare the convergence time of a supernet to that of the base model to determine an indication of how much faster – or slower – a supernet is trained compared to the speed that the base model is trained for a particular problem set. The Relative Convergence Time (RCT) is an indication determined for a supernet that indicates the relative speed at which the supernet converges compared to the speed the base model converges. The RCT for a supernet is calculated from the following formula:

$$RCT(\mathcal{M}) = \frac{CT(\mathcal{M})}{CT(\mathcal{B})},$$

where CT($\mathcal{M}$) is the convergence time of the supernet and CT($\mathcal{B}$) is the convergence time of the base model.

If a supernet model is shown to have a low RCT, then the model learns its training data more quickly that the base model; conversely a high RCT indicates the model learns it training data slowly than the base model. Note that the rate that a model learns it training data is not necessarily indicative of how well it can generalize.

The RCT value determined for the comparison of supernets is not equivalent to the

MCC_{early} component of the MCC. The MCC_{early} component indicates an early MTE for a supernet, while the RCT value shows a relative time quantity indicating a degree of how much faster or slower a supernet's training occurs in comparison to the base model.

3.2 Problem Sets

In this text a *problem set* refers to a training set and its related validation sets. A collection of problem sets represents a broad application domain. The problem sets chosen for this thesis represent applications whose training data is clustered in one of the following ways:

- Non-overlapping (determinate) clusters
- Overlapping clusters
- Intertwined but non-overlapping clusters

The problem sets used in this thesis are listed in **Table 1** together with information regarding the type of clusters found in the data, the number of clusters in the training set, the input and output dimensions and the number of training examples given.

Most of the problem sets comprise 2D inputs and a 1D output, which makes it easier to visualize outputs as a 3D surface. A higher dimension training set is used to verify operation for higher dimensionality, since all the supernets must be capable of using multiple input and output dimensions.

The 'Scatter4' problem set was developed as a simplistic test to determine how models make use of easily separable clusters. This is used as an initial test to see if the supernet works for the "trivial" case. As **Figure 12*(a)*** shows the 'Scatter4' training set contains a scattering of samples for each of the four clusters, where each example has as desired output the cluster number to which it belongs.

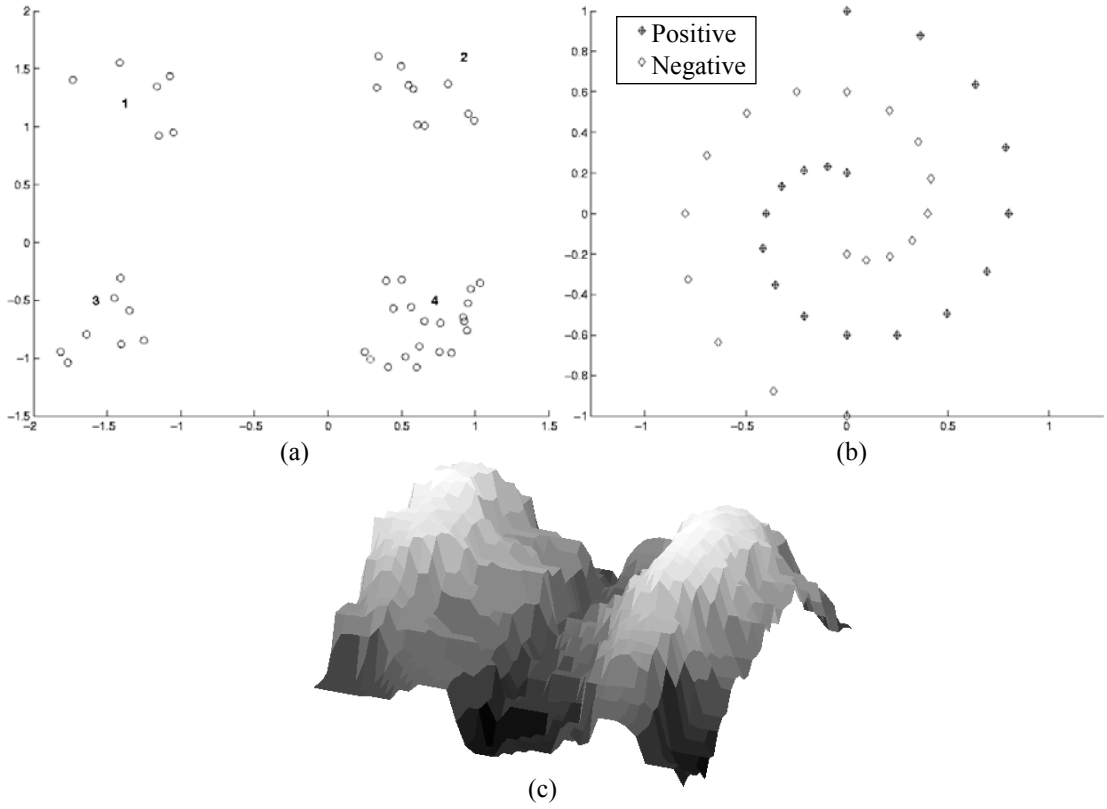


Figure 12: Plot of 2D training sets.

(a) Scatter4, (b) Spiral, (c) Crags2.

Table 1: Problem Set Listing.

Problem Set	Type	Clusters	Inputs	Outputs	Examples
Scatter4	Non-Overlapping	4	2	1	63
Spiral	Intertwined	2	2	1	34
Crags2	Overlapping	2	2	1	240
Wine	Overlapping	3	13	3	128

The 'Spiral' problem, shown in **Figure 12*(b),*** is the classical intertwined spiral problem, where one spiral has positive desired outputs, the other negative. The spiral was chosen because it cannot be solved by a simple backpropagation network, as shown by Baum and Lang (1991: 904-910). The spiral is a form of "intertwined" cluster in that the clusters are separable yet cannot be separated into two hyper spheres as a means to discriminate between them, as most simple clustering algorithms would attempt to do.

The 'Crags2' problem set was contrived from a mesh generated in MatLab. It comprises two "mountains" or "crags". The data points in each crag (or cluster) follow a gradual but noisy decline from some maximum value. This noisy decline is not the same in each crag nor is it equal at equidistant points from the highest point (if it were, it would be a trivial problem for a radial basis network). Each training example consists of a 3D coordinate representing a sampled point on the surface of one of the crags. The input part of each training example is two-dimensional and consists of the x and y components of this coordinate. The output part is the z component that represents the height of the mountain at a given grid point. A sampling of the height values at random xy-positions on the grid was used to develop the training and validation sets. **Figure 12*(c)*** shows a rendering of the original mesh that represents the imaginary mountain range.

The "Wine" problem set is multidimensional and originates form the "Wine recognition data" training set assembled from real data by Blake (1998) for the UCI machine learning database. The training data is used to predict from which winery a particular wine

originates using the chemical composition of the wine. There are three possible choices of winery. This sample was originally available only as a single sample set, so it was split into two parts to form separate training and validation sets.

A view of the wine training set mapped onto the 2D plane is given in **Figure 13**. The dataset was applied to the distance glance algorithm (see **Section 4.1.2**) in order to determine a suitable number of clusters to partition the data into, and to approximate the location of these clusters. The clusters are marked C1 to C3 in the figure and the estimated size of the clusters, according to their density, are indicated by the circles.

A three-dimensional view of the wine training set is give in **Figure 14**, and helps to illustrate the degree of overlap between the clusters. This problem is notably different from the Crags2 problem: For the Crags2 problem there is a small amount of overlap and the mountains behave similarly, but the clusters in the wine problem overlap to a higher degree and each cluster has different local behavior.

3.3 Optimizing Development and Evaluation Time

The following points were found to take the most amount of time in the development and evaluation for this thesis:

- Finding the correct training parameters for backpropagation
- Execution time to train the network
- Time taken to implement the model prototypes

The major problem with the standard backpropagation algorithm is the time it takes to train and the time spent finding appropriate training parameters. As this thesis involves comparing many models with a number of training sets, it was essential to find a strategy that could reduce the amount of human effort required to training the models without making any changes to the learning algorithm itself.

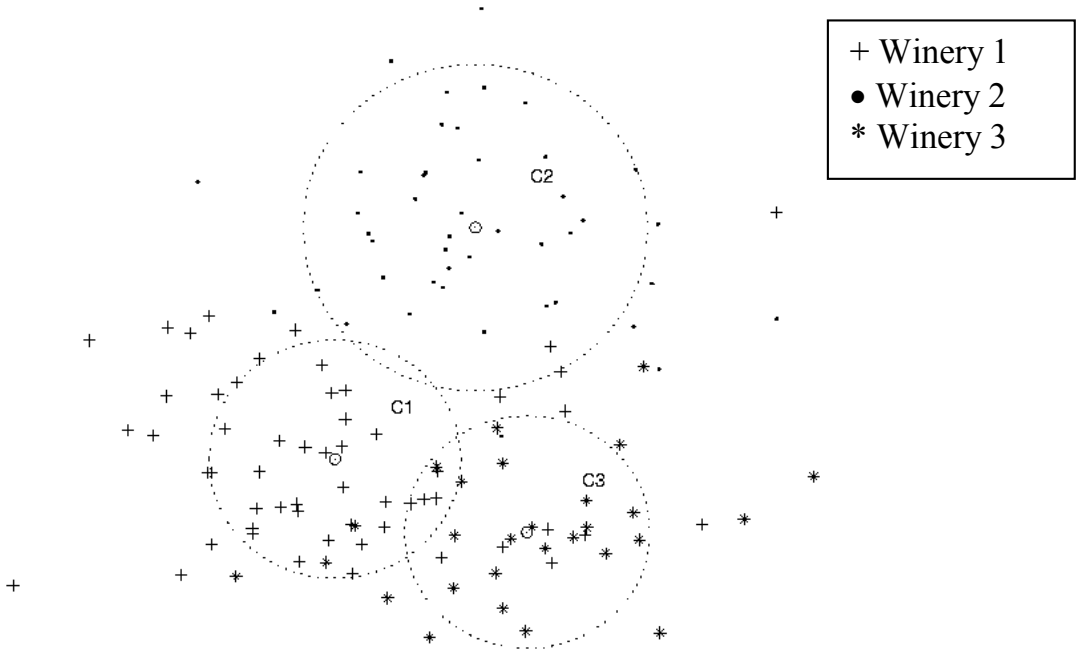


Figure 13: Plot of the "Wine" training set showing clusters according to DG algorithm.

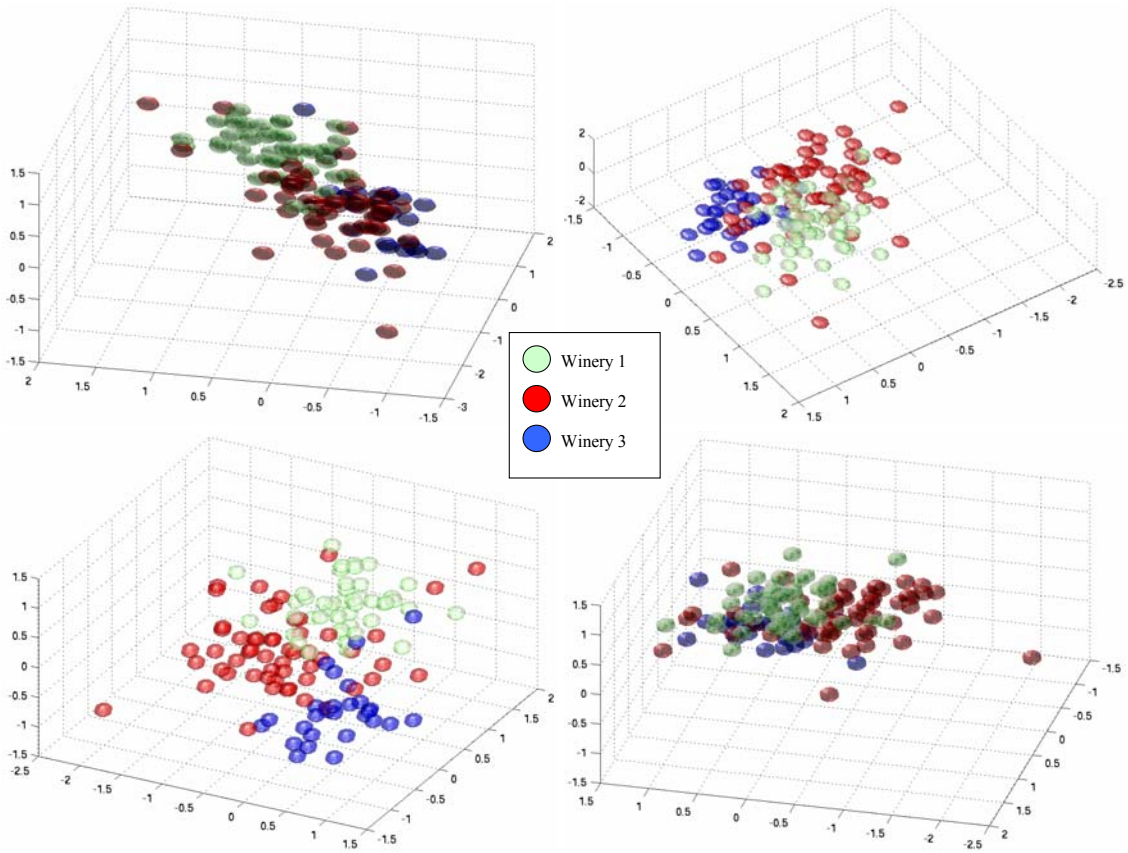


Figure 14: Four rotated views of a 3D representation of the wine training set.

3.3.1 Common Strategies

The first attempt to lessen the workload on the human operator was the implementation of commonly used strategies that accelerate the backpropagation algorithm without making changes to the implementation of the algorithm itself.

Joost and Schiffmann (1998) showed various strategies to optimize the traditional backpropagation formulation. This paper also pointed out another, more obvious fact, that training is further improved, without data loss, when the inputs are translated by the mean and scaled by the standard deviation. Therefore all the training and validation files used in this thesis are normalized to have a mean of zero and standard deviation of 1. The sigmoid functions were also suitably scaled to reach all the desired outputs in the normalized training data.

The following formula was used to perform the normalization of problem sets used in this thesis and was applied to each column of input and output values independently:

$$x_{i,j} \in X$$

$$\bar{x}_i = \frac{1}{N}\sum_{j=1}^{N} x_{i,j}$$

$$\sigma_i = \sqrt{\frac{1}{N-1}\sum (x_{i,j} - \bar{x}_i)^2}$$

$$x_{i,j}^{norm} = \frac{(x_{i,j} - xi)}{\sigma_i}$$

Note that the validation and training sets should be joined for this operation and then re-separated to ensure accurate normalization for the data.

Another useful technique commonly used to improve the training accuracy (not so much the training speed) is to gradually reduce the learning rates for both the hidden and output

layers as training processes. This causes the initial rate of training to be fast but inaccurate, and then to become progressively slower causing increasingly more accurate predictions. This helps to reduce the number of permutations required in selecting optimal learning rates.

3.3.2 Applying Computer Power: EP Scripts

Since the techniques in the previous chapter did not make it significantly easier for the human to select suitable training parameters, an attempt was made to delegate the bulk of the trial and error process used in parameter selection to a cluster of computers.

An "Embarrassingly Parallel" (EP) strategy was use to split-up the training workload. A script was developed that, given *n* processor nodes, *n* copies of the program would be executed simultaneously, a copy of the program executing on each processor node. Each node would be instructed to test a certain range of training parameters and save the training performance statistics together with the corresponding parameter combination to a log file.

The parameters specified in the EP scripts were dependent on the particular type of network that was being tested. The human operator was still required to review each of the log files to determine which combination of parameters were optimal (this procedure could also have been automated, but the means of selecting suitable parameters was already suitably improved upon)

3.3.3 Reusable Modules

The EbTide v2 Neural Net package (discussed in **Section 2.5**) provides a good interface design that was used as a starting point for the implementation design for the supernets. The EbTide method of saving and restoring states of the network was applied to the

supernet construct. Additional data structures were designed for handling dynamic matrices and object lists to provide the means of adding new subnets and other processing constructs on the fly.

As it was apparent that various classes of sub-processing modules would be reused, it was necessary to develop an efficient method of connecting these structures with the model code used to implement the high-level operation of a particular supernet. Since each processing class contains its own state information it was necessary to provide a means to serialize this state data without having to have the model code do this explicitly. Therefore specialized callback handlers (see terminology in **Appendix D**) were added to the EbTide library that call these modules so that they can perform serialization of arbitrary data structures and preprocessing tasks without having to code this explicitly in the model code.

A simplified design of the system is shown in **Figure 15**. Since the software was developed in C++ its properties of inheritance and polymorphism could be used effectively. Inheritance was used to facilitate streaming of state data to and from disk. Polymorphism was used to quickly substitute different classes of the same superclass to test the effect of alternate module configurations.

Using the techniques discussed in this chapter to improve the speed of backpropagation, to reduce the human workload of trial and error parameter selection, and to have an efficient means to develop supernets, it become quite possible to test and evaluate many different supernet configurations in a fairly short time.

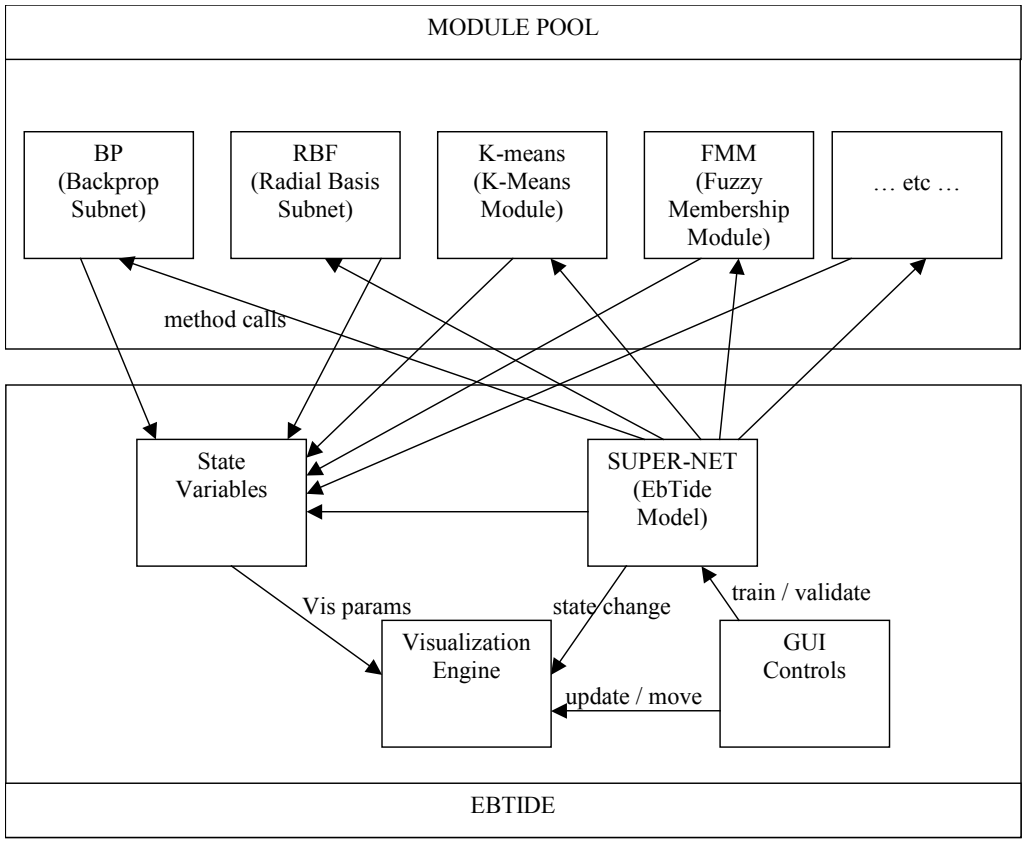


Figure 15: System Design Overview.

4 The Modules

Many of the experimental neural network models presented in this thesis are built from a selection of processing modules. The operation of each sub-process module is defined in this chapter together with a representative glyph. A glyph in the context of this work is a graphical representation of the model in the form of an icon with input and output links. These glyphs can be connected up to perform higher-level processing. They are used in **Chapter 5** to construct Concise Description Diagrams (CDDs) that provide a visual means to show the composition of a supernet. A CDD is divided into two parts, showing the configuration of the network in train mode on the left and the configuration for yield mode on the right. **Appendix A** provides a summarized listing of the glyphs and other symbols used in these diagrams.

4.1 Clustering Modules

Clustering modules are responsible for performing cluster processing for a supernet. A supernet is not necessarily limited to a single cluster module, but if only one type of clustering is used, as is the case with all the supernets presented here, then only one clustering module is used.

The glyph used to represent a clustering module (shown on the left of **Figure 16**) is represented as a box containing three circles meant to represent clusters. The number of clusters that the algorithm uses is shown in parentheses.

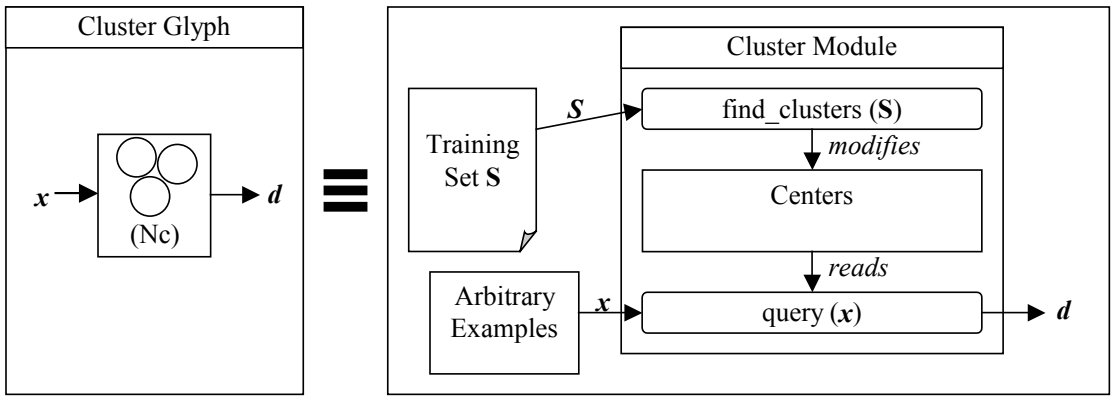


Figure 16: Cluster Module Design.

Clustering models provide two methods (illustrated on the right of **Figure 16**). The first function ("find_clusters") performs the act of funding clusters in a given training set ***S*** containing *Nx* vectors of dimension *Ni*. This function updates a matrix of size *Nc* rows by *Ni* columns, where each column ($\boldsymbol{c_j}$) represents a center for the j^{th} cluster.

The second function ("query") determines Euclidean distances between a given data point and the cluster clusters. The *query* takes as input a vector ***x*** of dimension *Ni* and returns a vector ***d*** of dimension *Nc*, where each component d_i of ***d*** represents a distance value between ***x*** and center $\boldsymbol{c_i}$. These distance values are then passed to an appropriate membership module (MM) to decide which cluster or clusters the data point should belong to.

The "find_clusters" function is applied to the training set before the training of subnets commences, this accounts for the missing input ***S*** to this module's glyph. For this thesis the clustering process was chosen to be done as a simple preprocess, although it could be done during training instead as Zeng and Liu (2002) show in their paper that discusses an online clustering paradigm.

The "query" function is generally used during training in conjunction with the fuzzy membership modules, and is also occasionally used in yield operations.

The following cluster algorithms were implemented and used as preprocesses in the supernet models:

- *K-means*: Requiring the number of clusters to be known prior to execution
- *Distance Glance (DG)*: Heuristic method that iterates a small number of times finding cluster centers according to minimal distance between points.

4.1.1 K-Means Clustering Module

The k-means algorithm is probably the most commonly used clustering algorithm (Theodoridis & Koutroumbas, 1999: 482). The basic version of the algorithm is computationally inexpensive and simple to implement. The k-means algorithm is essentially a minimization of a performance index which is defined as the sum of all squared Euclidean distances between points in a cluster domain and the cluster center (Tou & Gonzalez, 1974: 94-96).

The basic k-means algorithm needs to know the number of cluster centers (***Nc***) to use, thus this forms a parameter to supernet models that use this clustering module. The Isodata algorithm would be an alternate method that would not require the extra ***Nc*** parameter, but it would need other parameters that tells it what kind of clusters to expect. In the Isodata algorithm is basically the k-means algorithm with added heuristics (Tou & Gonzalez, 1974: 97). Since it was fairly obvious how many clusters were in the data sets chosen for this evaluation, it was not deemed necessary to implement the Isodata algorithm.

"find_clusters": Basic K-Means Clustering Algorithm

1. Choose *Nc* Initial cluster centers $\boldsymbol{c_1(1)}$, $\boldsymbol{c_2(1)}$ to $\boldsymbol{c_{Nc}(1)}$. These centers can be selected arbitrarily, but two methods were implemented for this thesis: either

selection of the first *Nc* points in the training set, or distributing the centers uniformly through the input space.

2. At the nth iteration distribute the inputs {**x**} amongst the *Nc* cluster domains using the relation:

$$\boldsymbol{x} \in S_j(n) \text{ if } \left\|\boldsymbol{x} - \boldsymbol{c}_j(n)\right\|^2 < \left\|\boldsymbol{x} - \boldsymbol{c}_i(n)\right\|^2$$
$$\text{for all } i = 1,2,...,Nc, i \neq j$$

Where $\boldsymbol{S_j(n)}$ represents the set of input vectors {**x**} that are within the j^{th} cluster with center $\boldsymbol{c_j(n)}$**.** Any ties are resolved arbitrarily.

3. From step 2, new cluster centers $\boldsymbol{c_j(n+1)}$, *j=1..Nc* are computed such that the sum of the squared distances from all points in $\boldsymbol{S_j(n)}$ to the new cluster center is minimized. The sample mean of $\boldsymbol{S_j(n)}$ achieves this:

$$\boldsymbol{c}_j(n+1) = \frac{1}{N_j} \sum_{\boldsymbol{x} \in S_j(n)} \boldsymbol{x}, \quad j = 1 \ldots Nc$$

4. If $\boldsymbol{c_j(k+1) = c_j(k)}$ for *j=1...Nc* then the algorithm has converged and the process is complete.

"Query": Distance Function

During training and yield operations, the k-means clustering module is requested to calculate the distances between an input vector and the cluster centers. The square of the Euclidean distance was used so as to avoid computationally expensive square root operations. The formula for the distance values returned by the query function is:

$$d_j = \sum_{i=1}^{Ni} (x_i - c_i)^2 \quad for \ j = 1...Nc$$

4.1.2 Distance Glance (DG)

The Distance Glance (DG) algorithm was developed for this thesis as an attempt to extract approximate cluster information from a data set relying only on the minimum density of clusters and expected distances between clusters. The method works well for data sets that comprise clusters of similar density and whose clusters are at fairly uniform distance from one another. The algorithm is able to handle low-density noise between clusters. This algorithm is an adaptation of the commonly known "Maximin" algorithm (Tou & Gonzalez, 1974: 92). This algorithm was named "Distance Glance" because it "glances" at a distance matrix in each iteration of the algorithm.

The DG algorithm requires three parameters:

- *min_density* : minimum density (num points) of a cluster
- *lumping factor* : approximate distance between points in a cluster
- *cluster_distance_ratio* : mean distance from old clusters to a new cluster divided by the mean distance between all old clusters.

The DG method starts in the same way as the Maximin algorithm by first finding the most distant point ($\boldsymbol{c_1}$) from the sample mean of a particular sample set $\boldsymbol{S}$. The point $\boldsymbol{c_1}$ is then selected as the first *cluster prototype* (where the term "cluster prototype" is a point representing a cluster location and is *not necessarily* the center of the cluster). Then squared distances from $\boldsymbol{c_1}$ to all other points are computed and stored in the first column of a matrix $\boldsymbol{D}$. Next, the *lumping process* is applied to this matrix as follows: rows in column 1 of $\boldsymbol{D}$ that have values greater than the *lumping factor* are kept and all the other rows are removed from $\boldsymbol{D}$ and placed into a temporary matrix $\boldsymbol{D_{dis}}$. The set of sample in $\boldsymbol{S}$

that correspond to distance values in $\boldsymbol{D_{dis}}$ are removed from $\boldsymbol{S}$ and placed in a temporary matrix $\boldsymbol{S_{dis}}$. The number of samples discarded (i.e. the number of rows in $\boldsymbol{D_{dis}}$) is stored in d_1 as the density estimate for cluster 1. The samples comprising $\boldsymbol{S_{dis}}$ are aggregated into an estimated cluster center $\boldsymbol{ce_1}$, and the matrix $\boldsymbol{S_{dis}}$ is deleted. The value r_1 is set to the mean of the distance values in $\boldsymbol{D_{dis}}$, and the matrix $\boldsymbol{D_{dis}}$ is deleted.

The index of the largest value in column 1 of $\boldsymbol{D}$ is found and the point corresponding to this distance is assigned to the second cluster prototype ($\boldsymbol{c_2}$). Then a second column is added to $\boldsymbol{D}$ that stores squared distances from $\boldsymbol{c_2}$ to the all the remaining points in sample set $\boldsymbol{S}$. The lumping process is then applied to column 2 of $\boldsymbol{D}$ removing more rows from matrices $\boldsymbol{D}$ and $\boldsymbol{S}$.

The next cluster prototype ($\boldsymbol{c_{n+1}}$) is found by looking at the matrix $\boldsymbol{D}$ and finding the maximum of the minimum squared distances between the remaining points and cluster prototypes ($\boldsymbol{c_1}$ to $\boldsymbol{c_n}$). Then another column is added to $\boldsymbol{D}$ and the lumping process is applied. This procedure is repeated until the new cluster prototype ($\boldsymbol{c_{n+1}}$) is too close to the other clusters (as determined by the *cluster_distance_ratio* parameter described below). When this happens this final prototype $\boldsymbol{c_{n+1}}$ is discarded, another column should not be added to $\boldsymbol{D}$ and neither should the lumping process be applied. At this point the second part of the algorithm commences.

The second part of the algorithm works on only the cluster centers ($\boldsymbol{ce_1}$ to $\boldsymbol{ce_n}$) and the density measures ($\boldsymbol{d_1}$ to $\boldsymbol{d_n}$). The first step involves discarding any density values that are below the *min_density* parameter (usually a value between 1 and 3) along with their corresponding centers and radii. Then the mean of the remaining densities are determined and any density values below 50% of this mean value are discarded together with the corresponding centers and radii. The remaining centers are then returned as the final result of the algorithm. Supernets that need an estimated size and population for each cluster can use the radius and density values.

Appendix C1 provides a complete listing for the DG algorithm in MatLab with explanation provided for each step.

The lumping process has the following advantages:

1. It helps to discard noise scattered at a density below that of the cluster density.
2. It helps to construct an approximate center for the cluster.
3. It reduces the amount of computation by decreasing the samples size per iteration.

Choosing the Parameters

A few heuristic methods were found in the course of testing the DG algorithm with different training sets. These heuristics were found to work suitably to determine the *lumping factor* and *min_density* parameters for the algorithm. Note that these heuristics have not been tested exhaustively, but the overall result of the algorithm was found to be fairly close to the results of the standard k-means algorithm when the number and initial location of centers could be determine roughly by visualizing the data (note that an automatic means to determine the *cluster_distance_ratio* was not found).

Lumping Factor Selection:

One way to choose a *lumping factor* is to select n sample points $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\}$ at random from the input samples $\boldsymbol{S}$. These points can be stored in the rows of a matrix $\boldsymbol{X}$. Then select the m closest neighbors to each point $\boldsymbol{x}_i$ and store them in the rows of matrix $\boldsymbol{Y}_i$. Then calculate the mean distance to neighbors for these n points (i.e. a mean of $m.n$ distance values) and set the *lumping factor* to about twice this value:

$$\boldsymbol{X} = \{\boldsymbol{x_1}, \boldsymbol{x_2}, ..., \boldsymbol{x_n}\} \mid \boldsymbol{x_i} \in \boldsymbol{S} \text{ for } i = 1...n$$

$$\boldsymbol{Y_i} = \{\boldsymbol{y_{i1}}, \boldsymbol{y_{i2}}, ... \boldsymbol{y_{im}}\} \mid \left\|\boldsymbol{y_{i1}} - \boldsymbol{x_i}\right\| \le \left\|\boldsymbol{y_{i2}} - \boldsymbol{x_i}\right\| \le ... \le \left\|\boldsymbol{y_{im}} - \boldsymbol{x_i}\right\| \text{ for } i = 1...n$$

$$\overline{Y_i} = \frac{1}{m}\sum_{j=1}^{m}\left\|\boldsymbol{y_{ij}} - \boldsymbol{x_i}\right\| \text{ for } i = 1...n$$

$$lumping = \frac{2}{n}\sum_{i}^{n}\overline{Y_i}$$

Min_Density Selection:

For *min_density* select a new set of *n* points $\{\boldsymbol{x_1}, \boldsymbol{x_2}, ..., \boldsymbol{x_n}\}$ at random (i.e. do not reuse the points chosen for the lumping factor). Then using the lumping factor computed above, calculate the average number of points that are within this distance of these *n* points as follows:

$$\boldsymbol{X} = \{\boldsymbol{x_1}, \boldsymbol{x_2}, ..., \boldsymbol{x_n}\} \mid \boldsymbol{x_i} \in \boldsymbol{S} \text{ for } i = 1...n$$

$$\boldsymbol{Y_i} = \{\boldsymbol{y_{ij}}\} \mid \left\|\boldsymbol{y_{ij}} - \boldsymbol{x_i}\right\| \le lumping \text{ for } i = 1...n, \forall j$$

$$min\_density = \frac{1}{n}\sum_{i=1}^{n}\left|\boldsymbol{Y_i}\right| \text{ , where } \left|\boldsymbol{Y_i}\right| = \text{number of rows in } \boldsymbol{Y_i}$$

Cluster_Distance_Ratio Selection:

The *cluster_distance_ratio* is more difficult to determine. Due to time constraints, this research could not be sidetracked to solve this problem. Therefore the only recommendation for this parameter is to use trial and error, keeping in mind the following rules:

1. If the *cluster_distance_ratio* is too big there will be dense areas that are not assigned to clusters.
2. If the ration it is too small clusters will overlap.
3. For some datasets a compromise in data points being unassigned to clusters,

and some data points being assigned to multiple clusters (i.e. cluster overlap) may be the required.

Some Optimizations

When applying the DG algorithm to a large dataset one of the critical issues is the size of the distance matrix ***D***. The size of ***D*** is *Nx* rows by *Nc* columns, so if there are many small clusters in the input set, this could pose a problem. The lumping process helps to lessen this effect by reducing the number of points that need to be compared for future iterations.

The placement of the cluster centers according to the DG algorithm can be applied to the basic k-means algorithm for further improvement. This is generally the case followed during the test scenarios in this thesis: training sets that did not have clusters easily determinable by eye were passed through the DG function.

Example Run

The DG algorithm was tested on a selection of data sets, both 2D and of higher dimension. This example shows the results from a particular data set that contains 6 clusters where one cluster is noticeably distant from the others. Random points were added uniformly to the data. This file was run through the DG algorithm and the results plotted in MatLab. The algorithm found the six clusters as desired (**Figure 17**) by finding points that were most distant from other points, but which also had many neighboring points. The estimated cluster centers are labeled B1 though B6 and the circles represent the cluster radii.

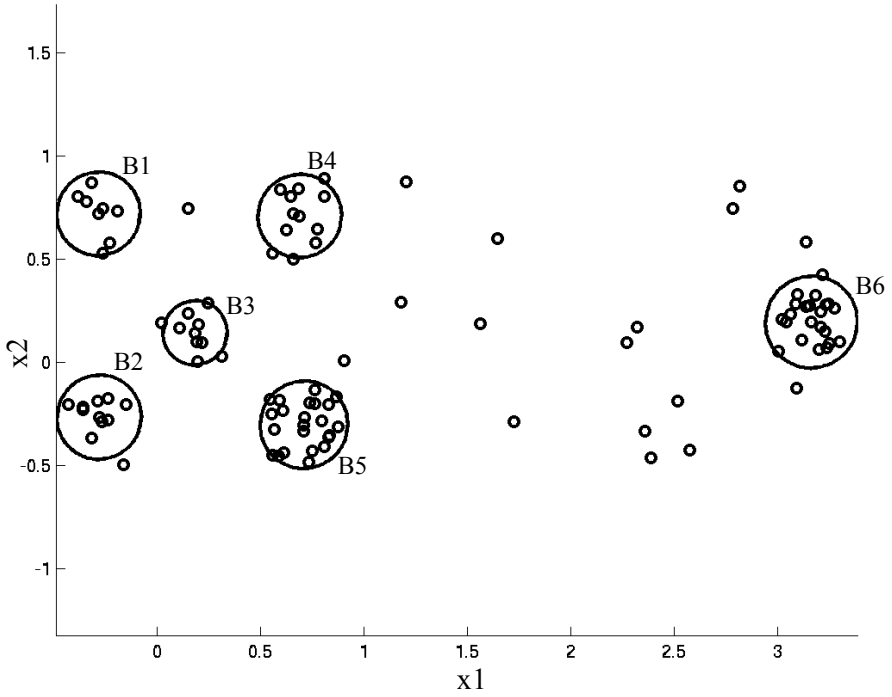


Figure 17: Result of DG algorithm applied to 6-cluster problem.

4.2 Membership Modules

Three types of membership module (MM) were implemented: 1) Determinate, 2) Fuzzy, and 3) Hybrid. Each type is implemented as a subclass of the MM super class.

A MM accepts a vector ***d*** containing n distance values where each distance value d_i represents a distance from a certain point ***x*** to cluster i. Each module returns an n dimensional membership vector ***m*** where each m_i represents the possibility (a value between 0 and 1 inclusive) of the point ***x*** being in cluster i. On the left of **Figure 18** the glyph used to represent the MM is shown (note that it is labeled according to the type of membership function implemented), on the right the top-level class structure of the MM is given.

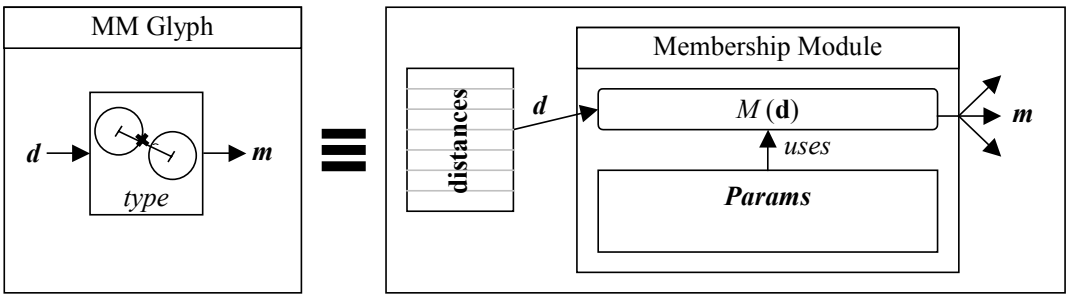


Figure 18: Fuzzy Membership Module Design.

The MM has one main interface function named *M* that implements a certain membership function for the subclass. The function *M* can access certain state parameters such as the size of the clusters.

4.2.1 Determinate Membership Module (DMM)

For a determinate membership function (M_{det}), each input vector belongs exclusively to a single cluster. The *M* function for this type of MM returns a binary output vector ***m*** where the element m_i corresponding to the maximum d_i value is set to true (**1**) and all the other elements of ***m*** are set to false (**0**) as follows:

$$m_i = M_{\det}(d_i) = \begin{cases} 1 & if\ d_i < d_j\ \forall\ j = 1 \ldots Nc, j \neq i \\ 0 & \text{otherwise} \end{cases}$$

$$where\ d_i, d_j \in \boldsymbol{d} \rightarrow d_i = \left\| \boldsymbol{x} - \boldsymbol{c_i} \right\|^2, d_j = \left\| \boldsymbol{x} - \boldsymbol{c_j} \right\|^2$$

4.2.2 Fuzzy Membership Module (FMM)

In the truly fuzzy membership function (M_{fuz}), an input vector can belong simultaneously to multiple clusters. The query function for such a module has an output vector ***m*** of

dimension *Nc* (i.e. equal to the number of clusters) with each m_i element representing a fuzzy possibility in the [0,1] range. The function $M_{fuz}(\boldsymbol{x},\boldsymbol{C})$ introduced in **Section 2.4** is revised to deal arbitrarily many clusters and to operate on distance values d_i that represent squared Euclidean distances between and input $\boldsymbol{x}$ and the cluster *i*, with limitations as follows:

$$m_i = M_{fuz}(d_i),\ where:$$
$$M_{fuz}(d_i) > M_{fuz}(d_j) \text{ if } \boldsymbol{x} \text{ belongs more in cluster } i \text{ than in cluster } j;$$
$$\text{restructed to:}$$
$$\sum_{i=1}^{Nc} M_{fuz}(d_i) = 1$$

The fuzzy membership function implemented for this thesis provides a nonlinear mapping from distance values to fuzzy possibility values using the gaussian function:

$$y_i = \exp\left(\frac{-d_i^2}{2r_i^2}\right)$$

$$m_i = \frac{y_i}{\sum_{j=1}^{Nc} y_j}$$

where r_i is a parameter for cluster *i* specifying its effective range of influence.

4.2.3 Hybrid Membership Module (HMM)

The hybrid membership function used in this thesis is a form of fuzzy membership that exhibits discontinuities in the membership function. This hybrid method assumes the clusters are hyper spheres, and each cluster *i* has a certain radius r_i. The algorithm assigns fuzzy possibility values m_i for a cluster *i* as follows:

- For input points falling within the hyper sphere of cluster *i* only then $m_i=1$.

- For input points not falling within cluster *i* or any other cluster the value of m_i is assigned to the output of a gaussian.
- For input points falling within multiple clusters *excluding* cluster *i*, $m_i = 0$
- For input points falling within multiple clusters *including* cluster *i*, m_i is passed through a gaussian.

The type of membership function chosen for a supernet can significantly affect its behavior. A membership function surface (MFS) is a useful technique that can be used to predict to what extent a certain clustering technique affects the yielded results of a supernet. This can help to determine in which way membership modules affect results of supernet prediction.

An MFS is produced for a single cluster *i* selected from the available set of clusters found in a problem set. The MFS for cluster *i* is generated by producing a hyper volume of input points and passing these points through the membership function ***M(x,i)*** used by the subnet. This hyper volume is mapped into the xy-plane and the output of the membership function is used as a height above this plane. The surface will thus be a height between 0 and 1 from the xy-plane.

Each class of membership function has its own inherent form for of MFS. For instance, the determinate case produces a step of height 1 in the area of the grid within the cluster, while hybrid clustering has a plateau of height 1 within the cluster area that gradually ramps down to 0 outside the cluster space, with occasional undulations where clusters overlap (See **Chapter 5** for examples).

4.3 Subnet Modules

There are two types of subnet module developed for supernets given in this thesis: Standard backpropagation subnets and radial basis subnets.

4.3.1 Standard Backpropagation Subnet

The standard backpropagation method using incremental training, as described in **Section 2.2** is packaged into the subnet module that is used by all the models evaluated in this thesis. The glyph for this module is represented by a sigmoid-like curve with a circle in the center. Subnet glyphs are labeled with a number if more than one is used in a particular CDD (as indicated by the "*i*" in the glyph shown on the right of **Figure 19**). Since a CDD has two parts that separate the train and yield modes of the subnet, it is necessary to have a means to indicate in which mode a subnet operates for a particular mode in which its supernet operates. This was achieved by displaying a tilde and z^{des} input for those subnets operating in train mode, while for subnets operating in yield mode the tilde and z^{des} input is removed (shown as a dotted lines in **Figure 19**).

The subnet module has the following methods: *update*, *place_weights*, *feed* and *train*. The *update* and *place_weights* methods replace the traditional initialization method as training parameters and weight operation change on the fly. The update method is used to apply a change to the training parameters, such as when a change is made to the subnets learning rate or number of weights in its hidden layer.

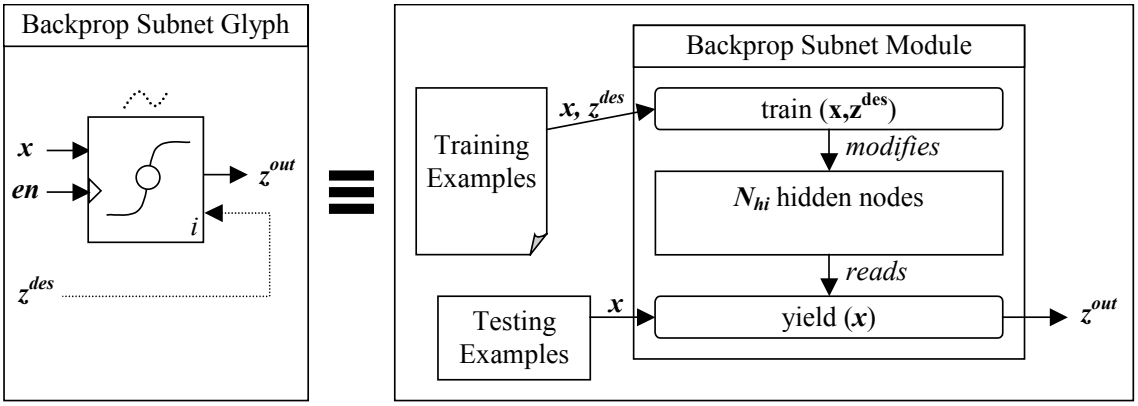


Figure 19: Design of the Backpropagation Subnet.

The *place_weights* method is used for initializing new weights according to a specific weight placement algorithm (pseudo random weights were used in this thesis). The *feed* method is used to yield a result for the network by passing an input through the network, while the *train* method is used for updating the weights according to a desired output.

Two transfer functions are used in the feed algorithm, one for nodes in the hidden layer ($\boldsymbol{f_{hidden}}$) and the other for nodes in the output layer ($\boldsymbol{f_{out}}$). The train algorithm makes use of the derivative function for these transfer functions.

The feed algorithm is given two input vectors: $\boldsymbol{x}$ representing an input example, and $\boldsymbol{z^{des}}$ that represents the desired output produced by the network. The output of the network is stored in a vector $\boldsymbol{z^{out}}$, and generates an error vector $\boldsymbol{z^{erorr}}$ that stores differences between $\boldsymbol{z^{out}}$ and $\boldsymbol{z^{des}}$ values. The intermediate vectors $\boldsymbol{y^{out}}$ and $\boldsymbol{y^{in}}$ are results produced from the hidden layer and are used by the train algorithm. The $\boldsymbol{z^{in}}$ vector represents combined input weights to nodes in the output layer and is used during training. Bias values of 1 are assigned to the 0^{th} entries for $\boldsymbol{x}$ and $\boldsymbol{y^{out}}$, this is used for shifting the hyper planes so as to fit the data better (Fausett, 1994: 21). The weight matrices $\boldsymbol{V}$ and $\boldsymbol{W}$ whose elements are referenced by v_{jk} and w_{ij} represent the weights for the hidden layer (links between input k and hidden node j) and output layer (links between hidden node j and output node i) respectively, where the size of matrix $\boldsymbol{V}$ is Nh by $Ni+1$ elements and the size of $\boldsymbol{W}$ is $\boldsymbol{No}$ by $Nh+1$ elements.

The subnet module has a binary enable line '*en*', that causes the subnet to be enabled if *en* is true (1) otherwise it is disabled. During yield operations, if the subnet is enabled the input is fed normally though the network and yields a prediction. But if it the network is disabled no processing is performed and the network output a value of zero for all elements of its output vector. Likewise, training the subnet only occurs if the enable line is set to true.

Incremental Backpropagation Algorithm

Feed: Yields a result $\boldsymbol{z}^{out}$ from an input vector $\boldsymbol{x}$ for the network

$$\text{bias values:}$$

$$x_0 = 1,\ y_0^{out} = 1$$

$$\text{apply to hidden layer:}$$

$$\left.\begin{aligned} y_j^{in} &= \sum_{k=0}^{Ni} v_{jk} \cdot x_k \\ y_j^{out} &= f_{hidden}(y_j^{in}) \end{aligned}\right\} for\ j = 1 \ldots Nh$$

$$\text{apply to output layer:}$$

$$\left.\begin{aligned} z_i^{in} &= \sum_{j=0}^{Nh} w_{ij} \cdot y_j^{out} \\ z_i^{out} &= f_{out}(z_i^{in}) \\ z_i^{error} &= z_i^{out} - z_i^{des} \end{aligned}\right\} for\ i = 1 \ldots No$$

Train: Adjusts the weights according to training example $\boldsymbol{x}$ and desired output $\boldsymbol{z}^{des}$.

$$call\ feed(\boldsymbol{x}, \boldsymbol{z}^{des})$$

$$\text{propagate error signal:}$$

$$z_i^{\varepsilon} = 2 \cdot f_{out}(z_i^{in}) \cdot z_i^{error} \quad for\ i = 1 \ldots No$$

$$\left.\begin{aligned} ces_j &= \sum_{i=1}^{No} w_{ij} \cdot z_i^{\varepsilon} \\ y_j^{\varepsilon} &= f_{hidden}(y_j^{in}) \cdot ces_j \end{aligned}\right\} for\ j = 1 \ldots Nh$$

$$\text{update weights:}$$

$$\left.\begin{aligned} \mathrm{dw_{ij}} &= -\eta_{\mathrm{out}} \cdot \mathrm{z_i^{\varepsilon}} \cdot \mathrm{y_j^{out}} \\ w_{ij} &= w_{ij} + dw_{ij} \end{aligned}\right\} for\ i = 1 \ldots No,\ j = 0 \ldots Nh$$

$$\left.\begin{aligned} \mathrm{dv_{jk}} &= -\eta_{\mathrm{hidden}} \cdot \mathrm{y_j^{\varepsilon}} \cdot \mathrm{x_k} \\ v_{jk} &= v_{jk} + dv_{jk} \end{aligned}\right\} for\ j = 1 \ldots Nh, k = 0 \ldots Ni$$

In the above algorithm η_{out} and η_{out} are the learning rates for the output layer and hidden layer respectively. The ces_j value is the combined error signal for the output of the *j'th* hidden node determined from back propagating the $\boldsymbol{z}^{error}$ value through each of the links connecting the *j'th* hidden node to the output nodes. The $\boldsymbol{y}^{\varepsilon}$ and $\boldsymbol{z}^{\varepsilon}$ values represent error signals that are back propagated for the hidden and output layers. The dv_{jk} and dw_{ij} values represent delta values for the amount by which a v_{jk} or w_{ij} weight should be increased (if a delta value is negative its corresponding weight is decreased).

4.3.2 Radial Basis Subnet

A radial basis neural subnet was developed for use in separation of input examples and unification of subnet outputs. The glyph for this subnet is shown on the left of **Figure 20**. As for the backpropagation subnet, the RBF subnet take as input a vector $\boldsymbol{x}$ and an optional 'en' line. The subnet is enabled only if the 'en' line is nonzero.

As described in **Section 2.3** the a standard RBF network determines its output by converting the input $\boldsymbol{x}$ into a list of distances between $\boldsymbol{x}$ and each centroid, then it passes these distance values through gaussians and returns as the final output a weighted sum of all the results produced by the gaussians. However, the RBF subnet designed for this thesis is designed for cluster prediction and was therefore modified in the following ways:

- The RBF subnet was limited to one centroid per output. Each centroid was "tied" to a certain output element. Each centroid had a corresponding gaussian function with a changeable standard deviation (σ) parameter.
- The gaussians were used to produce membership predictions indicating the possibility of a certain input belonging to a certain cluster.

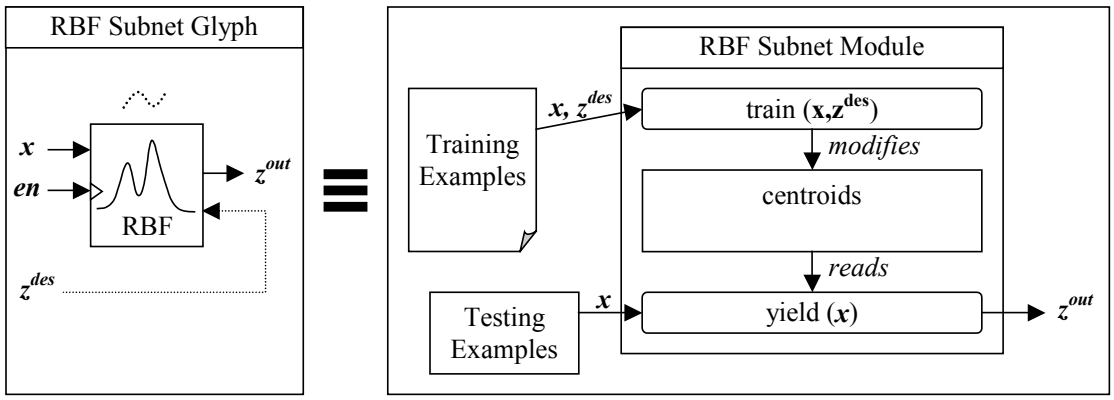


Figure 20: Design of the Radial Basis Subnet.

- Each desired output was binary and only one element of the desired output vector was permitted to be true at a time to indicate to which cluster the input point belongs.
- The elements of the output vector were limited to the [0,1] interval (as generated by the underlying gaussian function).
- The network was trained by moving the centroid that was "tied" to the nonzero desired output element slightly closer to an input vector, and to increase its σ parameter if the distance to the input was not already within 50% of its range (i.e. if the input was at a distance for which the output of the gaussian was less than 0.5 then the σ was increased slightly). All the other centroids were moved slightly away from the input and each of the corresponding gaussians for these centroids had their σ value slightly decreased if they were within a 50% range of the input point.

As in the backpropagation subnet, this subnet also has *train* and *yield* methods. Since the RBF subnet works in either train or yield mode depending on the mode that the supernet is in, a tilde and z^{des} input is only displayed in the case where the subnet is operating in train mode.

The RBF subnet is trained online, generally at the end of each training epoch of the

subnets. In this thesis the RBF subnet is used in the CBSE supernet as a means to predict the most optimal means to segregate training data. Then these predictions were revised at the end of each epoch after performance statistics regarding the subnets were gathered.

4.4 Connection Operators

As a means to make the CDD clearer, vector connection operators were devised. In the code developed for this thesis these connection operators took on the form of a dynamically resizable matrix class. The glyphs for these connection operators is given in **Appendix A** together with a brief description of what each operator does.

From the collection of sub-processing modules that were presented in this chapter it is now possible to provide a clear explanation of the supernet models developed for this thesis in terms of these sub-processing models.

5 The Models

This chapter presents the design of the base model and the supernet models together with their concise description diagrams and testing statistics. The motivation behind each supernet design is discussed and suggested improvements on the designs are given.

5.1 Base Model (BM)

As discussed in **Section 3.1.1**, the base model is a standard ANN having a three layer fully connected architecture comprising an input layer, a single hidden layer and an output layer. The performance of this base model is used to determine which supernet model, from a set of possible supernet models, is the most suitable design for a particular application. The base model is also used to compute a *minimum performance measure* that can be used to reduce the number of possibilities in choosing supernets for a particular application by discarding any supernet not meeting this minimum performance requirement.

5.1.1 Design

In terms of the *Concise Description Diagram* (CDD) schema devised in **Chapter 4**, the base model is represented by a single backpropagation subnet (see **Figure 21**). The diagram has two parts: the left side represents the training configuration while the right side represents the yield configuration. The input to the network is represented by "***x***" and is shown to be fed forward through the network for both train and yield operations.

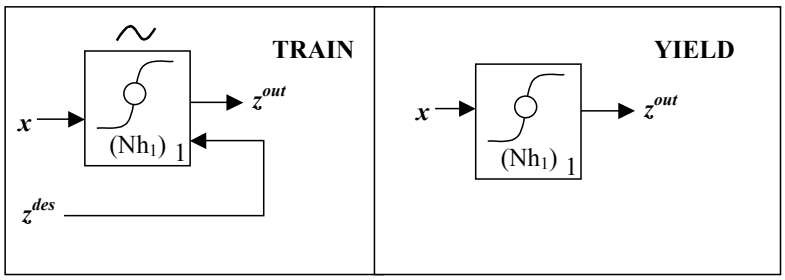


Figure 21: CDD for Base Model.

The desired output of the network, z^{des}, is used for training only. It is shown being fed backwards through the network (i.e. from right to left in the diagram) to emphasize the fact that the subnet is trained using backpropagation. When the network is in yield mode it is shown with a predicted output z^{out} calculated from the input $\boldsymbol{x}$.

Training Parameters

In addition to the number of inputs and number of outputs, the BM model has three major training parameters typical of a standard three layer backpropagation networks, these are shown in **Table 2**.

5.1.2 Testing

Separate training and validation sessions were performed for each of the problem sets chosen in **Section 3.2**. The performance results from testing the BM model are shown in **Table 3** together with the number of degrees of freedom (df) provided by the model (the final MTE values are not averaged since they are not used for evaluation).

The performance graphs for the base model are shown in **Figure B 1** in **Appendix B**. The solid line in each graph indicates the instantaneous normalized training error after each epoch of training, while the dotted line is the final MVE error calculated at the end of training.

Table 2: Base Model Training Parameters.

Parameter Name	Description
n_hidden	Number of hidden nodes in the network
lr_hidden	Hidden layer learning rate
lr_output	Output layer learning rate

From the results of these tests, it appears that the base model performs the best for the scatter4 problem, achieving almost a 0.1 MVE. For the other problems, it produces MVE values close to 0.3. The average MVE value produced by this model for the problem sets is 0.256, thus a supernet that outperforms this model must achieve a smaller mean MVE value.

Denominators for MCCs

The Model Comparison Coordinate (MCC) defined in **Section 3.1.5** is a 2D quantity that represents the performance of a supernet model rated against that of the base model for a specific collection of problem sets. The base model's early MTE values determined for each problem set is used to calculate the MCC_{early} part of MCC coordinates, while the final MVE values are used for the second component, MCC_{final}.

Table 3: Evaluation report for Base Model.

Problem	Final MTE	Early MTE	Final MVE	Hidden	df
Scatter4	0. 054	0.112	0.131	6	18
Spiral	0.311	0.966	0.279	6	18
Crags2	0.033	0.106	0.297	16	48
Wine	0.039	0.141	0.315	5	80
AVERAGE		0.331	0.256		

5.2 Cluster Clue (CC)

The aim of this Cluster Clue (CC) technique is to provide the network with an additional input, called a "clue", that tells it to which cluster a particular input vector belongs. The intent of this method is to determine if the training algorithm can pick up on this cluster information and use it as a rough ordering imposed on the data. If this is possible, then the training data would effectively be split into parts, and the training algorithm should be able to home in on patterns more quickly than in the case where this extra data is lacking.

5.2.1 Design

The main modules used in the design of the Cluster Clue (CC) model are the Cluster Prediction Subnet (CPS), and the Detail Extrapolation Subnet (DES). The clustering module is used only during training. Both the CPS and DES are standard backpropagation subnets as is required for this evaluation. See **Figure 22** for the CDD.

If this model performs successfully, then it should be possible to further increase its speed of training by performing the CPS and DES training separately. For example, a large training set could be split between two processors where one processor trains the CPS and the other trains the DES. Once both parts are suitably trained the weight values of each subnet could be combined into a single network.

During yield operations, the CPS is fed an input ***x*** and its duty is to estimate which cluster the input belongs to. The CPS is effectively a form of *dicer* since it can be viewed as an operation that organizes the training data into parts. The vector ***clue*** that is produced by the CPS is appended to the original input ***x*** and this larger vector is fed into the DES. In an abstract sense, the DES operates like a high-level *splicer* since it recombines processed input (although part of the input is still the same as the original input).

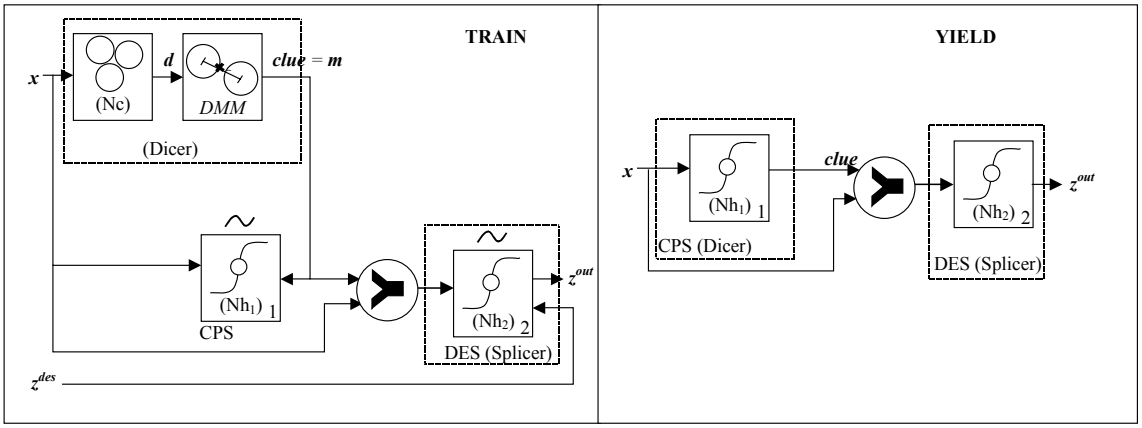


Figure 22: CDD for Cluster Clue.

During training, a k-means clustering module is used to train the CPS and to provide a ***clue*** vector input for the DES. For yield operations, an input is fed into the CPS and generates a predicted ***clue*** vector that is then appended with the supernet input ***x*** and this larger vector is fed into the DES. The output of the DES is used as the final output for the supernet. In this case, the *dicer* comprises the cluster module and the determinate membership module, and the DES remains as a splicing agent.

This method attempts to let the backpropagation algorithm decide for itself how best to use the cluster information. The method is also an attempt to determine whether backpropagation is able to make use of the clue input as a separate cluster weighting so that the inputs weights can focus on providing improved accuracy for certain cluster weightings.

Training Parameters

The significant training parameters used by this supernet are shown in **Table 4**. The number of clusters (n_clusters) is ascertained from performing a cluster analysis prior to training the supernet. Generally, the distance glance algorithm (**Section 4.1.2**) can be used to estimate a reasonable number of clusters automatically.

Table 4: Cluster Clue Training Parameters.

Parameter Name	Description
n_clusters	Number of clusters in training set
n_cps	Number of hidden nodes in CPS
n_des	Number of hidden nodes in DES
cps_out_lr	Learning Rate for output layer of CPS
cps_hidden_lr	Learning Rate for hidden layer of CPS
des_out_lr	Learning Rate for output layer of DES
des_hidden_lr	Learning Rate for hidden layer of DES

The number of hidden nodes in the CPS depends on how complex the cluster shapes are. The number of nodes depends on the number of hyperplanes required in approximating the Bayes decision boundary between the clusters (Lee & Landgrebe, 1993). For instance, two clusters that are far apart could be separated with a single hyperplane, thus needing only one hidden node in the CPS.

The following heuristic algorithm was used for choosing the number of hidden nodes in the CPS subnet:

1. First Let $Nh = Nc$-1 (where Nh is the number of hidden nodes in the CPS and Nc is the number of clusters in the training data)
2. Initialize the CPS weights with small random values
3. Train only the CPS part of the CC supernet with both the training set and clue values produced by the cluster module (this may require attempts using different learning rates)
4. Test the accuracy of the CPS using a validation set, by passing examples through both the clustering module and the CPS. Alter the MVE calculation to compare these two results.
5. If this MVE calculated in step 4 is suitably low (e.g. 0.1), then the value of Nh is acceptable and this algorithm terminates.

6. Either increase or decrease the *Nh* value, obviously not selecting a value that has already been tried. Then go back to step 2.

The choice for the number of nodes in the DES is *assumed* to depend on the desired accuracy of approximating detail within the clusters. However, since all the patterns in the training set, with the possible exception of the cluster bounds, need to be predicted by the DES the problem reverts back to the traditional trail and error technique. It is anticipated that the clue input should help to reduce some of these trials if the backpropagation method can use it effectively.

5.2.2 Testing

The results of the tests carried out on the CC model are shown in **Table 5**. Based on the average MVE value, the CC model appears to perform worse than the base model. However, the model did produce a better MVE for the "Wine" problem achieving 60% of the base model's MVE for the problem.

The other models did not produce significant improvements, and the spiral problem achieving a comparatively high MVE of 0.787. This may indicate that the CC model is better suited for training sets containing overlapping clusters. But such a hypothesis would not be true in all cases because the Crags2 problem was not noticeably improved upon in comparison with the base model. The fact that the two clusters in the Crags2 problem have a similar *range* of outputs for both clusters, while the Wine problem has totally different ranges for each cluster may account for this difference in MVE.

The training performance graphs for the four problem sets comparing the CC model to the BM model are shown in **Figure B 2** in **Appendix B**. These graphs show for all except the Crags2 problem set that the CC model obtains a lower training NMSE sooner than the BM model. This indicates that for some problems it appears that the clueing method does achieve a *higher training rate* than the base model.

Table 5: Evaluation report for Cluester Clue.

Problem	Final MTE	Early MTE	Final MVE	Hidden	df
Scatter4	0.014	0.040	0.121	4	14
Spiral	0.725	0.959	0.787	4	13
Crags2	0.047	0.105	0.305	16	49
Wine	0.032	0.100	0.189	5	81
AVERAGE		0.301	0.351		

In **Table 5**, the results from validation indicate that although the training rate is faster, the accuracy of predictions is generally not any better than that achieved by the base model.

The number of hidden nodes that this model required for each model was similar to that of the base model. This was expected since the cluster module is not using during yield operations, and would therefore not provide additional degrees of freedom to the model. Therefore it is necessary for both the CPS and the DES to have a sufficient number of free parameters (i.e. weights) so that they can produce suitable generalization.

5.2.3 Improvements

Three possible optimizations to this method were tested, but were not found to produce significant improvements over the basic implementation of the CC, since these methods focused on improving the speed of training the CPS. They are as follows:

1. Training the CPS with cluster centers: This drastically reduces the number of training points needed to train the CPS and made training the subnet very fast. However, the membership prediction determined by the CPS was significantly less accurate for points not close to the cluster centers. This was more effective for clusters having simple and non-overlapping shapes, but the method broke down for more complicated shapes and overlapping clusters.

2. Using Fuzzy membership instead of Determinate membership: This method introduces a degree of non-determinism to the ***clue*** vector used in training the CPS. The effect of this is that smaller changes in the weight values were made for points that were not close to any cluster, appearing to make the CPS retain better "memory" of points that were significant in determining the cluster boundaries. However further research into this was not carried out as it made no significant change in the results for the problem sets that the supernet was tested on.

3. Using an RBF network for the CPS: In the implementation of the CC model evaluated in this section, sigmoid transfer functions were chosen for both the CPS and DES subnets. It is more effective using an RBF network for the CPS as its purpose is to determine a rough partition of the input space. In such a case the RBF network would need no further training because the clustering module can assign the positions of the centroids directly. This allows the human operator to focus only on training the DES as there is no need to find suitable training parameters for the CPS.

5.3 Multi-Prop (MP)

Since the performance of the Cluster Clue model was not better than that of the base model it may be possible that the backpropagation algorithm cannot actually make beneficial use of the ***clue*** vector as a means to segregate its training data and thus improve training speed. It was therefore decided to develop a more rigid model that unconditionally forces the model to separate training examples before applying them to the backpropagation algorithm. This was the motivation behind the design of the Multi-Prop (MP) supernet.

5.3.1 Design

The design of the MP supernet comprises one subnet for each cluster in the training data. A clustering module (CM) is incorporated into the design and its distance outputs are linked to a determinate membership module (DMM) as shown in **Figure 23**. This implies that input clustering is treated as being determinate rather than fuzzy.

During train operations the CM and DMM comprise the *dicer* component of this supernet as the binary output vector of the DMM is fed directly into the enable lines of the subnet modules. This causes each subnet to be trained on only those input vectors that are closest to its corresponding cluster center. The output vectors of the subnets are joined by a vector sum connection operator, which represents the *splicer* agent for this model.

During yield operations the CM and DMM are again used to decide which cluster an input belongs to and the input is fed only into that subnet. The output of the subnet is again passed to a vector sum connection operator, which forms the final output for the network.

Since there is one subnet to handle each cluster in the training data it is necessary to specify training parameters for each of the clusters as shown in **Table 6**.

For training sets whose clusters are not similar (i.e. have vastly different sizes and do not follow the same local trends), the best strategy in training the network may be to focus on finding the correct training parameters for one subnet at a time. This can be done by setting the learning rates of all subnets but one to zero and treating the remaining subnet as if it were a standard network on its own. It was found that in general it is easier to find optimal parameters for the subnets in this way instead of guessing parameters for all the subnets at once. EP Scripts were used to accomplish this task more quickly (see **Section 3.3.2**).

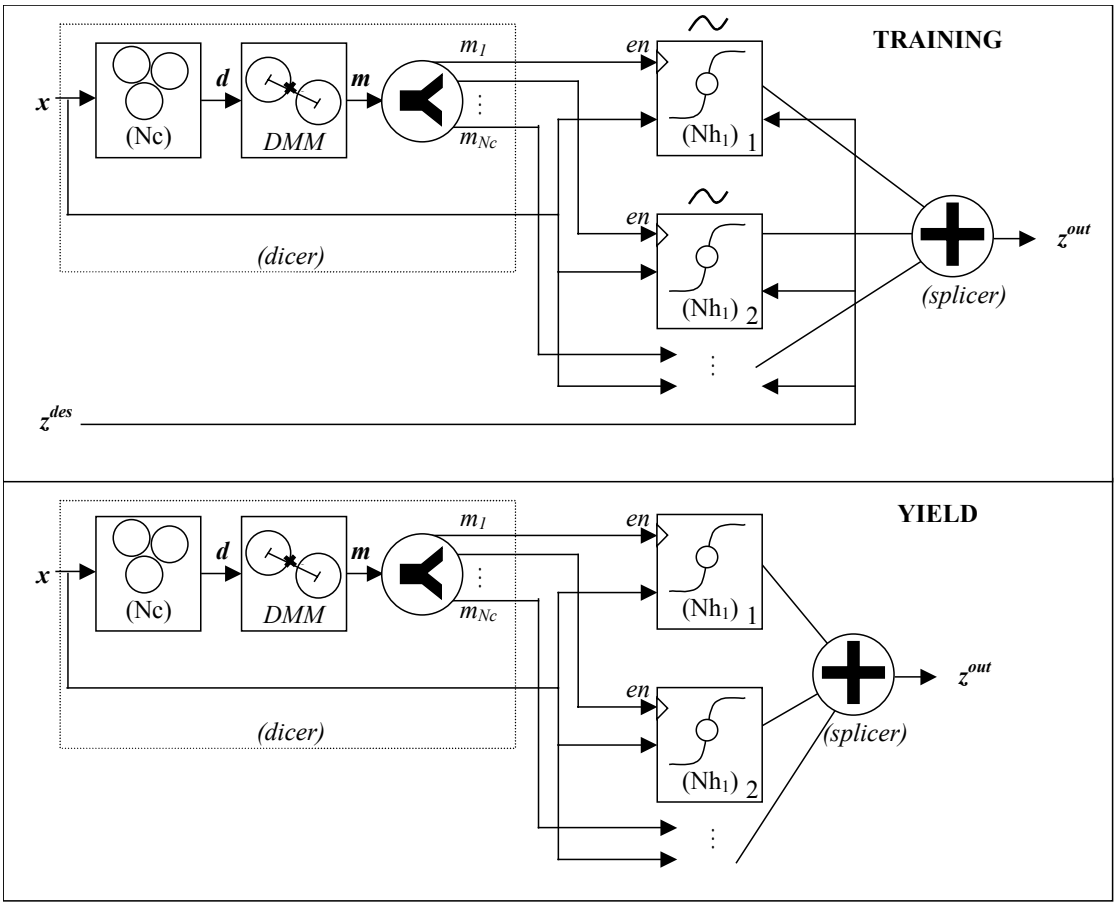


Figure 23: CDD for Multiprop supernet.

Table 6: Multi-Prop Training Parameters.

Parameter Name	Description
n_clusters	Number of clusters in training set
n_hidden	Array of n_clusters elements specifying the number of hidden nodes for each cluster.
lr_hidden	Array of n_clusters elements specifying the hidden layer learning rate for each cluster.
lr_output	Array of n_clusters elements specifying the output layer learning rate for each cluster.

5.3.2 Testing

The training performance results for the MP model are summarized in **Table 7**, and the training performance graphs are shown in **Figure B 3** in **Appendix B**. The Final MVE results shown in the table indicates that the model managed to learn the Scatter4 and Crags2 problems only slightly better than the base model did, but this improvement in performance is minimal. On average, this model did not surpass the performance of the base model.

An output prediction surface (OPS) for one of the problem sets was generated to gain better insight into possible reasons why this model did not perform any better than the base model. **Figure 24** shows the OPS for the Scatter4 problem set. The black points indicate training points, and the height of the points indicates their desired output value. Notice that the training points are almost perfectly predicted by the model as the sigmoids for each cluster have been positioned so as to produce an almost flat surface throughout the predicated cluster area.

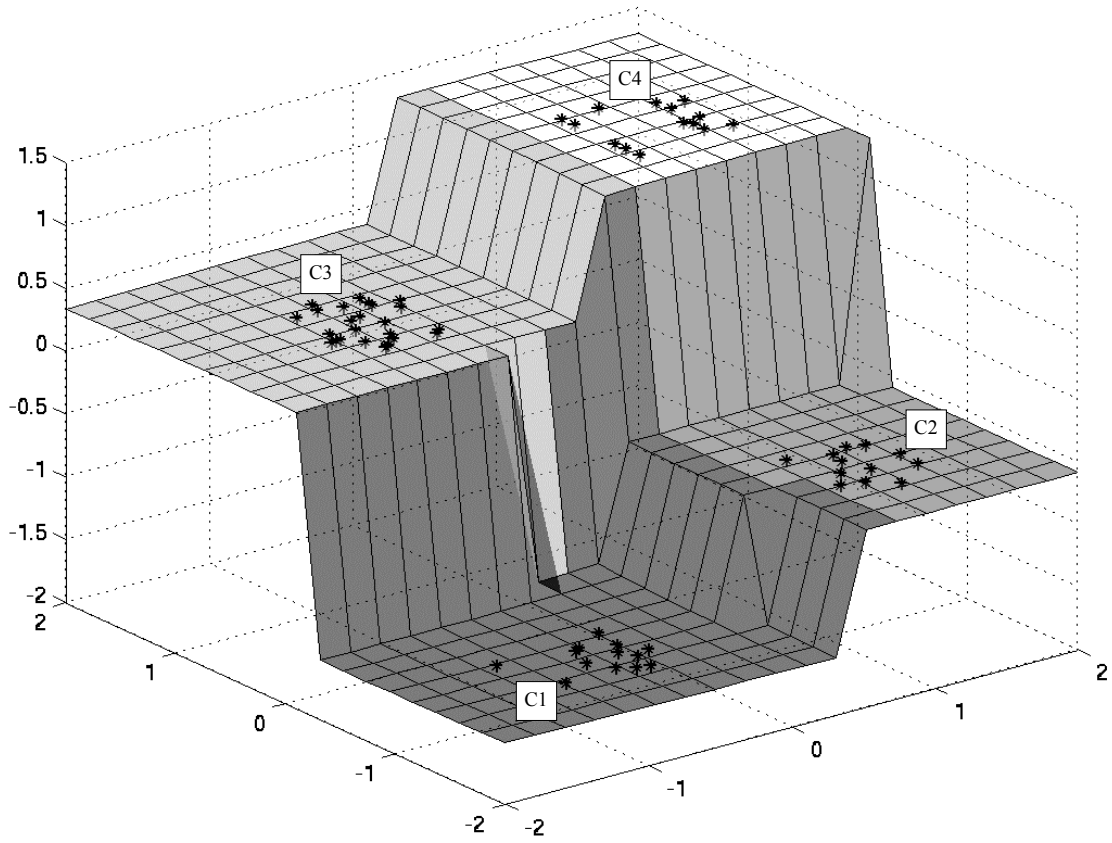


Figure 24: OPS for MP model using the Scatter4 problem set.

Table 7: Evaluation report for Multi-Prop.

Problem	Final MTE	Early MTE	Final MVE	Hidden	df
Scatter4	0.002	0.030	0.120	4	14
Spiral	0.724	0.932	0.677	6	19
Crags2	0.043	0.104	0.295	16	49
Wine	0.311	0.342	0.451	6	97
AVERAGE		0.352	0.386		

The most likely problem with this model, as shown by the "steps" in **Figure 24** is that the membership module does not know the actual size of the clusters. For instance an arbitrary input point that is meant to belong to cluster C4 may be incorrectly classified as belonging to cluster C1. Since there is a huge step of 3 between the average outputs of these two clusters, the effective prediction for such a point would have a high error value. Even if this problem is infrequent, the resultant MTE can be significantly degraded by such large prediction errors.

The number of hidden nodes used by the MP model for each problem set (except the Wine problem) was exactly the same as the number used by the base model. This may be attributable to the fact that the degrees of freedom provided by two subnets each of containing *Ni* inputs and *No* outputs and one hidden layer of *Nh* hidden nodes is equivalent to the degrees of freedom of a single subnet also having *Ni* inputs and *No* outputs, but 2·Nh hidden nodes.

The reason why the Wine problem has an additional hidden is probably due to the impossibility of dividing five nodes equally amongst three subnets. During testing of the Wine problem it was found that a minimum of two nodes were required in each subnet to produce MTE values below 0.4.

5.3.3 Improvements

Possible improvements to this method include:

- Extending the bounds of each cluster by some percentage so as to reduce the problem of cluster prediction error
- Distance measures could be cached for each training example so that the Euclidean distance does not need to be calculated more than once for each training point (since the clustering is performed as a preprocess the cluster centers will not more around during training).
- A means to lessen the effect of transient behavior at the cluster boundaries may reduce the effect of large cluster prediction errors close to these boundaries.

The Valve-Prop Attempt and its "Forgetfulness" Problem

A first attempt at solving the above cluster boundary problem was the development of a supernet that applied a clamping function to the desired outputs causing their value to drop towards zero as the distance from the cluster boundary increased. This model was given the name "Valve-Prop" because this clamping effect was similar to the way that valves operate by opening and closing to let a greater or lesser amount of fluid flow past.

Valve-Prop operated by training input points that were within a cluster boundary at their full desired output magnitude (i.e. $1 \cdot \mathbf{z}^{\mathbf{des}}$), and the other clusters were trained with zero outputs. Input points not within the bounds of any cluster were trained at a fraction of their output magnitude depending how far the point was from the cluster bound. During yield operations, a specific input was fed through *all* the clusters and the outputs were added together, thus eliminating the need to run an input through a clustering stage before applying it to the subnets.

Tests performed on this model were even *less* successful than the Multi-Prop model. The effect of the clamping action applied to the training output caused the training sets to *increase* in difficulty. This occurred because each cluster was required to learn the clamped outputs outside its cluster bound, and this caused the network to have a severe "forgetfulness problem".

A "forgetfulness problem" occurs in an ANN that is incrementally trained because it is forced to adapt its state according to the most recently training examples, which causes examples that were learned in the past to have increasingly less effect on the network's state as the new examples are processed (Shinozawa & Shimohara, 1999). Thus, if a network is fed with a large amount of data for which the first training examples have significantly different desired outputs to the examples learned last, then the network would exhibit poor training performance because the training state would best retaining the patterns learned from the most recent set of examples.

5.4 Fuzzy-Prop (FP)

Since the previous two supernet models attempted did not produce remarkably better results than the BM it was deemed necessary to attempt an alternate approach. This led to the development of the Fuzzy-Prop (FP) supernet. This supernet uses a Fuzzy Membership Model (FMM) or Hybrid Membership Model (HMM) to separate training data instead of a Determinate Membership Module (DMM). This helps to eliminate the problem of assuming that training sets have distinct clusters, and of assuming that the bounds of these clusters are well represented in the training data.

5.4.1 Design

The Fuzzy-Prop model uses one subnet per cluster as in the MP model. But, instead of strictly assigning each input to only one subnet, it provides for some indeterminacy in

this separation process by allowing one input to belong to more than one subnet.

The design of the FP model is shown in **Figure 25**. The cluster module is linked directly to the HMM, but the membership possibility (***m***) values that the HMM generates is not fed directly to the enable lines of the subnets but rather to a "M-E MAP" custom module that performs a mapping from membership possibilities (represented by the "M" in the name) to enable lines (hence the "E" in the name). The CM, HMM, and "M-E MAP" modules together perform the *dicer* agent for the supernet.

The outputs generated by the subnets are fed into a "Weighted Sum" custom module that use the membership possibility values from the HMM to produce a final output equal to the weighted sum of the outputs from the subnets.

The "M-E MAP" module is a special module used only in this supernet and is responsible for deciding which m_i components of the vector ***m*** have a sufficiently large value to merit training the subnet *i* with the input that generated the ***m*** vector. The output of this module is a set of enable lines (e_1 to e_{Nc}) that have value equal to either 1 or 0. Each e_i value is thus used to enable or disable the i'th subnet depending on whether or not the module decided that subnet should or should not be trained. The "M-E MAP" module implements the following function:

$$e_i = \begin{cases} 1 & if\ m_i \geq min\_prob \\ 0 & otherwise \end{cases}$$

The *min_prob* parameter (generally a value around 0.1) disables training of any subnets whose inputs have a small possibility of actually belonging to the cluster that the subnet is associated with. The "weighted sum" module implements the following function:

$$z_i^{out} = \sum_j^{Nc} y_i^j$$

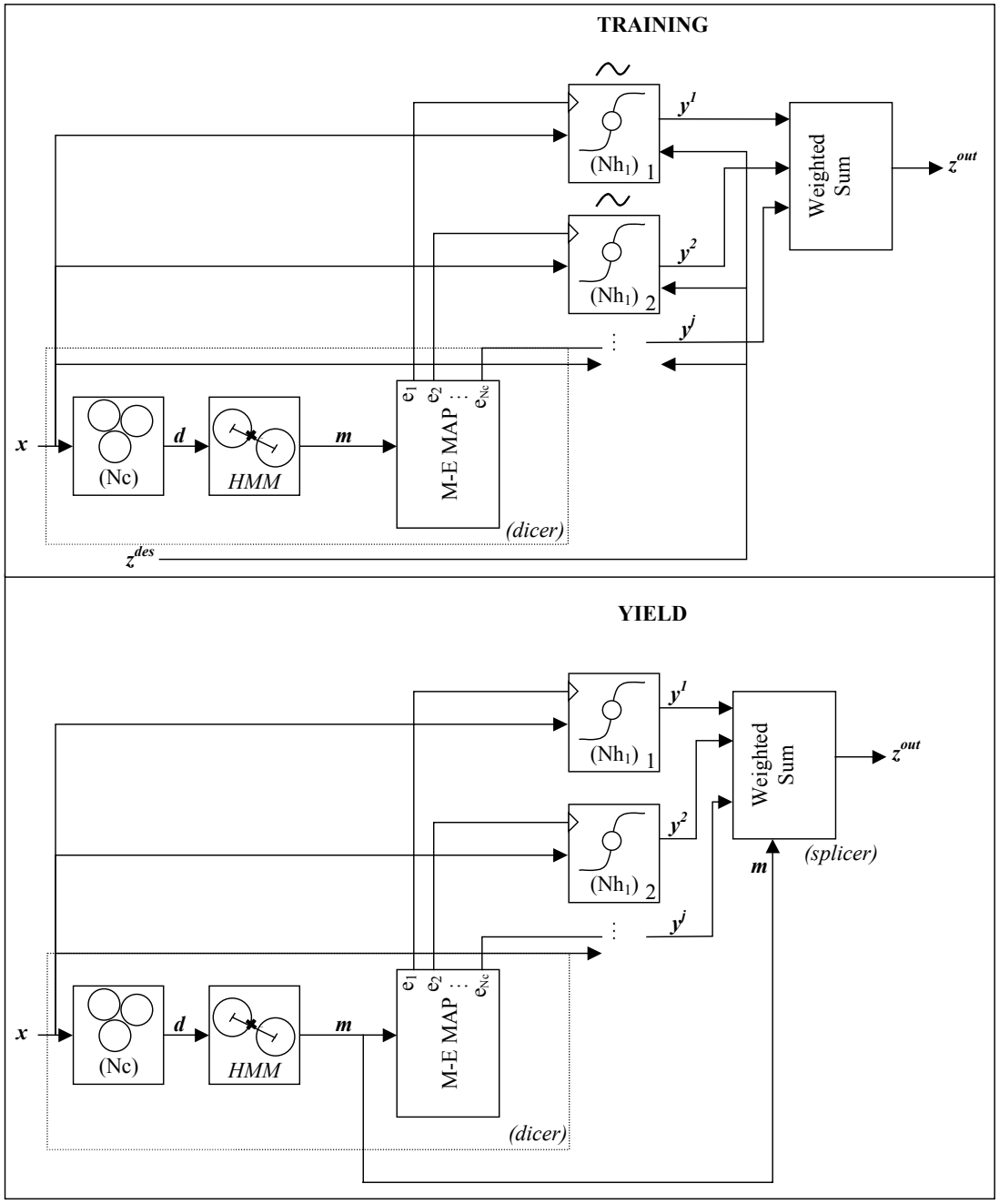


Figure 25: CDD for Fuzzy-Prop Supernet.

The training parameters for the FP model are shown in **Table 8**. Note that the number of hidden nodes and learning rates need to be specified for each subnet.

The "mm_id" training parameter specified which type of fuzzy membership model to use. The FP model was tested with both the continuous fuzzy membership function (see **Section 4.2.2**) and the hybrid membership function (**Section 4.2.3**). However, the results from the hybrid method were found to be consistently better than that of the continuous version.

5.4.2 Testing

The summary of the training performance for the FP is given in **Table 9**. The number of hidden nodes required for each problem set learned by this model was the same as the Multi-Prop model. As before, the Wine problem required one additional node than was needed by the base model.

The Final MVE values show a marked improvement over the base model's results for the Scatter4 and Wine problem sets. However these was no marked improvement for the Crags2 problem, and the Spiral problem performed considerably worse for this model than for the base model (see **Figure B 4** in **Appendix B** for training performance graphs).

In comparison to the final MVE value for the Wine and Scatter4 problem sets, the Crags2 problem set produced a notably higher MVE. This is surprising because the Crags2 problem produced a final MTE value that was almost a tenth below of the MTE calculated for the Wine problem.

Table 8: Fuzzy-Prop Training Parameters.

Parameter Name	Description
N_clusters	Number of clusters in training set
N_hidden	Array of n_clusters elements specifying the number of hidden nodes for each cluster.
Lr_hidden	Array of n_clusters elements specifying the hidden layer learning rate for each cluster.
lr_output	Array of n_clusters elements specifying the output layer learning rate for each cluster.
mm_id	ID number of membership function to use (0 = FMM, 1 = HMM)

Table 9: Evaluation report for Fuzzy-Prop.

Problem	Final MTE	Early MTE	Final MVE	Hidden	df
Scatter4	0.002	0.039	0.117	4	14
Spiral	0.751	0.822	0.758	6	19
Crags2	0.001	0.264	0.244	16	49
Wine	0.011	0.022	0.123	6	97
AVERAGE		0.287	0.311		

A membership function surface (MFS) for the first cluster of the Crags training set was generated in an attempt to determine why the MVE value for the Crags2 problem was worse than the MVE values for the Wine and Scatter4 problems (see **Figure 26**). From this figure, it appears that the size of the intersection between of the two clusters in the Crags2 problem is too small, causing the drop-off in the possibility values to become very sharp. This is likely to cause highly inaccurate prediction results for inputs in this area of intersection.

The MFS generated for the Scatter4 and Wine problem sets did not exhibit such steep transition between clusters. Using a continuous FMM for training the Crags2 problem instead of a discontinuous HMM did not produce noticeably better results. The likely reason for this is that in the FMM case the transition in membership possibilities in the overlap was still too sudden.

5.4.3 Improvements

In addition to the problem of small intersections, there were more noticeable problems caused by excessive overlap in the hyper sphere volumes representing the cluster domains. This causes the model to fail in situations where the clusters exhibit too much

overlap. This overlap causes an excessive degree of interference between the overlapping clusters making a reappearance of the "forgetfulness problem" discussed in **Section 5.3.3** with regard the Valve-Prop model. This problem of hyper sphere overlap is illustrated in **Figure 27**. This figure shows that even clusters whose membership of points are unambiguous can cause overlap of the hyper spheres used to classify them. In the clusters shown in the figure the cluster centers (each marked with an "X") were found to be close to one another since the points were most dense in those parts. But the scattering of points around the centers caused the radii of the clusters to become too large causing a high degree of cluster overlap.

A possible solution to this problem is to revise the clustering algorithm used by the CM so that hyper spheres are attracted by its data points but also repelled by clusters that are too close by. **Figure 28** shows the result of such a technique applied to the data set used in **Figure 27**.

Another means to improve the FP module would be to allow for more complex cluster shapes. For instance, using a polygon to define the cluster bounds instead merely a radius. Naturally, such a method would come at a price, such as making the procedure more computationally intensive since the arithmetic cost in computing whether or not an input is within a cluster would be more complex. There could still be the possibility of predicting cluster bounds inaccurately, causing the overlap problem to reoccur.

5.5 Classification Based on Subnet Error (CBSE)

The "Classification Based on Subnet Error" (CBSE) is a radically different approach to the previously discussed supernets. It uses a RBF network as a form of dicer agent, but this RBF network is only enabled after the input space has been partitioned.

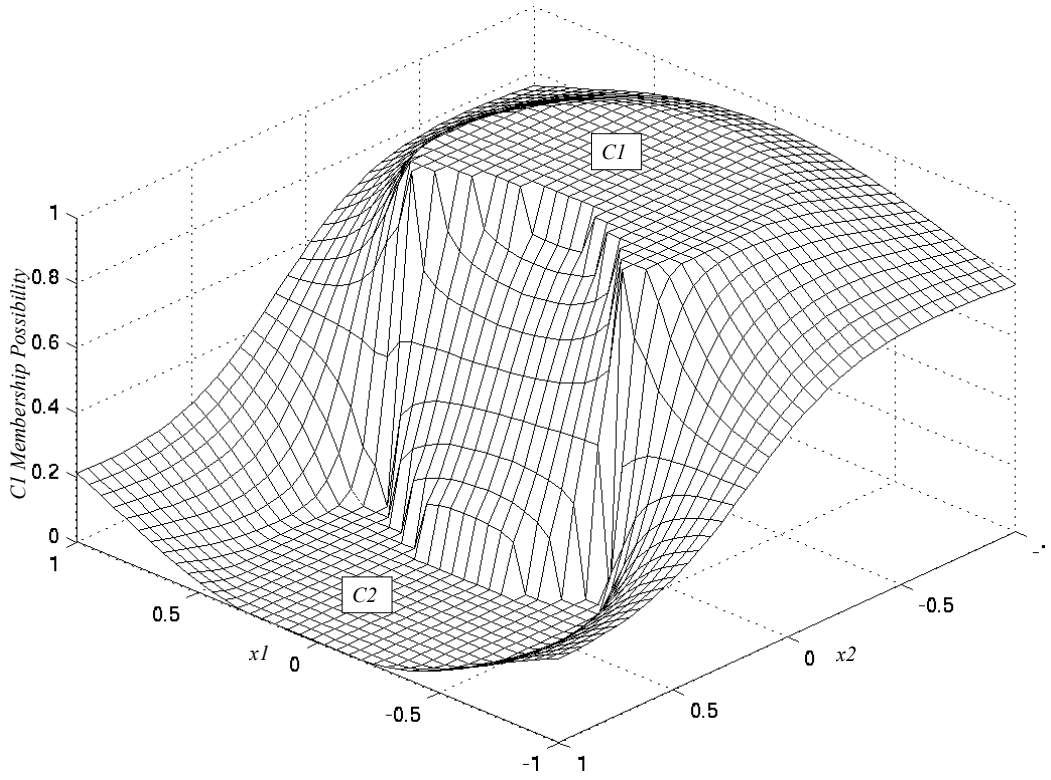


Figure 26: MFS for cluster 1 of Crags2 produced using HMM.

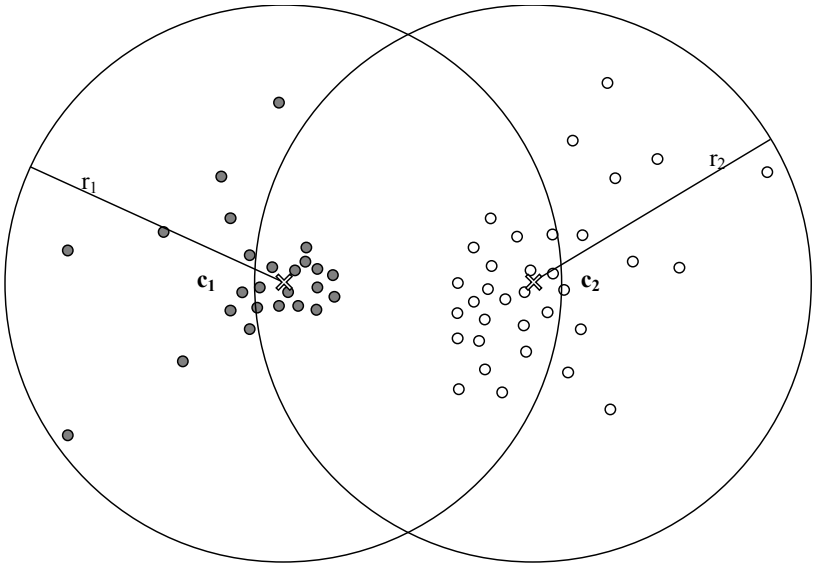


Figure 27: Overlapping Hypercube Problem.

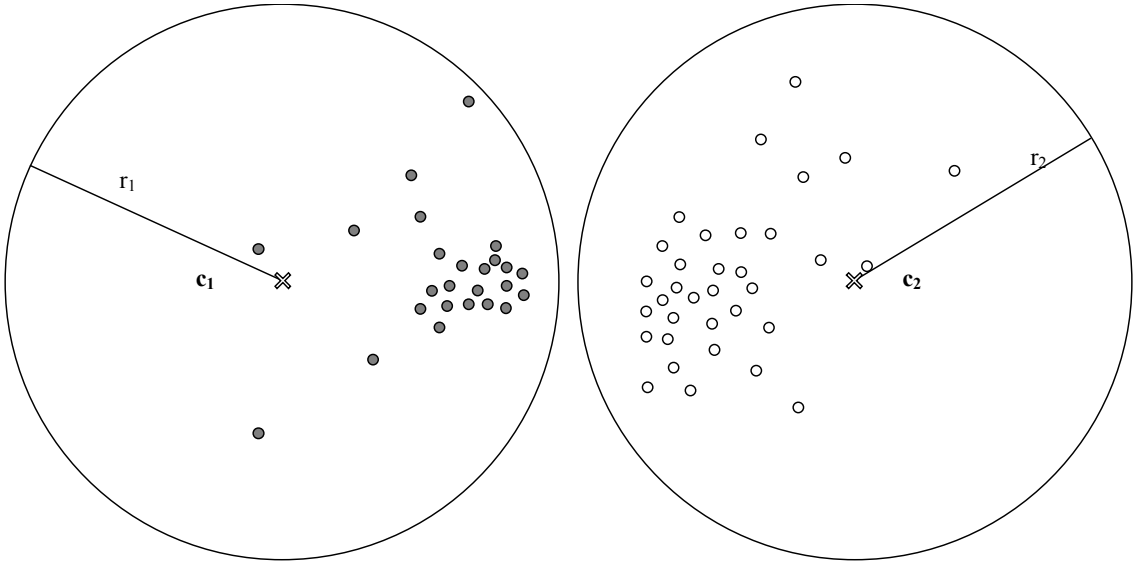


Figure 28: Effect of Cluster Repulsion.

The learning process of the CBSE model has two stages. The first stage causes a certain subnet's learning to converge to particular input regions. The second stage fixes the input regions for the subnets as found in the first stage, and continues training each subnet on only those training examples that belong to that subnet's input regions until suitable error levels are reached.

The algorithm for the first stage works is as follows:

1. Initialize:
    a. *Nc* subnets each with random initial weight vectors (note the number of nodes for each subnet is a parameter, and needs to be chosen by the user).
    b. An RBF dicer network containing at least *Nc* randomly positioned centroids.
2. For each training example in the training set:
    a. Feed the input ($\boldsymbol{x}$) of the training example through each subnet and determine an error value using the desired output for the training example.
    b. Train only the subnet with the smallest error value with that training example.
    c. Train the RBF dicer network with the input ($\boldsymbol{x}$) and the desired output equal to a bit pattern representing which subnet had the smallest error value for this input. This bit pattern can be represented as an binary array $[b_1, b_2, \ldots b_j, \ldots, b_n]$ where the values $b_i = 0 \ \forall i, i \neq j$, and value $b_j = 1$ (where subnet j has the smallest error for the input $\boldsymbol{x}$).
3. If the centroids in the RBF network moved significantly, then repeat step 2 (i.e. input space converges has not settled).
4. Proceed to stage 2.

This method was inspired from the k-means algorithm, and exhibits a similar behavior to the k-means algorithm for the following reasons: 1) both are competitive learning algorithms – for each training example, there is a competition to best represent that

training example, and only the winner of the competition is adjusted. 2) Both the k-means and the CBSE methods can start with an initially random state (the k-means algorithm has randomly placed centers, while the subnets in the CBSE model have randomly placed weights). 3) Both the methods cause convergence to a certain partitioning of the input space by repeatedly making small changes to their states: the k-means method makes small changes to the position of the winning center, while the CBSE method makes small changes to the winning subnet. 3) The position of the centers modified by the k-means algorithm converge to cluster centers that represent the centers of clusters in the input data (i.e. the location of these centers represent a partitioning of the input space), while the CBSE method partitions its input space according to which subnet has the minimum error value for a particular region.

Thus the first stage of the CBSE learning method uses a form of implicit clustering that clusters inputs based on both the input and output space (it includes the output space because the desired outputs affect the partitioning of the input space). This step effectively generates a mapping form input space to subnet number, and trains an RBF network to approximate this mapping.

The second stage of the CBSE learning method fixes the mapping of input space to subnet number, using only the RBF network to predict which subnet should be trained for a particular training example. The algorithm for this stage is simply:

1. For each training example with input ($\boldsymbol{x}$) and desired output ($\boldsymbol{z}^{des}$):
    a. Feed $\boldsymbol{x}$ into the RBF network to predict which subnet should be trained with this training example.
    b. The RBF network returns a vector $[y_1, y_2, \ldots, y_j, \ldots, y_n]$ where each y_i is a non-negative real value for i=1 to n. Find the index j of the first maximum value y_j in this array, i.e. yj >= yi $\forall$i≠j.
    c. Train only subnet j with the given training example.
2. Repeat step 1 until the MVE reaches a low enough value.

Each step in the first stage of this method takes considerably longer than each step in the second stage. This happens because the first stage has to feed each training example into every subnet (i.e. if there are *Nc* subnets there will be *Nc* feed operations for each training example in this stage), then the RBF network has to be trained with the subnet activation bitmap, and finally the winning subnet (i.e. the subnet that exhibited the lowest error value for the training example) has to be trained. In contrast, the second stage merely passes each example through the RBF network, and then through one subnet. Thus this method starts slowly by developing the input partitioning, and then suddenly speeds up once the input partitioning has converged sufficiently.

5.5.1 Design

The design of the CBSE is given in **Figure 29**. The supernet contains *Nc* subnets, where *Nc* is the number of clusters in the training set (this number needs to be calculated by the user). In train mode, each subnet is fed an input value $\boldsymbol{x}$ and produce an error value err_1 to err_{Nc} for subnets 1 to *Nc*. The error value err_j for subnet *j* is calculated from the vector difference between the predicted output ($\boldsymbol{z}^{out}$) generated by subnet *j* for the input $\boldsymbol{x}$ and the desired output ($\boldsymbol{z}^{des}$) given in the training example as follows:

$$err_j = \frac{1}{No}\sum_{i=1}^{No}\left(z^{des} - z^{out}\right)^2$$

These error values are then fed into a minimum index operator, which enabled training of only that subnet that produced the smallest error value.

Note that although there appear to be two sets of subnet modules in the train part of the CDD, there is in fact only one set because the subnets have the same index numbers shown on the bottom left corner of their glyphs. This was done to illustrate the sequence of operations more clearly (the operations start from the left and proceed right).

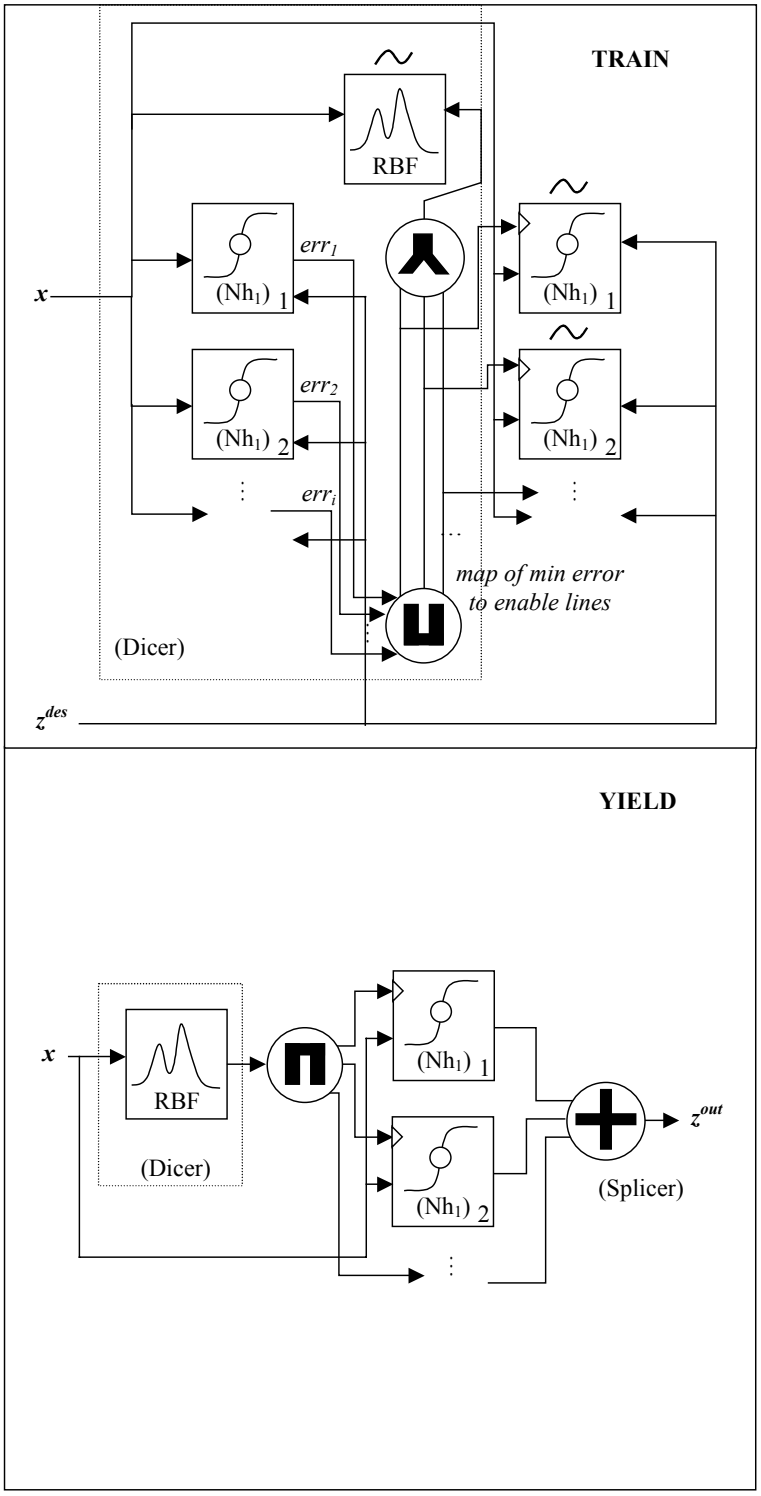


Figure 29: CDD for CBSE subnet.

During training a RBF subnet is instructed to learn the combination of enable lines for a particular input and desired output. This is used later for predicting which inputs belong to a certain cluster.

The CBSE model relies on the fact the subnets are initialized with highly dissimilar initial weights. An input is fed through each subnet and a set of error values is produced. The subnet that produced the smallest error value is trained with the input, now favoring inputs similar to the one just trained. This effectively causes each subnet to prefer certain characteristic inputs.

The training parameters used for the CBSE model are listed in **Table 10**. Note that that an "n_rbfs" parameter is required to specify the number of centroids in the RBF subnet.

5.5.2 Testing

The results from testing the CBSE model are summarized in **Table 11**. Graphs of the training performance are shown in **Figure B 5** in **Appendix B**.

A form of Membership Function Surface (MFS) can be generated during training to determine which subnet is being trained with which input examples. An example of such a surface generated for the first subnet (representing cluster 1) when training the Spiral problem is shown in **Figure 30**. Notice that the surface has two levels (at 1 and 0) that represent the possibilities of inputs being in subnet 1 or subnet 2. The shape that these levels make is similar to that of a spiral, which would cause each subnet to learn only one of the two spirals (i.e. to always output a 0.5 if it learns the upper spiral or –0.5 for the lower spiral). The shape of this cluster is different from that shown in the other models, because the CBSE method uses both the input and output space when partitioning the input space amongst its subnets.

Table 10: CBSE Training Parameters.

Parameter Name	Description
n_clusters	Number of clusters in training set
n_hidden	Array of n_clusters elements specifying the number of hidden nodes for each cluster.
lr_hidden	Array of n_clusters elements specifying the hidden layer learning rate for each cluster.
lr_output	Array of n_clusters elements specifying the output layer learning rate for each cluster.
n_rbfs	Number of nodes to use in the error prediction RBF subnet.

Table 11: Evaluation report for CBSE.

Problem	Final MTE	Early MTE	Final MVE	Hidden	df
Scatter4	0.002	0.009	0.056	8	24
Spiral	< 0.001	< 0.001	< 0.001	6	18
Crags2	0.171	0.250	0.511	12	36
Wine	0.015	0.027	0.041	6	96
AVERAGE		0.072	0.152		

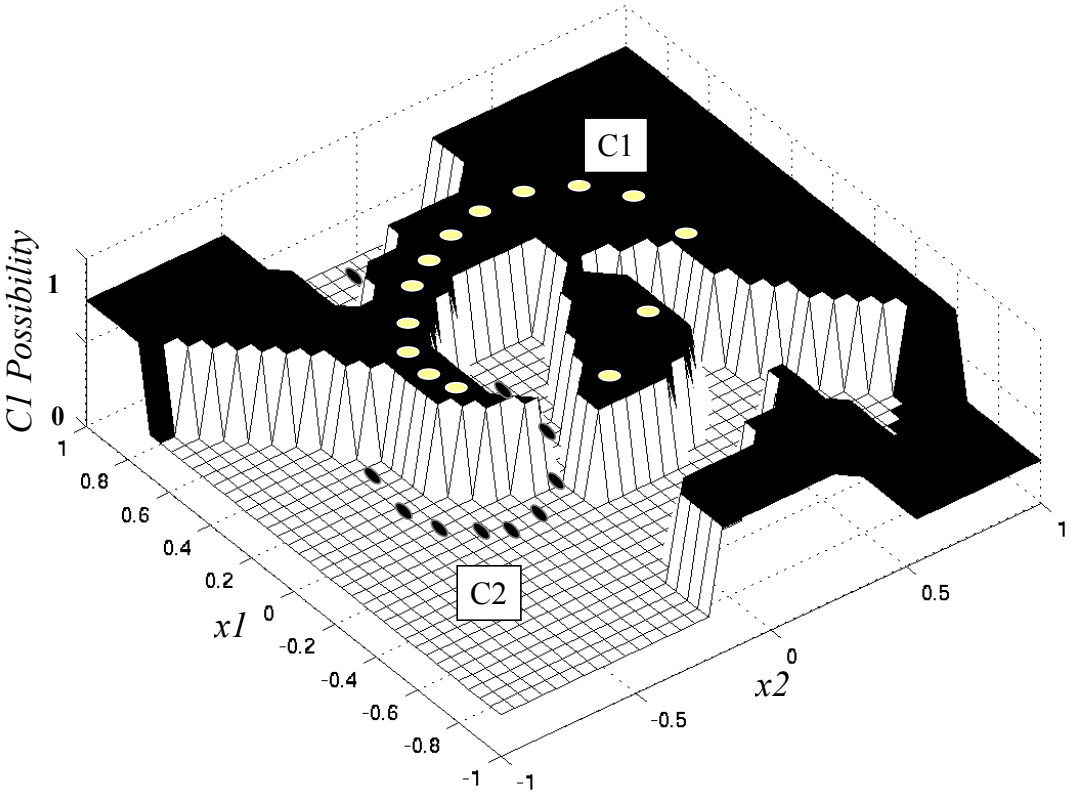


Figure 30: MFS generated during training CBSE model with Spiral Problem Set.

For each problem set, the CBSE required additional nodes to learn the problem when compared to the number of nodes needed by the base model to learn the same problem. This may be due to the supernet requiring additional free parameters for each subnet so that each subnet can define its input bounds. This is different from the base model and the other supernet models, since the base model has no need to define its input bounds (it does not need to decide which subnet to activate) and the other supernet models have their input bounds predefined by their clustering and membership modules. Here $log_2(Nc)$ degrees of freedom have not been added to the total number of degrees of freedom for the supernet as a clustering module is not used.

5.5.3 Improvements

An interesting phenomenon was observed during testing of the CBSE. It was found that if too few subnets are provided for the number of clusters in the training set, then the clustering performance deteriorates and tends to favor all subnets similarly (i.e. each subnet producing nearly the same error), or only one of the subnets is favored and the rest are not used (i.e. one subnet is doing all the work).

From this observation it may be possible to automatically select the number of subnets to use in a CBSE supernet. Such a method would undoubtedly still require some trial and error, but an indication as to whether the number of subnets is correctly selected would help to improve the speed of this process.

6 RESULTS & CONCLUSION

This chapter compares the supernet designs given in **Chapter 5** and discusses which types of problems the models are best suited to solving according to the selection of problem sets that they were tested on. The conclusion reviews the findings of this thesis and suggests further work on the topic.

6.1 Evaluation Results

The tests performed in **Chapter 5** determined the training speed and generalization accuracy for each model tested in relation to a collection of problems sets. These statistics were tabulated in the form of Final MVE and Early MTE values for each problem tested by a particular supernet model. From this data it is possible to generate model comparison coordinates (MCCs) that can be used to visualize the training performance of each supernet compared to other supernet models. This provides some insight into which supernet models are best suited to solve particular types of problems.

6.1.1 Average MCC Plot

Determining the average MCC coordinate for a certain supernet model indicates an average performance rating in comparison to the base model for a given collection of problem sets. The average MCC for a supernet $\mathcal{M}$ and problem set $\boldsymbol{p}$ is determined from the average MCC_{final} and MCC_{early} values as shown in **Section 3.1.5**.

A graph of the average MCC coordinates for each model is shown in **Figure 31** from which certain conclusions can be inferred regarding the *average performance* of each supernet model tested.

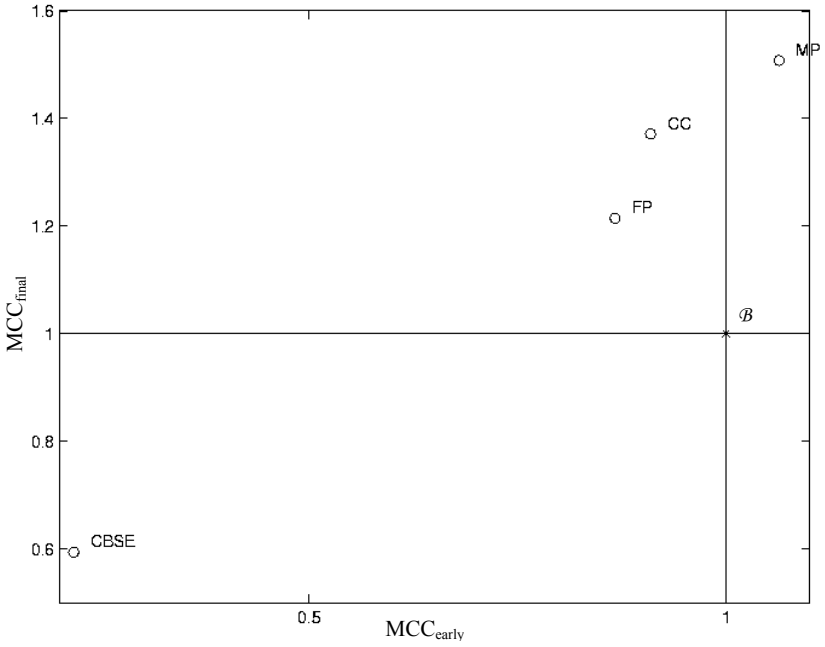


Figure 31: MCC Average Plot.

It is easy to see in **Figure 31** that the CBSE model *on average* outperforms the other models by a noticeable amount. Since the horizontal position of the CBSE point is much further left than the other models this indicates that the CBSE on average achieves a considerably lower MCC_{early} value than the other models, implying that it learns its training data more quickly than the other models do. The vertical position of the CBSE point is further from zero implying that it does not achieve generalization as quickly as it learns to predict its own training data. Since the CBSE point is noticeably lower than that of the other models the CBSE in general still noticeably outperforms the other models' ability to generalize.

Looking further at **Figure 31** it is apparent that all the points, except the one representing the CBSE model, are at a higher position than that of the base model. This indicates that these models in general produce worse generalization than the base model does. The CC and FP models may learn their training data more quickly that the base model, but this does not seem to imply that they will produce better generalization.

6.1.2 MCC Plots For Each Problem Set

Although some of the models do not appear to exhibit improvements in general with regard to all the problem sets considered, some of the models are still better suited to certain types of problem. **Figure 32** shows plots of MCC values for each of the four problem sets for which the supernets were tested. From these plots, the following conjectures were made as to which types of problem a specific supernet model is best suited to solve:

- The CBSE model performed well for all problem sets, except the Crags2 problem set. The clusters in the Crags2 problem have similar output ranges and their governing functions are almost identical. This strengthens the argument that the CBSE model is best suited to training data that contain clusters that exhibit dissimilar local behavior.
- The Fuzzy-Prop model showed improvement over the base model for all except the spiral problem. This indicates that this model can handle partly overlapping and non-overlapping clusters but cannot handle intertwined clusters.
- The Cluster Clue model performs similarly for both the Scatter4 and Wine problems, but does not perform well for the Spiral or Crags2 problem. This is an indication that this model works best for clusters that have distinct behavior and at most only slightly overlapping (i.e. there can be some overlap of the hyper spheres used to partition the clusters but too much overlap – as occurs in the spiral problem – causes the supernet to fail).
- The Multi-Prop model shows improved performance only for the Scatter4 problem, which implies that it is only suited to learning problems that exhibit non-overlapping clusters.

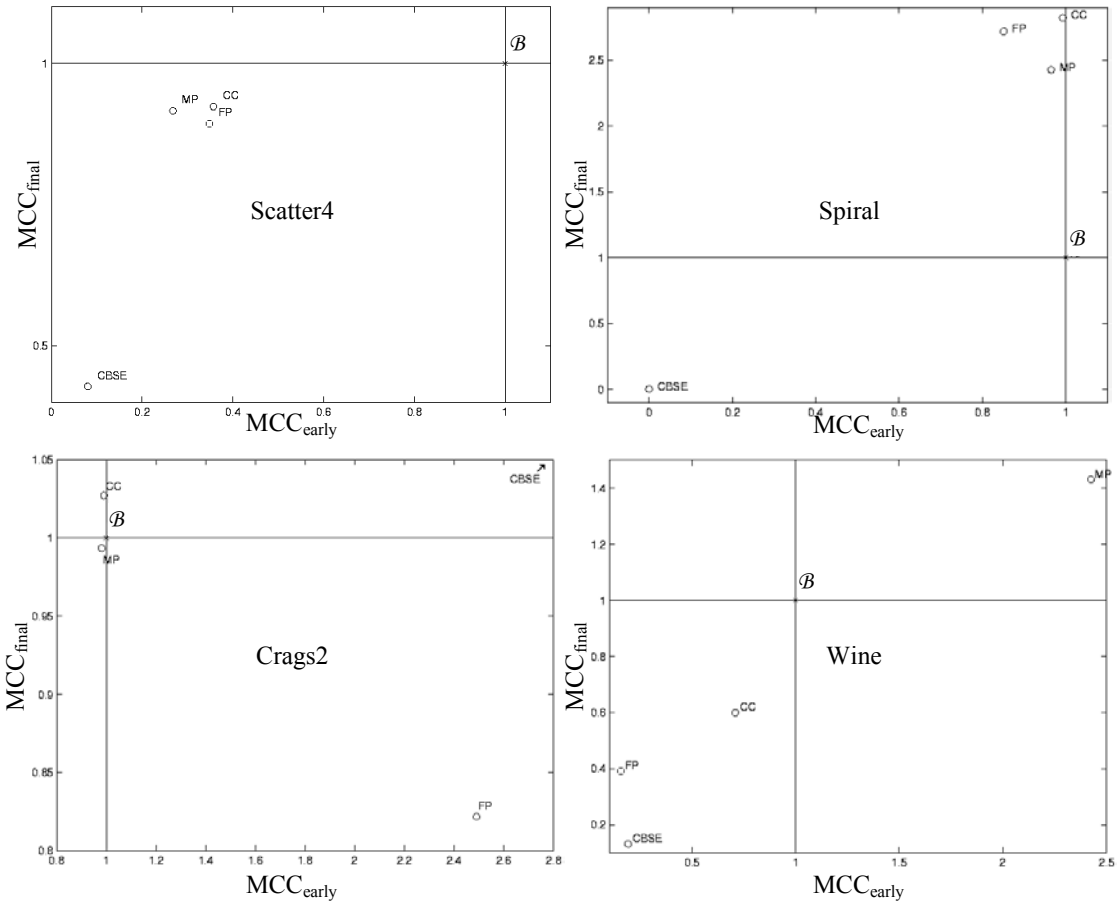


Figure 32: MCC plots for the individual problem sets.

6.1.3 RCT Plots

As discussed in **Section 3.1.6**, the Relative Convergence Time (RCT) of a supernet indicates how quickly the supernet learns its training data compared to the base model. The Convergence Time (CT) for each model are shown in **Table 12** expressed using the number of training epochs required to achieve a MTE that is 30% above the convergent MTE. **Figure B 7** in **Appendix B** illustrates the way that these values were calculated using the training performance graphs that were produced during testing of the models (**Figure B 8** shows a zoomed-in view).

From this table, the RCT values were calculated by dividing the number of epochs required for every model for a given problem set by the number of epochs required by the base model.

Average RCT values (shown in **Figure 33**) were generated for each model, and expresses *on average* how quickly a particular supernet learns its training data. From this it is possible to see that the CC and CBSE models on average produce faster convergence than the base model, with the CC model achieving the fastest convergence on average. The MP and FP models on average produce a slower convergence than that of the base model.

Table 12: Number of epochs required to achieve a MTE 30% above final MTE.

Problem	BM	CC	MP	FP	CBSE
Scatter4	90	120	290	120	90
Spiral	1804	1170	2180	390	80
Crags2	35	30	90	170	110
Wine	270	160	110	50	60

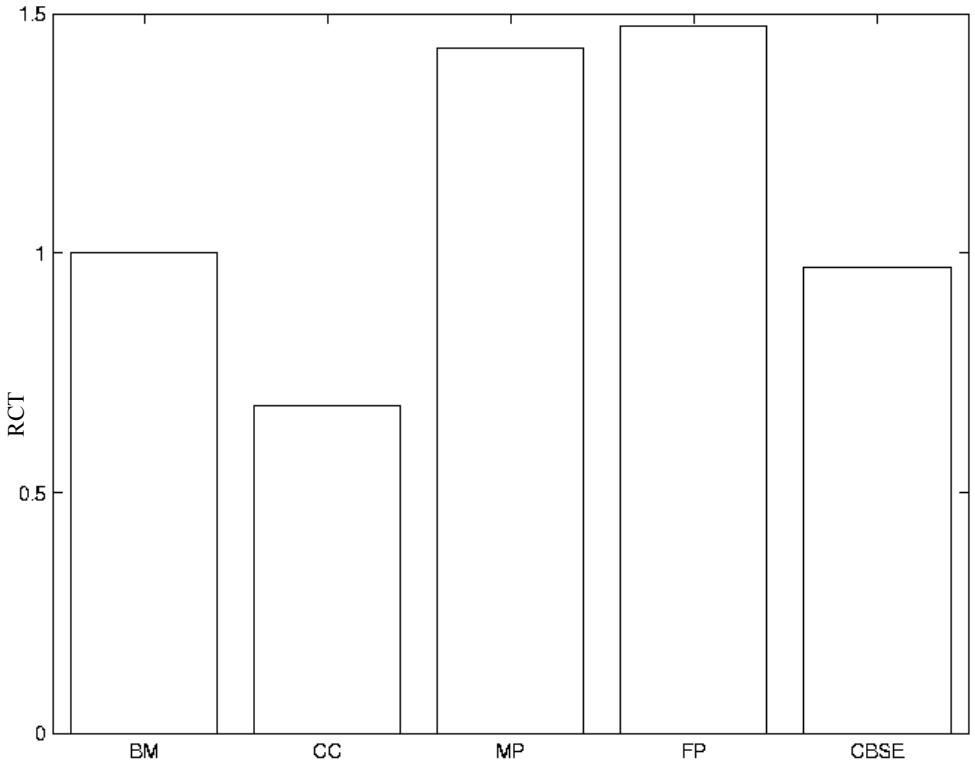


Figure 33: Barchart of average RCT values.

The RCT values for each problem are shown in **Figure 34**. This figure shows which models achieve the fastest convergence for certain problem sets:

- The CBSE model converges most quickly for Spiral and Scatter4 problems compared to all the other supernet models, but it converges comparatively slowly for the Crags2 problem.
- The CC model converges most quickly for the Crags2 problem, and is shown to converge more quickly than the base model for all the problems.
- The FP model exhibited faster convergence than the base model for all problems except the Crags2 problem.
- The MP model exhibited slower convergence that the base model for all the problems with the exception of the Wine problem.

6.1.4 Crags2 Compared To Gaussian Humps

The MCC plot for the Crags2 problem set shows that none of the supernet models produced significantly improved training results for the Crags2 problem. The two crags in this problem have many discontinuous due to the sharp edges that make up the sides of the crags (see **Figure 12**).

The discontinuous feature of the Crags2 training set can have a detrimental effect on networks trying to learn it using backpropagation and sigmoid functions, because: 1) backpropagation uses differentiation, and therefore works best if the underlying function it is attempting to approximate is smooth, and 2) sigmoid functions are essentially smooth functions (when they are used for interpolating a transition between multiple points), and therefore would not be ideal functions to approximate training data containing many sudden jumps (a unique sigmoid would be needed for every large jump in the data). It is therefore expected that the models presented here would have some difficulty in learning the Crags2 training set since they all rely on the backpropagation training algorithm and

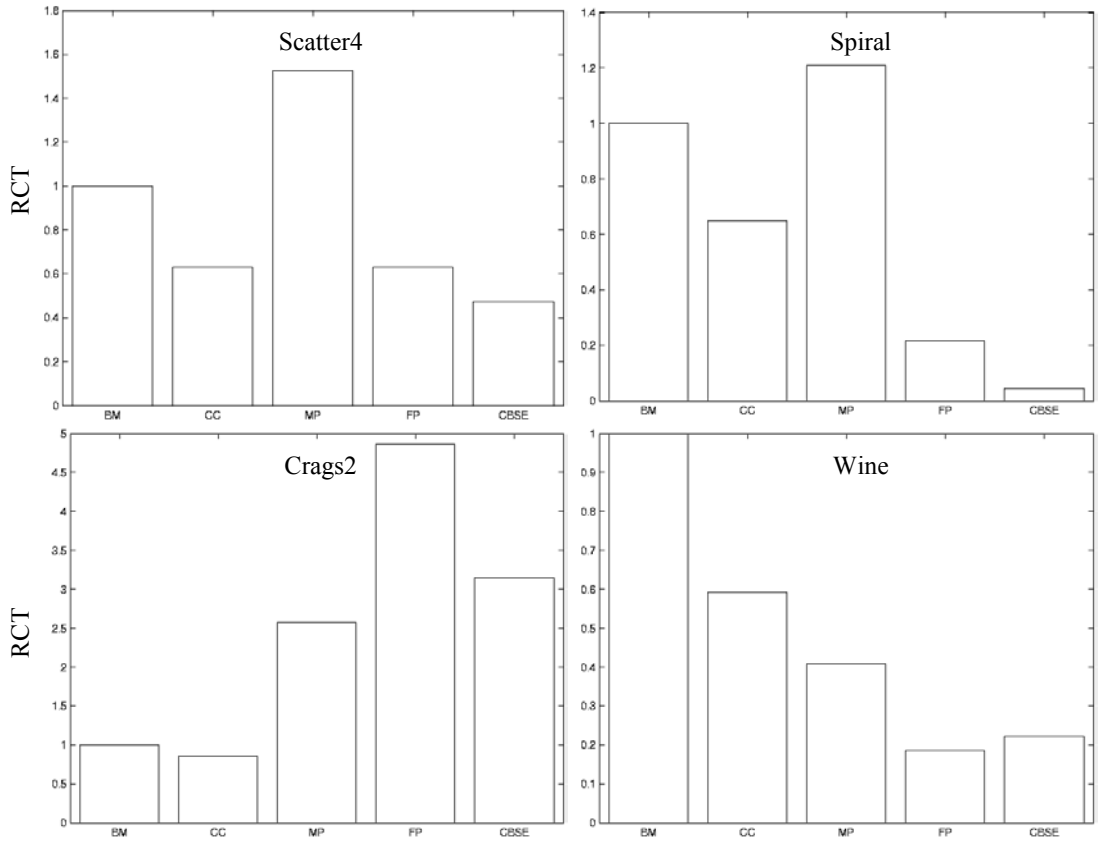


Figure 34: RCT values calculated for each model arranged according to problem set.

use sigmoid functions. As a means to verify this hypothesis, a simplified version of the Crags2 problem was generated that comprises two smooth gaussian humps (shown in **Figure 35**) instead of discontinuous crags. The expected result of training the base model on this gaussian humps problem should be a reduced mean validation error, because the smooth sigmoids used to approximate the surface should do a better job of approximating smooth gaussians than the rough crags.

Figure B 6 in **Appendix B** shows the training performance of the base model when learning the gaussian humps problem and the Crags2 problem. Both problems have the same size training and validation sets. As can be seen in the diagram, the mean training error for both problem sets are similar. However the mean validation error for these problems were significantly different. The MVE value for the Crags2 problem was 0.297, while the MVE for the gaussian humps problem was 0.073. This shows that the discontinuous property of the Crags2 problem set is at least partly responsible for the poor training results obtained for this problem set.

6.1.5 Consistent Behavior

The model that produces the most consistent behavior can be determined by finding the standard deviation in the list of MVE values for each problem that the model was tested on. The bar chart shown in **Figure 36** illustrates the standard deviation of MVE values calculated for each model. The base model (BM) has the most consistent behavior since its standard deviation is the lowest. The standard deviations for the supernets were noticeably higher than that of the base model, indicating that they each exhibited a certain amount of inconsistency in their training performance. This result is expected since supernets are not designed to solving arbitrary problems, while the base model is.

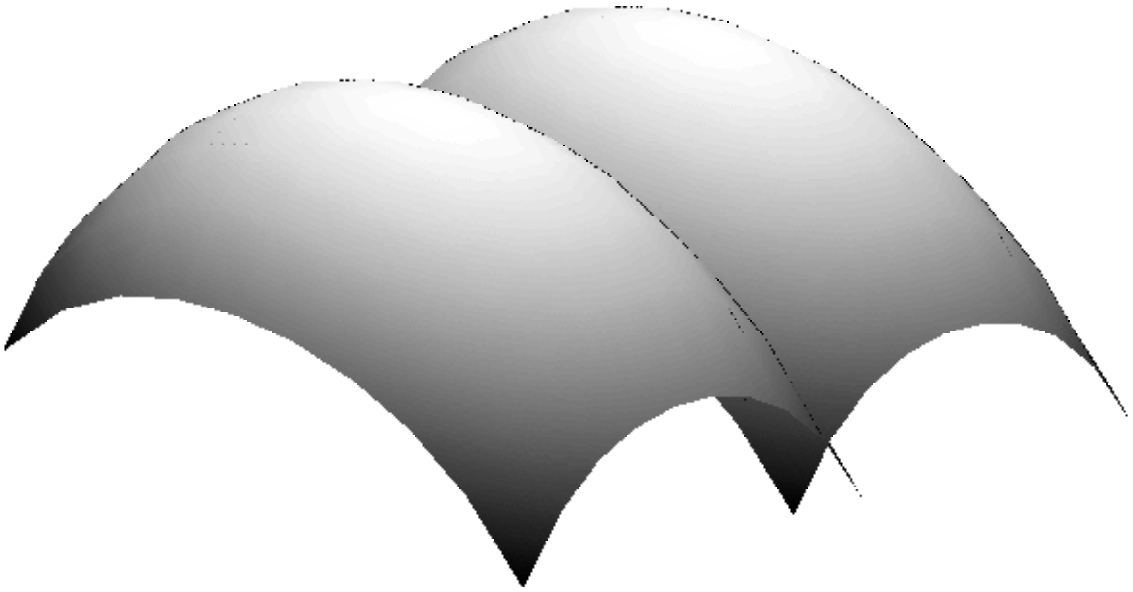


Figure 35: Smooth Gaussian Humps.

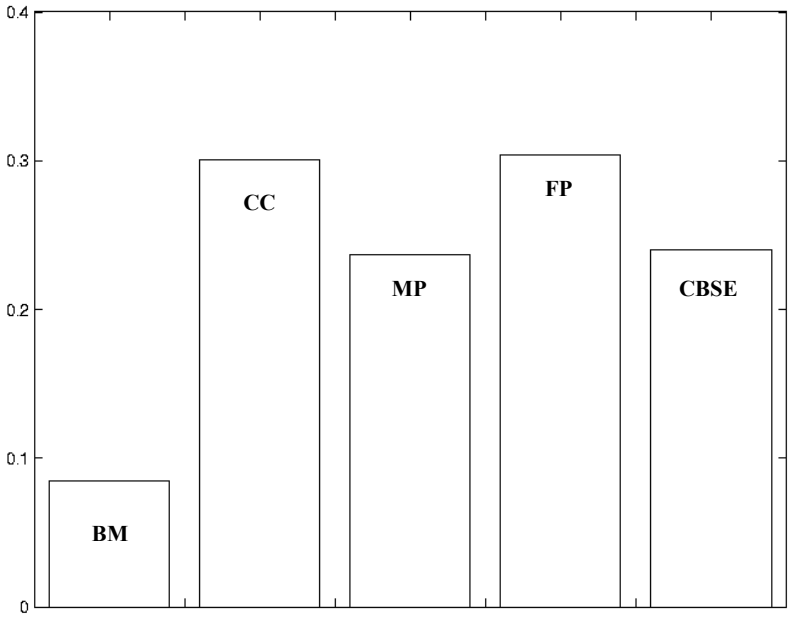


Figure 36: Standard Deviation of MVE values.

6.2 Conclusion

The objectives of the thesis discussed in **Section 1.4** were achieved, namely:

- A practical feasibility study of the supernet paradigm was investigated and found to be a suitable technique for specific types of applications.
- The benefits of supernets over the standard model were shown to be improved training rates and generalization, but only for problems that were suited to the specific supernet design
- The performance of supernets is limited to the capability of the training algorithm that it attempts to enhance (in this case backpropagation). Since a supernet is not designed for solving arbitrary problem it can result in degrading the performance of the training algorithm when processing training data for which it is not designed.
- The thesis presented an effective means to develop, train and evaluate supernet models in related to a base model.

From the findings of this thesis, further research into the application and design of supernet models is merited. Recommendations for further research on this topic include:

- Evaluation of the supernets using a larger selection of real-world problem sets.
- Other techniques aside from clustering should be tested to determine how they could improve the performance of standard training methods.
- Using different training algorithms to train different subnets could be tested to determine if such a technique improves the overall prediction performance of a supernet.
- Further research into the heuristics for choosing parameters required by the Distance Glance algorithm could help to solve the problem of selecting the number and placement of clusters in training sets.

- Development of an online version of the Distance Glance algorithm for training RBF subnets used in cluster membership prediction could eliminate the clustering preprocessing step, making the supernet method more compatible with online learning requirements.
- Since the CBSE method only works well for training sets that have dissimilar behavior for each cluster, there may be some means to make it perform better for similarly behaving clusters by combining the CBSE model with a standard clustering algorithm such as k-means.
- Further investigation of methods that use clustering of the input and output space.

The effectiveness of supernets depends on the number of times a particular supernet design can be reused for different applications. If a supernet is designed for one-time use, then the development time inherent in the design of a specific model for the application concerned may exceed the improvement gained from implementing such a model. However, if a supernet design is found to be effective for certain types of application which commonly occur, then the supernet approach is likely to offer long-term savings in human time as it would help to decrease the amount of time the time taken for the human operator to train a standard neural network to provide the same performance.

□

LIST OF REFERENCES

LIST OF REFERENCES

Abdi, H., Valentin, D., & Edelman, B. (1999). *Neural Networks.* Newbury Park CA, USA: Sage University Series.

Baum, E. B. & Lang, K. E. (1991). "Constructing hidden units using examples and queries" In Lippmann, R.P. Moody, J.E. & Touretzky, D.S. (Eds) *Advances in Neural Information Processing Systems 3,* pp. 904-910. San Mateo: Morgan Kaufmann Publishers.

Blake, C. (1998). *Wine recognition data*. (Web Document). Irvine, University of California. Available from: <ftp://ftp.ics.uci.edu/pub/machine-learning-databases/wine> [Accessed June 24, 2002].

Burke, L. (1993) "Assessing a Neural Net: Validation procedures". *PC AI*, 7(2) March-April, pp. 20-24.

Cybenko, G. (1989). "Approximation by Superposition of a Sigmoidal Function". *Mathematics of Control Signals Systems*, 2, pp. 303-314. London: Springer-Verlag

Duda, R., Hart, P. (1973). *Pattern Classification and Scene Analysis*. New York: Wiley.

Dwinnell, W. (1998). "Modeling Methodology 4: Localizing Global Models". PC AI, 12(3), pg. 24.

Fausett, L. (1994). "Fundamentals of Neural Networks: Architectures, Algorithms and Applications". London: Prentice Hall.

Haykin, S. (1994). *Neural Networks: A Comprehensive Foundation*. New York: Macmillan College.

Hornik, K., Stinchcombe, M., and White, H. (1989). "Multilayer feedforward networks are universal approximators". *Neural Networks*, 2(5), pp. 359-366.

Jigsaw Puzzle Tips and Techniques. (1997/2001) (Web Document). Jigsaw Jungle. Available from: <http://www.jigsawjungle.com/code/puzztips.html> [Accessed June 14, 2002].

Joost, M. & Schiffmann, W. (1998). "Cross-Entropy combined with Pattern Normalization". *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 6(2), pp 117-126.

Kosko, B. (1992). *Neural Networks and Fuzzy Systems: A Dynamical Systems Approach to Machine Intelligence*. New Jersey: Prentice-Hall.

Lee, C., Landgrebe, D. (1993). "Decision Boundary Feature Extraction for Non-Parametric Classification". *IEEE Transactions on System, Man, and Cybernetics*. 23(2) March-April, pp 433-444.

MatLab Documentation. (1994/2002) (Web Document). Math Works. Available from: <http://www.mathworks.com/access/helpdesk/help/techdoc/matlab.shtml> [Accessed May 20, 2002].

Moody, J., Darken, C. (1989). "Fast learning in networks of locally-tuned processing units", *Neural Computation*. 1, pp 281-294.

Saarinen, S., R.B. Bramley, and G. Cybenko. (1992). "Neural Networks, backpropagation

and automatic differentiation." In *Automatic Differentiation of Algorithms: Theory, Implementation, and Application* (A. Griewank and G.F. Corliss, eds.), pp. 31-42. Philidelphia, PA: SIAM.

Shinozawa, K., Shimohara, K. (1999). "An improved method for reducing the forgetfulness of incremental learning". *IEEE International Conference on Systems, Man and Cybernetics (SMC'99)*, *4* October, pp. 1068-1073

Theodoridis, S., & Koutroumbas, K. (1999). *Pattern Recognition*. San Diego: Academic Press.

Tou, J., Gonzalez R. (1974). *Pattern Recognition Principles*. London: Addison-Wesley.

Whitehead, B. A. and Choate, T. D. (1996). "Cooperative - competitive genetic evolution of radial basis function centers and widths for time series prediction". *IEEE Transactions on Neural Networks*, 7(4), pp. 869-880.

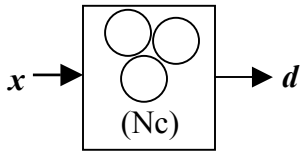
Zeng, Y., Liu, Z. "Self-Splitting Competitive Learning: A New On-Line Clustering Paradigm". IEEE Transactions on Neural Networks, 13(2), pp 369-380.

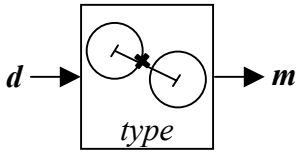
APPENDIX

Appendix A: Glyphs

Clustering And Membership Modules

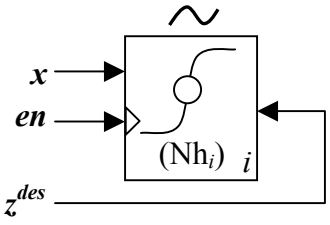
Determinate Clustering (**Section 4.1**)	Membership Module (**Section 4.2**)

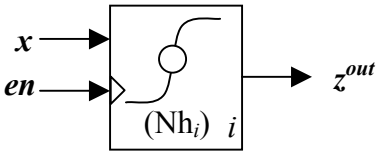




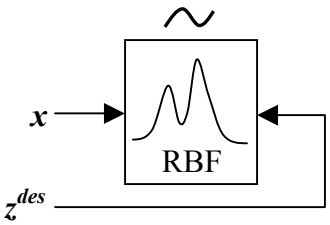
Subnet Modules

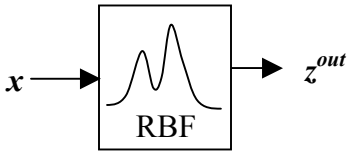
Backprop Training (**Section 4.3.1**)	Backprop Yield (**Section 4.3.1**)





RBF Training (**Section 4.3.2**)	RBF Yield (**Section 4.3.2**)

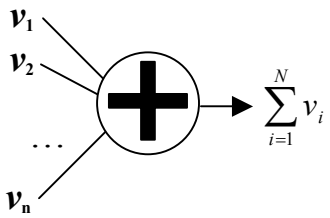




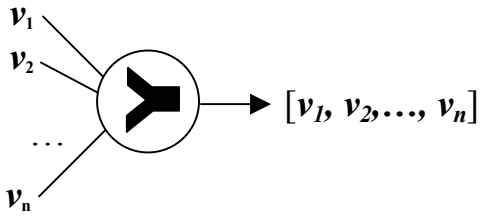
Notes: The '*en*' input is the enable line. Only if *en* is nonzero during training is the subnet trained, otherwise no operation is performed. During yield only if *en* is nonzero does the input ***x*** passed through the network producing a predicted output, otherwise the subnet merely returns zero.

Connection Operators (See Section 4.4)

Vector Sum	Vector Append

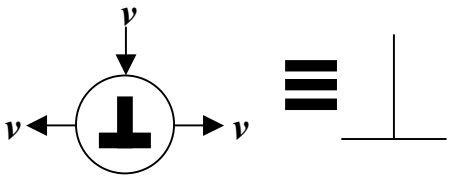


Produces an element-by element vector sum of the vectors v_1 to v_n.

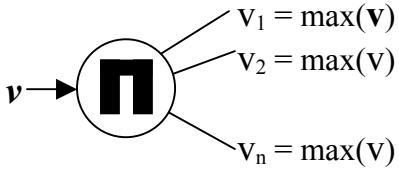


Joins the vectors v_1 to v_n into a larger vector.

Duplicate	Max Index

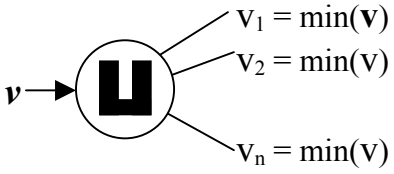


This "T-Junction" is used to copy the results of one vector to several places. Generally the lines are just split into two sections as shown on the right.



Returns a binary vector where the index corresponding to the maximum element of v is true.

Min Index



Returns a binary vector where the index corresponding to the minimum element of v is true.

Note: All the connection links between modules are vectors.

Appendix B: Performance Graphs

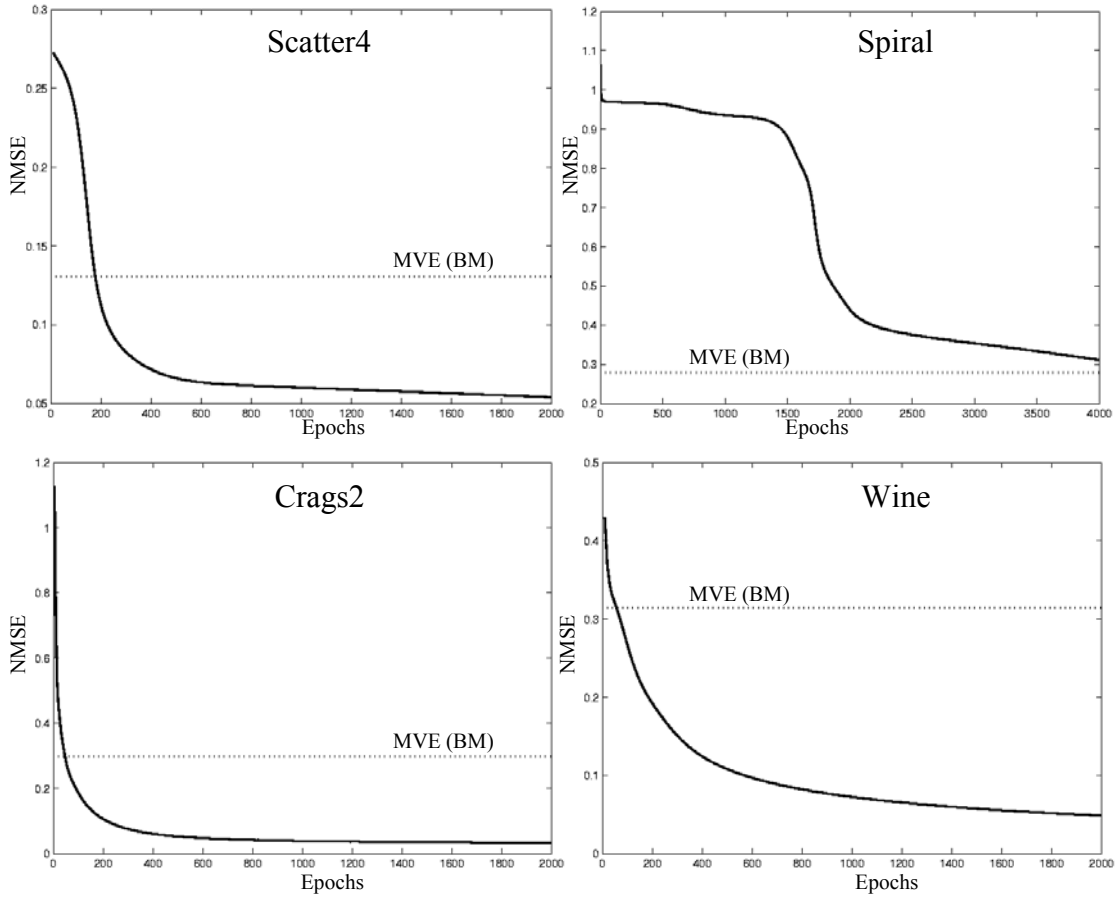


Figure B 1: Performance graphs for BM.

Note: The graphs in solid black represents the instantaneous mean training error of the model, and the dotted line provides a reference for the mean validation error (MVE) computed by the model.

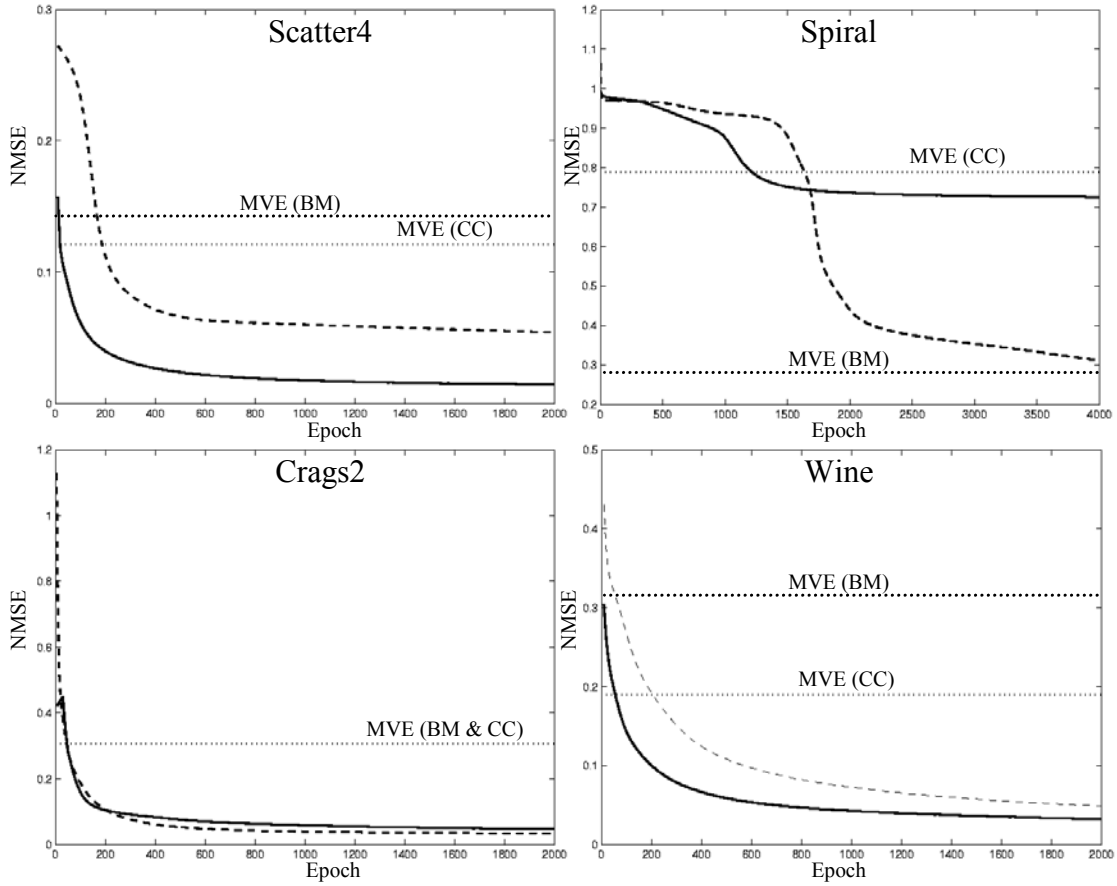


Figure B 2: Performance graphs for CC.

Note: The graphs in solid black represents the instantaneous mean training error of the supernet, the dashed line indicates the BM's instantaneous mean training error for the problem, and the dotted line provides a reference for the mean validation error (MVE) computed by the supernet.

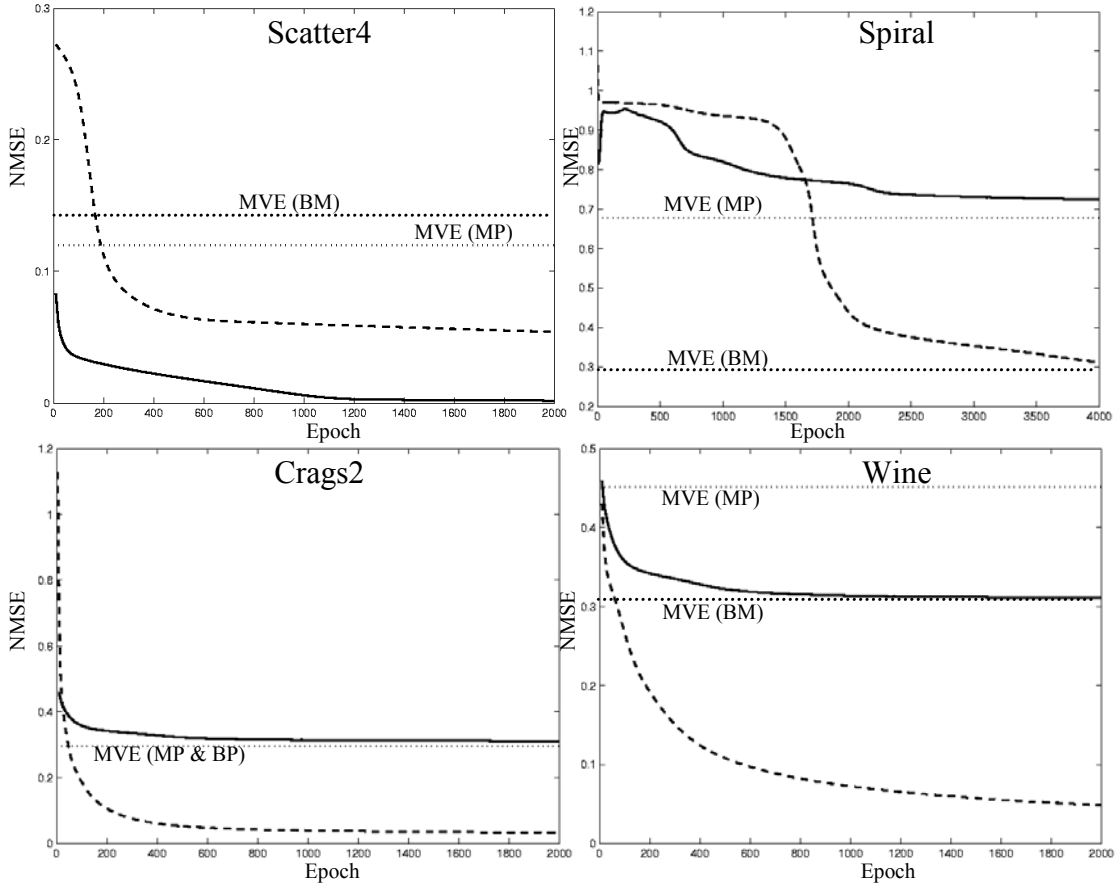


Figure B 3: Performance graphs for MP.

Note: The graphs in solid black represents the instantaneous mean training error of the supernet, the dashed line indicates the BM's instantaneous mean training error for the problem, and the dotted line provides a reference for the mean validation error (MVE) computed by the supernet.

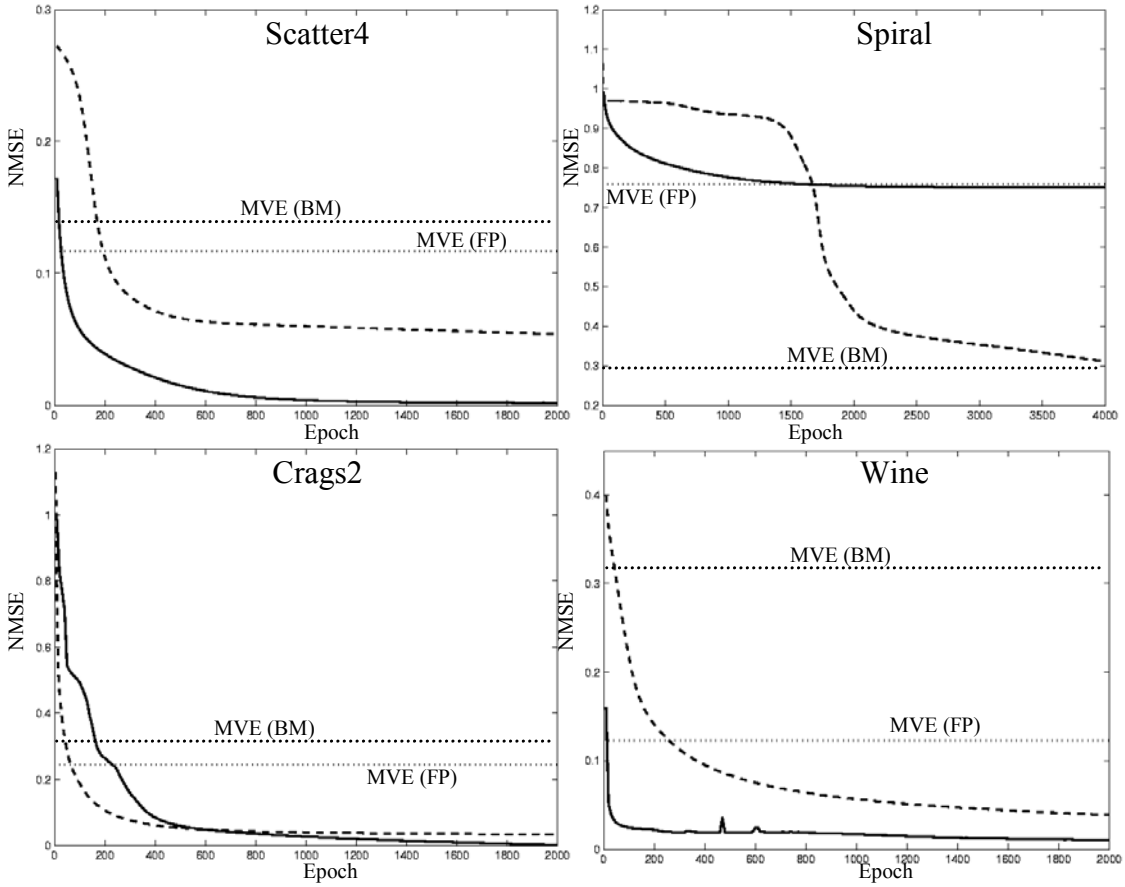


Figure B 4: Performance graphs for FP.

Note: The graphs in solid black represents the instantaneous mean training error of the supernet, the dashed line indicates the BM's instantaneous mean training error for the problem, and the dotted line provides a reference for the mean validation error (MVE) computed by the supernet.

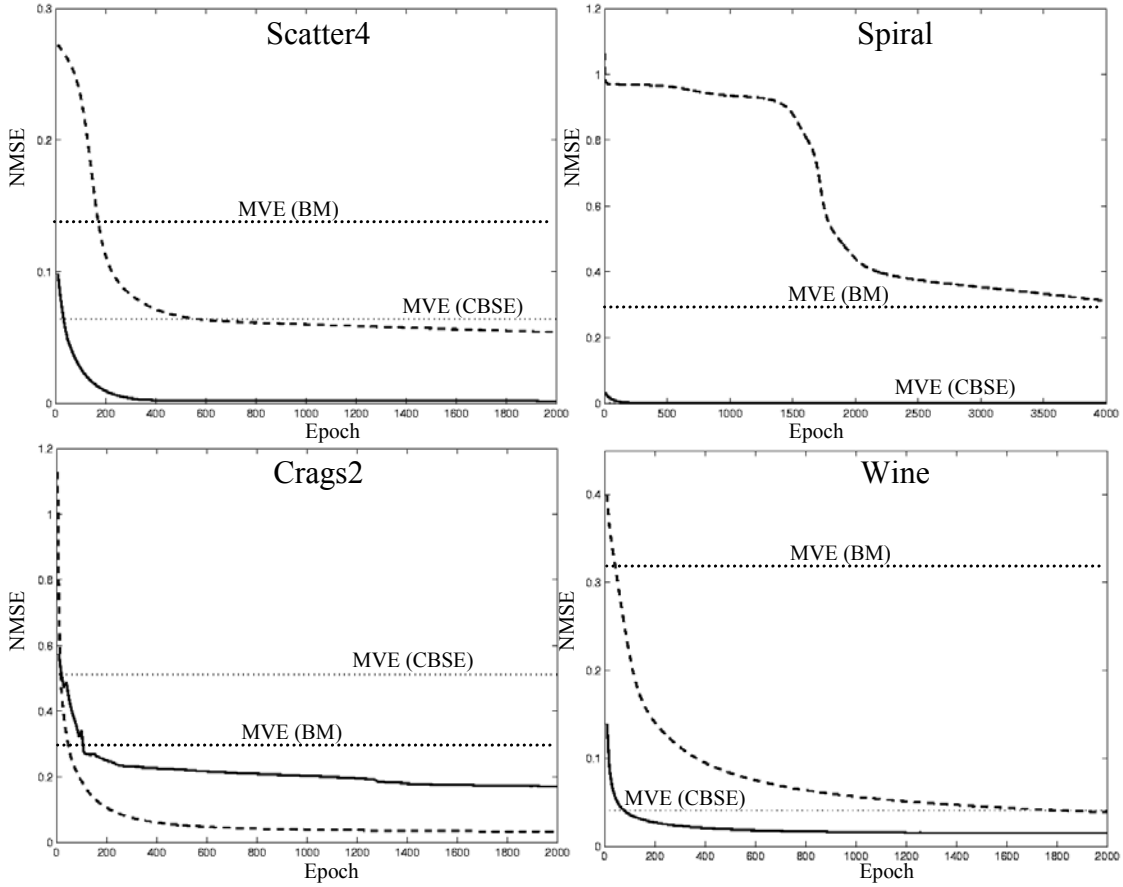


Figure B 5: Performance graphs for CBSE.

Note: The graphs in solid black represents the instantaneous mean training error of the supernet, the dashed line indicates the BM's instantaneous mean training error for the problem, and the dotted line provides a reference for the mean validation error (MVE) computed by the supernet.

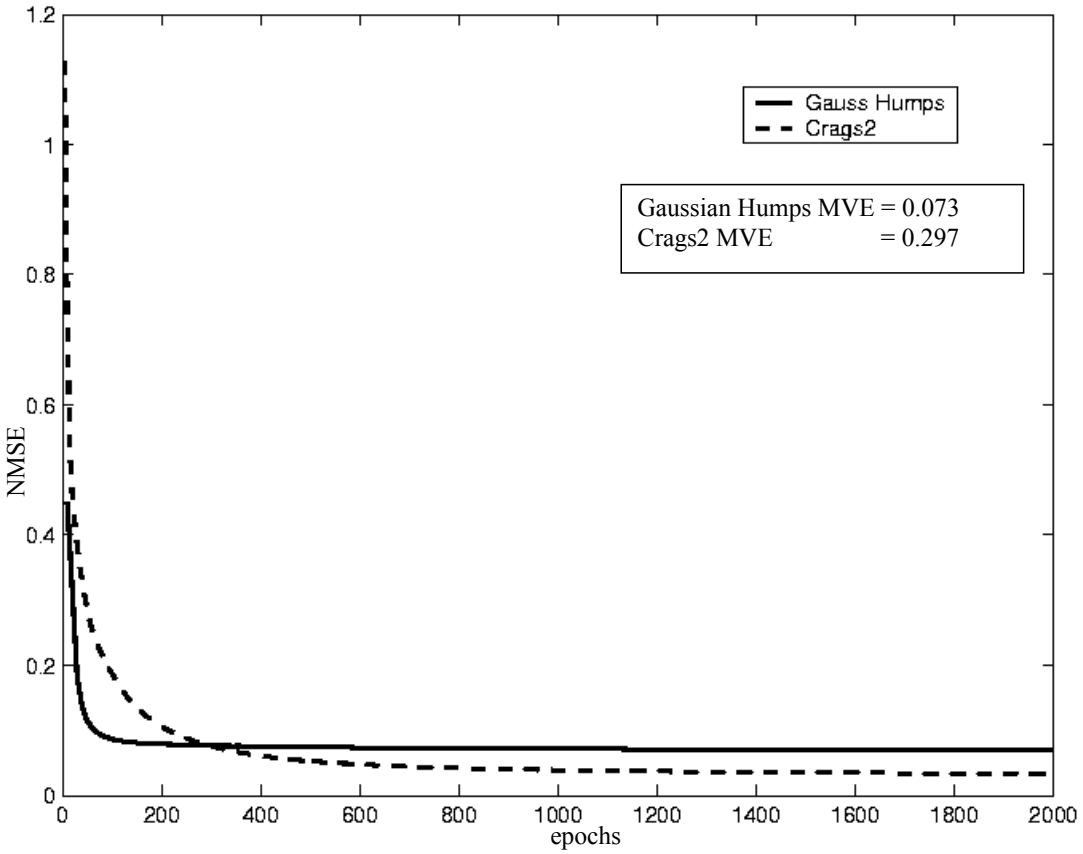


Figure B 6: Comparison of Gaussian Humps and Crags2.

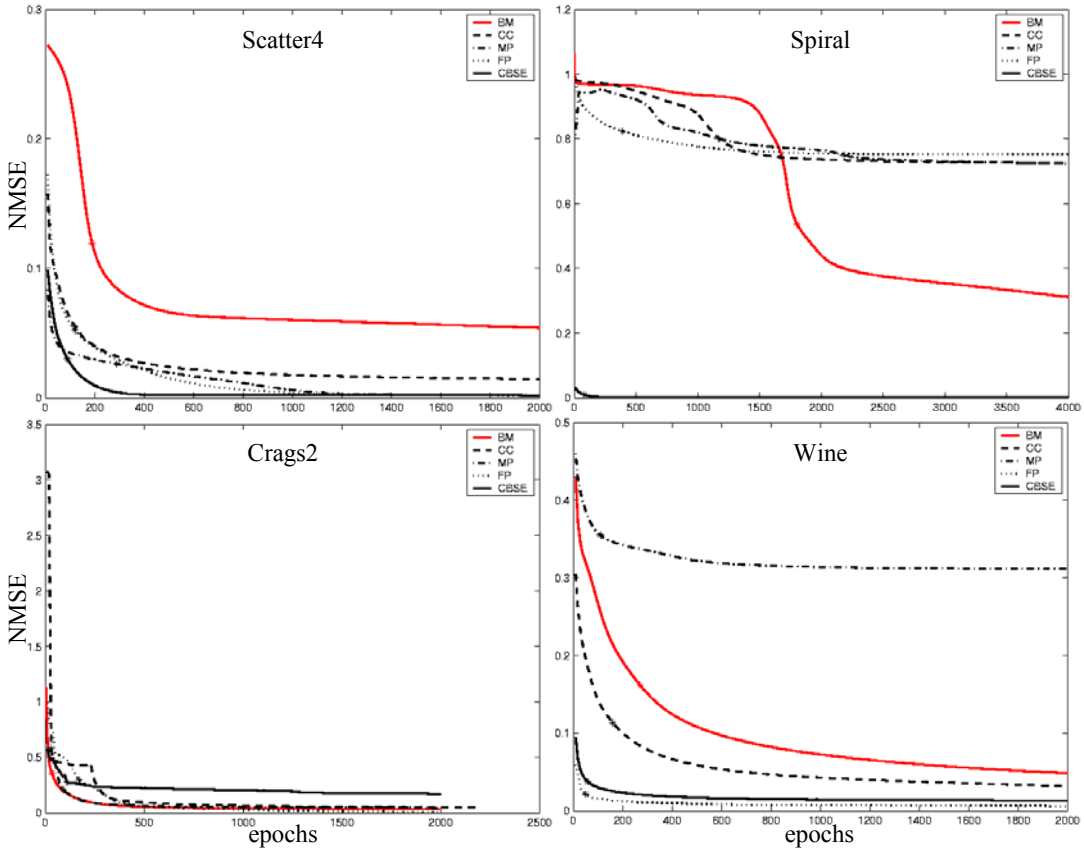


Figure B 7: Graph of instantaneous training error per problem set used for calculation of convergence times.

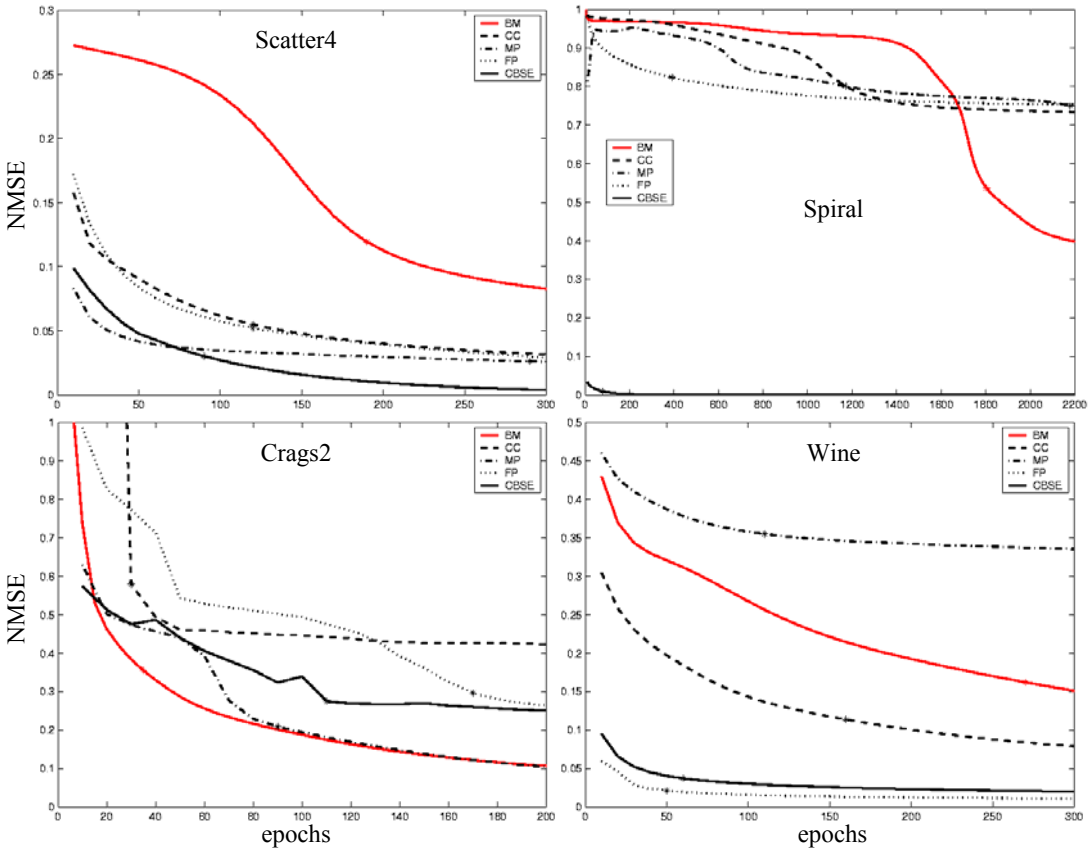


Figure B 8: Zoomed view of instantaneous training error graphs shown in Figure B 7.

Note: the starts on the curves indicate the 30% convergence positions.

Appendix C: MatLab Code

A free online reference guide to MatLab code is available from the MathWorks website (MatLab Documentation, 1994/2002).

Distance Glance Algorithm

The explanation of this algorithm is divided into:

- Inputs,
- Sub functions,
- Part I: Finding Cluster Prototypes, and
- Part II: Density Analysis of the prototypes

Inputs:

- ***S*** is set of input vectors of ***Ni*** dimensions for which clusters are to be determined.
- Lumping factor γ represents the minimum number of points in a cluster. This is used to combine points that are close to a given center.
- **Min_density,** the minimum density parameter of a cluster.
- Cluster distance ratio μ, used in deciding which clusters are too close together.

Sub-functions:

Define the function ***furthest (y, S)*** that determines the point **x** furthest from ***y*** in the set ***S***:

$$furthest(\vec{y}, S) = \vec{x} : \left\|\vec{x} - \vec{y}\right\|^2 > \left\|\vec{z} - \vec{y}\right\|^2 \forall \vec{x}, \vec{z} \in Q, \vec{x} \neq \vec{z}$$

$AOC_{furthest}$ = Nx·(3·Ni+1) *(same as the query function for K-means clustering)*

Define the function ***closest (y, S)*** that determines the point ***x*** closest to ***y*** in set ***S***:

$$closest(\vec{y}, S) = \vec{x} : \left\|\vec{x} - \vec{y}\right\|^2 < \left\|\vec{z} - \vec{y}\right\|^2 \forall \vec{x}, \vec{z} \in Q, \vec{x} \neq \vec{z}$$

Any ties in the above functions are resolved arbitrarily.

Define the function ***sqrdist* (*S*, *y*)** that returns a column vector of squared distance values between all remaining points in S and the point ***y***:

$$sqrdist(S, y) = \left\|y - x\right\|^2 \forall x \in S$$

Define the lumping function ***[D, S, r, ce, d] = lump (j, D, S, γ)*** that investigates the squared distances of column ***j*** in ***D*** and returns new ***D*** and ***S*** matrices together with a radius (***r***), approximated center (***ce***) for the newly found cluster, and the approximated density for this cluster (***d***). This is function operates as follows:

```
[D, S, r, ce, d] = lump ( j, D, S, lumping )
      % Lumping function as described in Section 4.1.2
      % j == the column of D to glance at
      above = find(D(:,j) > lumping);
      below = setdiff ([1:size(S,1)], above);
      d = size(below,1); % the density
      r = mean(D(below,j)); % squared radius for this cluster
      ce = mean(S(below,:));
      D = D(above,:);
      S = D(above,:);
```

PART I : Finding Cluster Prototypes

1. Determine the mean ***m*** of the dataset:

$$m = \frac{1}{Nx}\sum_{x \in S} x$$

2. Find the point c_1 furthest from ***m***:

$$c_1 = furthest(m, S)$$

3. Determine the first column of distances for ***D***:

$$D = sqrdist\ (S,\ c_1)$$

4. Apply the ***lumping*** for Nc = 1.
5. Determine the point c_2 that is furthest from c_1 by looking through the matrix ***D***.

$$c_2 = x_i : d_{i1} > d_{k1}\ , D = \{d_{11}, d_{21}, ...\}, S = \{x_1, x_2, ...x_i, ...\}$$

6. Compute a new column for ***D*** containing the distances from all points in ***S*** to c_2:

$$D = [\ D\ ,\ sqrdist\ (D, c_2)\ ]$$

7. Apply lumping for Nc = 2.
8. Find the distance between first two clusters:

$$dc_1 = sqrdist(c_1, c_2)$$

9. Find the minimums for distances between samples and clusters:

$$min\_dst = min(D);$$

10. Find the maximum of these minimums:

$$new\_idx = find(min\_dst == max(min\_dst))$$

11. Generate the new prototype:

$$c_{Nc+1} = S(new\_idx,:)$$

12. Determine the mean distance between this new cluster and the other clusters:

$$d\_cnew = mean(sqrdist(c, c_1) + sqrdist(c, c_1) + ... + sqrdist(c, c_{Nc}))$$

13. **IF** $d\_cnew/DC(Nc\text{-}1) > \mu$ **THEN** go to step 16 **ELSE** $Nc = Nc + 1$
14. Manage the new cluster prototype:
    a. Compute new distances:

$$D = [\ D\ ,\ sqrdist\ (D, c_i)\ ]$$

    b. Perform Lumping.

c. Compute the new mean distance between prototypes:

$$DC\ (Nc\text{-}1) = mean(sqrdist(c_i,c_j))\ for\ i=1..Nc,\ j=1:Nc,\ i \neq j$$

15. Go back to Step 9.
16. Proceed to Part 2.

At the end of this part the following variables contain information regarding the clusters:

- ***r*** is a column vector of squared radius values.
- ***ce*** is a matrix with center coordinates along the columns.
- ***c*** is a matrix containing the cluster prototypes.
- ***dc*** is a vector of mean distances between clusters

PART II : Density analysis

1. Find all points with a density above or equal to ***min_density***:

$$\boldsymbol{above = find(d \geq min\_density)}$$

2. Keep only the corresponding rows from the ***ce*** matrix and the ***d*** and ***r*** vectors.

$$\boldsymbol{ce = ce(above,:)}$$
$$\boldsymbol{r = r(above)}$$
$$\boldsymbol{d = d(above);}$$

3. Now determine the half mean of the remaining densities and keep only those clusters with densities above this value:

$$\boldsymbol{half\_mean = 0.5*mean(d)}$$
$$\boldsymbol{above = find\ (d > half\_mean)}$$
$$\boldsymbol{ce = ce(above,:)}$$
$$\boldsymbol{r = r(above)}$$
$$\boldsymbol{d = d(above)}$$

4. The algorithm is complete.

Appendix D: Terminology

TERM	DESCRIPTION
Activation	When used with a network the term "activation" refers to the process of performing some action to a network or subnet. During training of a network activation refers to the process of updating the weights in the network. During classification it refers to feeding an input to the network, the process of the neurons "firing" and producing an output.
ANN	*Abbr.* Artificial Neural Network.
Callback	A callback is a pointer to a function, which makes it possible for one segment of code to invoke a function whose name is not known. A callback handler is the function that is called when the callback pointer is invoked.
CDD	See Concise Descriptive Diagram
Concise Description Diagram (CDD)	In the scope of this thesis a CDD is a diagram constructed from sub-process glyphs and is a clear representation of a CANN.
Convergence Time (CT)	The number of training epochs needed to reach a training error that is a certain percentage above the convergent (i.e. final) training error.
Dicer/Dicer Agent	The set of processing modules responsible for organizing and splitting of training data amongst subnets.
Distance Glance (DG)	A heuristic determinative clustering algorithm that automatically determines the number of clusters in a given dataset based on the minimum distance between points.
Epoch	One cycle trough the training set
Feed / Fed	The process of applying an input to a feed-forward network,

	causing the neurons to be activated and a response to be yielded.
Firing	The process by which a neuron receives input stimuli and computes an output response.
Forgetfulness Problem	This refers to the problem of neurons forgetting previously learned trends by being exposed to greatly differing training examples. For instance, training a neuron on small values and large values makes it approximate neither particularly well.
Fuzzy Membership Value	A value between 0 and 1 that indicates the possibility of a given entity belonging to a certain set.
Generalization	The process by which a network produces suitable accurate predictions for arbitrary points within its input domain.
Input Domain	The space of all meaningful inputs to a network that yield meaningful results.
Kernel Functions	A pool of functions from which transfer functions are selected. Combinations of kernel functions are used as building blocks to construct an approximation function.
Mean Training Error (MTE)	The mean prediction error calculated for predicting outputs for a training set during training.
Mean Validation Error (MVE)	The mean prediction error calculated for predicting outputs for a validation set at the end of training.
Metadata	Data about data. Metadata is definitional data that provides information about data related to a specific application. For example $\boldsymbol{Nx}$, the number of examples in a training set, can be considered an element of metadata as it documents information related to data of which the training set comprises.
Model Comparison Coordinate (MCC)	The Model Comparison Coordinate is a 2D coordinate that represents the performance of a certain network in comparison to a base model. A small MCI value that is closer to (0,0) is considered better to one that is further from (0,0).

Network	In the context of this thesis, the term "network" refer to either a supernet or ANN.
Neuron	An information-processing unit that is fundamental to the operation of a neural network (Haykin 1994: 8)
Overfitting	Giving a neural too many degrees of freedom that causes predicted outputs to become inaccurate due too there being insufficient restriction on the behavior of inputs in regions of the input space between training examples.
Overseer	Human wanting to obtain a training network.
Prediction Error	An error measurement proportional to the difference between a network's predicted output and the desired output.
Relative Convergence Time (RCT)	A ratio between the convergence time of a supernet and the convergence time of the base model
Response	The output calculated from a feed operation
Splicer/Splicer Agent	The set of processing modules responsible for combining outputs from a set of subnets to produce a final output for the supernet.
Stimulus	An input to a network or subnet
Subnet	A component of a network that is in its own right a ANN.
Supernet	A set of subnets joined together with *dicer* and *splicer* agents.
Synapses	Elementary structural and functional units that mediate the interaction between neurons.
Transfer Function	A function is used in an artificial neuron to map the weighted sum of inputs to an output.
Yield	The process by which a network generates a predicted output.

VITA

Simon Winberg was born in Cape Town, South Africa on March 24, 1976. He attended schools in Stockholm, Sweden and Cape Town, South Africa. He graduated from Cape Town High School in 1994. He received a Bachelor of Science degree in Computer Science from the University of Cape Town in South Africa. Then he worked for two years in the electronics industry for the Electronic Development House – a South African company specializing in the development and manufacture of custom-made digital electronic products – before pursuing a Masters degree at the University of Tennessee Space Institute.